Transformers

Clase 20

Dra. Wendy Aguilar

Modelos Generativos Profundos

UN ENFOQUE DESDE LA CREATIVIDAD COMPUTACIONAL

Atención

El core de los transformers

Intuición:

¿cómo los humanos prestamos atención al escribir?

Y si cambiáramos de contexto y la frase fuera:

"The pink elephant tried to squash the car but it was too ..."

¿Qué palabra esperamos al final?

Ahora el sujeto de *it* sería el **car**, y la palabra más natural podría ser **"fast"** o **"strong"**.

En cada caso, **solo algunas palabras del contexto** son determinantes para elegir la siguiente.

El mecanismo de atención (también llamado **cabeza de atención**) permite que un modelo, igual que nosotros,

"mire" las palabras relevantes e **"ignore" las que no aportan información**.

Cabeza de atención

¿Cómo decide a qué palabras debe prestar más atención?

Podemos imaginar el *attention head* como un **sistema de recuperación de información**:

1. Hacemos una **pregunta (query)**

¿Qué tanto las palabras que me anteceden (a *"too") me pueden ayudar a saber qué palabra viene después de mí*?"

Tenemos que poder hacer esa pregunta en términos de números.

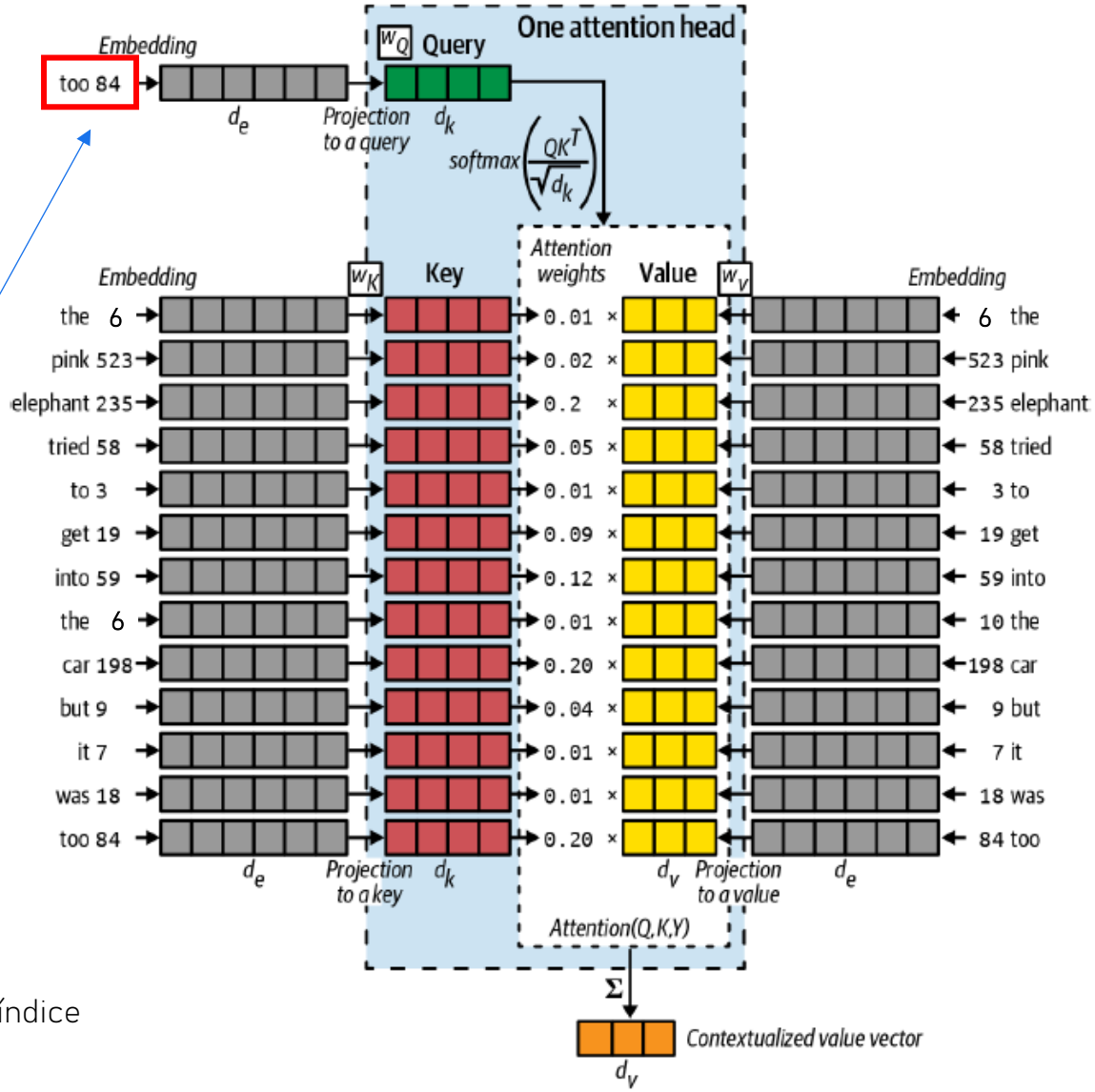
```
# Crea una capa de vectorización
vectorize_layer = layers.TextVectorization(
    standardize="lower",
    max_tokens=VOCAB_SIZE,
    output_mode="int",
    output_sequence_length=MAX_LEN + 1,
)

# Adapta la capa al conjunto de entrenamiento
vectorize_layer.adapt(text_ds)
vocab = vectorize_layer.get_vocabulary()
```

```
0:
1: [UNK]
2: :
3: ,
4: .
5: and
6: the
7: wine
8: a
9: of
```

Ese número **no tiene significado semántico por sí mismo** — es solo un identificador/índice dentro del vocabulario.

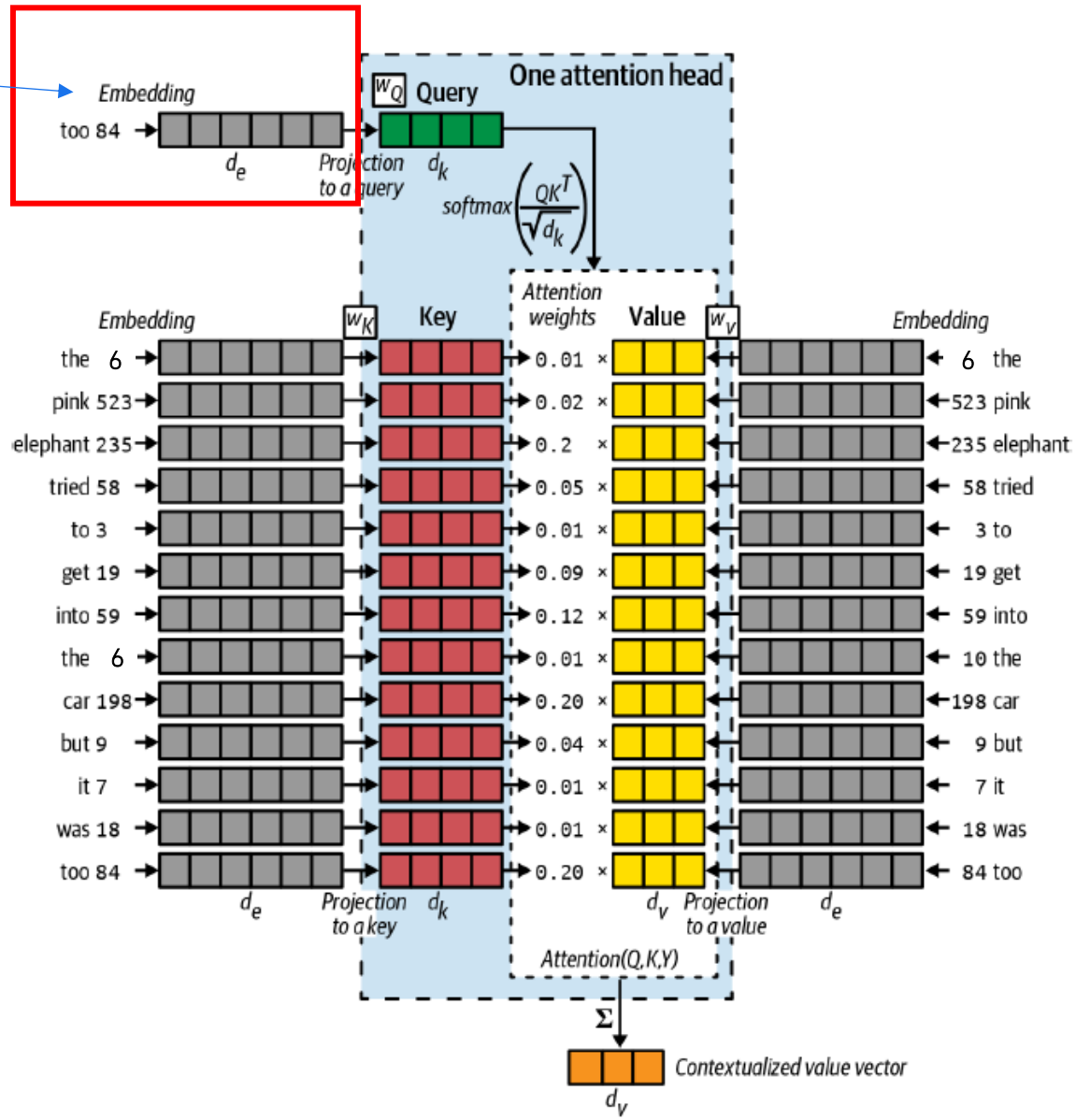
The pink elephant tried to get into the car but it was too ..."



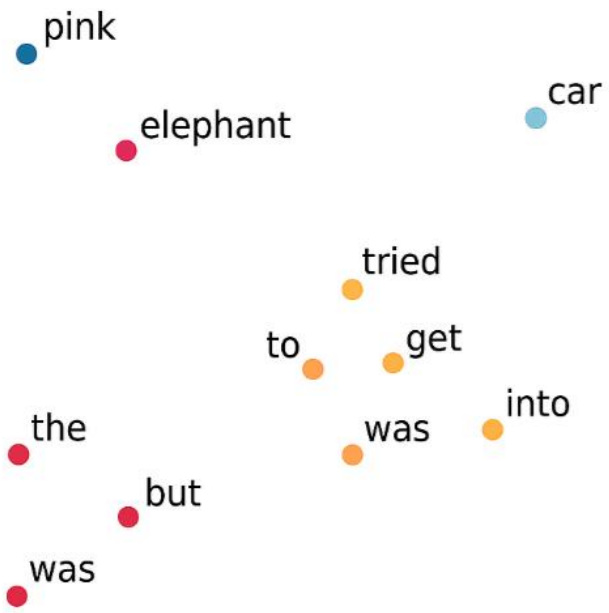
- La capa `Embedding` aprende una **matriz de pesos** $W_E \in \mathbb{R}^{|V| \times d_e}$, donde:
  - Cada **fila** corresponde a una palabra del vocabulario.
  - Cada **columna** representa una dimensión semántica aprendida (por ejemplo, "color", "tamaño", "sentimiento"...).

Palabra (vocabulario)	dim_1: color / apariencia	dim_2: tamaño / forma	dim_3: movimiento / acción	dim_4: objetos / entorno	dim_5: emoción / contexto	... (dim_n)
the	0.02	0.01	0.00	0.03	0.00	...
pink	0.88	0.10	0.05	0.12	0.02	...
elephant	0.32	0.91	0.18	0.07	0.10	...
tried	0.04	0.08	0.92	0.06	0.12	...
to	0.01	0.00	0.05	0.02	0.00	...
get	0.03	0.09	0.88	0.11	0.15	...
into	0.02	0.03	0.61	0.75	0.10	...
the	0.02	0.01	0.00	0.03	0.00	...
car	0.10	0.12	0.22	0.94	0.05	...
but	0.01	0.02	0.03	0.01	0.84	...
it	0.00	0.05	0.10	0.03	0.67	...
was	0.01	0.02	0.07	0.02	0.80	...
too	0.02	0.08	0.09	0.04	0.93	...

The pink elephant tried to get into the car but it was too ..."

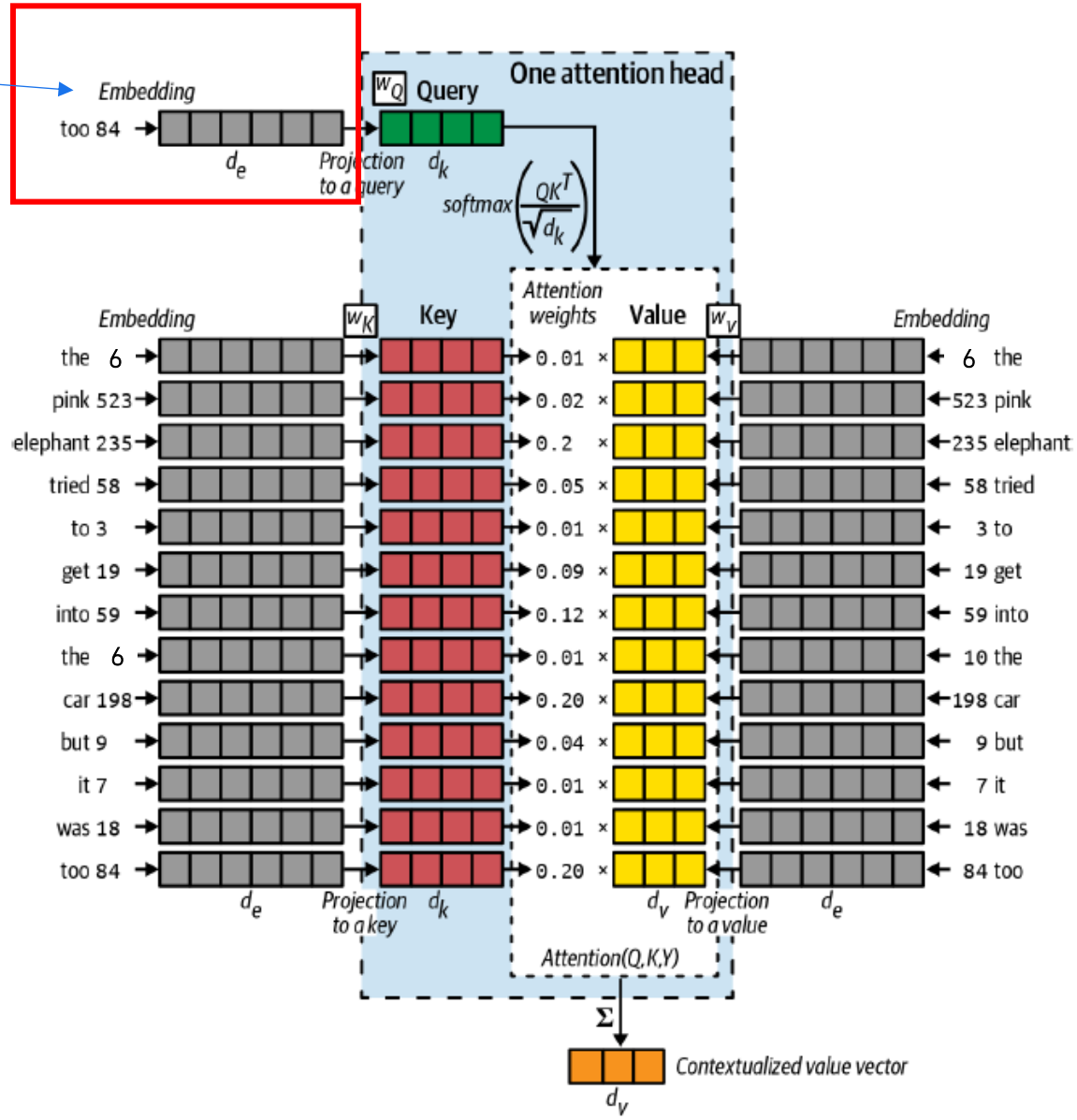


- Palabras que aparecen en contextos similares terminan con vectores cercanos.



- Cuando alimentamos el índice *84*, la red **busca la fila 84** de esa matriz y obtiene un vector e_{too} de longitud d_e.
- Así, *"too" → [0.12, –0.31, 0.45, …]* se convierte en una **representación numérica densa y significativa**.

The pink elephant tried to get into the car but it was too …"



De *embedding* a *Query*:

- Una vez que la palabra *too* tiene su vector de embedding $\mathbf{e}_{too}$, el modelo necesita convertirlo en una forma que le permita **hacer una pregunta al contexto**.

- Esa conversión se realiza mediante una **proyección lineal** usando una matriz de pesos W_Q que se aprende durante el entrenamiento.

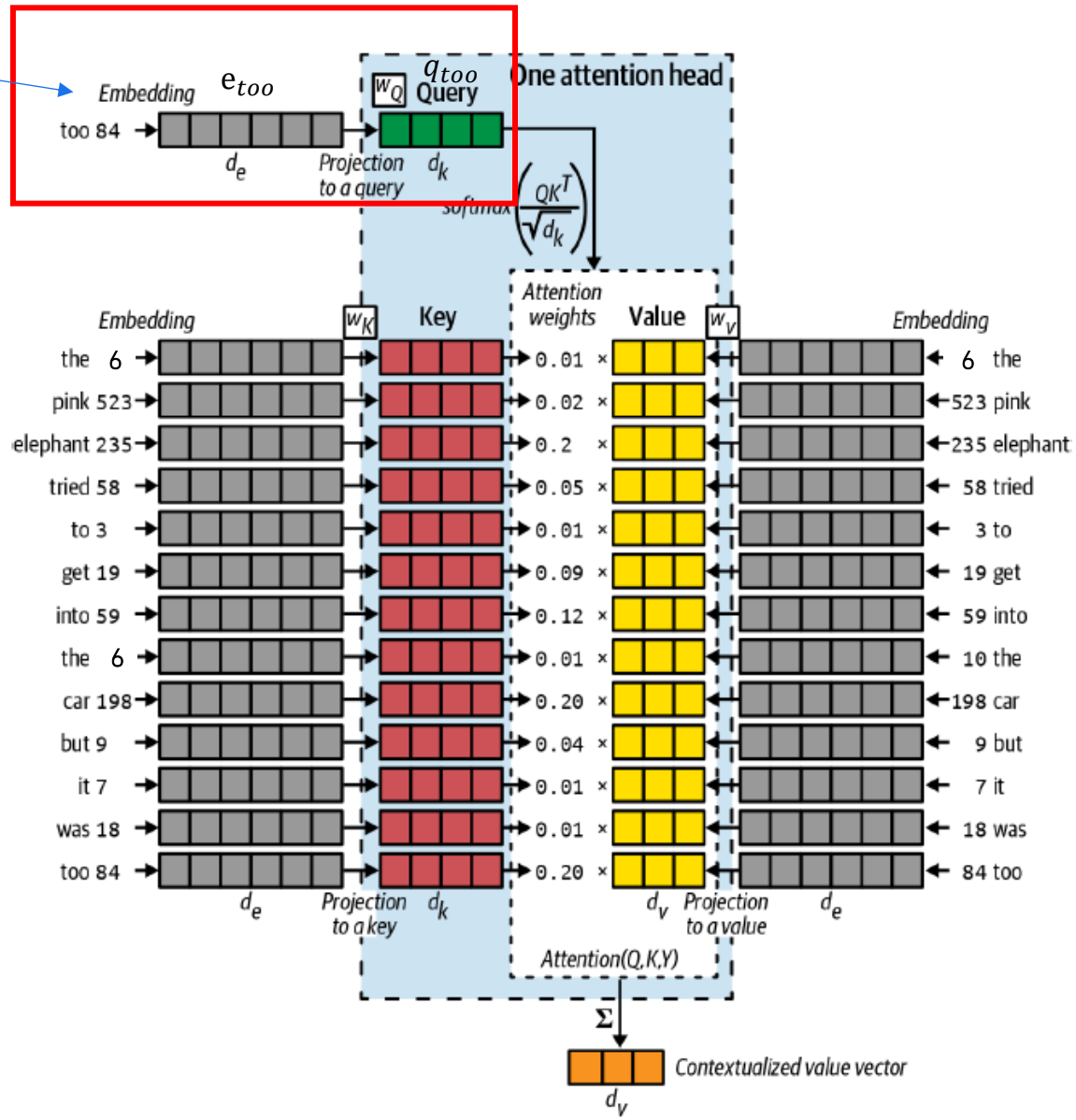
$$q_{too} = e_{too} W_Q$$

W_Q tiene tamaño $d_e \times d_k$:

  - d_e = dimensión del embedding (por ejemplo, 128)
  - d_k = dimensión del espacio de consultas (*query space*), generalmente menor.

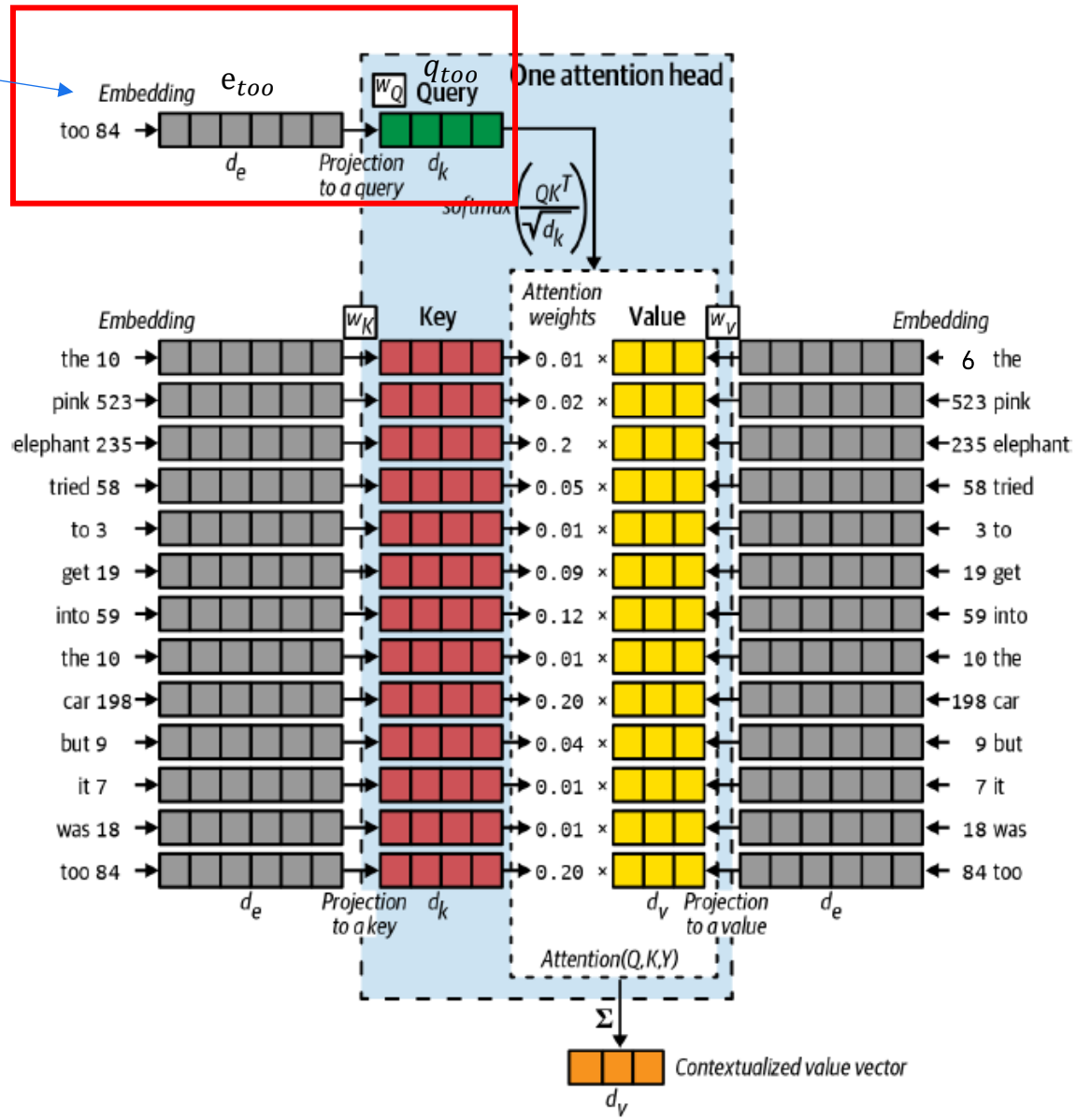
- Este paso no cambia el significado de la palabra, sino que **adapta su representación** al espacio donde se comparará con las demás palabras (*keys*).

The pink elephant tried to get into the car but it was too ..."



De *embedding* a *Query*:

The pink elephant tried to get into the car but it was too …"



Intuitivamente:

- El vector gris $\mathbf{e}_{too}$ representa *"yo soy la palabra too"*.
- El vector verde $\mathbf{q}_{too}$ representa *"quiero saber quién puede ayudarme a predecir la palabra que sigue"*.
- Así, *query* captura la intención de búsqueda de información que tendrá el *attention head*.

La cabeza de atención aprende su propio idioma de preguntas y respuestas.

The pink elephant tried to get into the car but it was too ..."

2. Creamos las *keys*

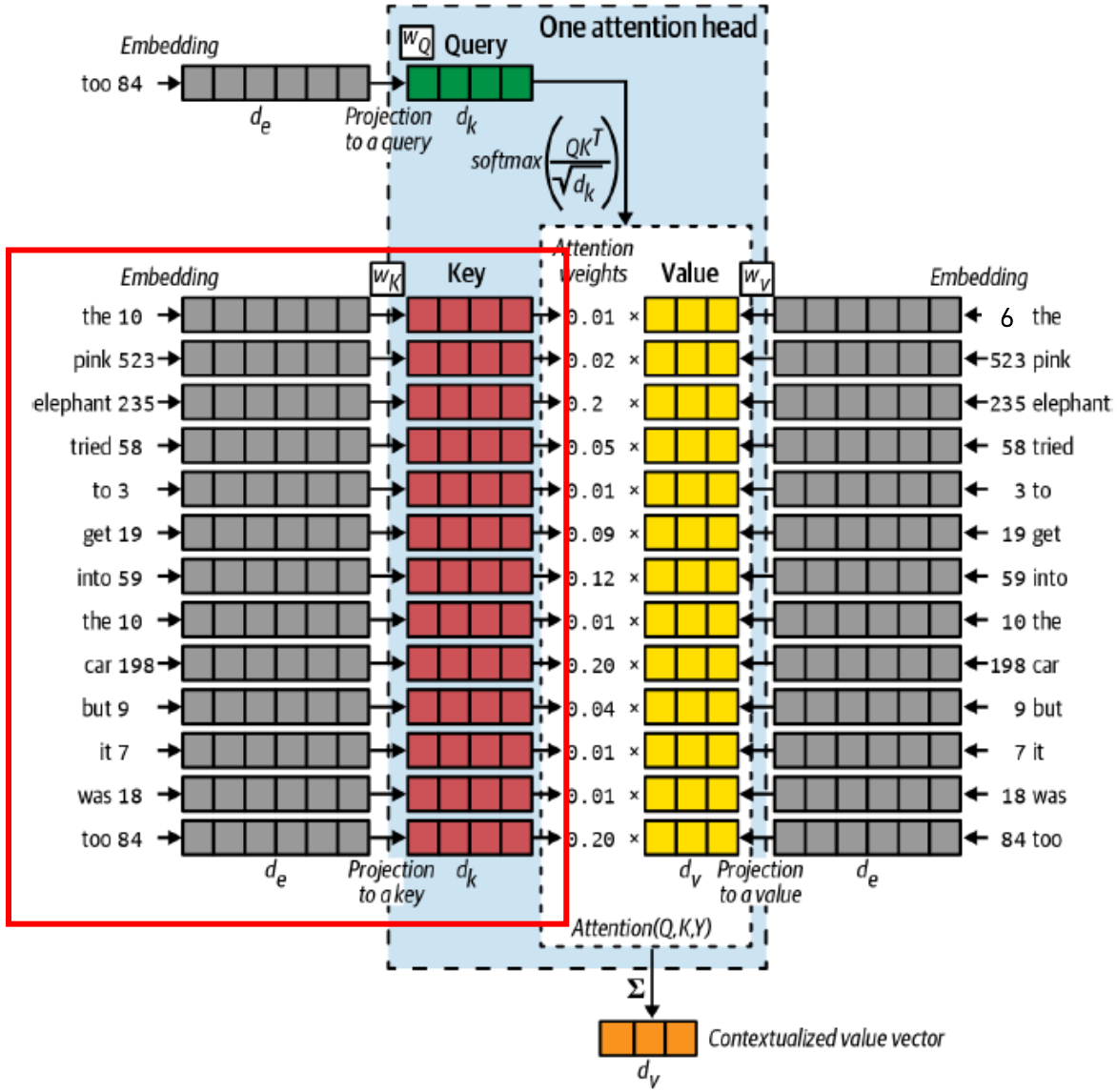
- Cada palabra del contexto genera su propio vector **key (clave)**.
- Las *keys* pueden pensarse como **descripciones del tipo de pregunta que esa palabra puede responder**.

Ejemplo:
elephant puede responder preguntas sobre tamaño.
car sobre objetos o movimiento.
was o *but* sobre relaciones gramaticales.

- Para obtener cada *key*, se toma el **embedding** de la palabra y se proyecta linealmente con su propia matriz de pesos W_K:

$$k_i = e_i W_K$$

- $W_K \in \mathbb{R}^{d_t \times d_k}$ también se aprende durante el entrenamiento, igual que W_Q.



The pink elephant tried to get into the car but it was too …"

3. Generamos los *values*

- Cada palabra del contexto se proyecta a un nuevo vector llamado ***value***.

- Los ***values*** representan **la información concreta** que esa palabra pondrá a disposición del *attention head*.

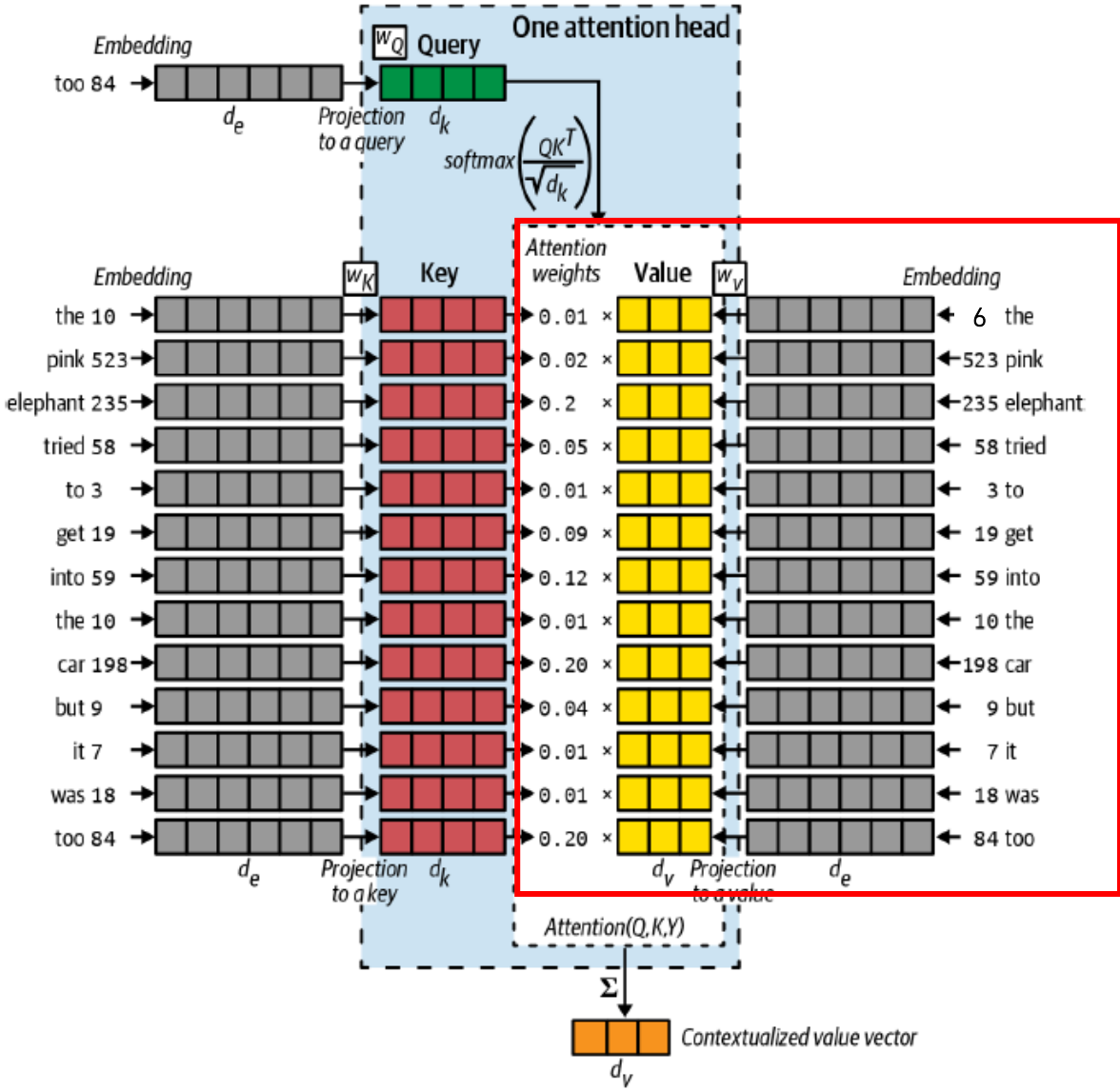
$$v_i = e_i W_V$$

- $W_V \in \mathbb{R}^{d_e \times d_v}$ también se aprende durante el entrenamiento.

Si el *query* es la pregunta
"¿qué palabra me ayuda a predecir lo que sigue?"

y las *keys* son las descripciones
"yo sé de tamaño, yo sé de movimiento",

los *values* son **las respuestas**
"grande", "rápido", "pesado", etc.



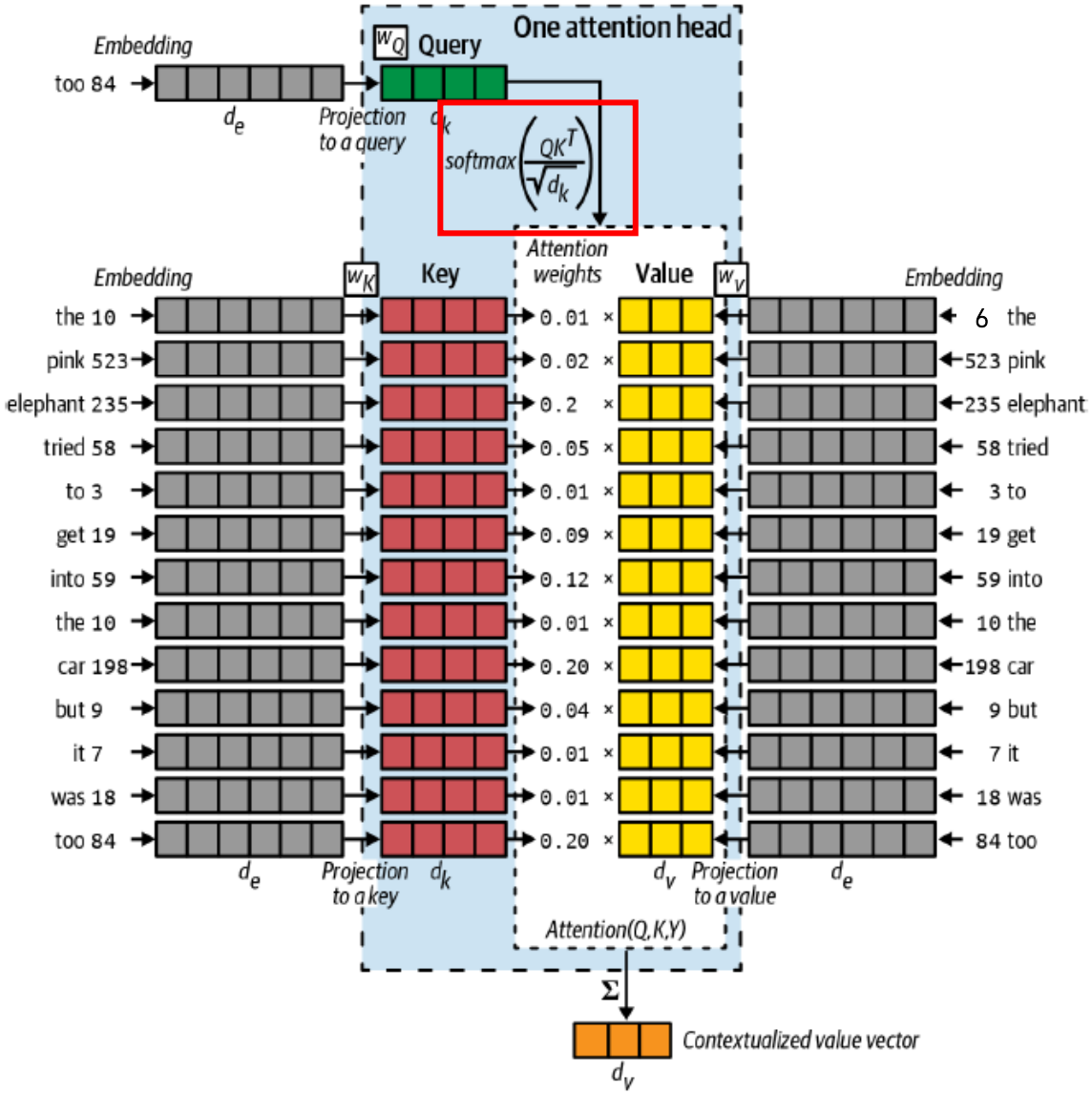
The pink elephant tried to get into the car but it was too ..."

4. Combinamos *queries*, *keys* y *values*

- 4.1 Comparar *queries* con *keys*
  - Se calcula el **producto punto** entre todas las *queries* y *keys*:

$$QK^T$$

  - Cada valor $Q_i \cdot K_j$ mide **qué tan bien la palabra** *i* (la pregunta) **resuena con la palabra** *j* (la descripción de la jésima key).
  - Es decir, mide **la relevancia contextual** entre ambas palabras.



Nota:

Durante el entrenamiento

- El *Transformer* **no procesa palabra por palabra** como una RNN, sino **toda la secuencia en paralelo**.
- Por eso, en la cabeza de atención:
  - Se calcula **una query por cada token** de entrada.
  - Se calculan **todas las keys y values** para la secuencia completa.
- Luego, **cada query** se compara (producto punto) con **todas las keys**.

The pink elephant tried to get into the car but it was too ..."

4. Combinamos *queries*, *keys* y *values*

- 4.2 Escalar y normalizar
  - Para evitar valores muy grandes, se divide entre $\sqrt{d_k}$

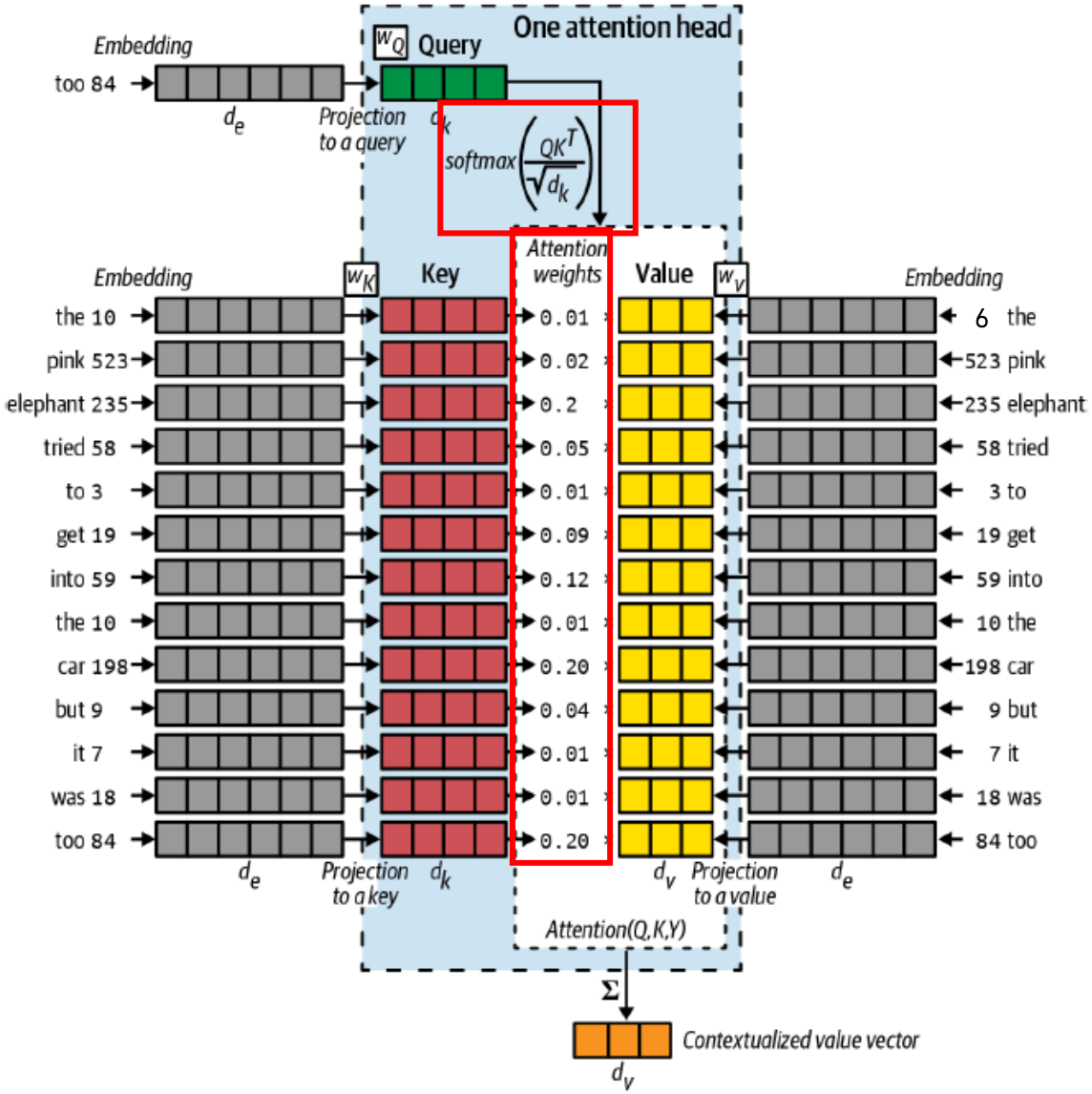
$$\frac{QK^T}{\sqrt{d_k}}$$

  - Luego se aplica **softmax**, convirtiendo esos puntajes en **pesos de atención** que suman 1:

$$\text{attention\_weights} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

  - Estos pesos indican **a qué palabras se les presta más atención**.

Por ejemplo, "*too*" prestará más atención a "*elephant*" y "*car*".



The pink elephant tried to get into the car but it was too ..."

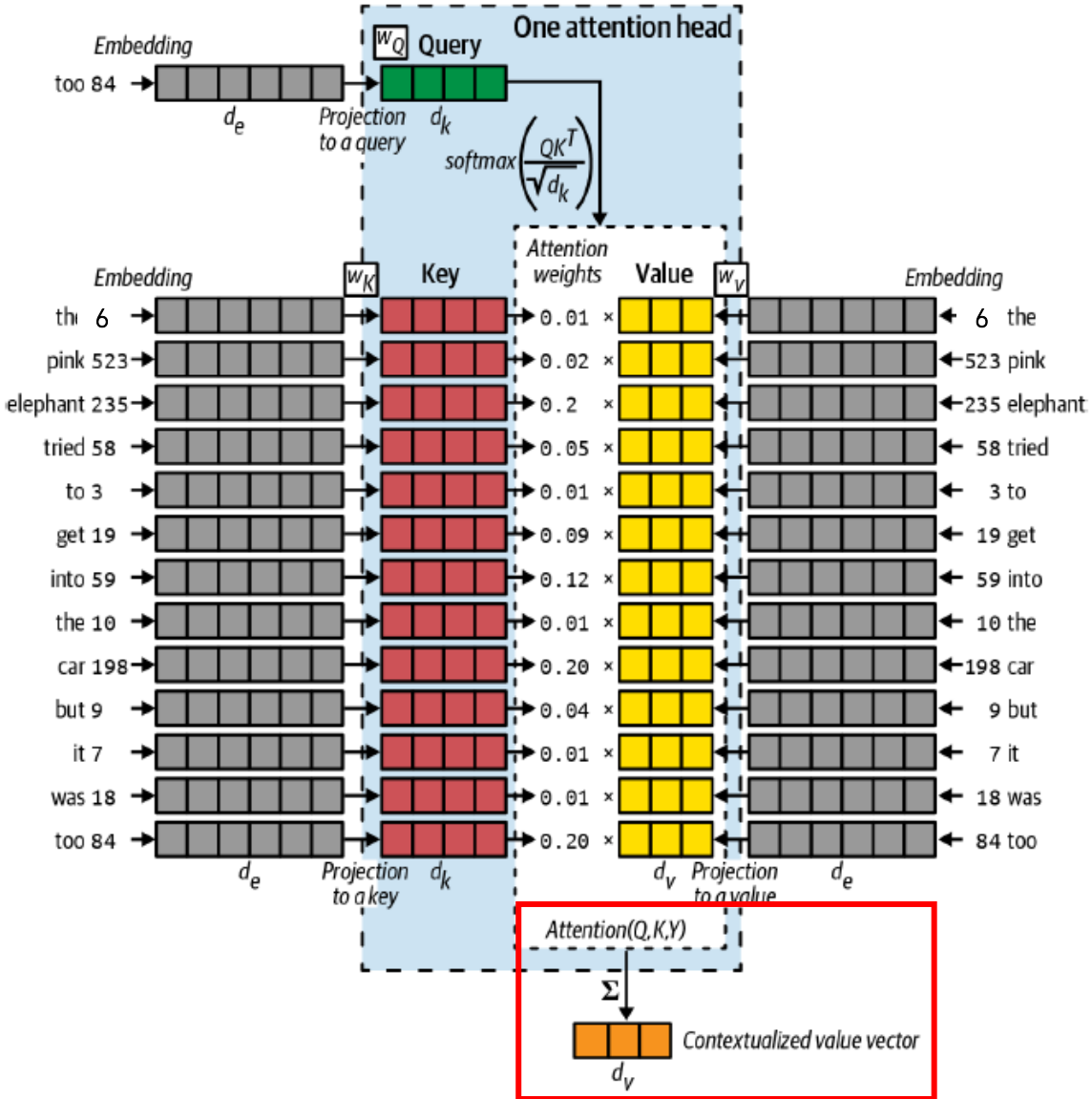
4. Combinamos *queries*, *keys* y *values*

- 4.3 Combinar los valores
  - Los pesos de atención se usan para hacer una **suma ponderada** de los *values*:

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

  - El resultado es el **vector contextualizado** $\mathbf{c} \in \mathbb{R}^{d_v}$:
    - combina la información de varias palabras,
    - enfatizando aquellas más relevantes para la actual.

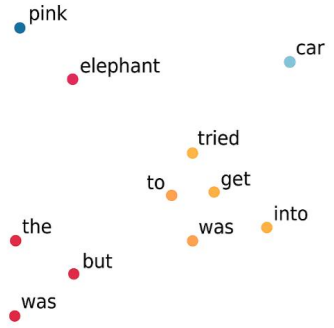
"Voy a construir mi comprensión actual combinando un poco de 'elephant' (tamaño), algo de 'car' (objeto), y un toque de 'was' (gramática)."



¿Qué se aprende durante el entrenamiento?

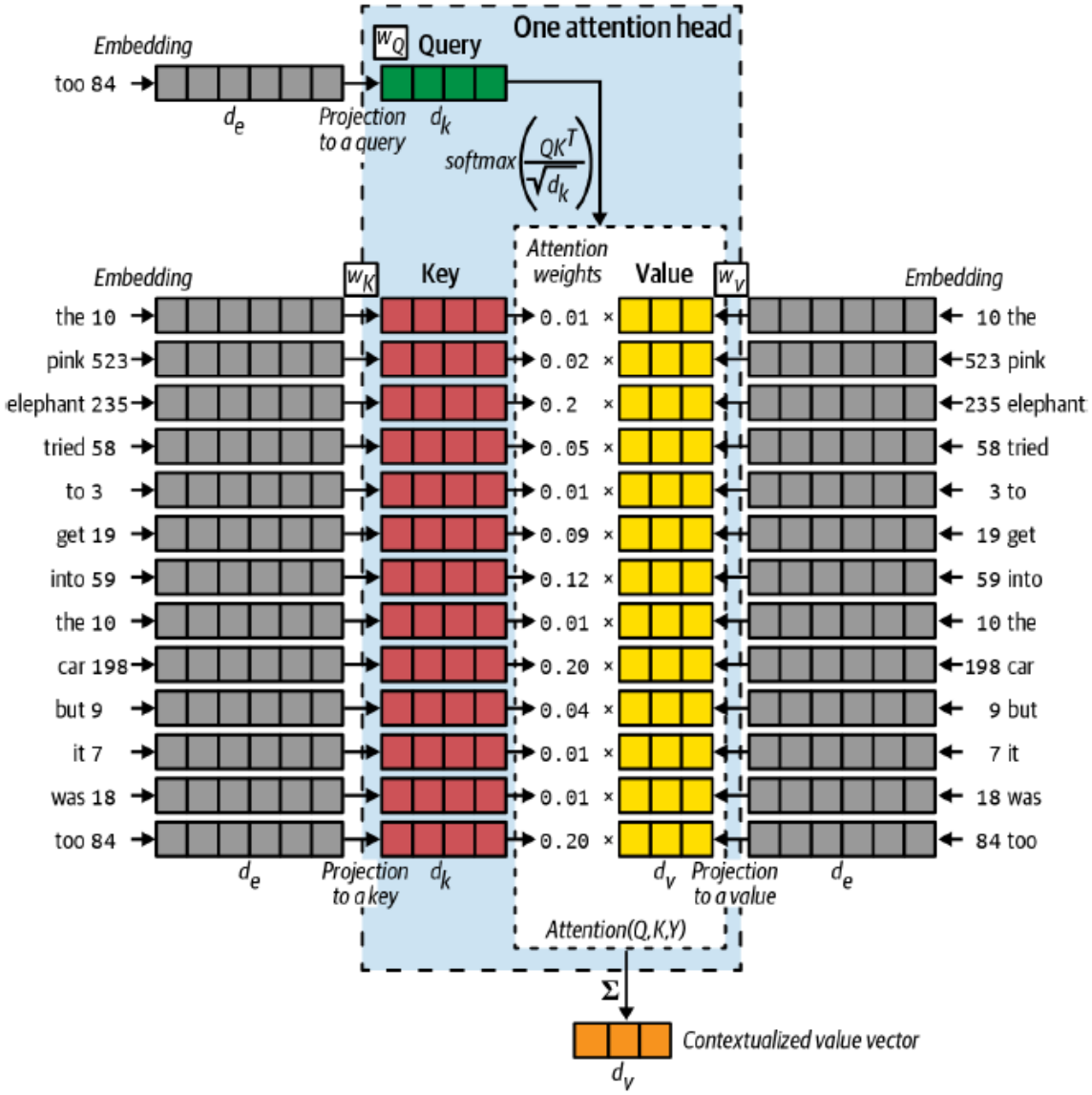
1. El embedding

- La **capa de embeddings** aprende a representar cada palabra en un **espacio semántico continuo**.
  - Al inicio, los vectores son aleatorios.
  - Durante el entrenamiento, se ajustan para que **palabras que aparecen en contextos similares** tengan **vectores cercanos**.



2. Las matrices de proyección W_Q, W_K, W_V

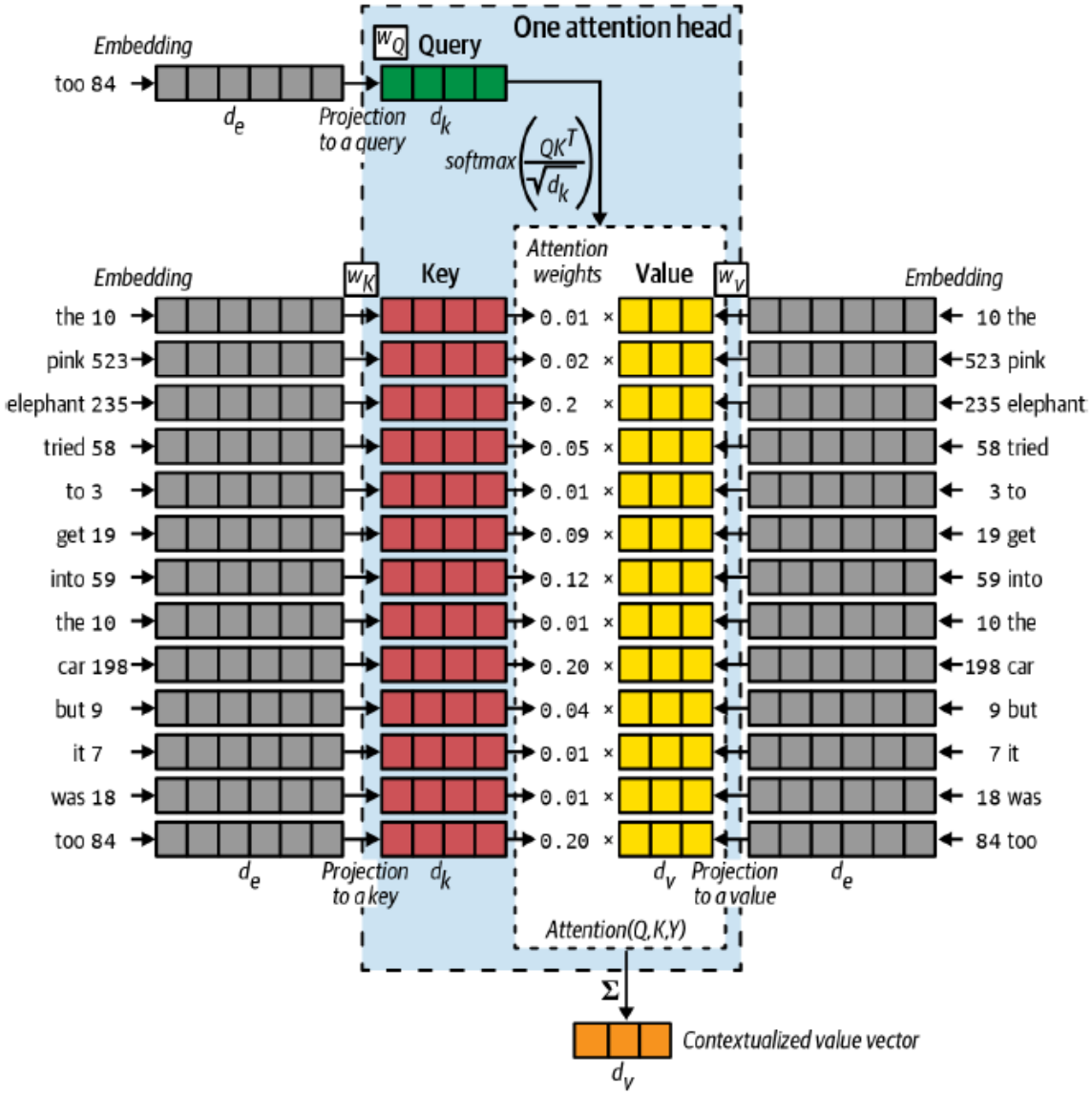
- Estas matrices son **parámetros entrenables** dentro del *attention head*.
  - W_Q aprende a proyectar los embeddings para **formular preguntas útiles** (*queries*).
  - W_K aprende a proyectarlos para **describir la información disponible** (*keys*).
  - W_V aprende a proyectarlos para **ofrecer la información más relevante** (*values*).
- Todas se inicializan aleatoriamente y se **ajustan con gradientes** según la pérdida (*loss*).



¿Cómo se aprende todo a la vez?

- El modelo calcula una **pérdida global** (qué tan bien predice la siguiente palabra).
- Esa pérdida se **propaga hacia atrás** a través de todas las capas:
  - Los gradientes llegan hasta las matrices $\boldsymbol{W_Q, W_K, W_V}$.
  - También alcanzan la capa de *embedding*, ajustando las representaciones de palabras.
- Así, el modelo aprende **simultáneamente**:
  - qué significan las palabras,
  - cómo hacer las preguntas correctas,
  - y cómo combinar la información más útil.

El Transformer aprende por sí mismo cómo representar las palabras y cómo atender a las más relevantes para comprender el contexto."

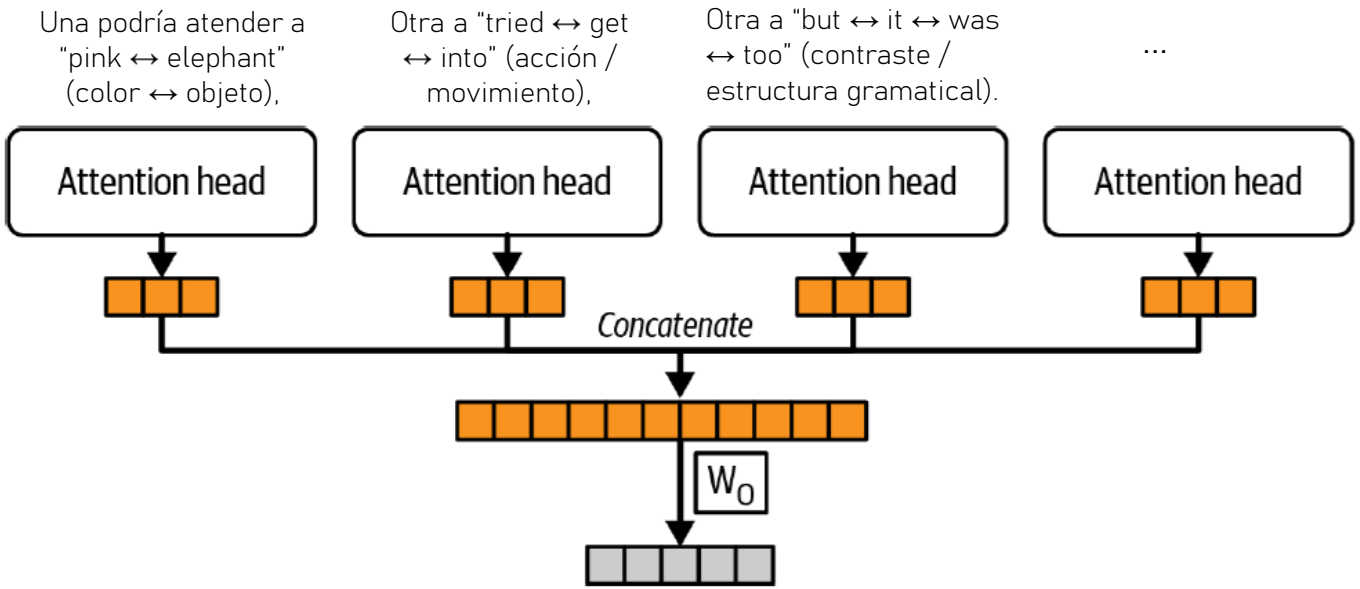


Multi-Head Attention

No hay razón para conformarse con una sola cabeza de atención.

1. ¿Por qué usar varias cabezas?

- Una sola cabeza de atención puede aprender **un tipo de relación** (por ejemplo, "color ↔ objeto").
- Pero el lenguaje y el significado son **multidimensionales**:
  - otra cabeza puede aprender relaciones **gramaticales**,
  - otra puede enfocarse en **acciones**,
  - otra en **causa-efecto** o **contraste**.
- Con varias cabezas, el modelo puede **mirar al contexto desde distintos ángulos a la vez**.



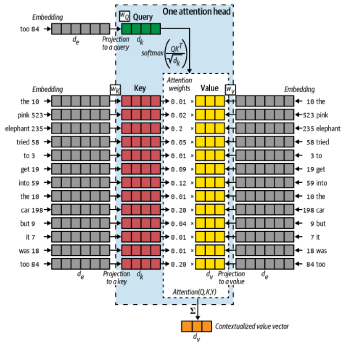
Cada cabeza aprende a mirar el texto de una manera diferente.
Al combinar sus perspectivas, el modelo capta relaciones complejas que una sola cabeza no podría ver.

Nota:

- La capa de embedding convierte cada palabra (token) en un **vector fijo** que captura su significado general, sin depender del contexto en el que aparece. Estos vectores **no cambian según el contexto**
- Cada cabeza de atención aprende **relaciones contextuales dinámicas** entre palabras dentro de una secuencia. Aquí, los pesos del *softmax* indican **qué palabras influyen más en la representación actual**.

Multi-Head Attention

No hay razón para conformarse con una sola cabeza de atención.

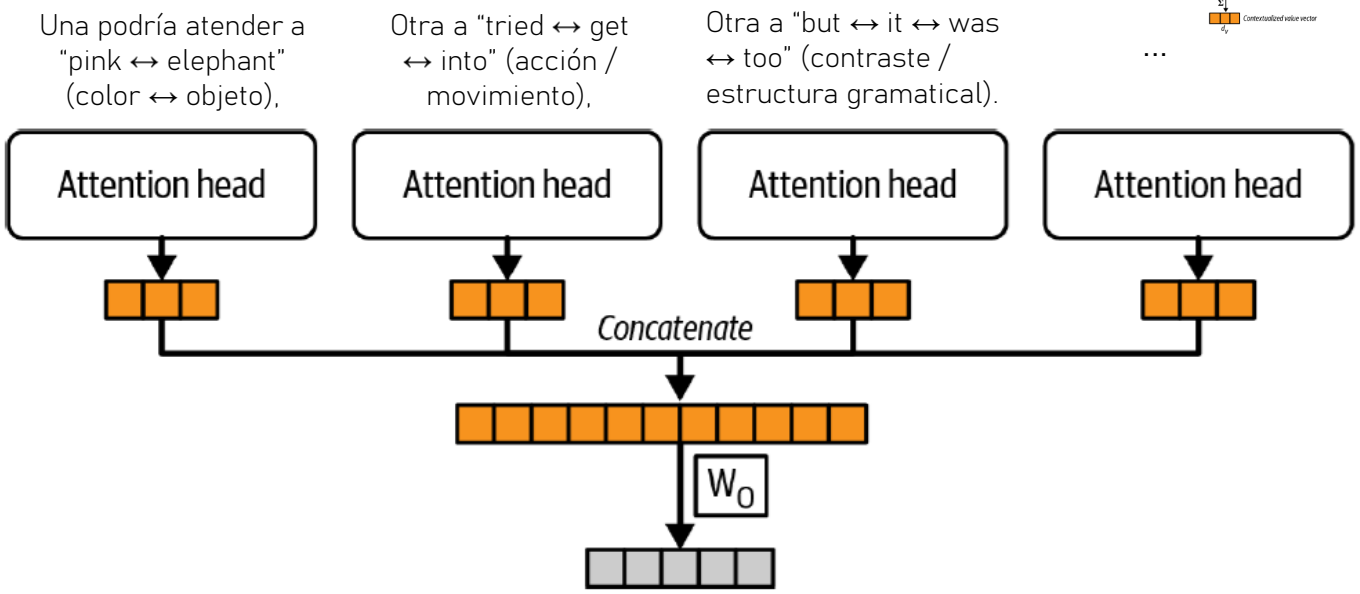


2. ¿Cómo funciona?

- Cada cabeza de atención tiene **sus propios pesos** W_Q, W_K, W_V, por lo que cada una aprende un patrón distinto de atención.
- Todas las cabezas procesan la misma secuencia en paralelo, generando varios **vectores contextualizados** (naranjas).
- Luego, estos vectores se **concatenan** en un solo gran vector.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, ..., \text{head}_h)W_O$$

- Finalmente, se aplica una nueva matriz W_O para **proyectar** el resultado combinado de nuevo a la dimensión original d_e.



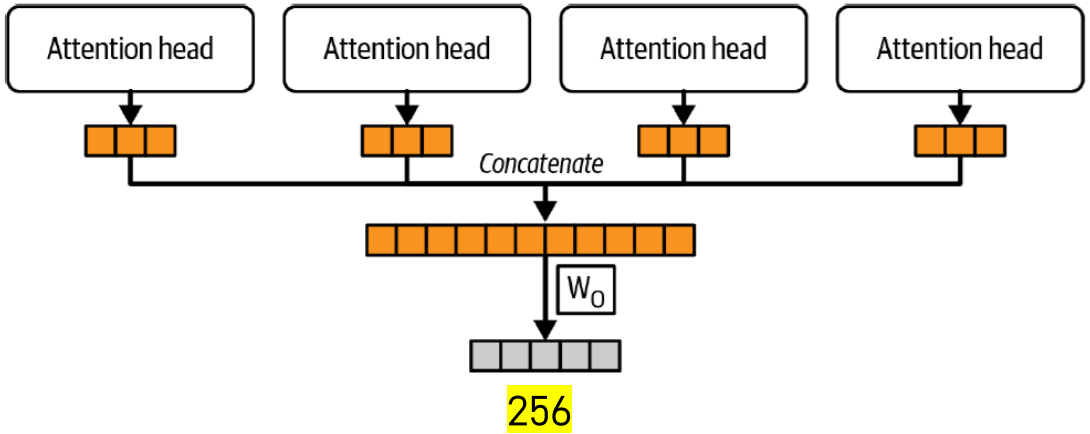
Multi-Head Attention

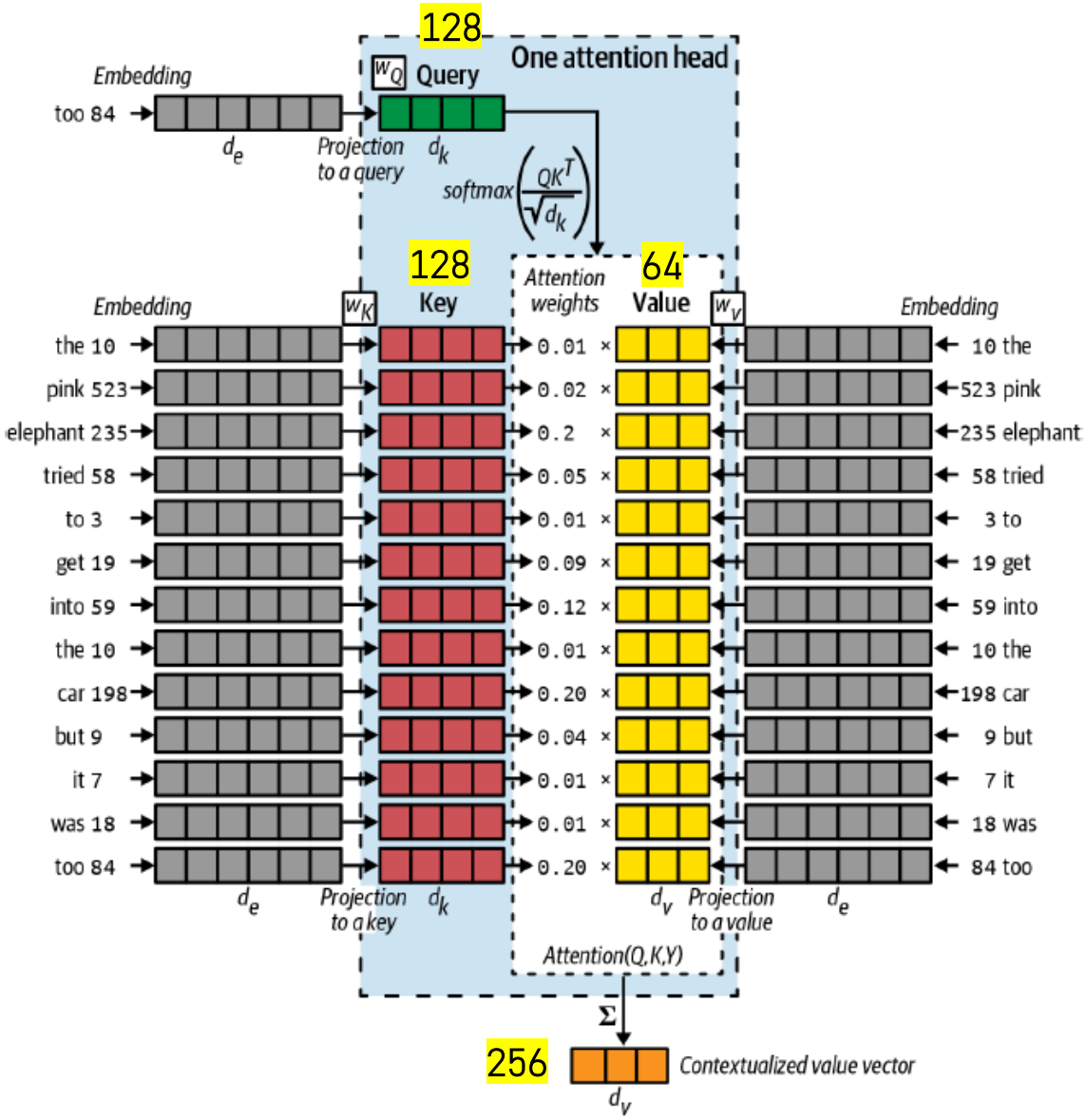
No hay razón para conformarse con

3. En Keras

```
layers.MultiHeadAttention(
    num_heads = 4,
    key_dim = 128,
    value_dim = 64,
    output_shape = 256
    )
```

- 4 cabezas de atención
- Las *keys* y *query* son vectores de longitud 128
- Los values (y por lo tanto la salida de cada cabeza) son v
- El vector de salida tiene longitud 256





Causal Masking

Evita que el modelo vea el futuro mientras aprende a predecir el siguiente token.

1. ¿Por qué necesitamos una máscara causal?

- En el entrenamiento de un modelo tipo **GPT**, queremos que aprenda a **predecir la siguiente palabra** usando **solo el contexto previo**.
- Sin una máscara, cada *query* podría "ver" todas las *keys*, incluso las que vienen **después** en la oración.
- Eso causaría fugas de información:
  - el modelo usaría información del futuro → tendría **pérdidas artificialmente bajas** y **no aprendería a generar texto correctamente**.

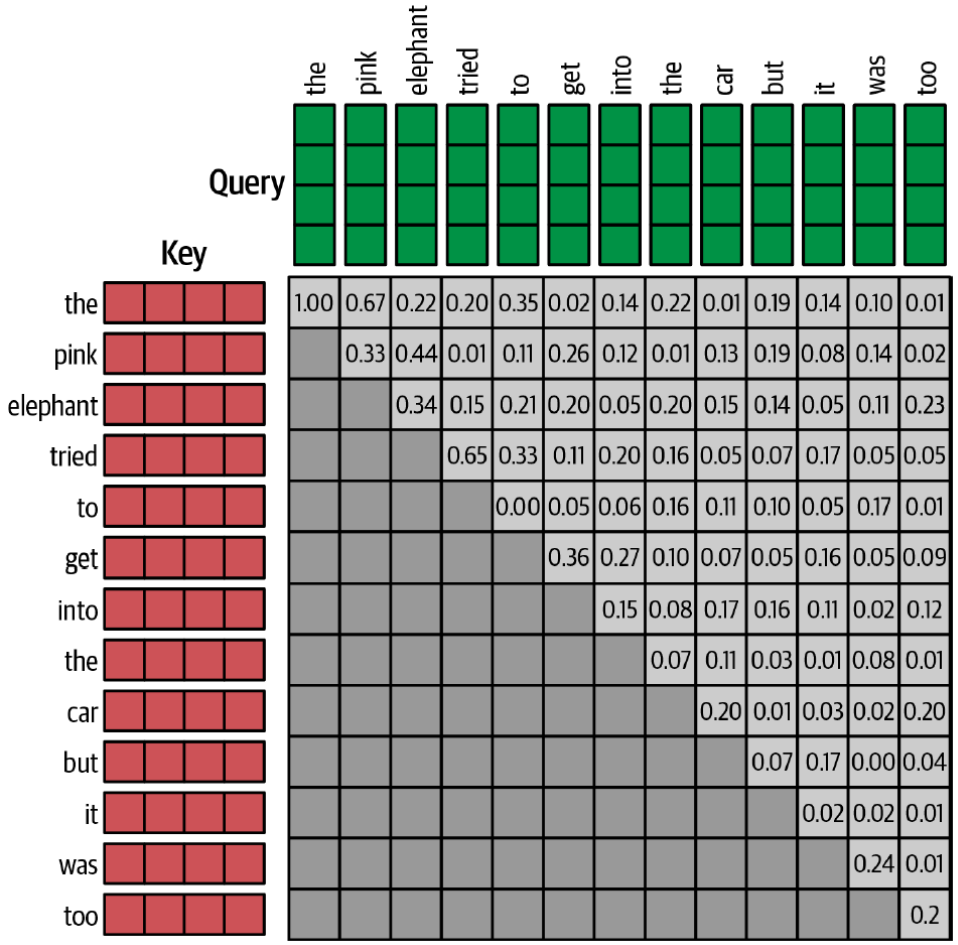
Causal Masking

Evita que el modelo vea el futuro mientras aprende a predecir el siguiente token.

2. ¿Qué hace la máscara causal?

- La máscara causal **bloquea las posiciones futuras** (las que aún no deberían ser visibles).
- En la matriz de atención QK^T, los valores correspondientes a palabras futuras se reemplazan por **−∞** antes del *softmax*, de modo que sus pesos de atención se vuelven **0**.
- Esto asegura que la atención de cada palabra solo use **palabras anteriores o ella misma**.

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



Causal Masking

gpt.ipynb

Google colab

Evita que el modelo vea el futuro mientras aprende a predecir el siguiente token.

Implementación en Keras

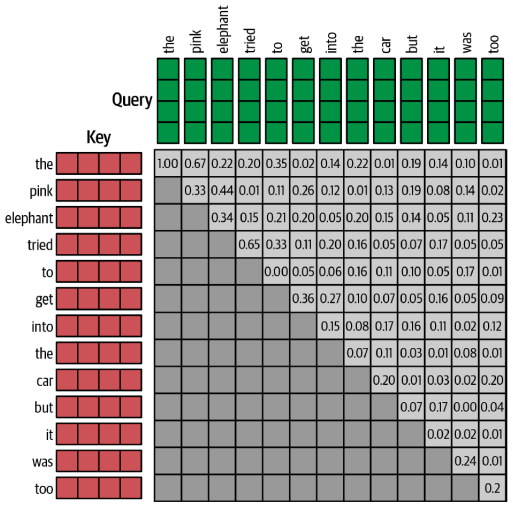
5. Crear la máscara de atención causal

```
def causal_attention_mask(batch_size, n_dest, n_src, dtype):
    i = tf.range(n_dest)[:, None]
    j = tf.range(n_src)
    m = i >= j - n_src + n_dest
    mask = tf.cast(m, dtype)
    mask = tf.reshape(mask, [1, n_dest, n_src])
    mult = tf.concat(
        [tf.expand_dims(batch_size, -1), tf.constant([1, 1], dtype=tf.int32)], 0
    )
    return tf.tile(mask, mult)
```

- `i = tf.range(n_dest)[:, None]` → Representa los índices de las *queries* (filas de la matriz).
- `j = tf.range(n_src)` → Representa los índices de las *keys* (columnas).
- `m = i >= j - n_src + n_dest` → Genera una **matriz booleana** del tamaño `(n_dest, n_src)`. El resultado es `True` (1) cuando una *query* puede ver esa *key*, y `False` (0) cuando la *key* está en el futuro.
- `mask = tf.cast(m, dtype)` → Convierte el booleano en un número (int, float32, etc.).
- `mask = tf.reshape(mask, [1, n_dest, n_src])` → Agrega una dimensión extra al inicio , esto permite aplicar la misma máscara a un *batch* completo.
- `return tf.tile(mask, mult)` → Repite la máscara para cada elemento del *batch* (cada ejemplo en el entrenamiento). Así, todos los ejemplos del lote usan el mismo patrón de enmascaramiento.

```
np.transpose(causal_attention_mask(1, 10, 10, dtype=tf.int32)[0])
```

Genera una matriz 10×10 (para una secuencia de 10 tokens) y la transpone para visualizarla como en la figura.
El resultado es una matriz triangular inferior: los "1" indican posiciones visibles y los "0" las enmascaradas (futuras).



¿Cuándo se necesita causal masking?

1. En los Transformers decodificadores (como GPT)

- El objetivo del modelo es **generar texto secuencialmente**, **prediciendo la siguiente palabra** a partir de las anteriores.
- Durante el entrenamiento, debe aprender a hacerlo **sin ver el futuro**.
- Por eso se aplica el **causal masking**:
  - Evita que el modelo "haga trampa" mirando tokens posteriores.

Dirección del contexto
Izquierda → Derecha

2. En los Transformers codificadores (como BERT)

- Su tarea no es generar texto, sino **comprenderlo**.
- Se entrenan con oraciones completas, por lo que pueden usar **contexto de ambos lados** (antes y después de la palabra).
- No requieren *causal masking*, porque no predicen el "siguiente token".
- Su tarea típica de entrenamiento se llama **Masked Language Modeling (MLM)**:
  - Se ocultan algunas palabras del texto (por ejemplo, *"The pink [MASK] tried to get into the car"*), y BERT debe **predecir la palabra faltante** usando todo el contexto.

Dirección del contexto
Bidireccional

El *causal mask* solo es necesario cuando el modelo aprende a **predecir el futuro**.
Si el modelo solo necesita **entender el presente**, puede mirar en ambas direcciones.

Una arquitectura sorprendentemente simple

- Todo el poder del Transformer proviene de **operaciones puramente algebraicas**:
  - Productos punto,
  - normalización (*softmax*),
  - y combinaciones lineales de vectores.
  - No hay *loops* como en las RNN, ni *filtros deslizantes* como en las CNN.

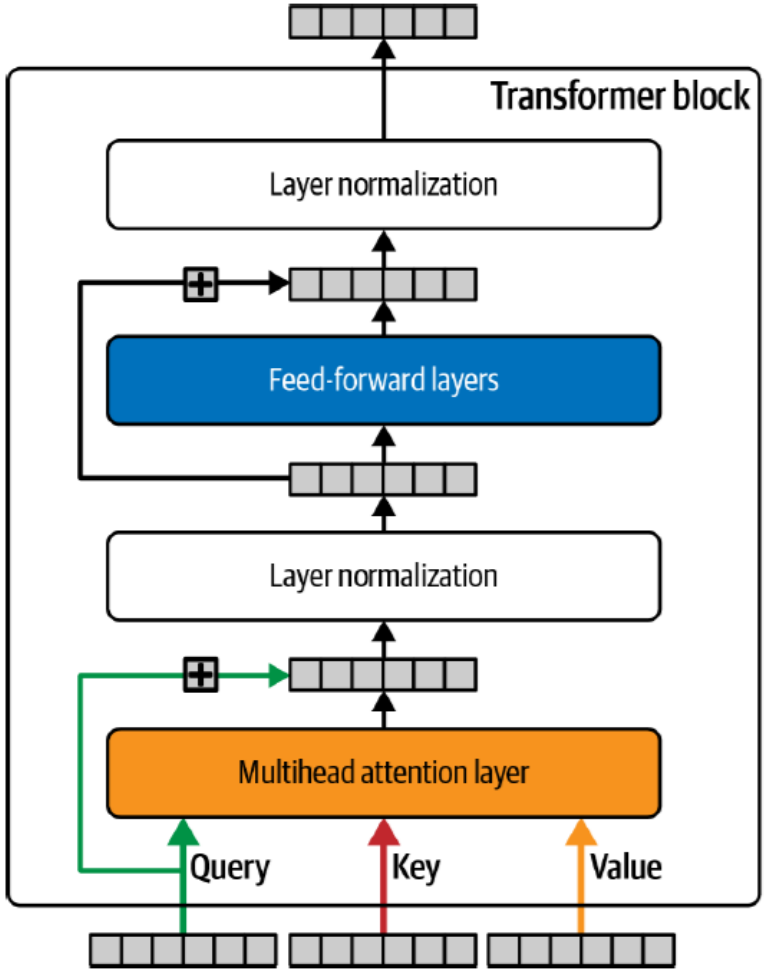
Los parámetros aprendibles consisten únicamente en **tres matrices de pesos totalmente conectadas** para cada cabeza de atención (W_Q, W_K, W_V) y **una matriz adicional de pesos** para **reajustar la salida** (W_O).

- Aun así, esta estructura logra:
  - capturar relaciones de largo alcance,
  - modelar dependencias contextuales complejas,
  - y escalar de manera eficiente con hardware moderno (GPU/TPU).

- El *multi-head attention* no actúa solo.
- Forma parte de un módulo más grande: el **Transformer block**, que integra:
  - *Layer Normalization*
  - *Residual Connections*
  - *Feed-Forward Networks*
- Este bloque se repite en múltiples capas para construir modelos como **GPT** o **BERT**.

Un bloque Transformer

- Es la **unidad básica** que se repite múltiples veces en un Transformer.
- Combina tres elementos principales alrededor del *multi-head attention*:
  - **Skip connections** (conexiones residuales)
  - **Normalización de capa** (*Layer Normalization*)
  - **Capas** *Feed-Forward* (densas)
- Cada bloque recibe una secuencia de embeddings y produce una **representación más abstracta y contextualizada**.



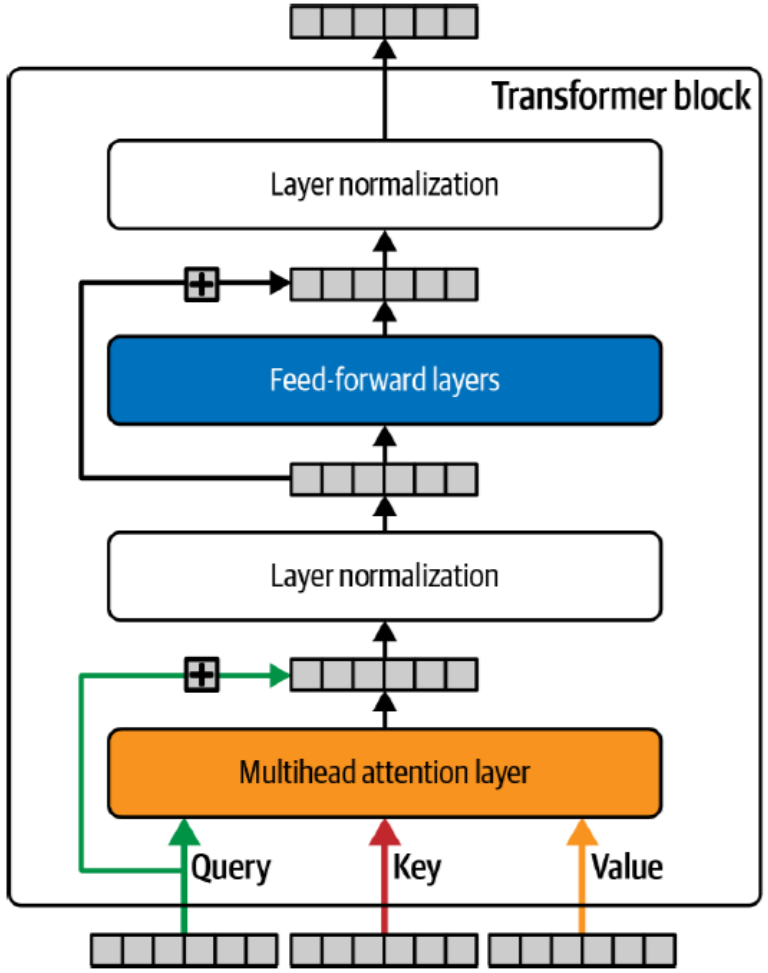
Un bloque Transformer

Conexiones residuales (*Skip connections*)

- La salida del *attention head* se **suma directamente** a su entrada original.

$$\text{output} = \text{LayerNorm}(x + \text{MultiHeadAttention}(x))$$

- Esto permite que **el gradiente fluya sin obstáculos**, evitando el *vanishing gradient problem* en redes profundas.
- En términos intuitivos:
  - el modelo "recuerda" lo que ya sabía y solo añade nueva información útil.
- Este tipo de conexión se introdujo originalmente en las **ResNets**, y hoy es un pilar en arquitecturas profundas como los Transformers.

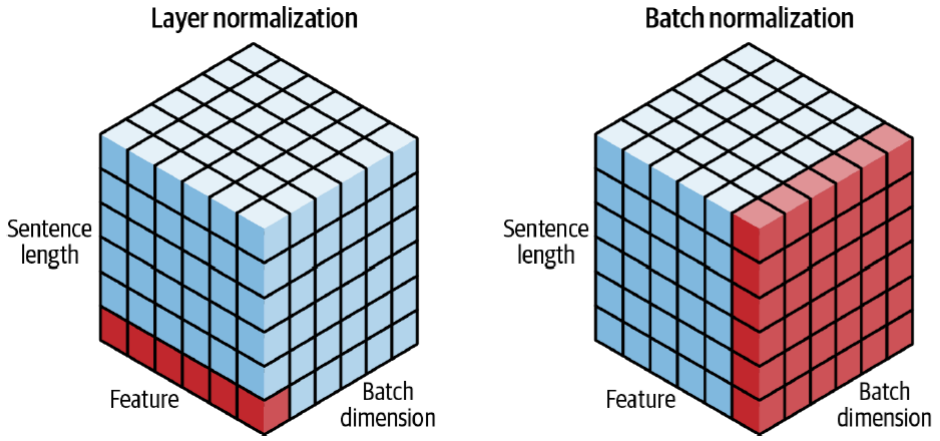


Un bloque Transformer

Capa de Normalización

- Se aplica después del *multi-head attention* y también después del *feed-forward*.
- Su función: **estabilizar el entrenamiento** y acelerar la convergencia.
- A diferencia de la *Batch Normalization*:
  - **BatchNorm:** normaliza por todo el *batch* (a lo largo de los ejemplos).
  - **LayerNorm:** normaliza por las **características dentro de cada token** (a lo largo de las dimensiones del embedding).

LayerNorm adapta la normalización a secuencias,
lo que permite que cada palabra sea tratada de forma independiente.



- Las celdas azules indican los valores usados para calcular las estadísticas de normalización.

* *LayerNorm* opera "en vertical", mientras que *BatchNorm* lo hace "en profundidad" sobre el lote.

Un bloque Transformer

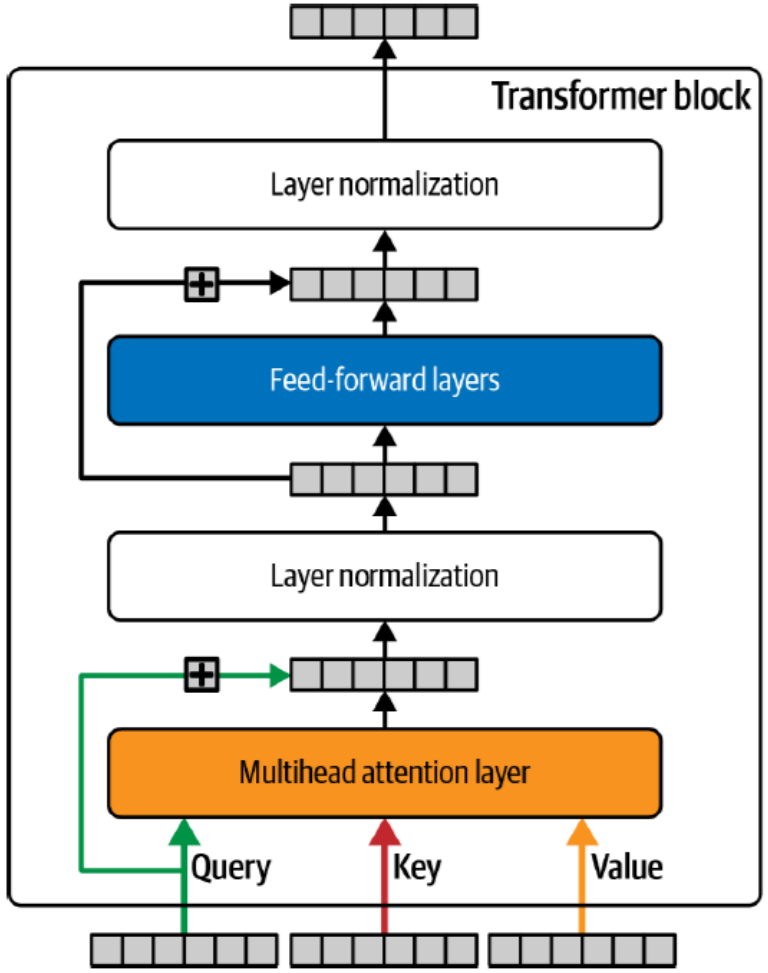
Capas Feed-Forward

- Después de la atención y la normalización, el bloque incluye **una red densa** (dos capas lineales con activación ReLU o GELU).

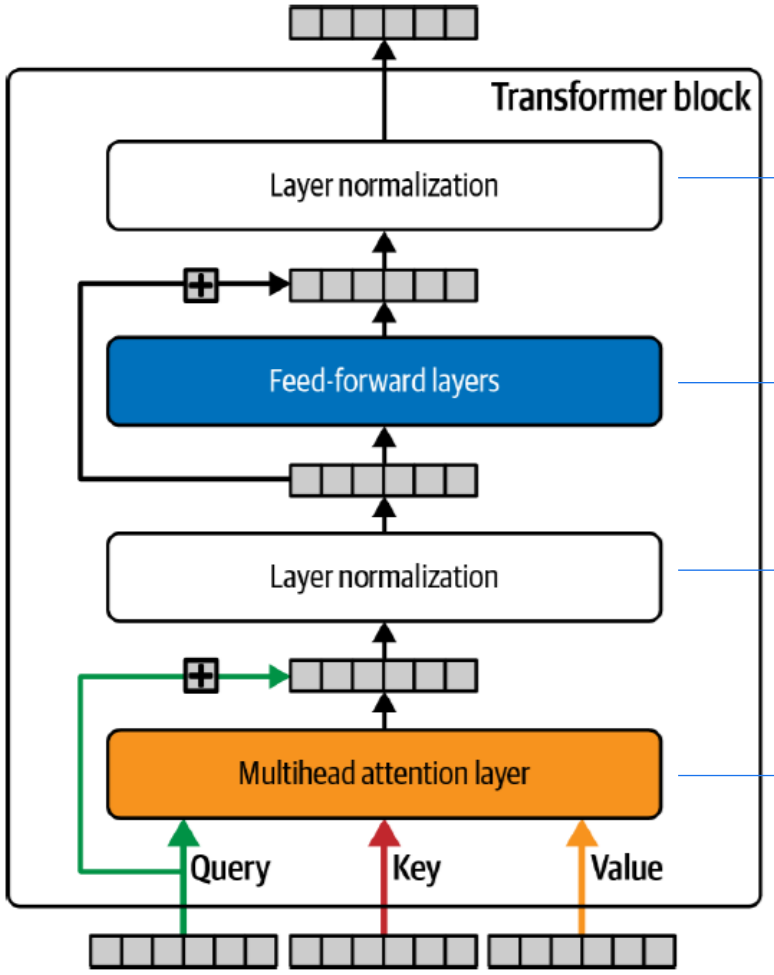
- Su propósito:
  - combinar y transformar las características aprendidas,
  - generar **representaciones de mayor nivel** a medida que se avanza por las capas.

- Estas capas se aplican **de manera idéntica** a cada posición de la secuencia (de forma paralela).

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$



Un bloque Transformer



Otra LayerNorm:
Prepara la salida para el siguiente bloque.

Feed-forward layers:
Transforman las representaciones.

Skip connection + LayerNorm:
Estabiliza y conserva información previa.

Multi-head attention:
Analiza relaciones entre tokens.

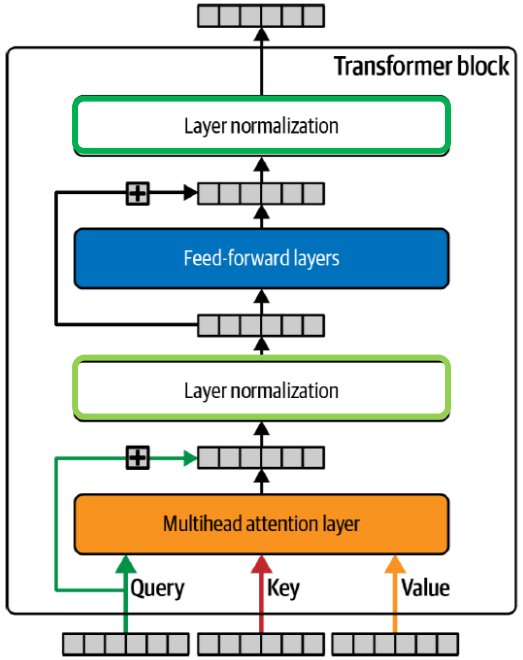
Cada *Transformer block* actúa como un mini-cerebro:
primero atiende al contexto, luego refina la información y finalmente normaliza lo aprendido antes de pasar al siguiente nivel.

gpt.ipynb

Google colab

```
VOCAB_SIZE = 10000
MAX_LEN = 80
EMBEDDING_DIM = 256
KEY_DIM = 256
N_HEADS = 2
FEED_FORWARD_DIM = 256
VALIDATION_SPLIT = 0.2
SEED = 42
LOAD_MODEL = False
BATCH_SIZE = 32
EPOCHS = 5
```

Un bloque Transformer



6. Crear una capa de bloque Transformer

```
class TransformerBlock(layers.Layer):
    def __init__(self, num_heads, key_dim, embed_dim, ff_dim, dropout_rate=0.1):
        super(TransformerBlock, self).__init__()
        self.num_heads = num_heads
        self.key_dim = key_dim
        self.embed_dim = embed_dim
        self.ff_dim = ff_dim
        self.dropout_rate = dropout_rate
        self.attn = layers.MultiHeadAttention(
            num_heads, key_dim, output_shape=embed_dim
        )
        self.dropout_1 = layers.Dropout(self.dropout_rate)
        self.ln_1 = layers.LayerNormalization(epsilon=1e-6)
        self.ffn_1 = layers.Dense(self.ff_dim, activation="relu")
        self.ffn_2 = layers.Dense(self.embed_dim)
        self.dropout_2 = layers.Dropout(self.dropout_rate)
        self.ln_2 = layers.LayerNormalization(epsilon=1e-6)
```

Crea la **capa de atención multi-cabeza**, que aprende las matrices W_Q, W_K, W_V, W_O y combina información contextual entre los tokens.

Define una **capa de Dropout** que se aplicará después de la atención, para reducir sobreajuste y mejorar la generalización.

Define la primera **normalización de capa** (*LayerNorm*) que estabiliza los valores tras la atención + conexión residual.

Primera capa densa de la red **Feed-Forward**: expande la dimensionalidad y aplica una transformación no lineal (ReLU).

2nda capa densa reduce de nuevo la dim. a `embed_dim` para mantener la compatibilidad con la entrada original.

Segunda capa de Dropout, aplicada tras la red *feed-forward*.

Segunda **normalización de capa**, aplicada después de la conexión residual final.
Estabiliza la salida antes de pasarla al siguiente bloque Transformer.

Un bloque Transformer

gpt.ipynb

Google colab

Define **el flujo de datos** dentro del bloque Transformer

```
def call(self, inputs):
    input_shape = tf.shape(inputs)
    batch_size = input_shape[0]
    seq_len = input_shape[1]
    causal_mask = causal_attention_mask(
        batch_size, seq_len, seq_len, tf.bool
    )
    attention_output, attention_scores = self.attn(
        inputs,
        inputs,
        attention_mask=causal_mask,
        return_attention_scores=True,
    )
    attention_output = self.dropout_1(attention_output)
    out1 = self.ln_1(inputs + attention_output)
    ffn_1 = self.ffn_1(out1)
    ffn_2 = self.ffn_2(ffn_1)
    ffn_output = self.dropout_2(ffn_2)
    return (self.ln_2(out1 + ffn_output), attention_scores)
```

Ejemplo: si el lote tiene 32 frases de 10 palabras cada una → `input_shape = [32, 10, embed_dim]`.

Crea la **máscara causal**

Aplica la **capa de atención multi-cabeza**:

Usa la misma entrada como *query*, *key* y *value* (por eso es *self-attention*).

Multiplica los vectores $QK^T/\sqrt{d_k}$,aplica *softmax*, y combina con V.

Regresa: `attention_output`: los embeddings contextualizados .
`attention_scores`: los pesos de atención (matriz que muestra qué tokens se atendieron entre sí).

Aplica **dropout** a la salida de la atención para evitar sobreajuste.

Primera capa de normalización

Pasa el resultado por la **primera capa densa** de la red *feed-forward* (expande la dimensionalidad y aplica ReLU).

Segunda capa densa: **reduce** la dimensionalidad de nuevo a `embed_dim` para mantener la compatibilidad con el tamaño de la entrada.

Aplica **dropout** tras el feed-forward para regularizar.

- Suma el resultado del *feed-forward* a su entrada (out1).
- Aplica LayerNormalization.
- Regresa:
  - La salida final del bloque Transformer (lista para el siguiente bloque).
  - Los pesos de atención (attention_scores), útiles para visualización o análisis.

Input
↓
MultiHeadAttention + CausalMask
↓
Dropout + Residual + LayerNorm
↓
FeedForward (Dense → Dense)
↓
Dropout + Residual + LayerNorm
↓
Output + Attention Scores

Un bloque Transformer

gpt.ipynb

Permite **guardar y volver a cargar** modelos personalizados.

```
def get_config(self):
    config = super().get_config()
    config.update(
        {
            "key_dim": self.key_dim,
            "embed_dim": self.embed_dim,
            "num_heads": self.num_heads,
            "ff_dim": self.ff_dim,
            "dropout_rate": self.dropout_rate,
        }
    )
    return config
```

Llama al método `get_config()` de la clase base (`layers.Layer`) para obtener la configuración **básica** de la capa.
Esto devuelve un diccionario con información estándar, por ejemplo:

```
{ "name": "transformer_block",
  "trainable": True,
  "dtype": "float32" }
```

Agrega al diccionario los hiperparámetros específicos de nuestra capa personalizada.
Así, al guardar el modelo, Keras sabrá exactamente con qué parámetros se construyó.

Devuelve el diccionario final con **toda la configuración** de la capa:
tanto la estándar (heredada) como la personalizada.

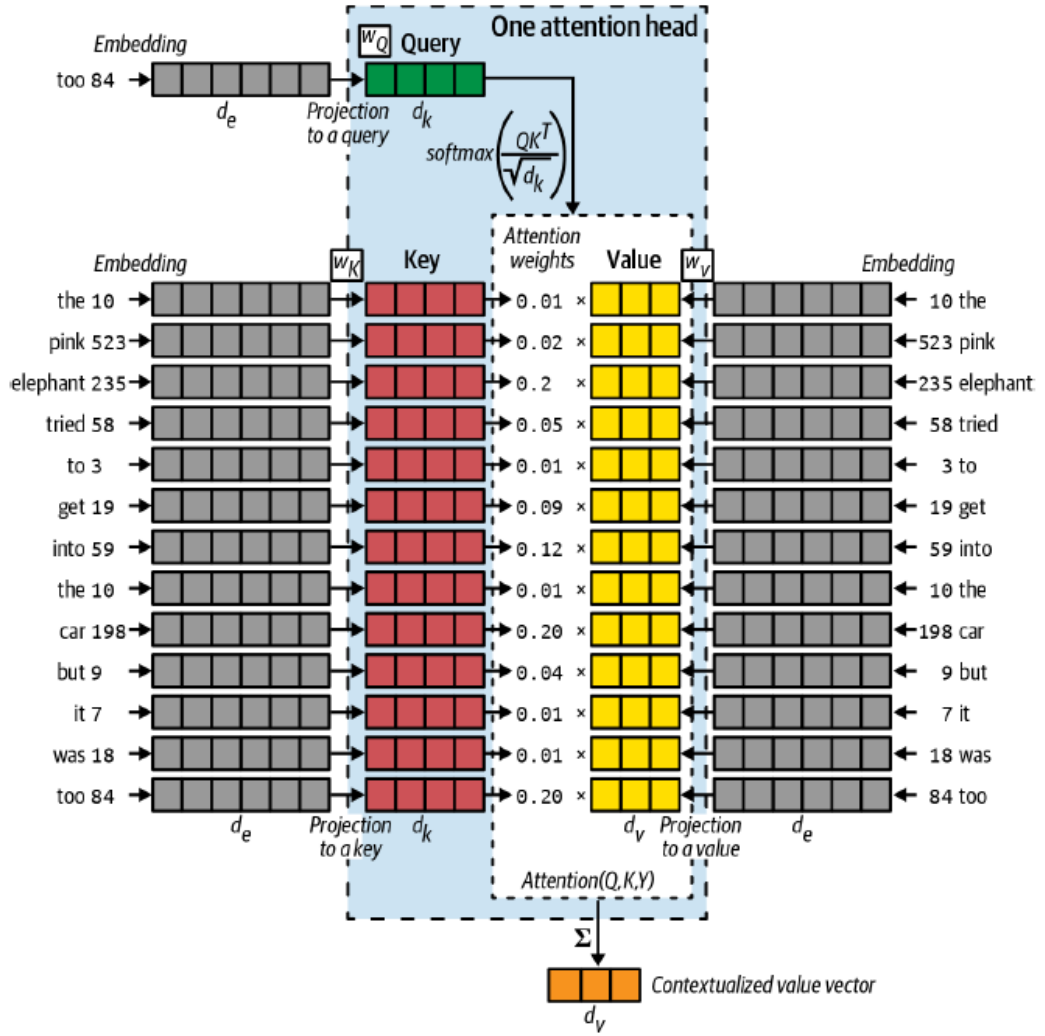
Guarda una receta completa para poder reconstruir este bloque más adelante

Positional Encoding

- Las **capas de atención** no tienen una noción del **orden de las palabras**.
  Cada *query–key* se compara **en paralelo**, sin saber si una palabra viene antes o después.

- Esto es una ventaja (permite **entrenar en paralelo**, a diferencia de las RNNs),
  pero también un problema: el modelo no puede distinguir frases como:

- "The **dog** looked at the **boy** and … (barked?)"
- "The **boy** looked at the **dog** and … (smiled?)"



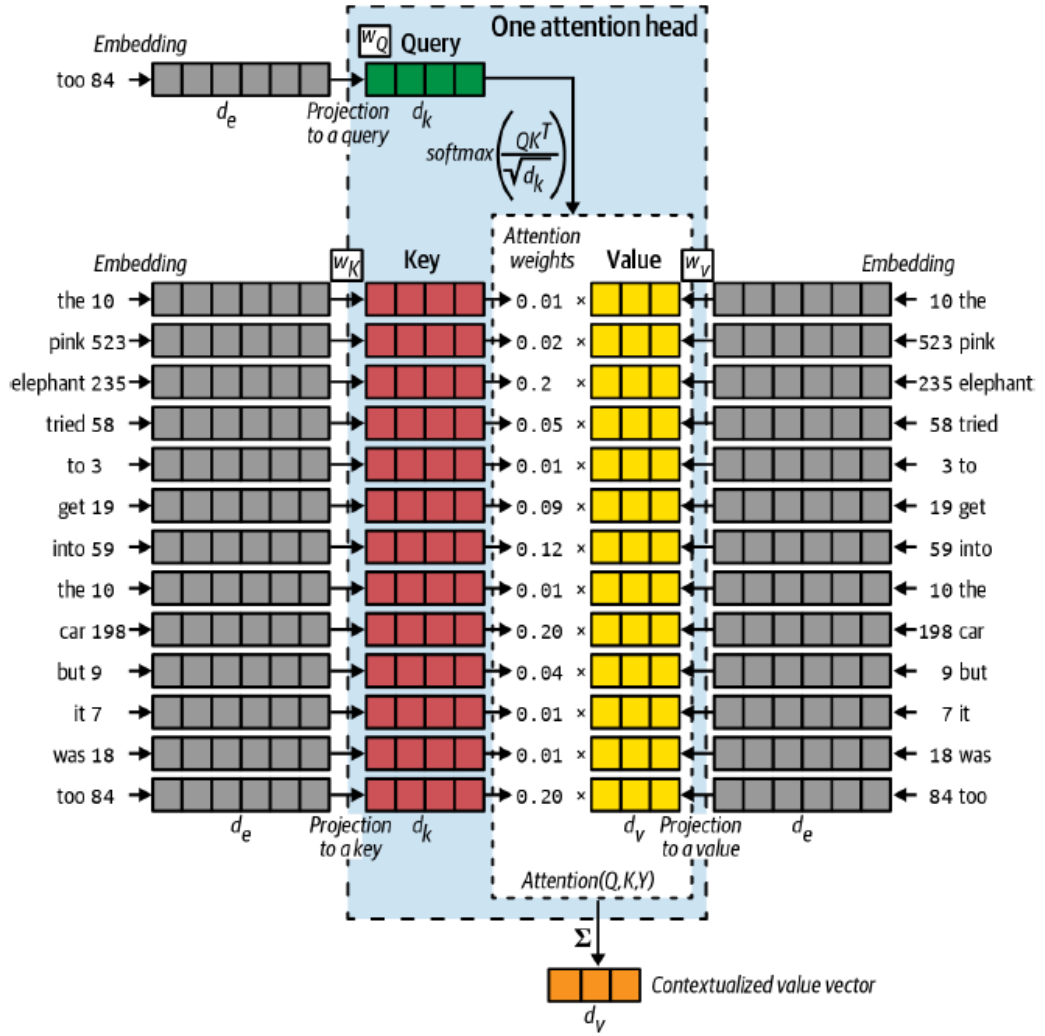
Positional Encoding

¿Cómo se resuelve?

- Se agrega información sobre la **posición** de cada palabra mediante una técnica llamada **positional encoding**.

- Cada palabra se representa con **dos embeddings**:

  - **Token embedding:** vector que captura el *significado* del token (por ejemplo, "dog", "boy").
  - **Positional embedding:** vector que codifica la *posición* del token dentro de la secuencia.



Positional Encoding

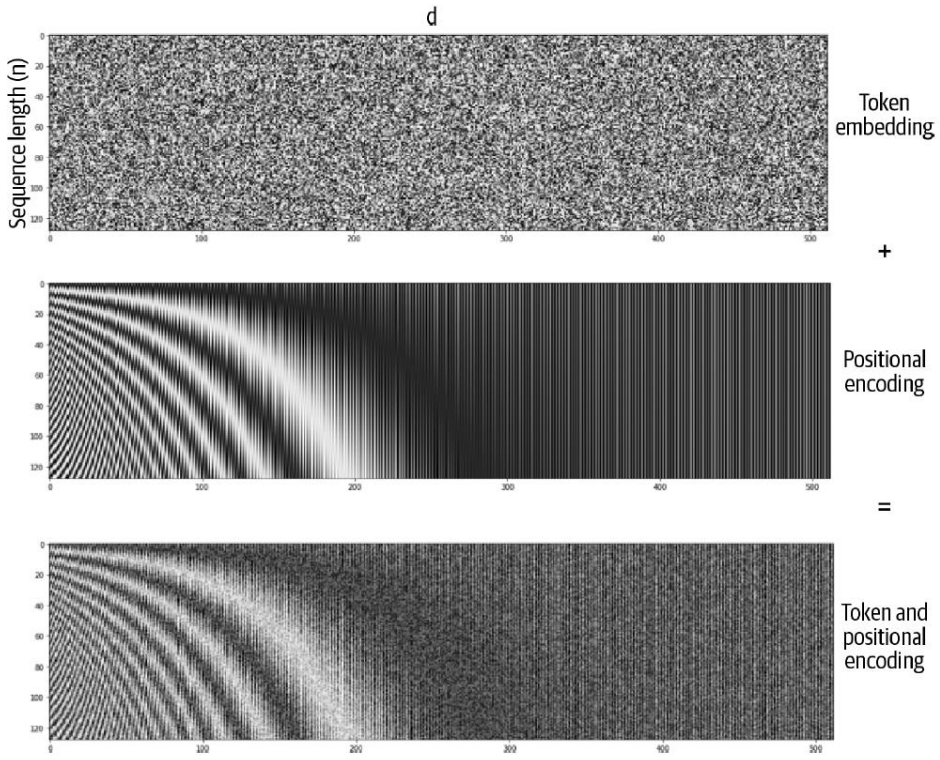
¿Cómo se combinan?

- Ambos vectores se **suman elemento a elemento**, para crear una representación conjunta:

$$\text{Input embedding} = \text{Token embedding} + \text{Positional embedding}$$

- Así, cada palabra tiene un vector que contiene simultáneamente:
  - Su **significado semántico**, y
  - Su **posición relativa** en la oración.

Cada fila representa una palabra de la oración, codificada como un vector denso.



A simple vista parece "ruido", porque los valores del embedding son **aprendidos** — no hay una estructura visible, pero sí información semántica interna.

Los valores varían suavemente a lo largo de la secuencia, de modo que el modelo puede inferir relaciones relativas como "antes", "después" o "cerca de".

Las franjas onduladas reflejan **patrones periódicos** que dependen de la posición del token.

Al sumar ambos, obtenemos un patrón más complejo que combina:
La **semántica** (del embedding de palabras), y
la **estructura secuencial** (del encoding posicional).
Así, cada vector final codifica **qué palabra es** y **dónde está**.

Positional Encoding

Implementación en GPT

- En GPT, el **positional embedding** se aprende con otra capa Embedding, igual que el embedding de palabras (es *aprendido*, no fijo).
- En el **Transformer original** (Vaswani et al., 2017),
  la posición se codificaba con **funciones trigonométricas** (seno y coseno).

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

$$\mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

p: El índice de la palabra en la secuencia (0, 1, 2, ..., *n*)

i: El índice de la dimensión dentro del vector de embedding (0, 1, 2, ..., *d/2*), porque cada i genera **dos componentes** (una de seno y una de coseno).

d: Tamaño total del vector de embedding (por ejemplo, 512)

"The pink elephant"

- `pos = 0` para "The"
- `pos = 1` para "pink"
- `pos = 2` para "elephant"
- `d = 4` (el embedding de cada token tiene 4 dimensiones → i = 0, 1)

Paso 1: Calcular denominadores

$$10000^{2i/d} \Rightarrow \begin{cases} i = 0 \Rightarrow 10000^{0} = 1 \\ i = 1 \Rightarrow 10000^{2/4} = 100 \end{cases}$$

Paso 2: Calcular cada componente

Token	pos	Dimensión	Fórmula
"The"	0	sin(pos/1)	sin(0/1) = 0
		cos(pos/1)	cos(0/1) = 1
		sin(pos/100)	sin(0/100) = 0
		cos(pos/100)	cos(0/100) = 1
"pink"	1	sin(1/1)	0.841
		cos(1/1)	0.540
		sin(1/100)	0.010
		cos(1/100)	0.999
"elephant"	2	sin(2/1)	0.909
		cos(2/1)	-0.416
		sin(2/100)	0.020
		cos(2/100)	0.999

Paso 3: Vector de posición

[0, 1, 0, 1]

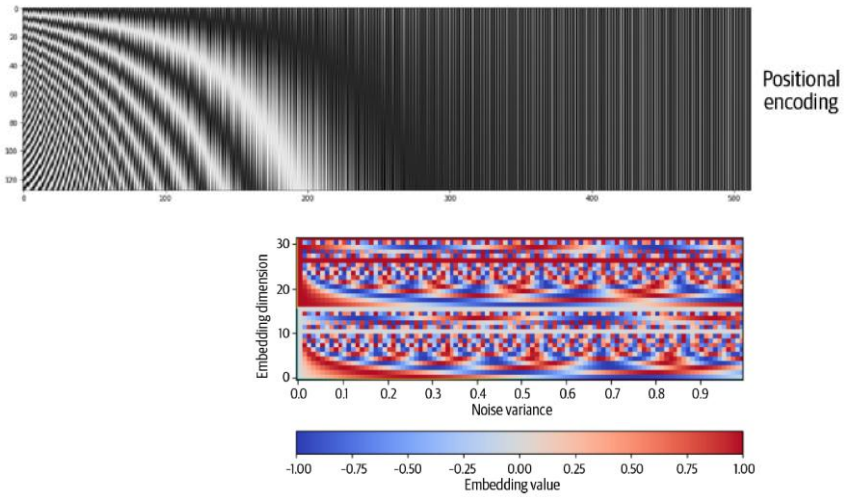
[0.841, 0.540, 0.010, 0.999]

[0.909, -0.416, 0.020, 0.999]

Positional Encoding

Es **la misma idea matemática que usamos en el modelo de difusión**: se necesitaba una forma de codificar el valor de un escalar en un espacio vectorial rico y continuo.

Porque las redes neuronales solo operan con vectores — no pueden razonar directamente sobre un número escalar como "posición 7" o "paso 450".



$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

$$\mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

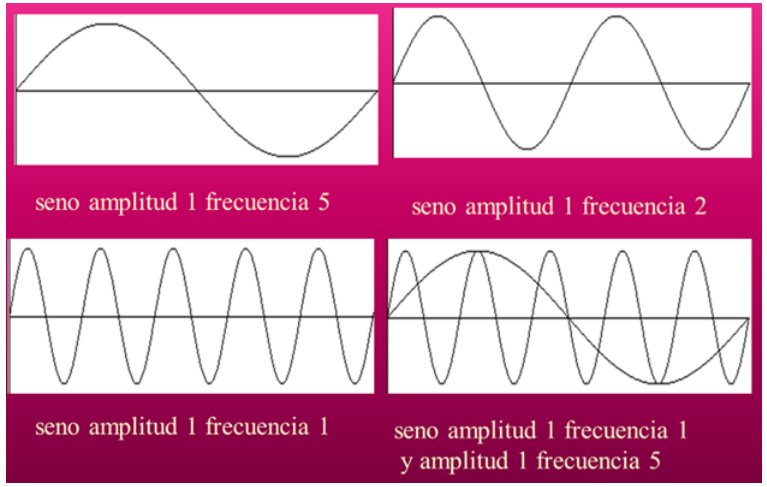
$$\gamma(\beta_t) = [\sin(2\pi e^0 f\beta_t), \ldots, \sin(2\pi e^{L-1} f\beta_t), \cos(2\pi e^0 f\beta_t), \ldots, \cos(2\pi e^{L-1} f\beta_t)]$$

Concepto	Usado en	Qué codifica	Fórmula base	Interpretación
Positional Encoding	Transformers	Posición del token	senos y cosenos con distintas frecuencias	"Dónde está cada palabra"
Time Embedding	Modelos de difusión	Paso temporal (timestep)	senos y cosenos con distintas frecuencias	"Qué tan avanzado está el proceso de ruido"

¿Por qué senos y cosenos?

Se buscaba que los vectores resultantes tuvieran las siguientes propiedades:

Propiedad	En Transformers	En Modelos de Difusión
Continuidad	Posiciones cercanas (tokens consecutivos) deben generar embeddings parecidos.	Tiempos cercanos (ruido similar) deben generar embeddings parecidos.
Periodicidad	Permite generalizar a secuencias más largas.	Permite extrapolar a pasos fuera del rango visto en entrenamiento.
Diferenciabilidad	Las ondas son suaves, ideales para backpropagation.	Igual: facilita el aprendizaje continuo del "tiempo" como variable.
Superposición de frecuencias	Codifica relaciones locales y globales entre palabras.	Codifica efectos del ruido en distintas escalas (finos y globales).



¿Por qué no una función no periódica?

Si usáramos una función **no periódica** (por ejemplo, lineal o exponencial):

$$PE(pos) = [pos, pos^2, pos^3, ...]$$

obtendríamos vectores:

- que **crecen indefinidamente** con el índice,
- con **distancias que aumentan sin control**,
- y con **pérdida de sensibilidad local** (las posiciones grandes se vuelven inaprendibles).

En cambio, las funciones seno/coseno:

- tienen **magnitud acotada** (entre -1 y 1),
- **mantienen continuidad**,
- y su **fase envuelve naturalmente** el espacio, lo cual da estabilidad numérica y geométrica.

Observación:

Una sola función seno o coseno repite su valor cada 2π unidades:

$$\sin(x) = \sin(x + 2\pi)$$

Entonces, si usáramos **una sola frecuencia** habría colisiones:

- dos posiciones distintas tendrían exactamente el mismo embedding → el modelo no podría distinguirlas.

La solución: usar *muchas frecuencias al mismo tiempo*

Cada i representa una frecuencia distinta:

- las de **baja frecuencia** varían lentamente (captan relaciones globales),
- las de **alta frecuencia** oscilan rápido (captan relaciones locales).

gpt.ipynb

Positional Encoding

7. Crear la incrustación de tokens y posiciones

Esta clase crea **embeddings de tokens + embeddings posicionales aprendidos** y los **suma** para entregar el vector de entrada al Transformer.

```python
class TokenAndPositionEmbedding(layers.Layer):
    def __init__(self, max_len, vocab_size, embed_dim):
        super(TokenAndPositionEmbedding, self).__init__()
        self.max_len = max_len
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim
        self.token_emb = layers.Embedding(
            input_dim=vocab_size, output_dim=embed_dim
        )
        self.pos_emb = layers.Embedding(input_dim=max_len, output_dim=embed_dim)

    def call(self, x):
        maxlen = tf.shape(x)[-1]
        positions = tf.range(start=0, limit=maxlen, delta=1)
        positions = self.pos_emb(positions)
        x = self.token_emb(x)
        return x + positions

    def get_config(self):
        config = super().get_config()
        config.update(
            {
                "max_len": self.max_len,
                "vocab_size": self.vocab_size,
                "embed_dim": self.embed_dim,
            }
        )
        return config
```

Guarda hiperparámetros: longitud máxima de secuencia, tamaño del vocabulario y dimensión del embedding.

Crea la **capa de embedding de tokens** (matriz `|V| × d_e`).
- Entrada: IDs de palabras.
- Salida: vectores de tamaño `embed_dim`.
- **Entrenable**: aprende representaciones semánticas.

Crea la **capa de embedding posicional aprendida** (matriz `max_len × d_e`).
- Entrada: índices de posición `0..max_len-1`.
- **Entrenable**: aprende cómo codificar la posición (no usa senos/cosenos aquí).

Obtiene la **longitud de la secuencia** del batch en tiempo de ejecución

Crea el vector `[0, 1, 2, ..., seq_len-1]` con las **posiciones** de cada token.

Convierte esas posiciones a **embeddings posicionales**.

Convierte los IDs de tokens en **embeddings de tokens**.

Suma ambos embeddings (token + posición).

Devuelve un diccionario con la **configuración** de la capa para poder **guardar/cargar** el modelo y reconstruirla con los mismos hiperparámetros.

Entrenamiento del modelo GPT

En este punto ya contamos con **todas las piezas del modelo Transformer**: embeddings, atención, bloques Transformer y capa final de salida.

Entrada:
El texto (por ejemplo, *"the pink elephant tried to get into the car but it was too …"*) se convierte en una secuencia de índices numéricos.

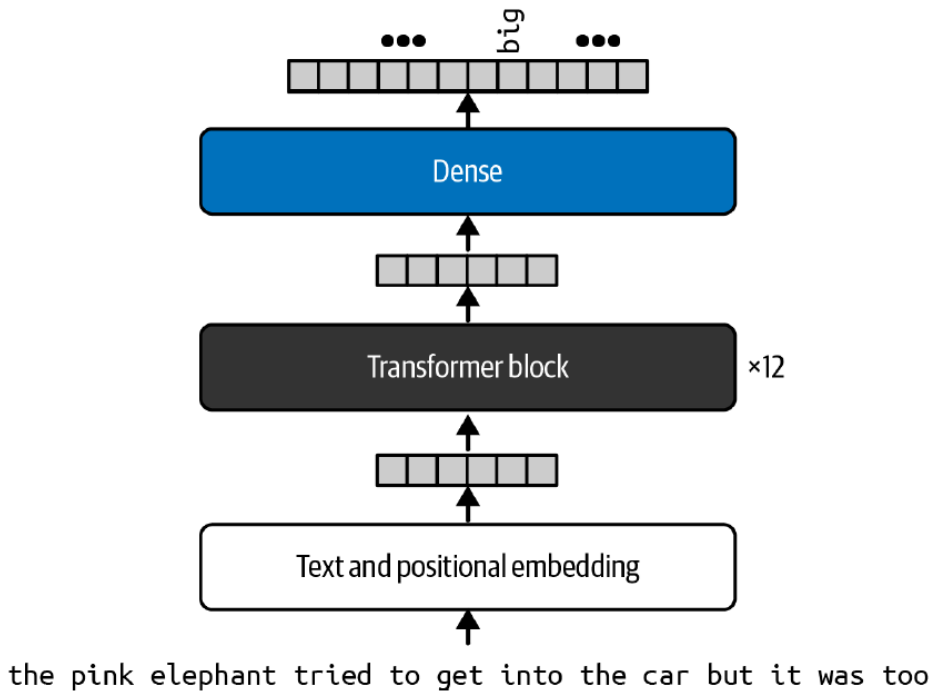
Text and positional embedding:
Cada índice se transforma en un vector denso que combina **significado (token embedding)** y **posición (positional embedding)**.

Transformer block:
El embedding pasa por uno o varios **bloques Transformer**, donde el modelo aprende a **atender a diferentes palabras del contexto** para predecir la siguiente palabra.

* En el paper original de GPT se usan **12 bloques**, pero aquí se emplea **uno solo** para simplificar el entrenamiento.

Dense layer + Softmax:
La salida del Transformer pasa por una **capa densa final**, que asigna una probabilidad a **cada palabra del vocabulario**.
El modelo predice cuál es **la palabra más probable** que sigue en la secuencia.

Objetivo de entrenamiento:
Ajustar los pesos de todo el modelo para **minimizar el error de predicción**, comparando la palabra predicha con la palabra real siguiente.



gpt.ipynb

Entrenamiento del modelo GPT

8. Construir el modelo del Transformer

```
inputs = layers.Input(shape=(None,), dtype=tf.int32)
x = TokenAndPositionEmbedding(MAX_LEN, VOCAB_SIZE, EMBEDDING_DIM)(inputs)
x, attention_scores = TransformerBlock(
    N_HEADS, KEY_DIM, EMBEDDING_DIM, FEED_FORWARD_DIM
)(x)
outputs = layers.Dense(VOCAB_SIZE, activation="softmax")(x)
gpt = models.Model(inputs=inputs, outputs=[outputs, attention_scores])
gpt.compile("adam", loss=[losses.SparseCategoricalCrossentropy(), None])
```

Define la **entrada** del modelo: una secuencia de IDs de tokens de longitud variable (B, T) (enteros).

Convierte los IDs en **embeddings** y **embeddings posicionales** y los **suma:** (B, T, EMBEDDING_DIM)

Aplica un **bloque Transformer** (self-attention con **máscara causal**, residuals, layer norm y feed-forward). Devuelve:

- `x`: representaciones **contextualizadas** `(B, T, EMBEDDING_DIM)`.
- `attention_scores`: **pesos de atención** `(B, N_HEADS, T, T)` para inspección.

Proyecta cada posición a una distribución de probabilidad sobre el **vocabulario**.
Salida: (B, T, VOCAB_SIZE) — probabilidades del **siguiente token** por posición.

Construye el modelo Keras con **dos salidas**:
1.. probabilidades sobre el vocabulario
2.`attention_scores` (solo para monitoreo/visualización).

$$\mathrm{SCCE}(y, \hat{y}) = -\frac{1}{N}\sum_{n=1}^{N}\log(\hat{y}_{n,y_n})$$

Es la probabilidad predicha para la clase correcta

- Usa **Adam** como optimizador.
- Define **pérdida solo para la primera salida** (clasificación del próximo token) con `SparseCategoricalCrossentropy` (compara la predicción del token en la posición *t* con la palabra real *t* del target.- targets como enteros).
- `None` indica **sin pérdida** para `attention_scores` (no se entrena contra ellos).

Nota práctica: Estamos entrenando de manera autoregresiva, entonces los **labels *y*** son la secuencia **desplazada 1**

```
gpt.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, None)	0
token_and_position_embedding (TokenAndPositionEmbedding)	(None, None, 256)	2,580,480
transformer_block (TransformerBlock)	[(None, None, 256), (None, 2, None, None)]	658,688
dense_2 (Dense)	(None, None, 10000)	2,570,000

Total params: 5,809,168 (22.16 MB)
Trainable params: 5,809,168 (22.16 MB)
Non-trainable params: 0 (0.00 B)

Entrenamiento del modelo GPT

gpt.ipynb

Google colab

9. Train the Transformer

```
# Creatr un TextGenerator checkpoint
class TextGenerator(callbacks.Callback):
    def __init__(self, index_to_word, top_k=10):
        self.index_to_word = index_to_word
        self.word_to_index = {
            word: index for index, word in enumerate(index_to_word)
        }

    def sample_from(self, probs, temperature):
        probs = probs ** (1 / temperature)
        probs = probs / np.sum(probs)
        return np.random.choice(len(probs), p=probs), probs

    def generate(self, start_prompt, max_tokens, temperature):
        start_tokens = [
            self.word_to_index.get(x, 1) for x in start_prompt.split()
        ]
        sample_token = None
        info = []
        while len(start_tokens) < max_tokens and sample_token != 0:
            x = np.array([start_tokens])
            y, att = self.model.predict(x, verbose=0)
            sample_token, probs = self.sample_from(y[0][-1], temperature)
            info.append(
                {
                    "prompt": start_prompt,
                    "word_probs": probs,
                    "atts": att[0, :, -1, :],
                }
            )
            start_tokens.append(sample_token)
            start_prompt = start_prompt + " " + self.index_to_word[sample_token]
        print(f"\ngenerated text:\n{start_prompt}\n")
        return info

    def on_epoch_end(self, epoch, logs=None):
        self.generate("wine review", max_tokens=80, temperature=1.0)
```

Guarda el vocabulario y también crea el diccionario inverso palabra → índice para poder convertir texto en tokens.

Recibe las probabilidades predichas por el modelo (`probs`) y la temperatura deseada.

Ajusta las probabilidades aplicando el **control de temperatura**.

Normaliza para que la suma de las probabilidades sea 1 (vuelve a ser una distribución válida).

Toma una **muestra aleatoria ponderada** según las probabilidades (`np.random.choice`).

Genera texto a partir de un *prompt inicial*.

Convierte el texto inicial en índices usando el diccionario creado.

Genera tokens hasta alcanzar el máximo o encontrar el token de final (0).

Predice: `y`: distribución de probabilidad para el siguiente token, `att`: pesos de atención para esa predicción.

Selecciona el próximo token aplicando temperatura al softmax.

Guarda la información de cada paso: el texto generado, las probabilidades y la atención.

Agrega el nuevo token a la secuencia y actualiza el texto visible.

Imprime el texto final generado y devuelve el registro de probabilidades y atención.

Se ejecuta automáticamente **al final de cada época** de entrenamiento.

Genera un texto de ejemplo para ver cómo evoluciona el modelo mientras aprende.

Entrenamiento del modelo GPT

gpt.ipynb

Google colab

```
gpt.fit(
    train_ds,
    epochs=EPOCHS,
    callbacks=[tensorboard_callback, text_generator],
)
```

```
Epoch 1/5
4060/4060 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - loss: 2.5973
generated text:
wine review : france : southwest france : bordeaux - style red blend : from one of 90 acres of two small vineyards in margaux , the garonne river . cabernet franc adds an older wine with

4060/4060 ━━━━━━━━━━━━━━━━━━━━ 212s 49ms/step - loss: 2.5972
Epoch 2/5
4059/4060 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - loss: 1.9768
generated text:
wine review : france : burgundy : merlot : rich and extracted [UNK] - textured . this wine needs a year going to have to be ready to drink .

4060/4060 ━━━━━━━━━━━━━━━━━━━━ 114s 28ms/step - loss: 1.9768
Epoch 3/5
4059/4060 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - loss: 1.8990
generated text:
wine review : us : california : cabernet sauvignon : spicy black fruit and cedar mark this wine known in malbec . integrated oak supports its red - cherry , oak and plum aromas on this f

4060/4060 ━━━━━━━━━━━━━━━━━━━━ 123s 30ms/step - loss: 1.8990
Epoch 4/5
4059/4060 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - loss: 1.8507
generated text:
wine review : brazil : protect : petite sirah : this wine dates - like [UNK] ' s standard [UNK] - smelling , it has the palate - based aromas of vanilla , cola , roasted meat and plum .

4060/4060 ━━━━━━━━━━━━━━━━━━━━ 122s 30ms/step - loss: 1.8507
Epoch 5/5
4060/4060 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - loss: 1.8236
generated text:
wine review : us : california : chardonnay : this has a beautiful chardonnay [UNK] notable for its vibrant , appley apple , banana and buttered toast flavors that are smooth and suave ,

4060/4060 ━━━━━━━━━━━━━━━━━━━━ 116s 28ms/step - loss: 1.8236
<keras.src.callbacks.history.History at 0x7e6c375ad490>
```