

Transformers

Orlando Ramos Flores

Contenido

- Transformers
 - Encoder
 - Self-Attention
 - Multi-Head Attention
 - Positional Encoding
 - Residual connections
 - Decoder
- Implementación en Pytorch

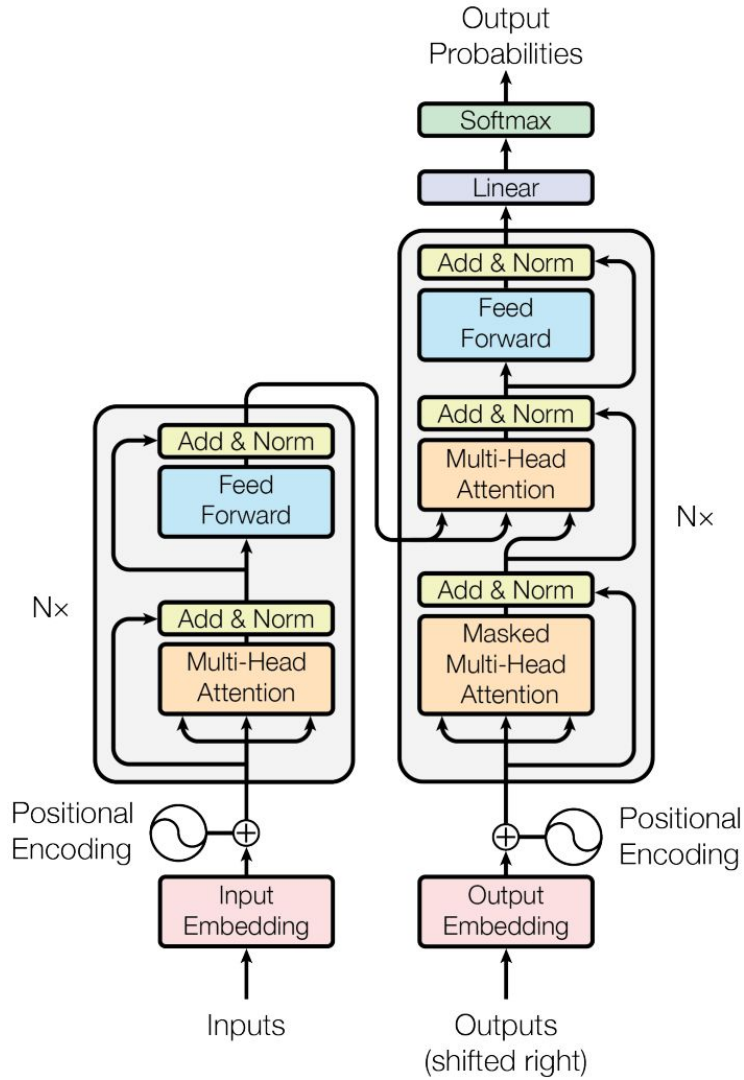
Transformers

¿Qué es un Transformer?

- Un Transformer es una arquitectura para transformar una secuencia en otra con la ayuda de un Encoder y Decoder (seq2seq, con algunas diferencias).
- La principal diferencia de los Transformers con los modelos seq2seq es que eliminan la necesidad de usar Redes Recurrentes (GRU, LSTM , Bi-LSTM).
- El Transformer es un modelo bastante complejo que utiliza varios tipos nuevos de bloques de construcción o capas.

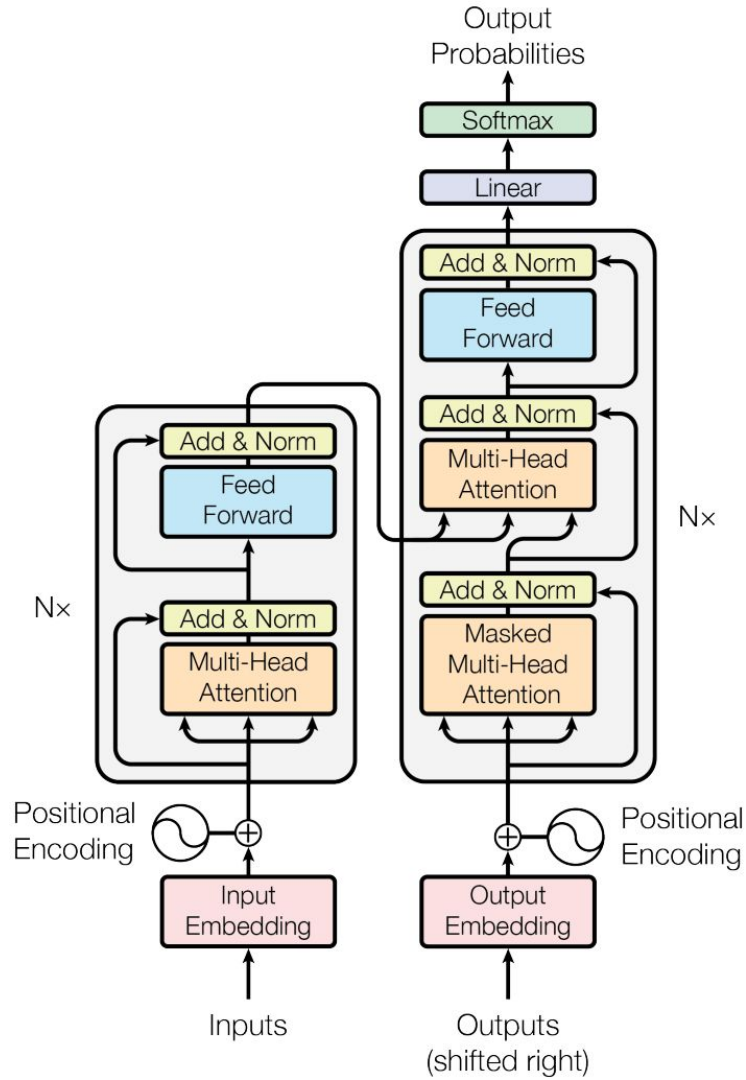
Transformer: Arquitectura

- **Encoder** (izquierda).
 - Compuesto por una pila de $N=6$ capas idénticas.
 - Cada capa tiene dos subcapas.
 - El primero es un mecanismo multi-head self-attention.
 - El segundo es una red position-wise FFN simple.
 - Una conexión residual alrededor de cada una de las dos subcapas,
 - Seguida de de una capa de normalización.



Transformer: Arquitectura

- **Decoder** (derecha).
 - Compuesto por una pila de $N=6$ capas idénticas.
 - Además de las dos subcapas en cada capa del encoder, el decoder inserta una tercera subcapa, que realiza multi-head attention sobre la salida de la pila del encoder.
 - De forma similar al encoder, se emplean conexiones residuales alrededor de cada una de las subcapas,
 - Seguidas de una capa de Normalización
 - También se modifica la subcapa self-attention en la pila del decoder para evitar que las posiciones presten atención a las posiciones posteriores.



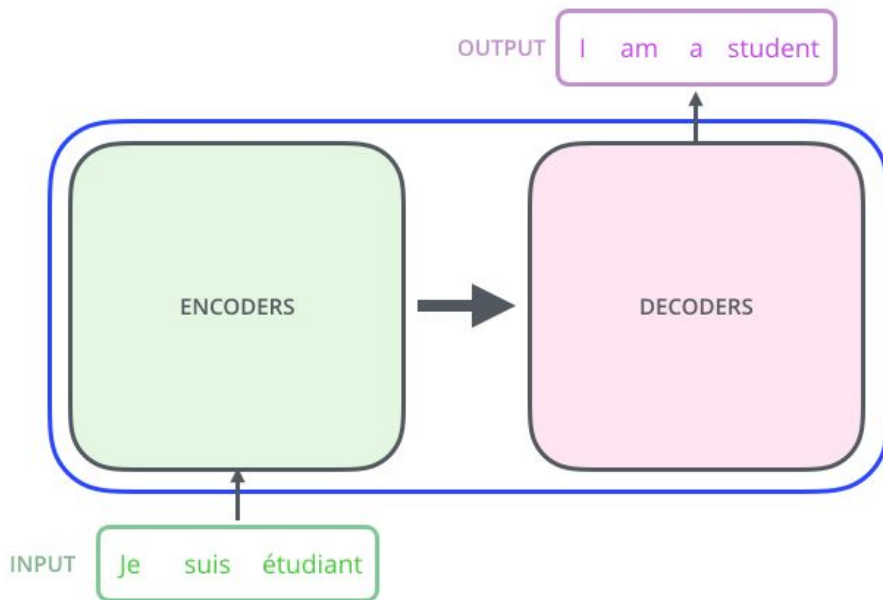
Transformers: Intuición ilustrada

Transformer



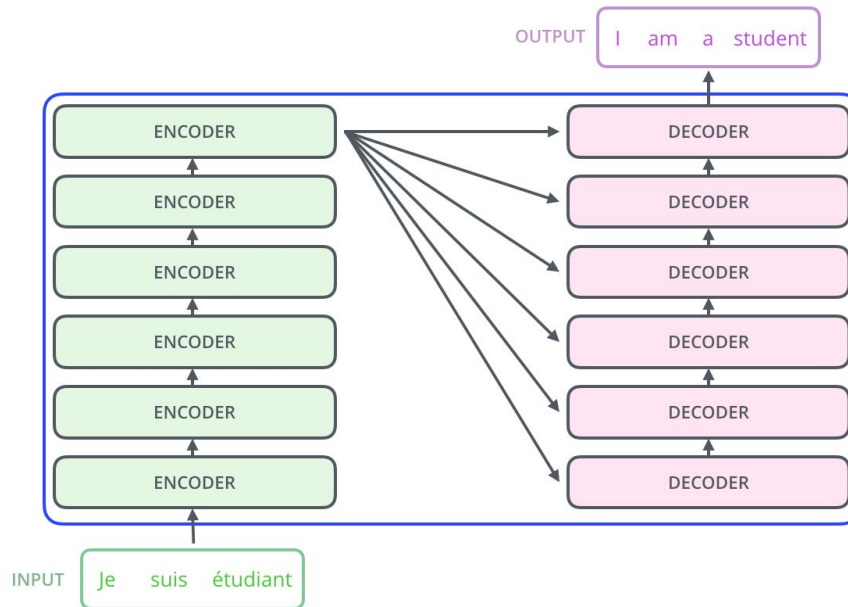
Transformer

- Al abrir el Transformer, vemos un componente de **codificación**, un componente de **decodificación** y conexiones entre ellos.



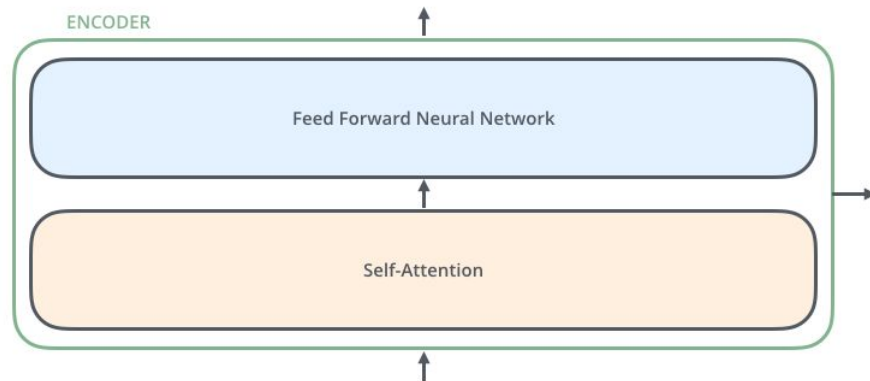
Transformer

- El componente **encoder** es una pila de codificadores
- Se apilan seis de ellos uno encima del otro
- El componente **decoder** es una pila de decodificadores del mismo número



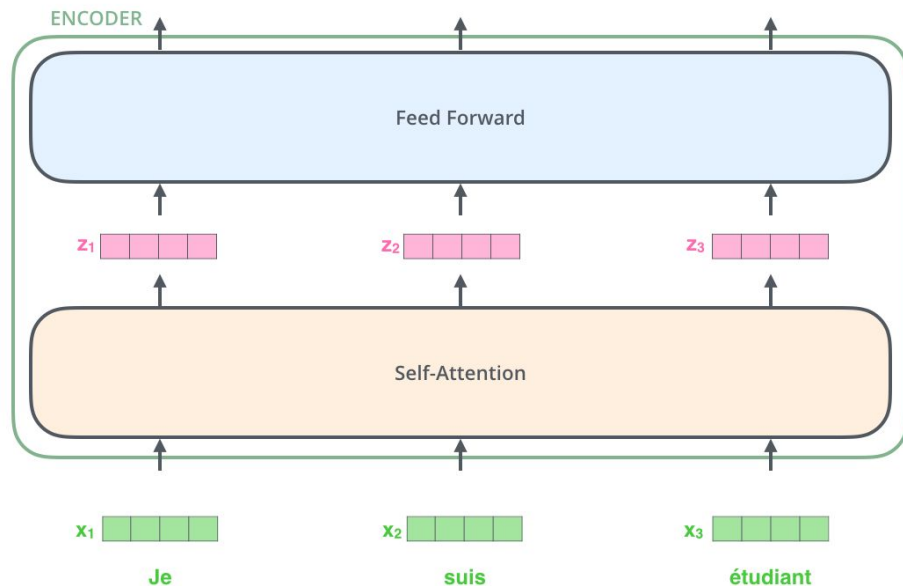
Transformer

- Los **encoders** son todos idénticos en estructura (pero no comparten pesos)
- Cada uno se divide en dos subcapas
- Las entradas del **encoder** fluyen primero a través de una capa **self-attention**, una capa que ayuda al **encoder** a ver otras palabras en la oración de entrada mientras codifica una palabra específica.
- Las salidas de la capa **self-attention** se envían a una red neuronal de FFN.
- La misma red FFN se aplica de forma independiente a cada posición.



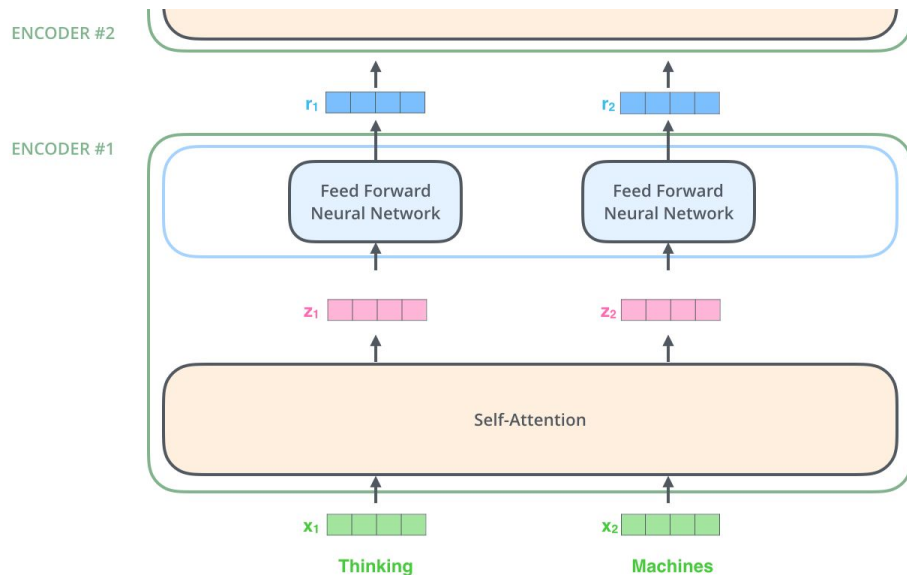
Transformer

- Los embeddings solo ocurren en el **encoder** de más abajo.
- La abstracción que es común para todos los encoders, es que reciben una lista de vectores, cada uno de tamaño 512: en el **encoder** inferior, serían word embeddings, pero en otros **encoders**, sería la salida del **encoder** que está directamente debajo.
- El tamaño de esta lista es un hiper-parámetro que se puede configurar; básicamente, sería la longitud de la oración más larga en nuestro conjunto de datos de entrenamiento.



Transformer

- Como se ha mencionado, un encoder recibe una lista de vectores como entrada.
- Procesa esta lista pasando estos vectores a una capa de "**self-attention**",
- Luego a una red neuronal de avance (FFN),
- Después se envía la salida hacia arriba al siguiente encoder.



Transformer: Self-Attention

- Anteriormente se ha visto cómo el **decoder** de una RNN podría prestar atención a una secuencia de entrada para capturar incrustaciones contextuales de cada entrada.
- Sin embargo, en lugar de que el **decoder** atienda al **encoder**, se puede modificar el modelo para que el **encoder** se atienda a sí mismo. Esto se llama **self-attention** (autoatención).
- En términos sencillos, el mecanismo **self-attention** permite que las entradas interactúen entre sí ("**self**") y descubran a quién deben prestar más atención ("**attention**").
- Dada una secuencia de entrada $\mathbf{x}_1, \dots, \mathbf{x}_n$ donde $\mathbf{x}_i \in \mathbb{R}^d$, self-attention puede generar una secuencia de salida del mismo tamaño usando: $\mathbf{y}_i = \text{Attn}(\mathbf{x}_i, (\mathbf{x}_1, \mathbf{x}_1), \dots, (\mathbf{x}_n, \mathbf{x}_n))$
- Donde la **query** es \mathbf{x}_i , y las **keys** y **values** son todas las entradas (válidas) $\mathbf{x}_1, \dots, \mathbf{x}_n$
- Para usar esto en el **decoder**, se establece $\mathbf{x}_i = \mathbf{y}_{i-1}$, $n = i - 1$, de modo que todas las salidas generadas previamente estén disponibles.

Transformer: Self-Attention

- En el momento del entrenamiento, ya se conocen todas las salidas, por lo que podemos evaluar la función anterior en paralelo, superando el cuello de botella secuencial del uso de RNN.
- Además de mejorar la velocidad, la **self-attention** puede brindar mejores representaciones del contexto.
- Como ejemplo, considere traducir las oraciones:

The animal didn't cross the street because it was too tired.
L'animal n'a pas traversé la rue parce qu'il était trop fatigué.

The animal didn't cross the street because it was too wide.
L'animal n'a pas traversé la rue parce qu'elle était trop large.

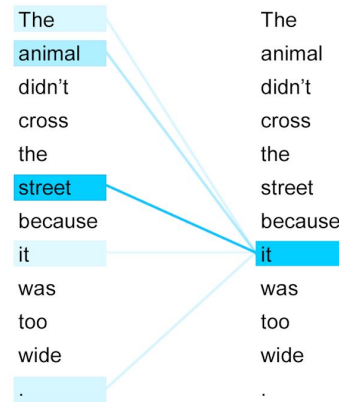
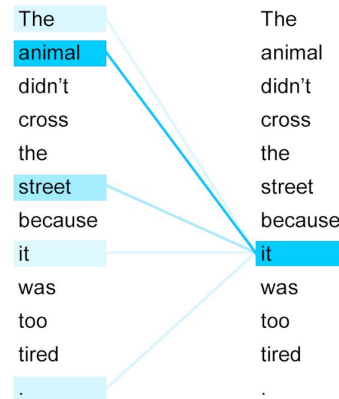
Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2021). Dive into deep learning. arXiv preprint arXiv:2106.11342.

Murphy, K. P. (2022). Probabilistic machine learning: an introduction. MIT press.

[Google AI Blog: Transformer: A Novel Neural Network Architecture for Language Understanding](#)

Transformer: Self-Attention

- Traducir del inglés:
 - *"The animal didn't cross the street because it was too tired"*
 - *"The animal didn't cross the street because it was too wide"*
- Para generar un pronombre del género correcto en francés, se necesita saber a qué se refiere **"it"**, esto se llama resolución de coreferencia.
- En el primer caso, la palabra **"it"** se refiere al animal.
- En el segundo caso, la palabra **"it"** se refiere a la calle.
- La **self-attention** aplicada a la oración en inglés puede resolver esta ambigüedad.
- En la primera oración, la representación de **"it"** depende de las representaciones anteriores de **"animal"**, mientras que en la última depende de las representaciones anteriores de **"street"**.



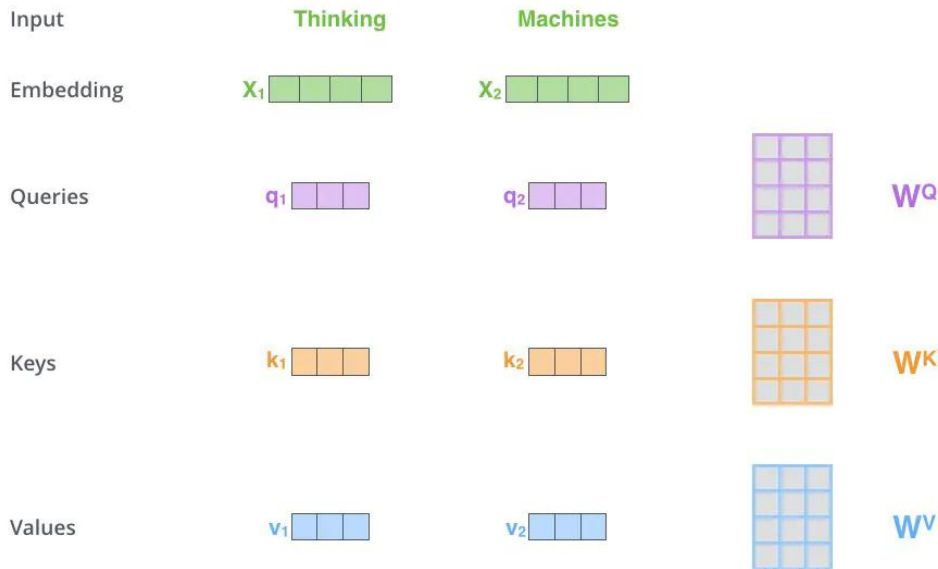
Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2021). Dive into deep learning. arXiv preprint arXiv:2106.11342.

Murphy, K. P. (2022). Probabilistic machine learning: an introduction. MIT press.

[Google AI Blog: Transformer: A Novel Neural Network Architecture for Language Understanding](#)

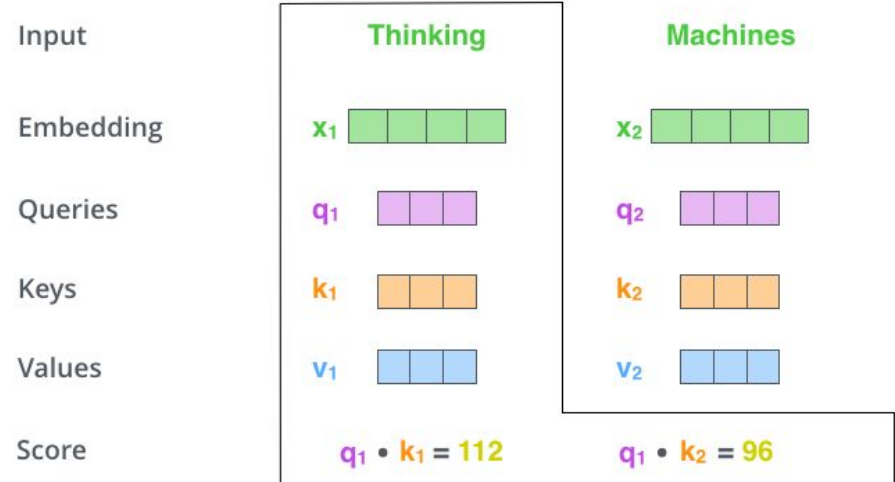
Transformer: Self-Attention

- El primer paso para calcular la **self-attention** es crear tres vectores de cada uno de los vectores de entrada del **encoder** (en este caso, la incrustación de cada palabra).
- Entonces, para cada palabra, creamos un vector de consulta (**Q**), un vector clave (**K**) y un vector de valor (**V**).
- Estos vectores se crean multiplicando la incrustación por tres matrices que se entrenaron durante el proceso de entrenamiento.
- Estos nuevos vectores son más pequeños en dimensión que el vector de embeddings
- Query, Key y Value: son abstracciones (vectores) útiles para calcular y pensar la atención.



Transformer: Self-Attention

- El segundo paso para calcular **self-attention** es calcular un score.
- Se está calculando **self-attention** de la primera palabra en este ejemplo, "Thinking".
- Se necesita calcular cada palabra de la oración de entrada contra esta palabra.
- El score determina cuánto enfoque colocar en otras partes de la oración de entrada a medida que se codifica una palabra en una posición determinada.



Transformer: Self-Attention

- El score se obtiene calculando el producto punto del **query vector** con el **key vector** de la palabra respectiva que se está calculando.
- Entonces, si se está procesando **self-attention** de la palabra en la posición #1, el primer score sería el producto punto de **q1** y **k1**.
- La segunda puntuación sería el producto escalar de **q1** y **k2**.

Input

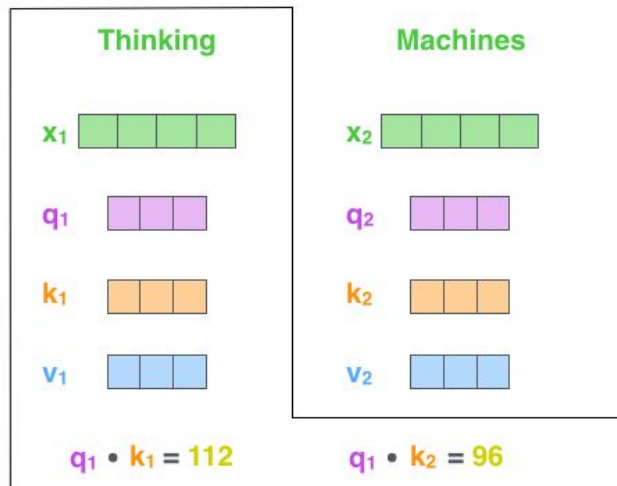
Embedding

Queries

Keys

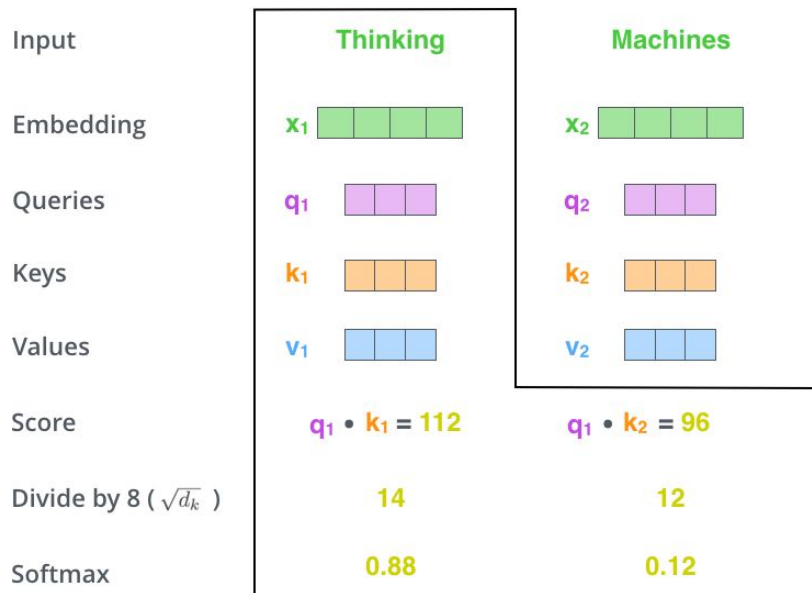
Values

Score



Transformer: Self-Attention

- El tercer y cuarto paso consisten en dividir las puntuaciones por 8 (la raíz cuadrada de la dimensión de los **key vector** utilizados en el [paper](#): 64).
- Esto lleva a tener gradientes más estables.
- Podría haber otros valores posibles aquí, pero este es el predeterminado), luego se pasa el resultado a través de una operación *softmax*.
- *Softmax* normaliza los scores para que todos sean positivos y sumen 1.
- Los scores *softmax* determinan cuánto expresa cada palabra en esta posición.
- La palabra en esta posición tendrá el score de *softmax* más alto, pero a veces es útil prestar atención a otra palabra que sea relevante para la palabra actual.



Transformer: Self-Attention

- El quinto paso es multiplicar cada **value vector** por el score *softmax* (en preparación para sumarlos).
- La intuición aquí es mantener intactos los valores de las palabras en las que queremos centrarnos y “ahogar” las palabras irrelevantes (multiplicándolas por números pequeños como 0.001).
- El sexto paso es sumar los **weighted value vectors**.
- Esto produce la salida de la capa de **self-attention** en esta posición (para la primera palabra).
- Eso concluye el cálculo de **self-attention**.

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

Transformer: Self-Attention

- El vector resultante es uno de los que se envían a la red FFN.
- Sin embargo, en la implementación real, este cálculo se realiza en forma de matriz para un procesamiento más rápido.

Input

Embedding

Queries

Keys

Values

Score

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

Softmax

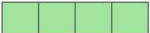
Softmax

X

Value

Sum

Thinking

x_1 

q_1 

k_1 

v_1 

$q_1 \cdot k_1 = 112$

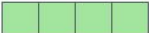
14

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 

Transformer: Self-Attention

- El primer paso es calcular las matrices **Query**, **Key** y **Value**.
- Se realiza empaquetando los embeddings en una matriz **X** y multiplicándola por las matrices de peso que se han entrenado (**W_Q**, **W_K**, **W_V**).

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

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

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

Transformer: Self-Attention

- Finalmente, dado que se está tratando con matrices, se pueden condensar los pasos dos a seis en una fórmula para calcular los resultados de la capa de self-attention.

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

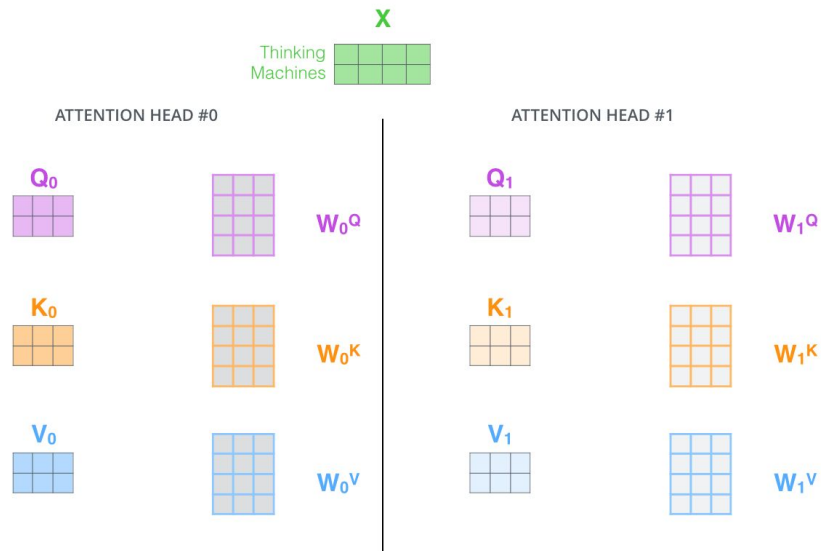
Transformer: Multi-Head Attention

El mecanismo llamado atención de "múltiples cabezas" mejora el rendimiento de la capa de atención de dos maneras:

1. Amplía la capacidad del modelo para enfocarse en diferentes posiciones.
2. Da a la capa de atención múltiples "subespacios de representación".
 - Con la atención de varios cabezales no solo se tiene uno, sino varios conjuntos de matrices de ponderación de Query/key/Value (el Transformador usa ocho cabezas de atención, por lo que se termina con ocho conjuntos para cada encoder/decoder).
 - Cada uno de estos conjuntos se inicializa aleatoriamente.
 - Luego, después del entrenamiento, cada conjunto se usa para proyectar los embeddings de entrada (o vectores de encoders/decoders inferiores) en un subespacio de representación diferente.

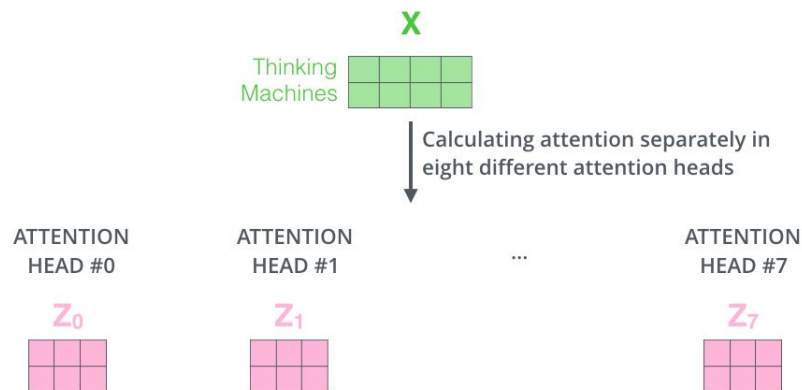
Transformer: Multi-Head Attention

- Con Multi-head attention, se mantienen matrices de peso Q/K/V separadas para cada cabeza, lo que da como resultado matrices Q/K/V diferentes.
- Como se hizo previamente, se multiplica X por las matrices $W_Q/W_K/W_V$ para producir matrices Q/K/V.



Transformer: Multi-Head Attention

- Si se hace el mismo cálculo self-attention descrito anteriormente, solo ocho veces diferentes con diferentes matrices de peso, se termina con ocho matrices Z diferentes.



Transformer: Multi-Head Attention

- Esto deja un pequeño desafío.
- La capa FFN no espera ocho matrices, espera una sola matriz (un vector para cada palabra).
- Así que se necesita una forma de condensar estas ocho en una sola matriz.

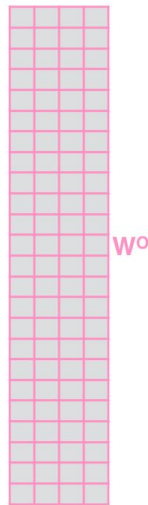
1) Concatenate all the attention heads



2) Multiply with a weight matrix W^O that was trained jointly with the model

x

3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN



Transformer: Multi-Head Attention

1) This is our
input sentence*

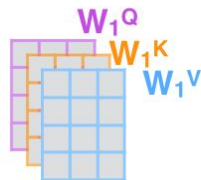
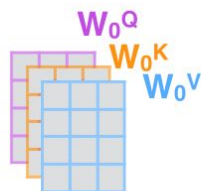
2) We embed
each word*

3) Split into 8 heads.
We multiply X or
 R with weight matrices

4) Calculate attention
using the resulting
 $Q/K/V$ matrices

5) Concatenate the resulting Z matrices,
then multiply with weight matrix W^O to
produce the output of the layer

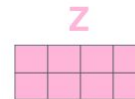
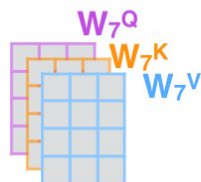
Thinking
Machines



...

...

...

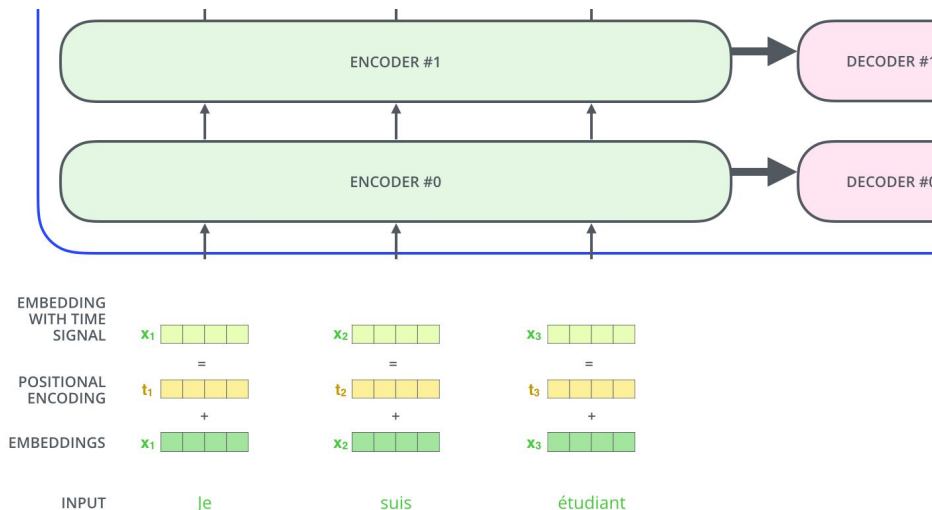


* In all encoders other than #0,
we don't need embedding.
We start directly with the output
of the encoder right below this one



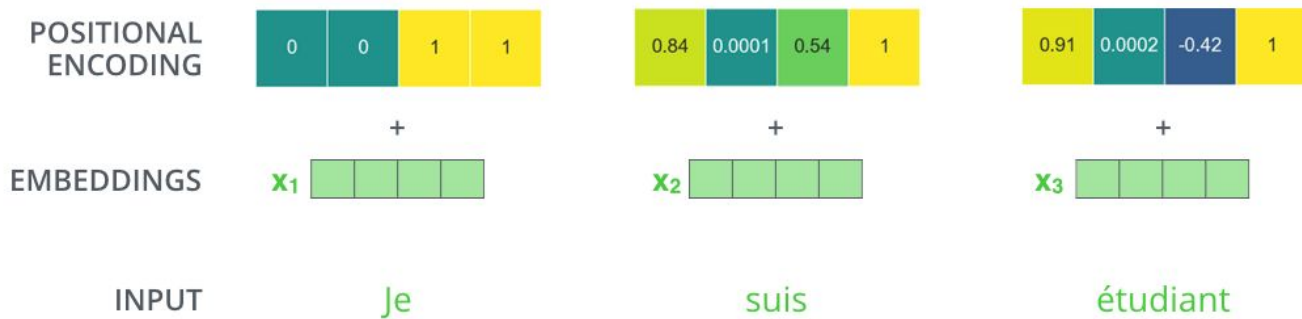
Transformer: Positional Encoding

- El Positional Encoding explica el orden de las palabras en la secuencia de entrada.
- El Transformer agrega un vector a cada embedding de entrada.
- Estos vectores siguen un **patrón específico** que aprende el modelo, lo que le ayuda a determinar la posición de cada palabra o la distancia entre diferentes palabras en la secuencia.
- La intuición aquí es que agregar estos valores a los embeddings proporciona distancias significativas entre los vectores de embeddings una vez que se proyectan en vectores Q/K/V y durante la atención del producto punto.



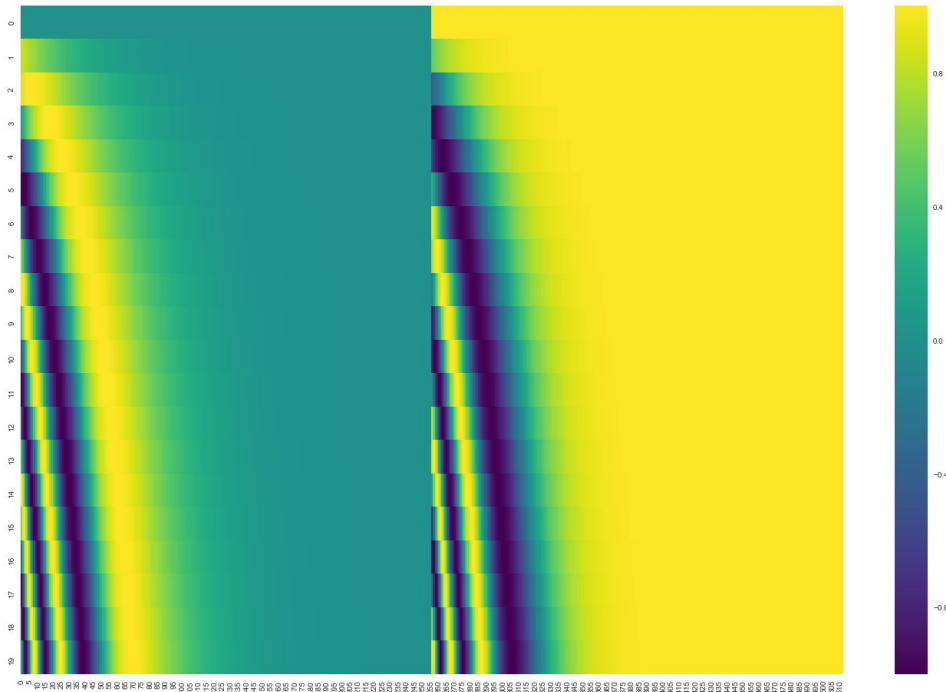
Transformer: Positional Encoding

- Si se asume que la incrustación tiene una dimensionalidad de 4, los positional encodings reales se verían:



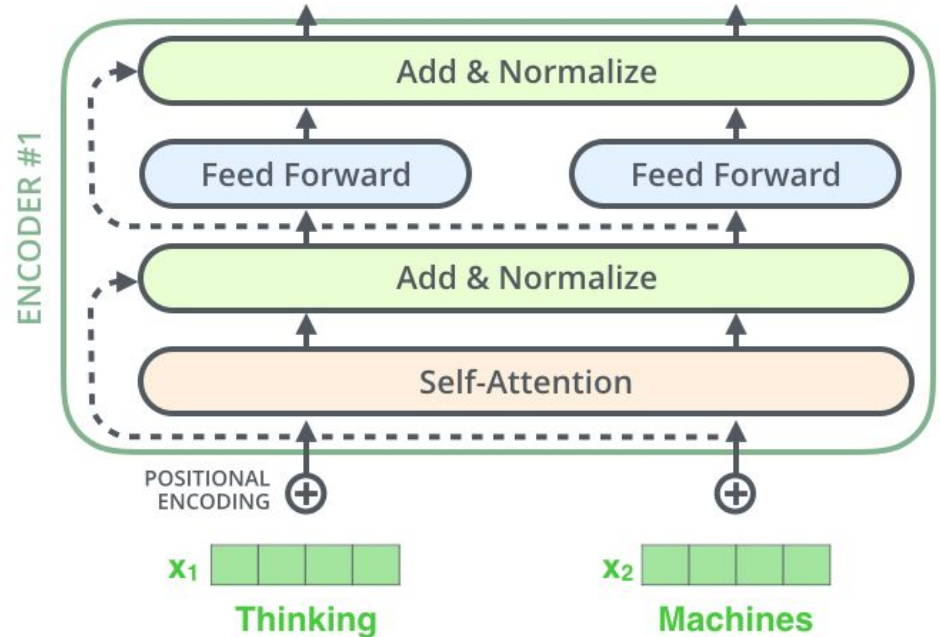
Transformer: Positional Encoding

- En la siguiente figura, cada fila corresponde a una positional encoding de un vector.
- La primera fila sería el vector que se agregaría al embedding de la primera palabra en una secuencia de entrada.
- Cada fila contiene 20 palabras (filas) con un tamaño de incrustación de 512 (columnas), cada uno con un valor entre 1 y -1.
- Están codificados por colores para que el patrón sea visible.
- Los valores de la mitad izquierda los genera una función (que usa el seno),
- Y los valores de la mitad derecha los genera otra función (que usa el coseno).
- Luego se concatenan para que cada uno forme los vectores positional encoding.



Transformer: Residual connections

- Un detalle de la arquitectura del encoder es que cada subcapa (self-attention, FFN) en cada **encoder** tiene una **conexión residual** a su alrededor, y va seguida de un paso de normalización de capa.

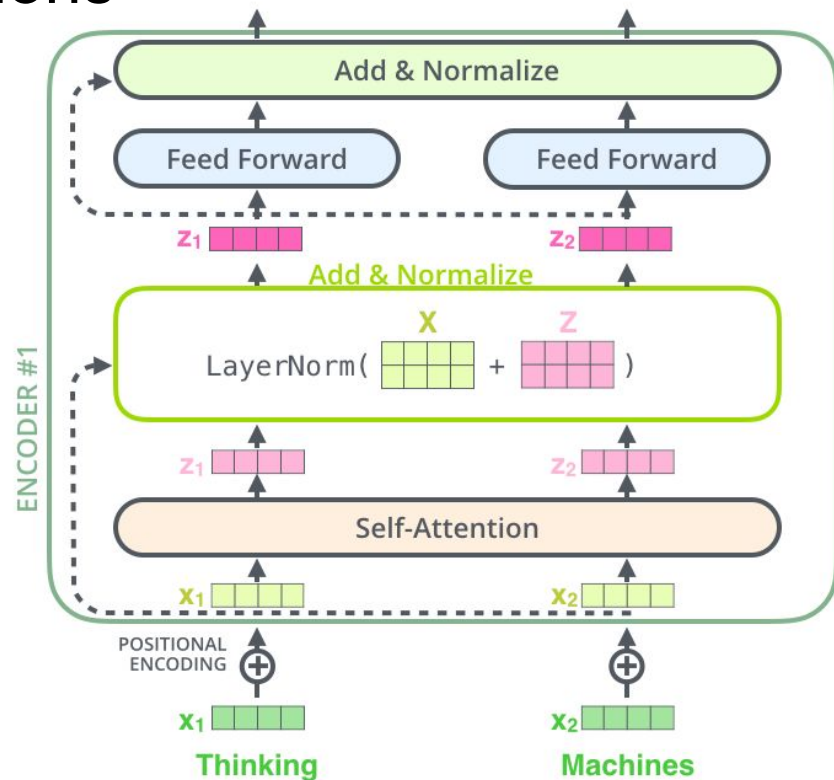


[The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time](#)

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

Transformer: Residual connections

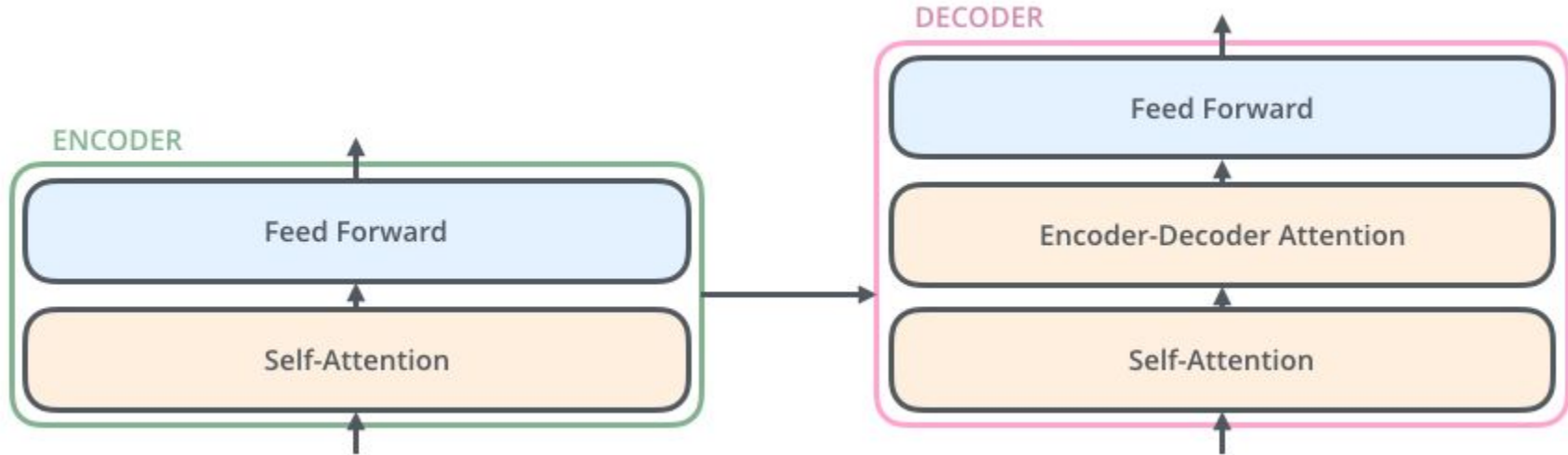
- Un detalle de la arquitectura del encoder es que cada subcapa (self-attention, FFN) en cada **encoder** tiene una **conexión residual** a su alrededor, y va seguida de un paso de normalización de capa.
- Si los vectores se pudieran visualizar y la operación de capa de normalización asociada con la self-attention, se vería así:



[The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time](#)

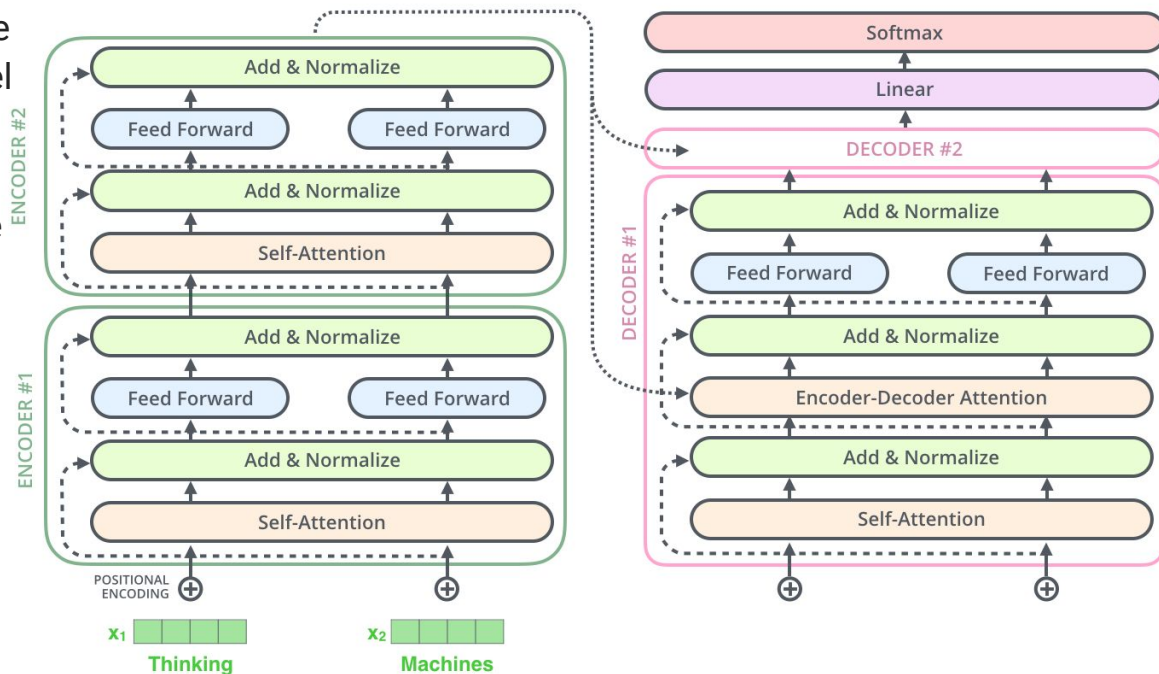
He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

Transformer: Decoder



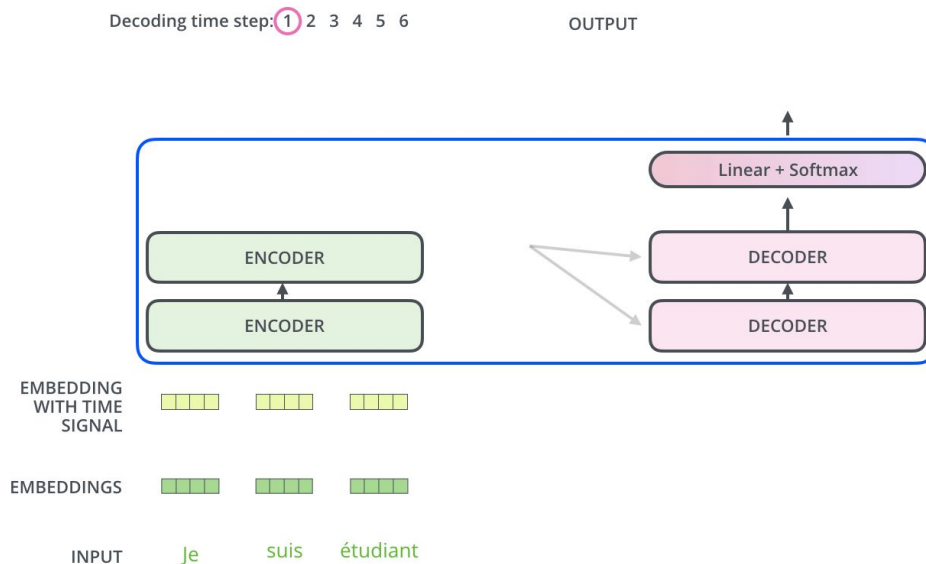
Transformer: Decoder

- Todo lo descrito anteriormente se aplica también a las subcapas del **decoder**.
- Si se asume un Transformer de 2 encoders y decoders apilados, se verían así:



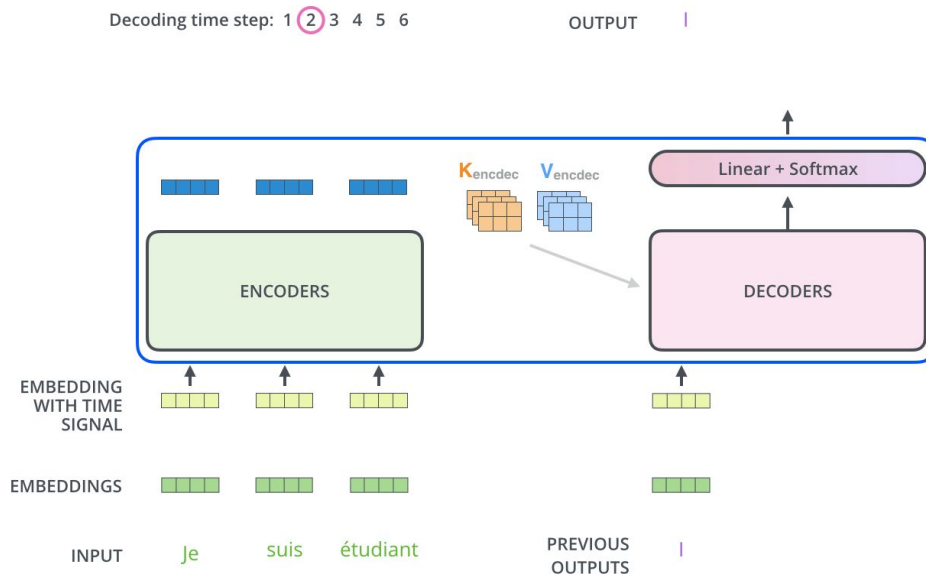
Transformer: Decoder

- El **encoder** comienza procesando la secuencia de entrada.
- La salida del **encoder** superior se transforma luego en un conjunto de vectores de atención **K** y **V**.
- Estos son utilizados por cada **decoder** en su capa de "encoder-decoder attention" que ayuda al **decoder** a enfocarse en los lugares apropiados en la secuencia de entrada
- Cada paso en la fase de decodificación genera un elemento de la secuencia de salida (la oración de traducción al inglés en este caso).



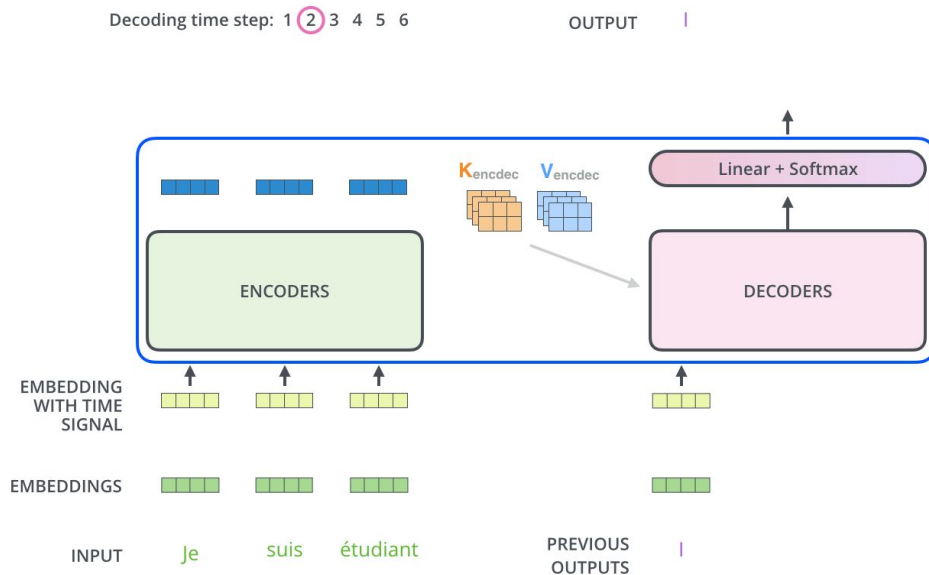
Transformer: Decoder

- Los siguientes pasos repiten el proceso hasta que se alcanza un símbolo especial que indica que el decodificador del transformador ha completado su salida.
- La salida de cada paso se alimenta al **decoder** inferior en el siguiente paso de tiempo, y los **decoders** aumentan sus resultados de decodificación al igual que lo hicieron los **encoders**.
- Tal como se hizo con las entradas del **encoder**, se incrustan y agrega el positional encoding a esas entradas del **decoder** para indicar la posición de cada palabra.



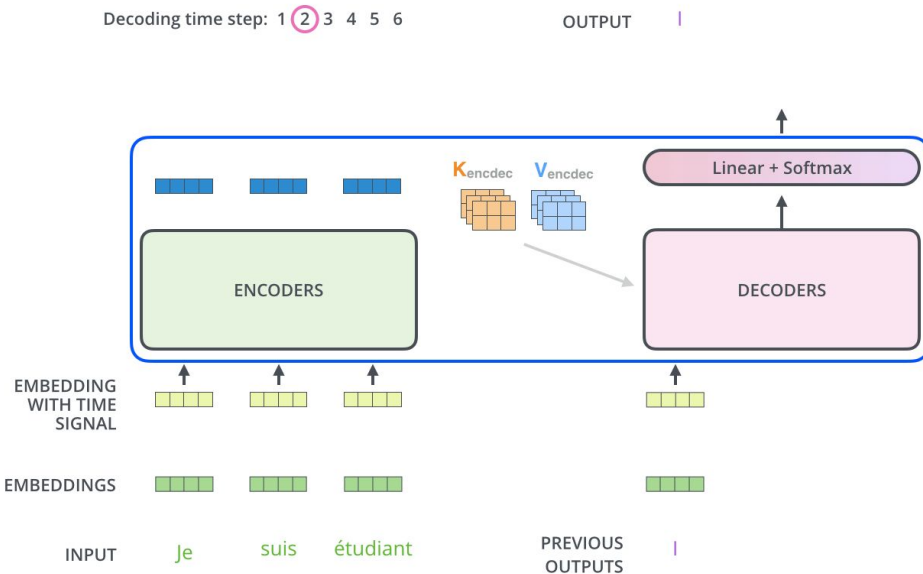
Transformer: Decoder

- Las capas **self-attention** en el **decoder** funcionan de una manera ligeramente diferente a la del **encoder**:
 - En el **decoder**, la capa **self-attention** solo puede atender posiciones anteriores en la secuencia de salida.
 - Esto se hace enmascarando posiciones futuras (configurándolas en $-\infty$) antes del paso *softmax* en el cálculo de **self-attention**.



Transformer: Decoder

- Las capas **self-attention** en el **decoder** funcionan de una manera ligeramente diferente a la del **encoder**:
 - En el **decoder**, la capa **self-attention** solo puede atender posiciones anteriores en la secuencia de salida.
 - Esto se hace enmascarando posiciones futuras (configurándolas en $-\infty$) antes del paso *softmax* en el cálculo de **self-attention**.
 - La capa "Encoder-Decoder Attention" funciona como la self-attention de múltiples cabezas, excepto que crea su matriz de *Queries* a partir de la capa debajo de ella y toma la matriz de *Keys* y *Values* de la salida de la pila del codificador.

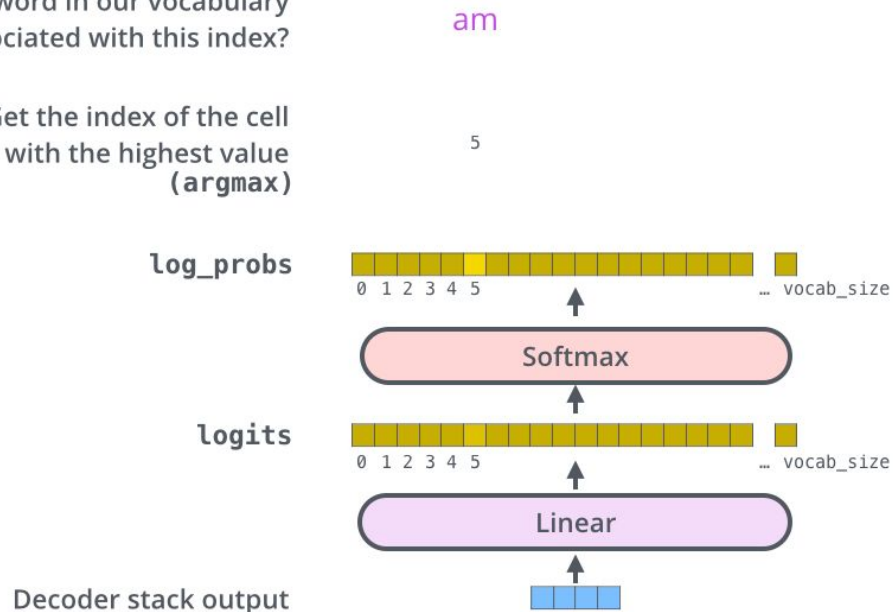


Transformer: La capa FC final y Softmax

- La pila del **decoder** genera un vector de *floats*. ¿Cómo se convierte eso en una palabra? Ese es el trabajo de la capa final Linear, seguida de una capa *Softmax*.
- La capa lineal final es una red neuronal simple totalmente conectada (FC) que proyecta el vector producido por la pila de decoders en un vector mucho, mucho más grande llamado vector logits.

Which word in our vocabulary
is associated with this index?

Get the index of the cell
with the highest value
(**argmax**)

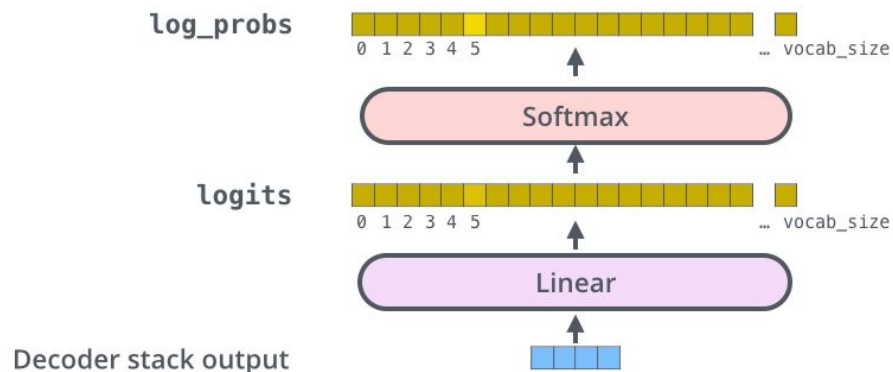


Transformer: La capa FC final y Softmax

- Supongamos que el modelo conoce 10,000 palabras únicas en inglés (el "vocabulario de salida") que aprendió de su conjunto de datos de entrenamiento.
- Esto haría que el **vector logits** tenga 10,000 celdas de ancho, cada celda correspondiente al score de una palabra única.
- Así es como interpretamos la salida del modelo seguido de la capa FC.

Which word in our vocabulary
is associated with this index?

Get the index of the cell
with the highest value
(**argmax**)

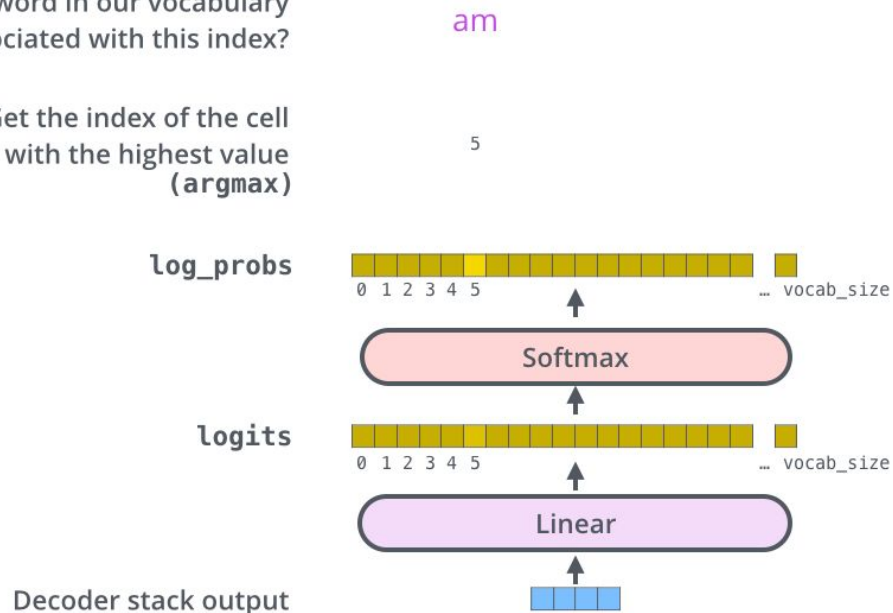


Transformer: La capa FC final y Softmax

- La capa *softmax* entonces convierte esos scores en probabilidades (todos positivos, todos suman 1.0).
- Se elige la celda con la probabilidad más alta y la palabra asociada a ella se produce como salida para este paso de tiempo.
- Esta figura comienza desde abajo con el vector producido como salida de la pila del decodificador.
- Luego se convierte en una palabra de salida.

Which word in our vocabulary
is associated with this index?

Get the index of the cell
with the highest value
(**argmax**)

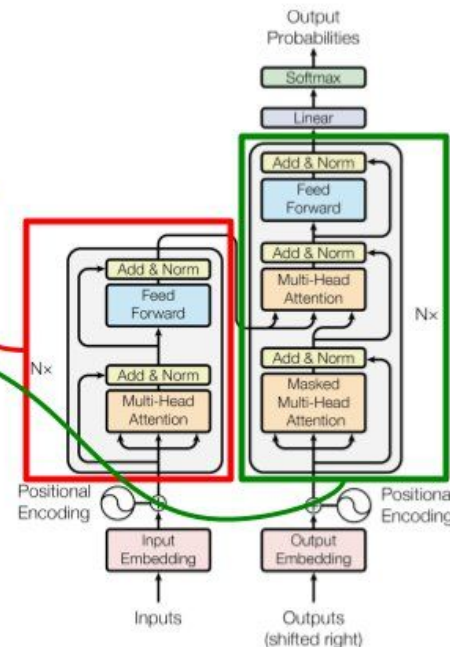


Implementación Transformers Pytorch

Implementación Transformer

```
class Transformer(nn.Module):
    def __init__(
        self, d_model=512, num_heads=8, num_encoders=6, num_decoders=6
    ):
        super().__init__()
        self.encoder = Encoder(d_model, num_heads, num_encoders)
        self.decoder = Decoder(d_model, num_heads, num_decoders)

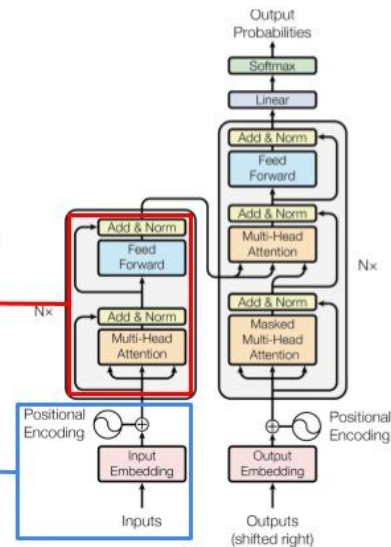
    def forward(self, src, tgt, src_mask, tgt_mask):
        enc_out = self.encoder(src, src_mask)
        dec_out = self.decoder(tgt, enc_out, src_mask, tgt_mask)
        return dec_out
```



Implementación Transformer

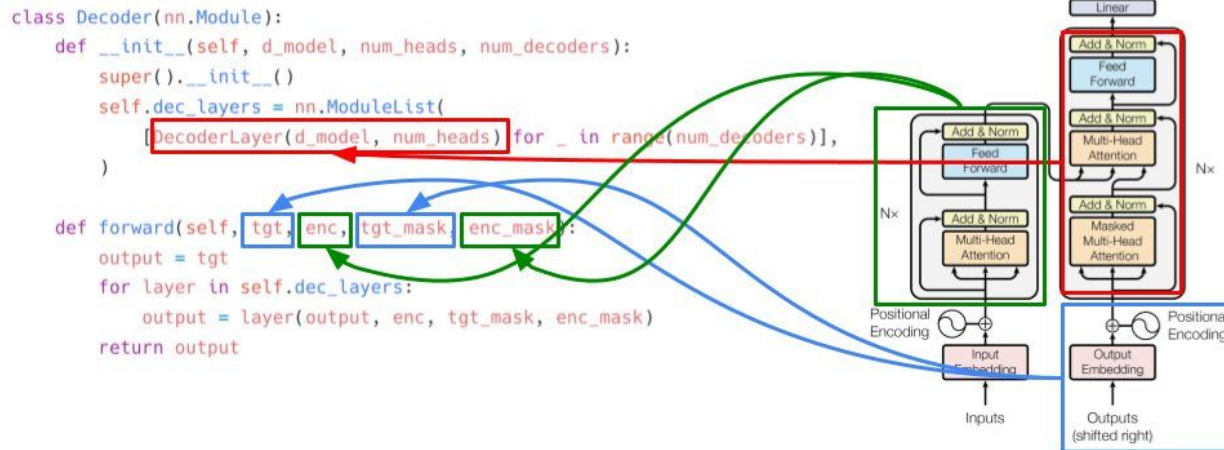
- El **encoder** se compone de N capas de encoders.
- Implementemos esto también como una caja negra.
- La salida de un **encoder** pasa como entrada al siguiente **encoder** y así sucesivamente.
- La máscara de origen sigue siendo la misma hasta el final.

```
class Encoder(nn.Module):  
    def __init__(self, d_model, num_heads, num_encoders):  
        super().__init__()  
        self.enc_layers = nn.ModuleList(  
            [EncoderLayer(d_model, num_heads) for _ in range(num_encoders)],  
        )  
  
    def forward(self, src, src_mask):  
        output = src  
        for layer in self.enc_layers:  
            output = layer(output, src_mask)  
        return output
```



Implementación Transformer

- De forma similar, el **decoder** está compuesto por capas de decoders.
- El **decoder** toma de entrada la última capa del **encoder** y los target embeddings y la target mask.
- `enc_mask` es lo mismo que `src_mask`



Implementación Transformer

- La capa del **encoder**.
- Consiste en multi-headed attention, una FFN y dos capas de normalización.
- En la función **forward** se observa cómo funcionan las conexiones de salto.
- Es solo agregar entradas originales a las salidas.

```
class EncoderLayer(nn.Module):  
    def __init__(self, d_model, num_heads, d_ff=2048, dropout=0.3):  
        super().__init__()
```

```
        # attention  
        self.attn = MultiHeadedAttention(d_model, num_heads, dropout=dropout)
```

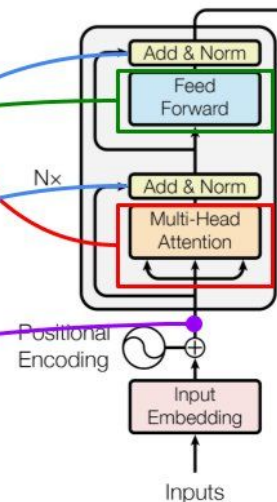
```
        # ffn
```

```
        self.ffn = nn.Sequential(  
            nn.Linear(d_model, d_ff),  
            nn.ReLU(inplace=True),  
            nn.Dropout(dropout),  
            nn.Linear(d_ff, d_model),  
            nn.Dropout(dropout),  
        )
```

```
        # layer norm
```

```
        self.attn_norm = nn.LayerNorm(d_model)  
        self.ffn_norm = nn.LayerNorm(d_model)
```

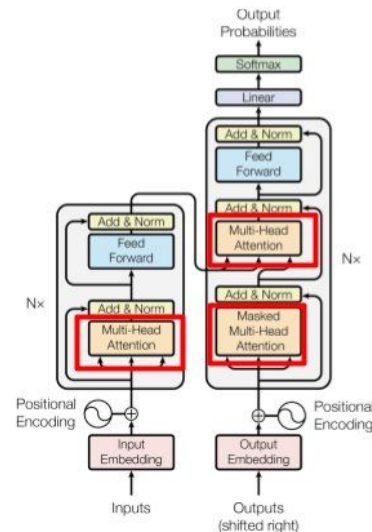
```
    def forward(self, src, src_mask):  
        x = src  
        x = x + self.attn(q=x, k=x, v=x, mask=src_mask)  
        x = self.attn_norm(x)  
        x = x + self.ffn(x)  
        x = self.ffn_norm(x)  
        return x
```



Implementación Transformer

- **Multi-head attention.**
- Lo vemos 3 veces en la arquitectura.
- **Multi-head attention** no es más que muchas capas diferentes de **self-attention**.
- Las salidas de estas **self-attention** se concatenan para formar una salida con la misma forma que la entrada.

```
class MultiHeadedAttention(nn.Module):  
    def __init__(self, d_model, num_heads, dropout):  
        super().__init__()  
        self.d_model = d_model  
        self.num_heads = num_heads  
        self.dropout = dropout  
        self.attn_output_size = self.d_model // self.num_heads  
        self.attentions = nn.ModuleList(  
            [  
                SelfAttention(d_model, self.attn_output_size)  
                for _ in range(self.num_heads)  
            ],  
        )  
        self.output = nn.Linear(self.d_model, self.d_model)  
  
    def forward(self, q, k, v, mask):  
        x = torch.cat(  
            [  
                layer(q, k, v, mask) for layer in self.attentions  
            ], dim=-1  
        )  
        x = self.output(x)  
        return x
```

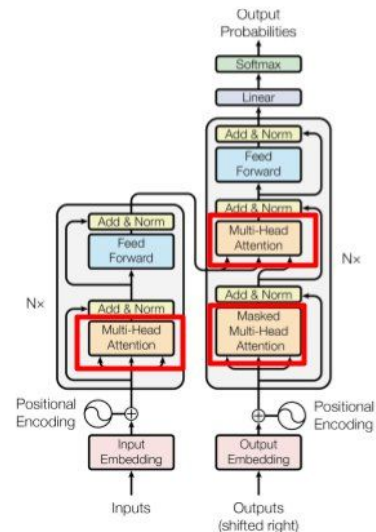


Implementación Transformer

- Si el número de cabezas es 8 y d_model (embedding size) es 512, cada self-attention producirá una salida de tamaño 64.
- Estos se concatenan para dar una salida final de tamaño $64 \times 8 = 512$.
- Esta salida se pasa a través de una capa densa.

```
class MultiHeadedAttention(nn.Module):
    def __init__(self, d_model, num_heads, dropout):
        super().__init__()
        self.d_model = d_model
        self.num_heads = num_heads
        self.dropout = dropout
        self.attn_output_size = self.d_model // self.num_heads
        self.attentions = nn.ModuleList(
            [
                SelfAttention(d_model, self.attn_output_size)
                for _ in range(self.num_heads)
            ]
        )
        self.output = nn.Linear(self.d_model, self.d_model)

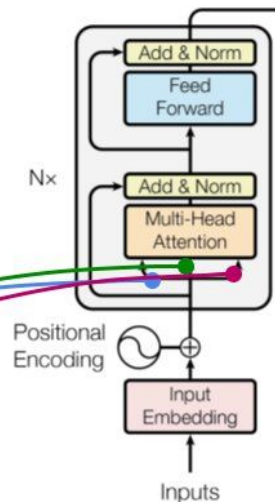
    def forward(self, q, k, v, mask):
        x = torch.cat(
            [
                layer(q, k, v, mask) for layer in self.attentions
            ], dim=-1
        )
        x = self.output(x)
        return x
```



Implementación Transformer

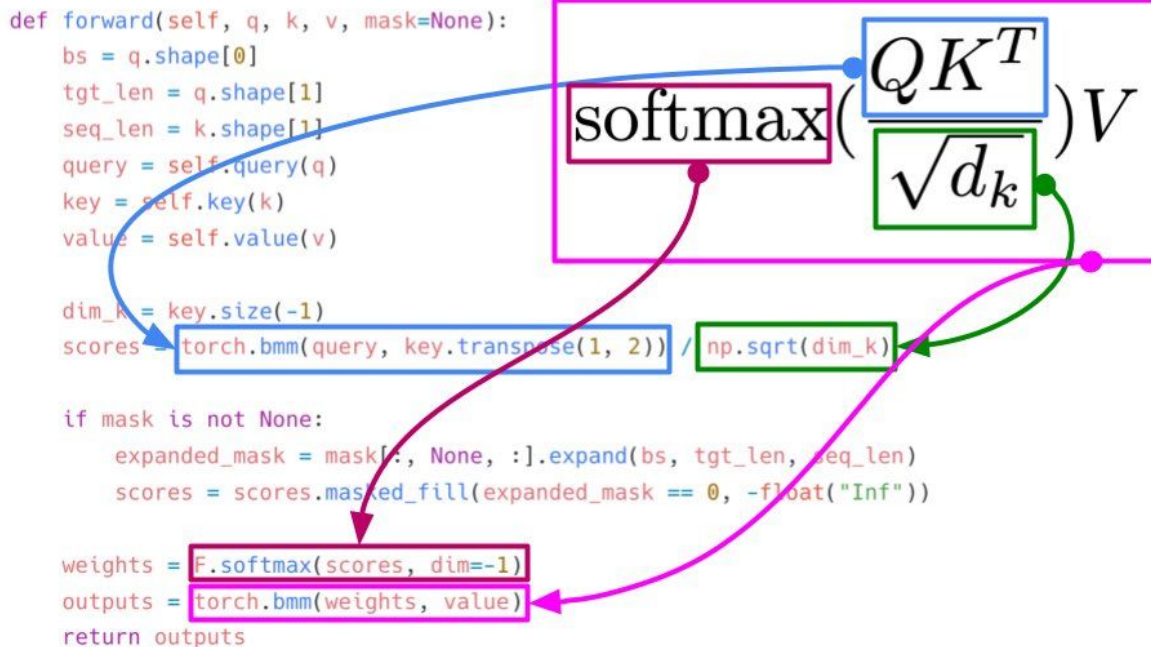
- La self-attention en palabras simples es atención en la misma secuencia.
- “Me gusta definirlo como una capa que te dice qué token ama a otro token en la misma secuencia”.
- Para la autoatención, la entrada se pasa a través de 3 capas lineales: query, key y value.

```
class SelfAttention(nn.Module):  
    def __init__(self, d_model, output_size, dropout=0.3):  
        super().__init__()  
        self.query = nn.Linear(d_model, output_size)  
        self.key = nn.Linear(d_model, output_size)  
        self.value = nn.Linear(d_model, output_size)  
        self.dropout = nn.Dropout(dropout)  
  
    def forward(self, q, k, v, mask=None):  
        bs = q.shape[0]  
        tgt_len = q.shape[1]  
        seq_len = k.shape[1]  
        query = self.query(q)  
        key = self.key(k)  
        value = self.value(v)  
  
        dim_k = key.size(-1)  
        scores = torch.bmm(query, key.transpose(1, 2)) / np.sqrt(dim_k)
```



Implementación Transformer

- En la función **forward**, se aplica la fórmula para self-attention: $\text{softmax}(Q \cdot K^T / \dim(k))V$.
- **torch.bmm** realiza la multiplicación matricial por lotes.
- $\dim(k)$ es la raíz cuadrada de k .
- Tener en cuenta: q, k, v (entradas) son las mismas en el caso de self-attention.



Implementación Transformer

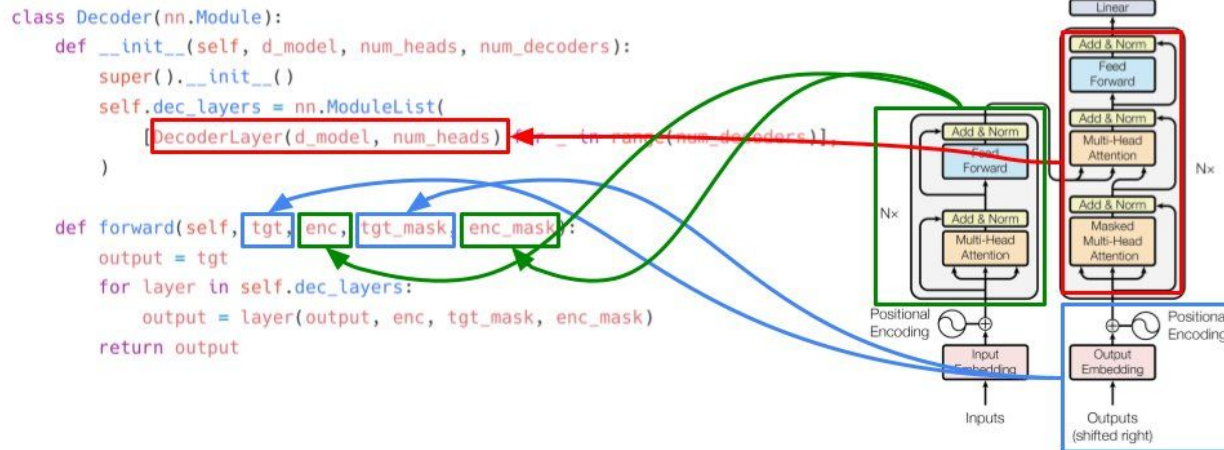
- La máscara solo dice dónde no mirar (por ejemplo, tokens de relleno)

```
if mask is not None:
    expanded_mask = mask[:, None, :].expand(bs, tgt_len, seq_len)
    scores = scores.masked_fill(expanded_mask == 0, -float("Inf"))
```

- mask tells the model where not to look
- padding = 0, tokens = 1
- mask: batch size x sequence length
- expanded mask: batch size x sequence length x sequence length
- don't use mask = 0 for scores

Implementación Transformer

- La implementación del **decoder** es similar a la del **encoder** excepto por el hecho de que cada **decoder** también toma la salida del **encoder** final como entrada.



Implementación Transformer

- La capa del **decoder** consta de dos tipos diferentes de atención.
- La versión enmascarada tiene una máscara adicional además de la máscara de relleno.
- La multi-head attention normal toma la key y el value de la salida final del encoder.
- La key y el value aquí son los mismos.

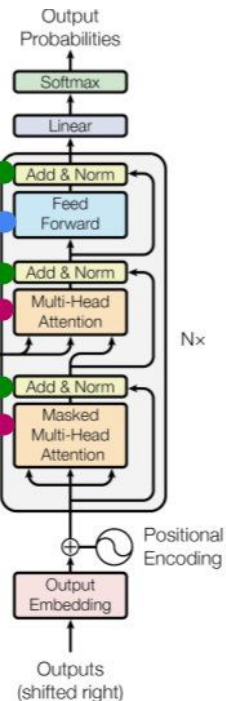
```
class DecoderLayer(nn.Module):  
    def __init__(self, d_model, num_heads, d_ff=2048, dropout=0.3):  
        super().__init__()
```

```
        # masked attn  
        self.masked_attn = MultiHeadedAttention(  
            d_model, num_heads, dropout=dropout  
        )
```

```
        # attn  
        self.attn = MultiHeadedAttention(  
            d_model, num_heads, dropout=dropout  
        )
```

```
        # ffn  
        self.ffn = nn.Sequential(  
            nn.Linear(d_model, d_ff),  
            nn.ReLU(inplace=True),  
            nn.Dropout(dropout),  
            nn.Linear(d_ff, d_model),  
            nn.Dropout(dropout),  
        )
```

```
        # layer norm  
        self.masked_attn_norm = nn.LayerNorm(d_model)  
        self.attn_norm = nn.LayerNorm(d_model)  
        self.ffn_norm = nn.LayerNorm(d_model)
```



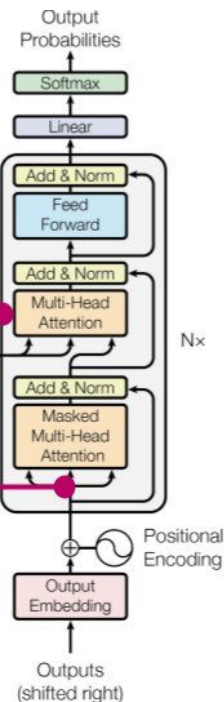
Implementación Transformer

- La query proviene de la salida de la masked multi-head attention enmascaradas (después de la capa de normalización).

```
# ffn
self.ffn = nn.Sequential(
    nn.Linear(d_model, d_ff),
    nn.ReLU(inplace=True),
    nn.Dropout(dropout),
    nn.Linear(d_ff, d_model),
    nn.Dropout(dropout),
)

# layer norm
self.masked_attn_norm = nn.LayerNorm(d_model)
self.attn_norm = nn.LayerNorm(d_model)
self.ffn_norm = nn.LayerNorm(d_model)

def forward(self, tgt, enc, tgt_mask, enc_mask):
    x = tgt
    x = x + self.masked_attn(q=x, k=x, v=x, mask=tgt_mask)
    x = self.masked_attn_norm(x)
    x = x + self.attn(q=x, k=enc, v=enc, mask=enc_mask)
    x = self.attn_norm(x)
    x = x + self.ffn(x)
    x = self.ffn_norm(x)
    return x
```



Implementación Transformer

- En la máscara especial para targets, también conocida como subsequent mask.
- La subsequent mask simplemente le dice al **decoder** que no mire tokens en el futuro.
- Esto se usa además de la máscara de relleno y se usa solo para la parte de entrenamiento.

```
if mask is not None:
    expanded_mask = mask[:, None, :].expand(bs, tgt_len, seq_len)
    subsequent_mask = 1 - torch.triu(
        torch.ones((tgt_len, tgt_len), device=mask.device, dtype=torch.uint8), diagonal=1
    )
    subsequent_mask = subsequent_mask[None, :, :].expand(bs, tgt_len, tgt_len)
    scores = scores.masked_fill(expanded_mask == 0, -float("Inf"))
    scores = scores.masked_fill(subsequent_mask == 0, -float("Inf"))
```

	T-1	T-2	T-3	T-4	T-5	T-6	T-7	T-8
T-1	1	0	0	0	0	0	0	0
T-2	1	1	0	0	0	0	0	0
T-3	1	1	1	0	0	0	0	0
T-4	1	1	1	1	0	0	0	0
T-5	1	1	1	1	1	0	0	0
T-6	1	1	1	1	1	1	0	0
T-7	1	1	1	1	1	1	1	0
T-8	1	1	1	1	1	1	1	1

Implementación Transformer

- Hasta este punto se tienen todos los componentes básicos, excepto el positional encoding.
- El positional encoding le da al modelo una idea sobre dónde se ubican los tokens entre sí.
- Para implementar la codificación posicional, ¡simplemente podemos usar una capa de incrustación!

Implementación Transformer

- Así es como se verán las entradas y salidas.
- Aquí, batch size = 32, longitud de secuencia de entrada = 128, longitud de secuencia de salida = 64.
- Agregamos un FC + softmax a la salida del **decoder**.
- Esto da una predicción simbólica para cada posición (un problema de clasificación)

