Transformers for Natural Language Processing

Build, train, and fine-tune deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, and GPT-3

Estructura general del taller

1. **Parte I: Introducción a las arquitecturas Transformer**

   - **¿Qué son los Transformers?**
   - **Comenzando con la arquitectura del modelo Transformer**
   - **Ajuste fino de modelos BERT**
   - **Preentrenando un modelo RoBERTa desde cero**

Estructura general del taller

2. **Parte II: Aplicando Transformers para la comprensión y generación del lenguaje natural**

   - **Traducción automática con el Transformer**
   - **El ascenso de los Transformers suprahumanos con los motores GPT-3,**
   - **Aplicando Transformers a documentos legales y financieros para el resumen de textos con IA**
   - **Coincidencia de tokenizadores y conjuntos de datos**
   - **Etiquetado de roles semánticos con transformers basados en BERT**

Estructura general del taller

3. **Parte III: Técnicas avanzadas de comprensión del lenguaje**

- **Deja que tus datos hablen: historia, preguntas y respuestas**
- **Detectando emociones de clientes para hacer predicciones**
- **Analizando noticias falsas con Transformers**
- **Interpretando modelos Transformer de caja negra**

¿Qué son los Transformers?

¿Qué necesitaremos?

La mayoría de los programas son cuadernos de Colaboratory.

Todo lo que necesitarás es una cuenta gratuita de Google Gmail, y podrás ejecutar los cuadernos en la máquina virtual gratuita de Google Colaboratory.

Para algunos de los programas educativos, necesitarás tener Python instalado en tu máquina.

¿Qué son los Transformers?

- Los transformers son modelos post-aprendizaje profundo industrializados y homogeneizados.
- A través de la homogeneización, un solo modelo transformer puede realizar una amplia gama de tareas sin necesidad de afinación adicional.
- Estas arquitecturas particulares del post-aprendizaje profundo se llaman modelos fundacionales.

El ecosistema de los transformers

- Los modelos transformer representan un cambio de paradigma tan grande que requieren un nuevo nombre para describirlos: modelos fundacionales.

- Los modelos fundacionales no fueron creados por la academia, sino por la gran industria tecnológica.

- Las grandes tecnológicas tuvieron que encontrar un modelo mejor para enfrentar el aumento exponencial de petabytes de datos que fluyen hacia sus centros de datos.

Modelos fundacionales

Los transformers tienen dos características distintas:

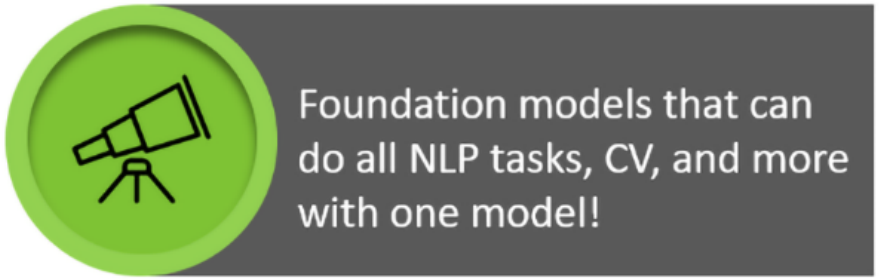
1. un alto nivel de homogeneización.
2. Propiedades emergentes asombrosas.

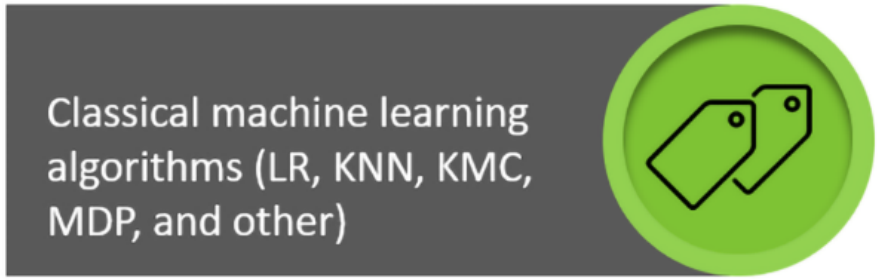
Los modelos fundacionales, aunque diseñados con una arquitectura innovadora, se basan en la historia de la IA.

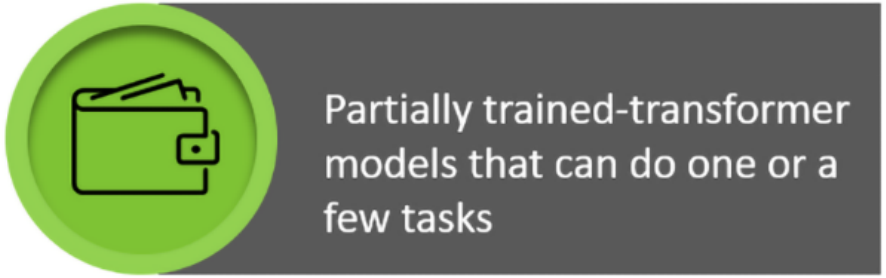
Como resultado, ¡el rango de habilidades de un especialista en inteligencia artificial está expandiéndose!

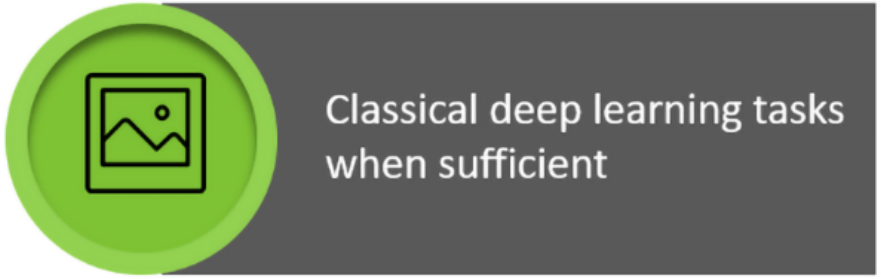
Modelos fundacionales

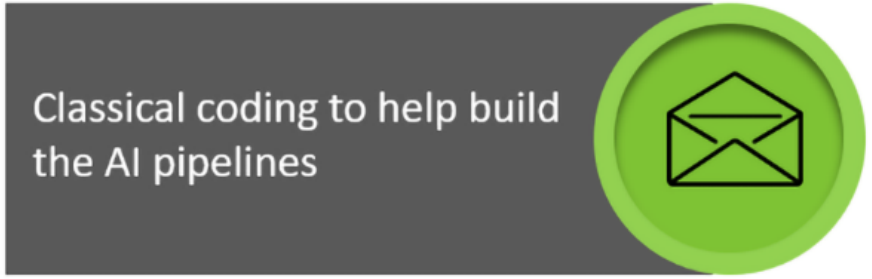
The new paradigm of AI











Modelos fundacionales

El ecosistema actual de modelos transformer es único en la evolución de la inteligencia artificial y se puede resumir en cuatro propiedades:

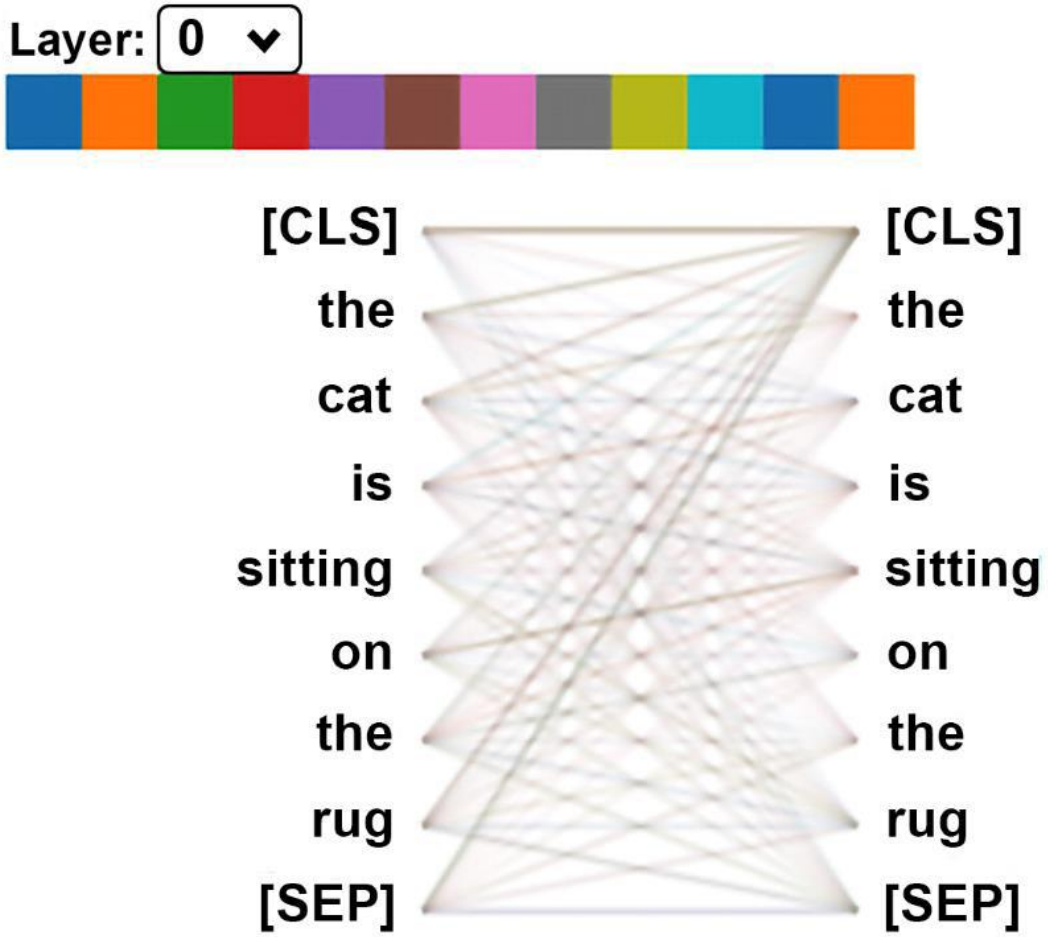
1. **Arquitectura del modelo**
2. **Datos**
3. **Poder de cómputo**
4. **Ingeniería de prompts**

El futuro de los especialistas en inteligencia artificial

- El impacto social de los modelos fundacionales no debe subestimarse.
- La ingeniería de *prompts* se ha convertido en una habilidad requerida para los especialistas en inteligencia artificial.
- Un especialista en IA estará involucrado en algoritmos de máquina a máquina utilizando IA clásica, IoT, computación en el borde (edge computing) y más.

Optimizando modelos de PLN con transformers

- Las Redes Neuronales Recurrentes (RNNs), incluidas las LSTMs, han aplicado redes neuronales a modelos de secuencias de PLN durante décadas.
- El uso de la funcionalidad recurrente llega a su límite cuando se enfrenta a secuencias largas.
- El concepto central de un transformer puede resumirse de manera aproximada como "mezclar tokens".

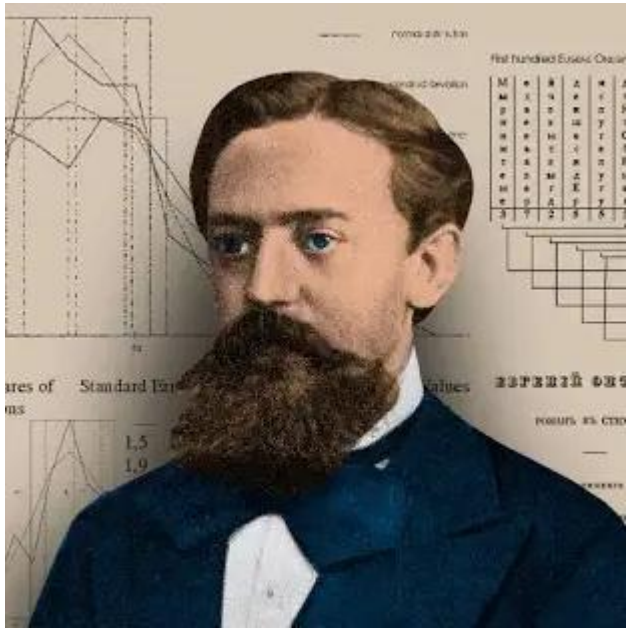


El origen de los transformers

Andrey Markov introdujo el concepto de valores aleatorios y creó una teoría de procesos estocásticos.

Los conocemos en IA como:

- El Proceso de Decisión de Markov (MDP)
- Cadenas de Markov.
- Procesos de Markov.

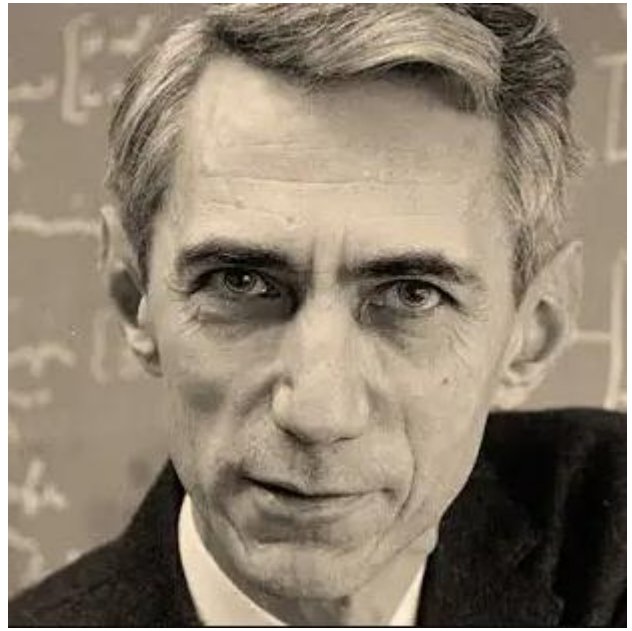


Andrei Andreyevich Markov (Rusia)
14 de junio 1856 – 20 julio 1922

El origen de los transformers

En 1948, se publicó *The Mathematical Theory of Communication* de Claude Shannon.

Él creó la teoría de la información tal como la conocemos hoy.



Claude Shannon (EEUU)
30 de abril 1916 – 24 febrero 2001

El origen de los transformers

En 1950, Alan Turing publicó su artículo seminal: *Computing Machinery and Intelligence*.

Turing basó este artículo sobre la inteligencia de las máquinas en la exitosa máquina de Turing.

Alan Turing (GB)
23 de junio 1912 – 7 junio 1954

El origen de los transformers

En 1954, el experimento Georgetown-IBM utilizó computadoras para traducir oraciones en ruso al inglés usando un sistema de reglas.

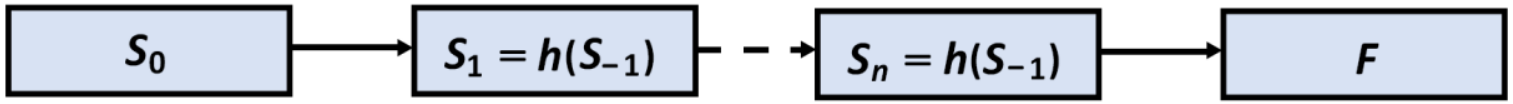
La expresión "Inteligencia Artificial" fue utilizada por primera vez por John McCarthy en 1956, cuando se estableció que las máquinas podían aprender.

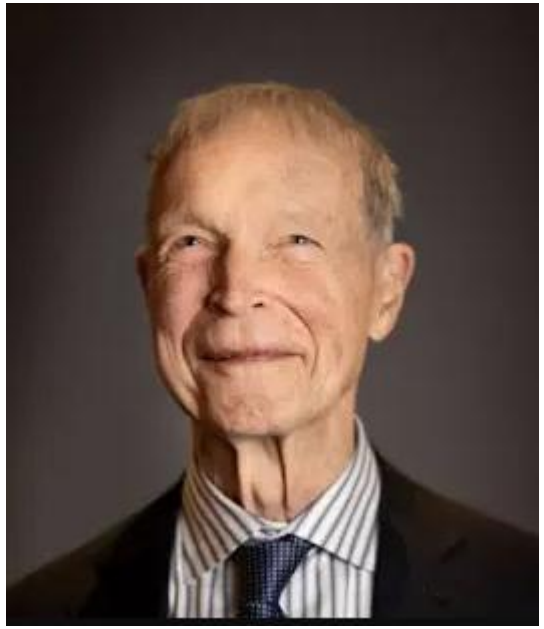
7 enero 1954

El origen de los transformers

En 1982, John Hopfield introdujo una RNN, conocida como redes Hopfield o redes neuronales "asociativas".

Un RNN memoriza los estados persistentes de una secuencia de manera eficiente, como se muestra en la figura:





John Hopfield (EEUU)
15 de julio 1933

El origen de los transformers

En los años 80, Yann LeCun diseñó la Red Neuronal Convolucional (CNN) multipropósito.

En los años 90, resumiendo varios años de trabajo, Yann LeCun produjo LeNet-5, lo que dio lugar a muchos de los modelos CNN que conocemos hoy en día.

Yann LeCun (Fr)
8 de julio 1960

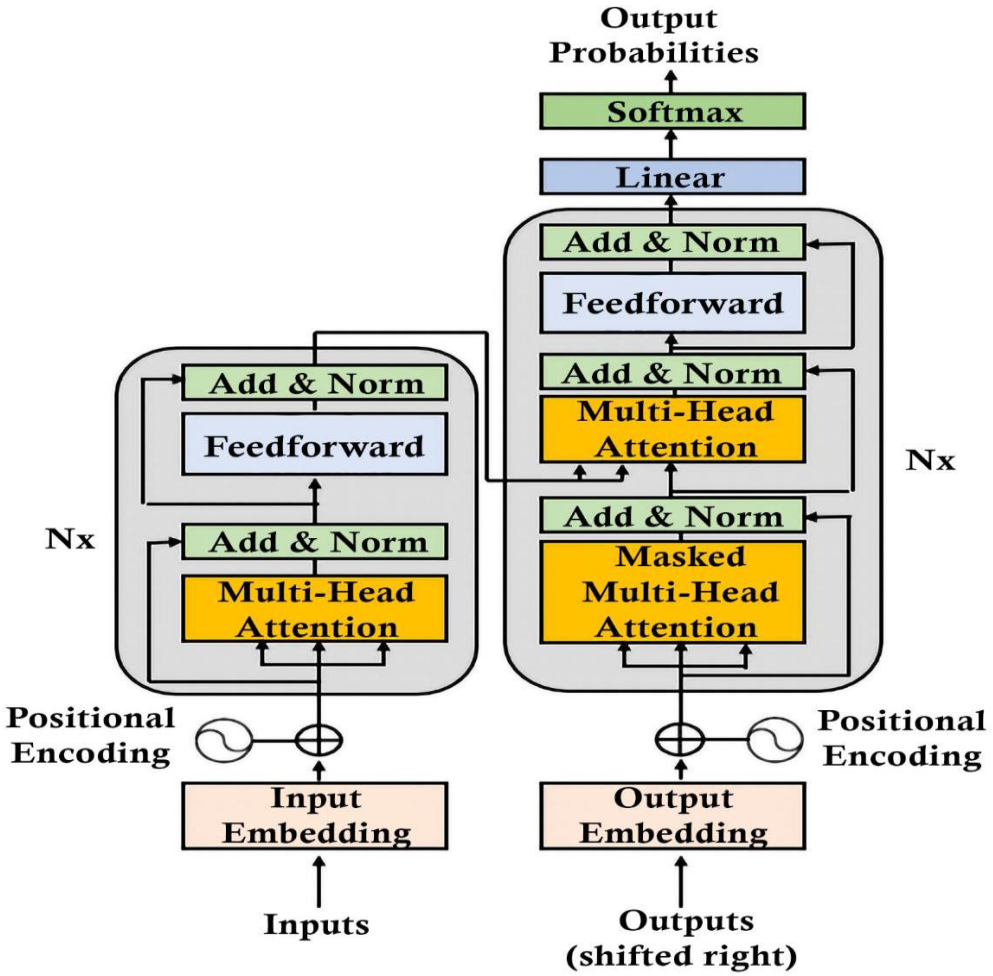
Comenzando con la arquitectura del modelo Transformer

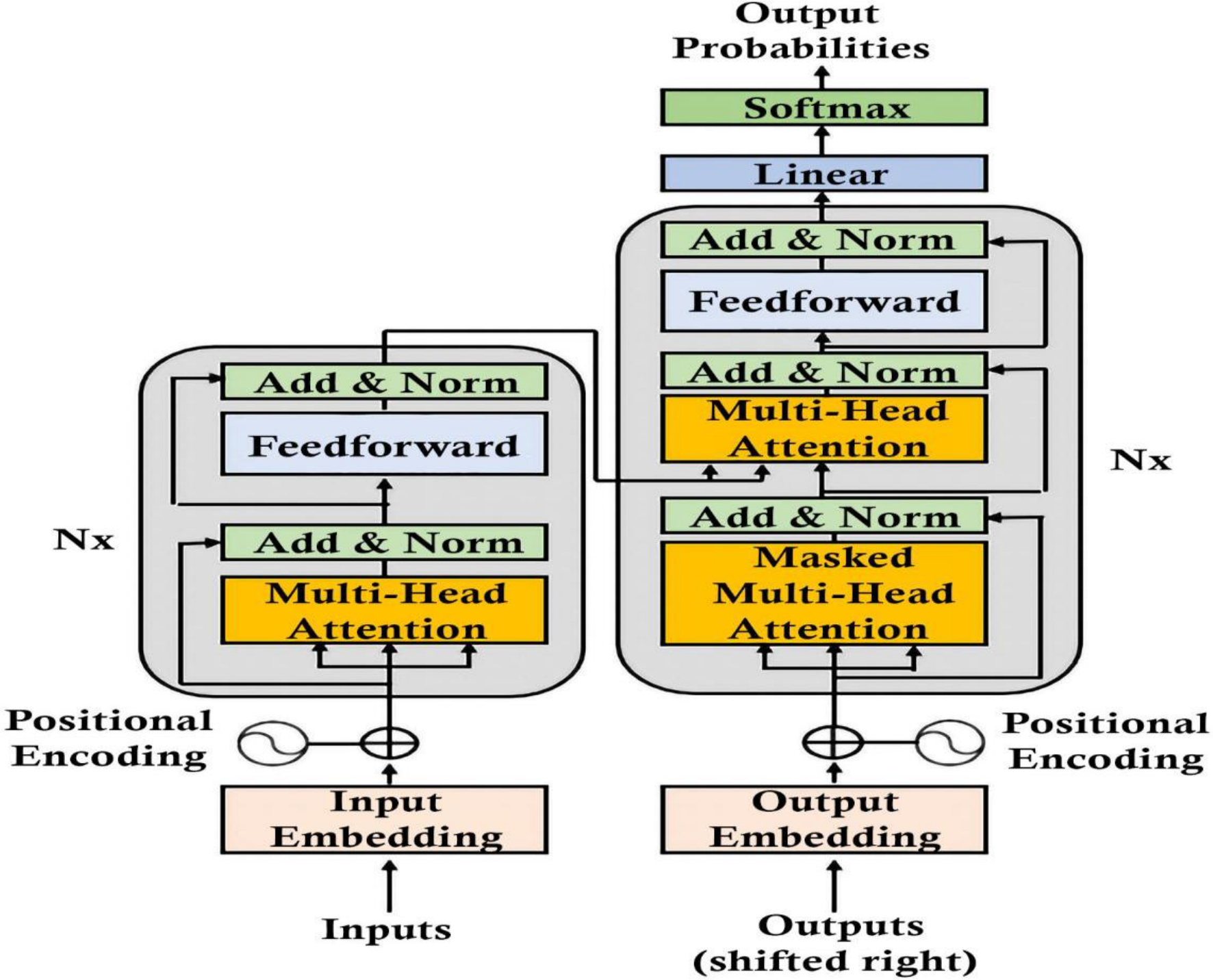
Comenzando con la arquitectura del modelo Transformer

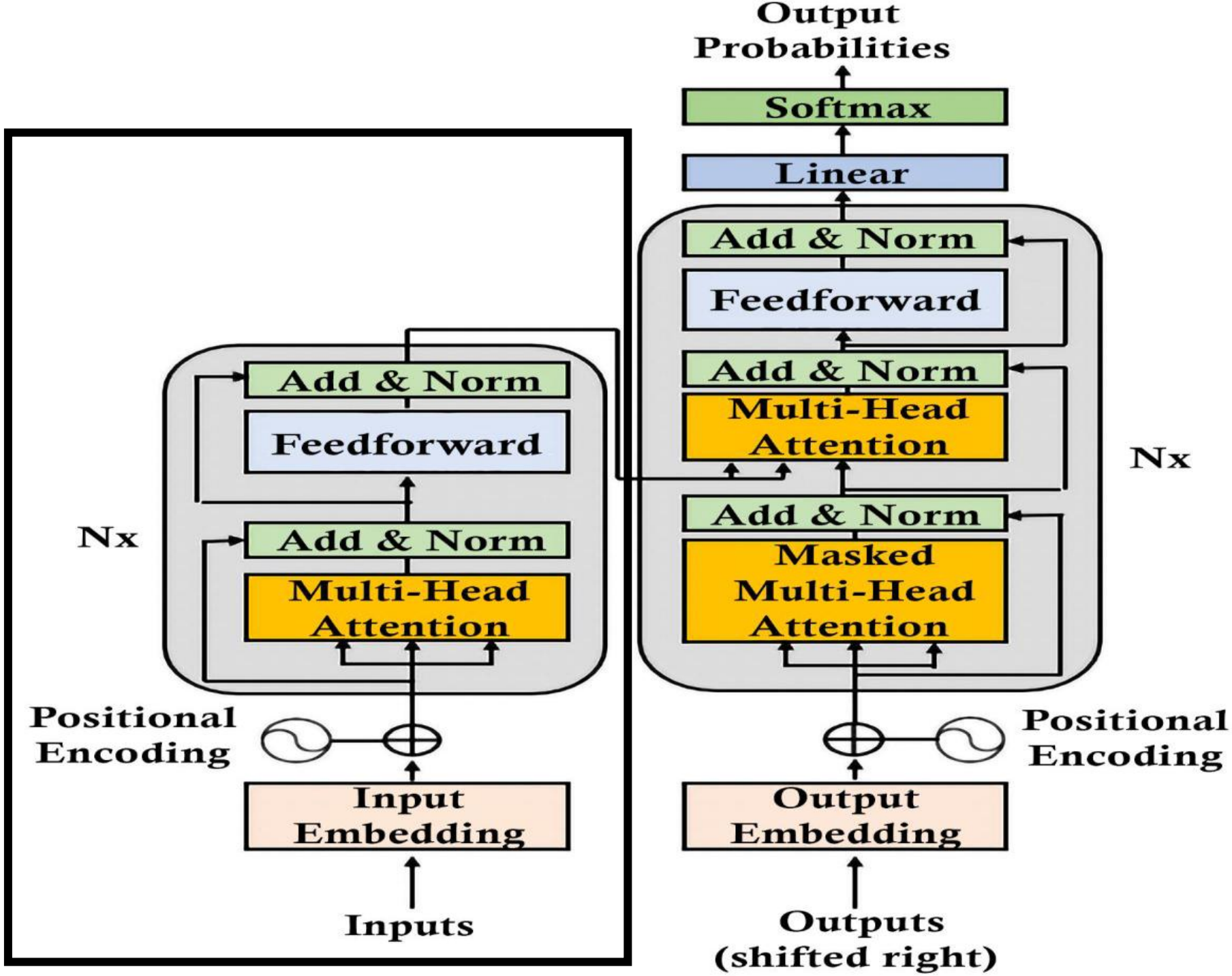
- En diciembre de 2017, Google Brain y Google Research publicaron el artículo de Vaswani et al., *Attention is All You Need*.
- El Transformer superó a los modelos de NLP existentes en ese momento.
- Entrenaba más rápido que las arquitecturas anteriores y obtenía mejores resultados de evaluación.
- La idea del "attention head" (cabeza de atención) del Transformer es eliminar las características de las redes neuronales recurrentes.

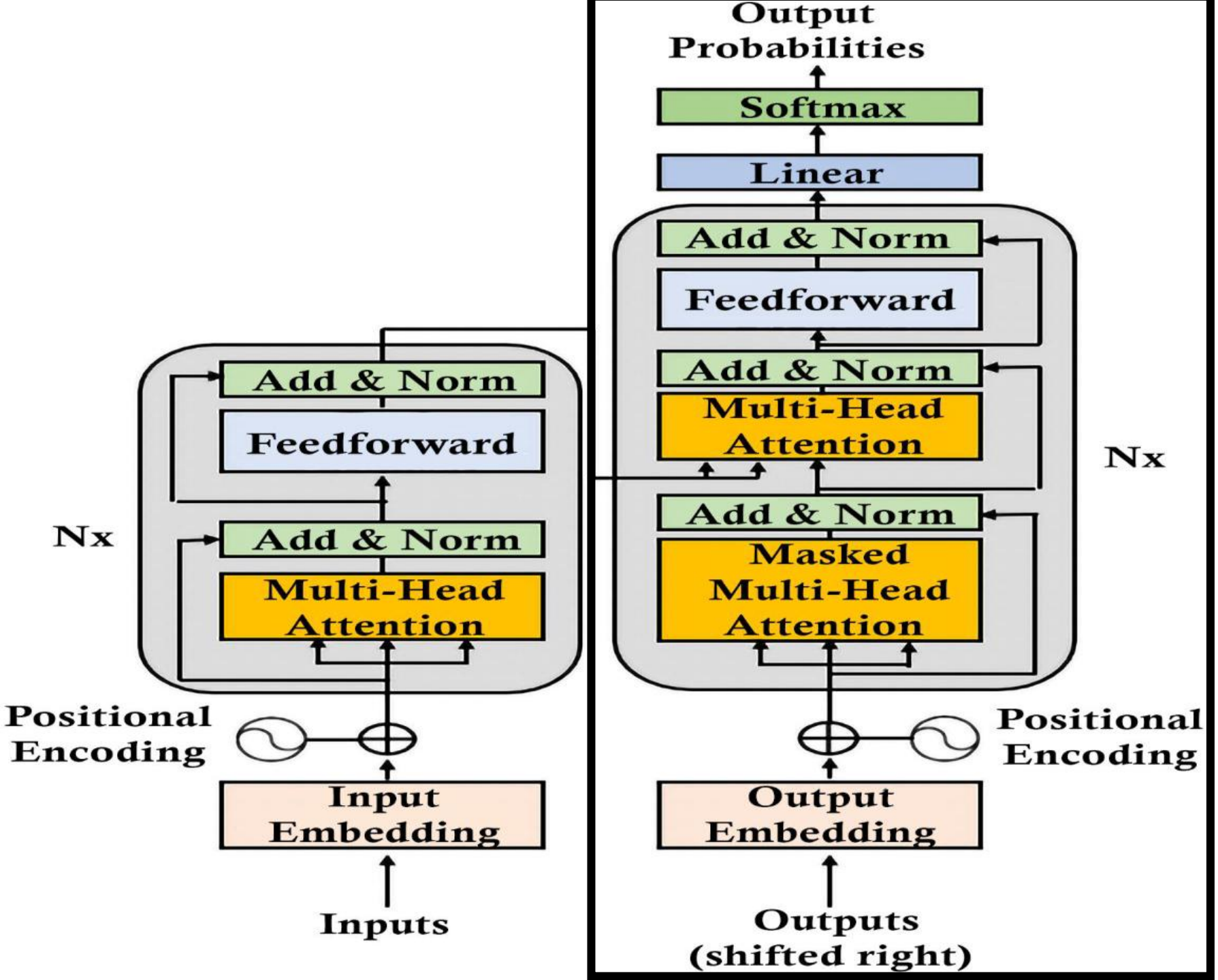
Comenzando con la arquitectura del modelo Transformer

- El modelo original de Transformer es una pila de 6 capas.
- La salida de la capa l es la entrada de la capa l+1 hasta que se alcanza la predicción final.
- Hay una pila de codificadores de 6 capas a la izquierda y una pila de decodificadores de 6 capas a la derecha.




Output Probabilities
Softmax
Linear
Add & Norm
Feedforward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feedforward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

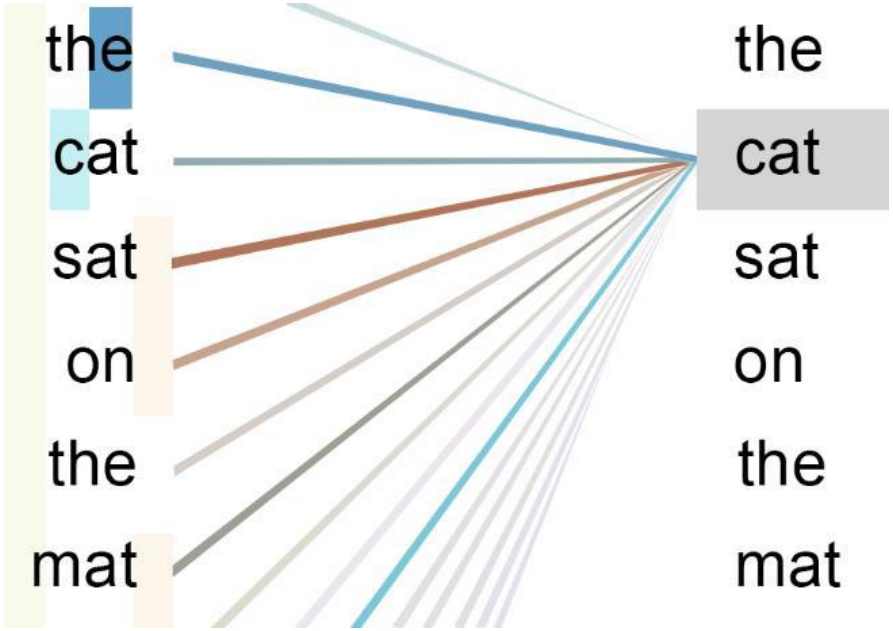

Output Probabilities
Softmax
Linear
Add & Norm
Feedforward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feedforward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)


Output Probabilities
Softmax
Linear
Add & Norm
Feedforward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feedforward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Comenzando con la arquitectura del modelo Transformer

```
The cat sat on the mat.
```

La atención realizará productos punto entre los vectores de palabras y determinará las relaciones más fuertes de una palabra con todas las demás palabras, incluida ella misma (“cat” y “cat”)



La pila del codificador

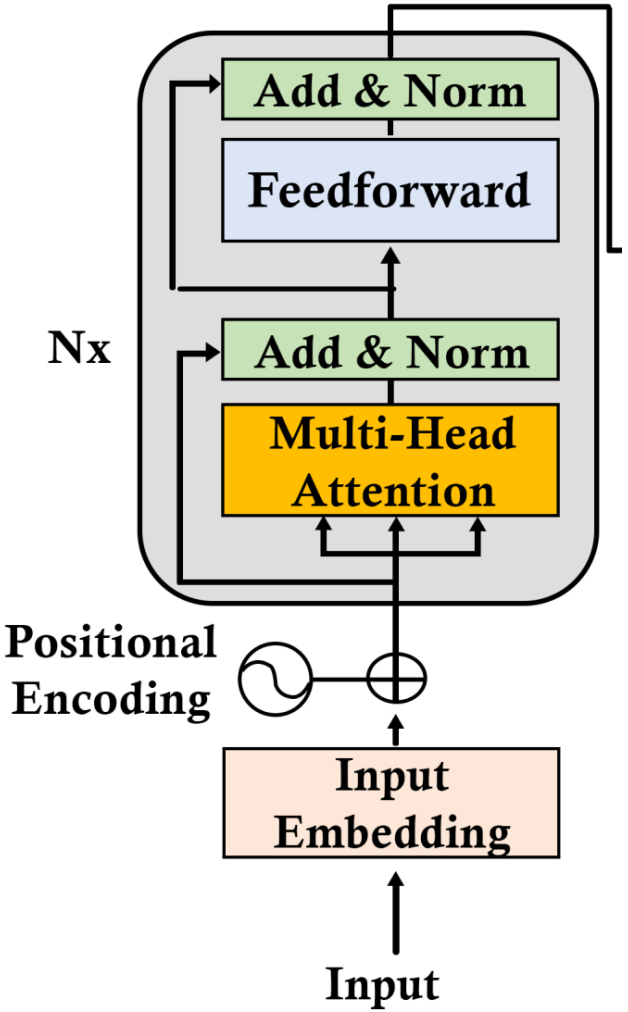
Las capas del codificador y decodificador del modelo Transformer original son pilas de capas.

Cada capa contiene dos subcapas principales: un mecanismo de atención de múltiples cabezas y una red completamente conectada de avance posicional.

El output normalizado de cada capa es así:

$$LayerNormalization\ (x + Sublayer(x))$$

Aunque la estructura de cada una de las N=6 capas del codificador es idéntica, el contenido de cada capa no es estrictamente idéntico al de la capa anterior.

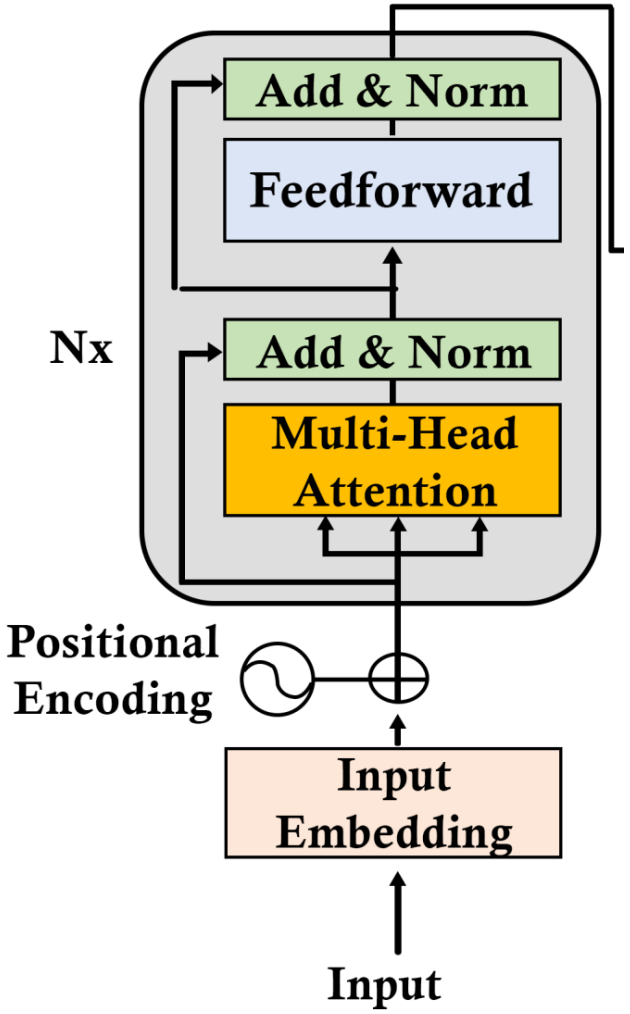


La pila del codificador

Los mecanismos de atención de múltiples cabezas realizan las mismas funciones de la capa 1 a la 6. Sin embargo, no realizan las mismas tareas.

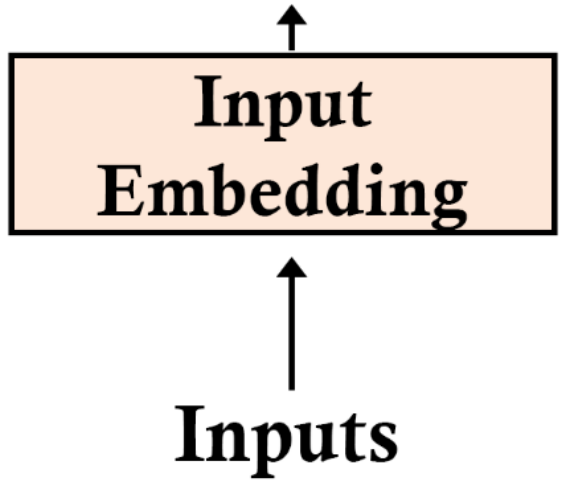
El output de cada subcapa del modelo tiene una dimensión constante, incluida la capa de embedding y las conexiones residuales.

Prácticamente todas las operaciones clave son productos punto. Como resultado, las dimensiones permanecen estables, lo que reduce el número de operaciones a calcular, reduce el consumo de la máquina y facilita el rastreo de la información a medida que fluye a través del modelo.



Embedding de entrada

La subcapa de embedding de entrada convierte los tokens de entrada en vectores de dimensión *dmodel = 512* usando embeddings aprendidos en el modelo Transformer original.



Cada tokenizador tiene sus métodos, como BPE (Byte-Pair encoding), *word piece* y *sentence piece*.

Por ejemplo, un tokenizador aplicado a la secuencia *the Transformer is an innovative NLP model!* producirá los siguientes tokens en un tipo de modelo:

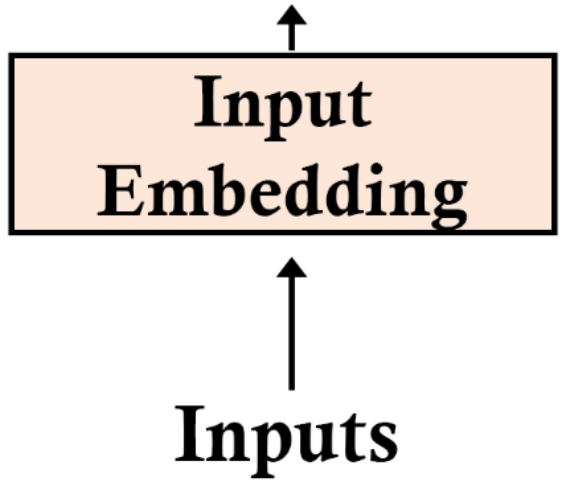
```
['the', 'transform', 'er', 'is', 'an', 'innovative', 'n', 'l', 'p',
'model', '!']
```

Embedding de entrada

Notarán que este tokenizador normalizó la cadena a minúsculas y la truncó en subpartes.

Un tokenizador generalmente proporcionará una representación entera que se usará para el proceso de embedding. Por ejemplo:

```
text = "The cat slept on the couch.It was too tired to get up."
tokenized text= [1996, 4937, 7771, 2006, 1996, 6411, 1012, 2009, 2001,
2205, 5458, 2000, 2131, 2039, 1012]
```

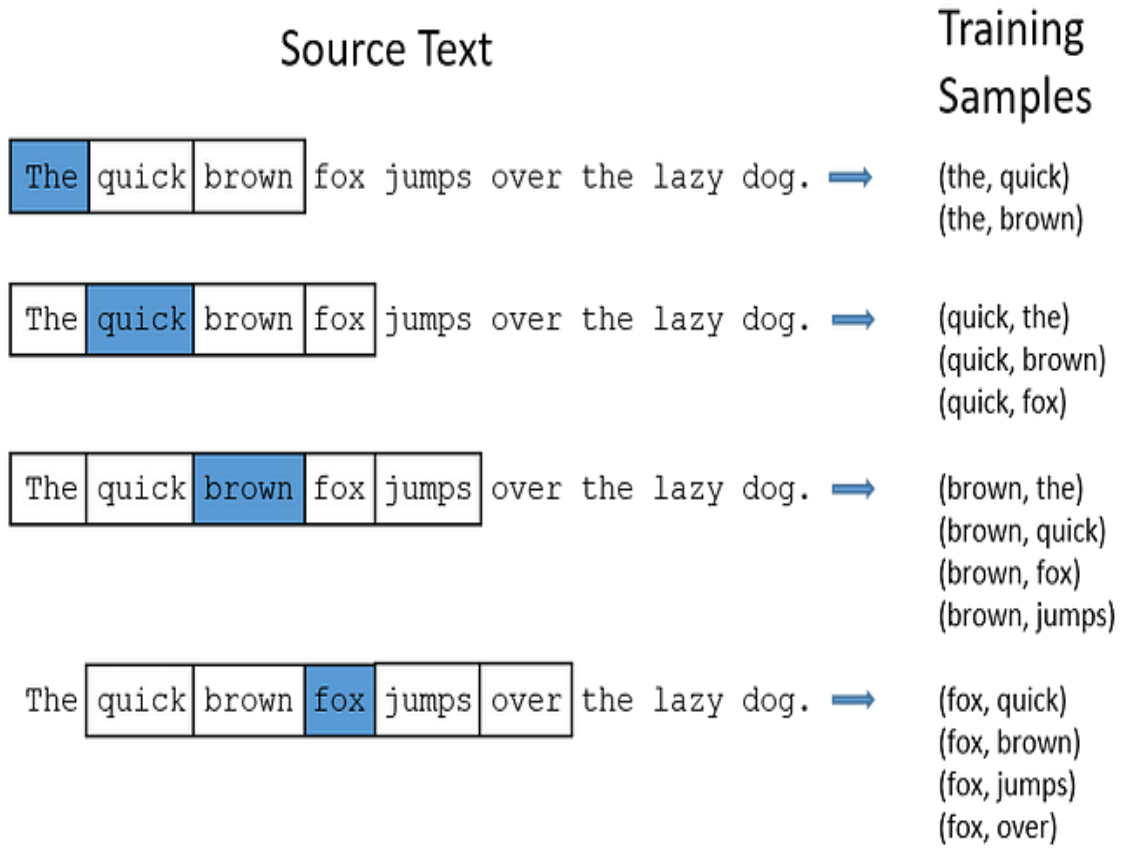


No hay suficiente información en el texto tokenizado en este punto para continuar. El texto tokenizado debe ser embebido.

Embedding de entrada

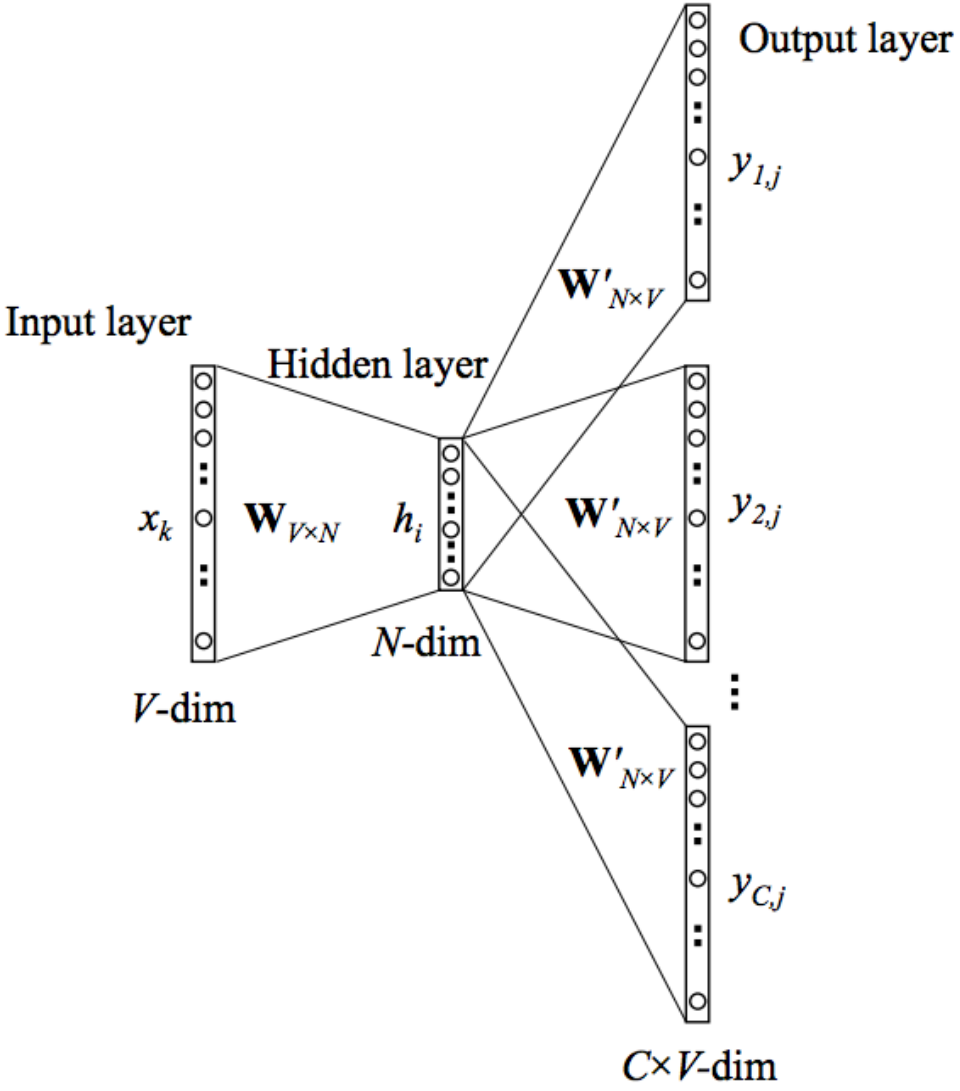
Un skip-gram se enfocará en una palabra central en una ventana de palabras y predecirá las palabras de contexto.

Por ejemplo, si word(i) es la palabra central en una ventana de dos pasos, un modelo skip-gram analizará word(i-2), word(i-1), word(i+1) y word(i+2). Luego la ventana se deslizará y repetirá el proceso.



Embedding de entrada

Un modelo skip-gram generalmente contiene una capa de entrada, pesos, una capa oculta y una salida que contiene los embeddings de palabras de las palabras tokenizadas de entrada.



Embedding de entrada

Supongamos que necesitamos realizar el embedding para la siguiente oración.

```
The black cat sat on the couch and the brown dog slept on the rug.
```

```
black=[[-0.01206071  0.11632373  0.06206119  0.01403395  0.09541149
0.10695464 0.02560172  0.00185677 -0.04284821  0.06146432  0.09466285
0.04642421 0.08680347  0.05684567 -0.00717266 -0.03163519  0.03292002
-0.11397766 0.01304929  0.01964396  0.01902409  0.02831945  0.05870414
0.03390711 -0.06204525  0.06173197 -0.08613958 -0.04654748  0.02728105
-0.07830904
    ...
0.04340003 -0.13192849 -0.00945092 -0.00835463 -0.06487109  0.05862355
-0.03407936 -0.00059001 -0.01640179  0.04123065
-0.04756588  0.08812257 0.00200338 -0.0931043  -0.03507337  0.02153351
-0.02621627 -0.02492662 -0.05771535 -0.01164199
-0.03879078 -0.05506947  0.01693138 -0.04124579 -0.03779858
-0.01950983 -0.05398201  0.07582296  0.00038318 -0.04639162
-0.06819214  0.01366171  0.01411388  0.00853774  0.02183574
-0.03016279 -0.03184025 -0.04273562]]
```

```
brown=[[ 1.35794589e-02 -2.18823571e-02  1.34526128e-02  6.74355254e-02
   1.04376070e-01  1.09921647e-02 -5.46298288e-02 -1.18385479e-02
   4.41223830e-02 -1.84863899e-02 -6.84073642e-02  3.21860164e-02
   4.09143828e-02 -2.74433400e-02 -2.47369967e-02  7.74542615e-02
   9.80964210e-03  2.94299088e-02  2.93895267e-02 -3.29437815e-02
...
  7.20389187e-02  1.57317147e-02 -3.10291946e-02 -5.51304631e-02
  -7.03861639e-02  7.40829483e-02  1.04319192e-02 -2.01565702e-03
   2.43322570e-02  1.92969330e-02  2.57341694e-02 -1.13280728e-01
   8.45847875e-02  4.90090018e-03  5.33546880e-02 -2.31553353e-02
   3.87288055e-05  3.31782512e-02 -4.00604047e-02 -1.02028981e-01
   3.49597558e-02 -1.71501152e-02  3.55573371e-02 -1.77437533e-02
  -5.94457164e-02  2.21221056e-02  9.73121971e-02 -4.90022525e-02]]
```

Embedding de entrada

Para verificar el *word embedding* producido para estas dos palabras, podemos usar la similitud del coseno (*cosine similarity*) para ver si los embeddings de las palabras *black* y *brown* son similares.

```
cosine_similarity(black, brown)= [[0.9998901]]
```

El modelo *skip-gram* produjo dos vectores que están cercanos entre sí. Detectó que *black y brown* forman un subconjunto de colores dentro del diccionario de palabras.

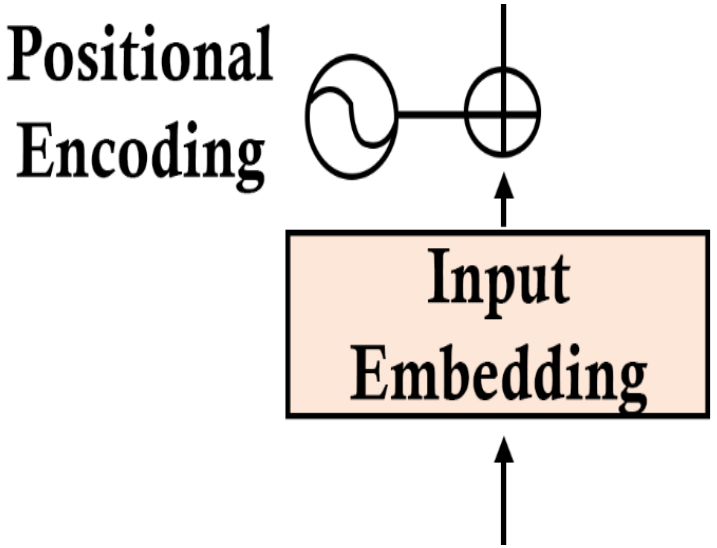
Codificación posicional

Entramos en esta función de codificación posicional del Transformer sin tener idea de la posición de una palabra en una secuencia.

La idea es agregar un valor de codificación posicional al *embedding* de entrada en lugar de tener vectores adicionales para describir la posición de un *token* en una secuencia.

El Transformer espera un tamaño fijo dmodel = 512 (u otro valor constante para el modelo) para cada vector de la salida de la función de codificación posicional.

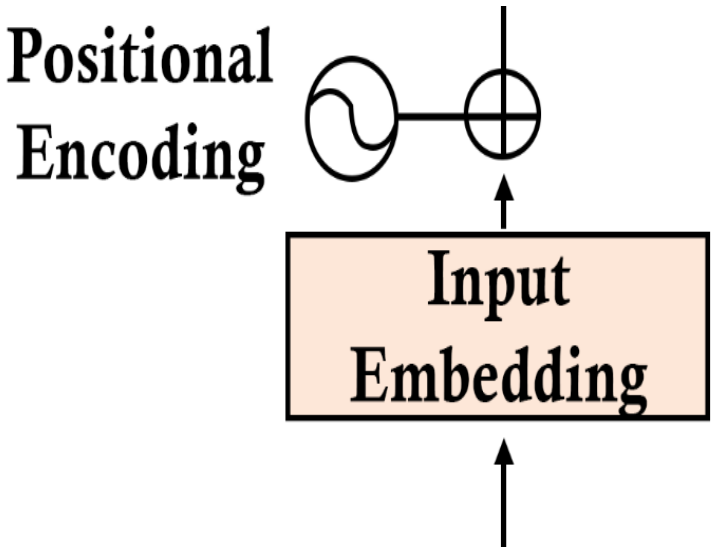
Si volvemos a la oración que usamos en la subcapa de *word embedding*, podemos ver que *black* y *brown* pueden ser semánticamente similares, pero están lejos entre sí en la oración:




```
The black cat sat on the couch and the brown dog slept on the rug.
```

Codificación posicional

Vaswani et al. (2017) proporcionan funciones de seno y coseno para que podamos generar diferentes frecuencias para la codificación posicional (PE) para cada posición y cada dimensión *i* del dmodel = 512 del vector de *word embedding:*

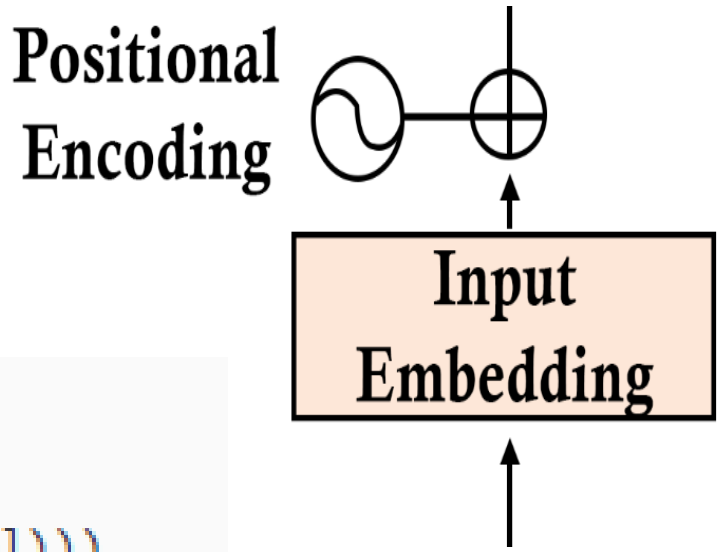


$$PE_{(pos\ 2i)} = sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE_{(pos\ 2i+1)} = cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

Codificación posicional

Utilizaremos las funciones de la manera en que fueron descritas por Vaswani et al. (2017). Una traducción literal al pseudo código de Python produce el siguiente código para un vector posicional pe[0][i] para una posición pos:




```
def positional_encoding(pos,pe):
for i in range(0, 512,2):
        pe[0][i] = math.sin(pos / (10000 ** ((2 * i)/d_model)))
        pe[0][i+1] = math.cos(pos / (10000 ** ((2 * i)/d_model)))
return pe
```

Codificación posicional

Si aplicamos las funciones seno y coseno literalmente para pos=2 y pos=10, obtenemos un vector de codificación posicional de tamaño=512:

```
PE(2)=
[[ 9.09297407e-01 -4.16146845e-01  9.58144367e-01 -2.86285430e-01
   9.87046242e-01 -1.60435960e-01  9.99164224e-01 -4.08766568e-02
   9.97479975e-01  7.09482506e-02  9.84703004e-01  1.74241230e-01
   9.63226616e-01  2.68690288e-01  9.35118318e-01  3.54335666e-01
   9.02130723e-01  4.31462824e-01  8.65725577e-01  5.00518918e-01
   8.27103794e-01  5.62049210e-01  7.87237823e-01  6.16649508e-01
   7.46903539e-01  6.64932430e-01  7.06710517e-01  7.07502782e-01
…
   5.47683925e-08  1.00000000e+00  5.09659337e-08  1.00000000e+00
   4.74274735e-08  1.00000000e+00  4.41346799e-08  1.00000000e+00
   4.10704999e-08  1.00000000e+00  3.82190599e-08  1.00000000e+00
   3.55655878e-08  1.00000000e+00  3.30963417e-08  1.00000000e+00
```

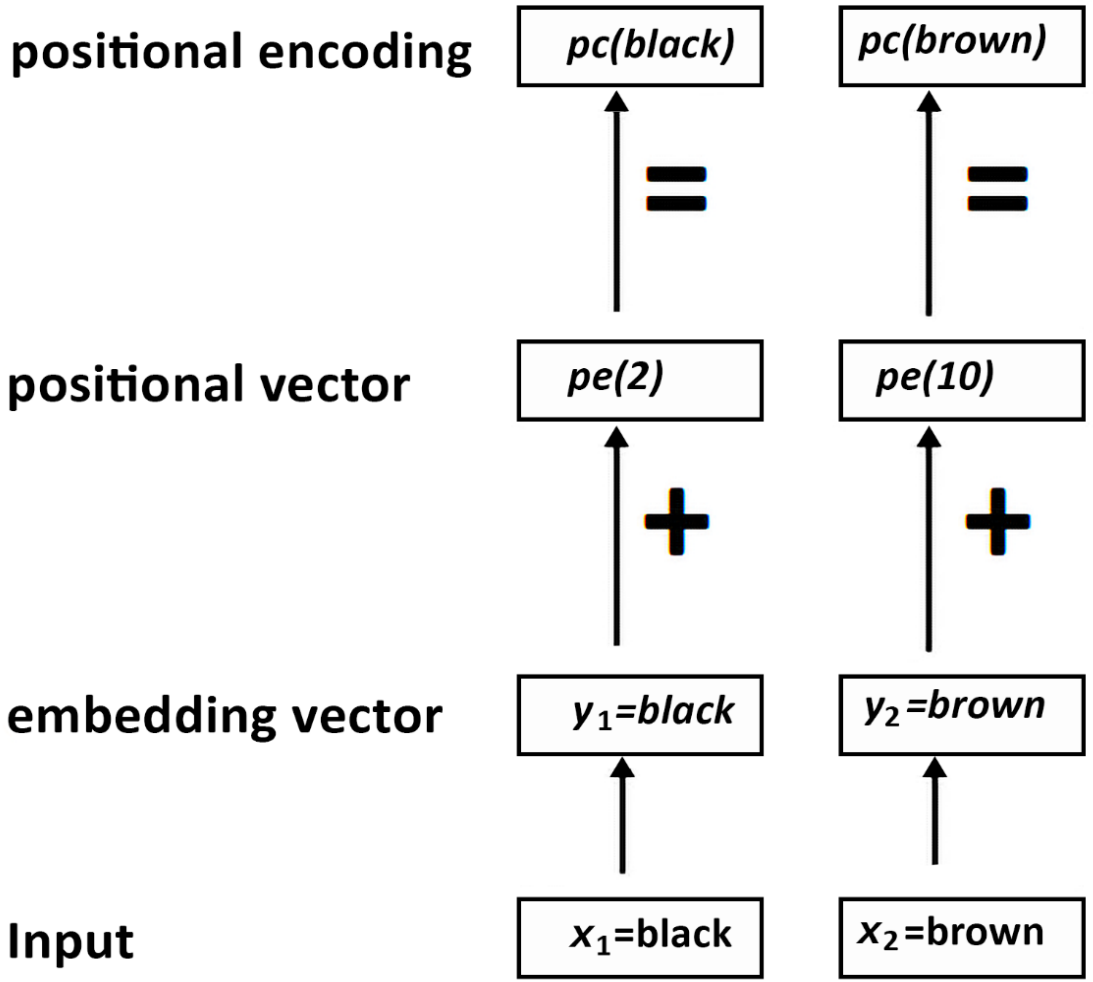
```
PE(10)=
[[-5.44021130e-01 -8.39071512e-01  1.18776485e-01 -9.92920995e-01
   6.92634165e-01 -7.21289039e-01  9.79174793e-01 -2.03019097e-01
   9.37632740e-01  3.47627431e-01  6.40478015e-01  7.67976522e-01
   2.09077001e-01  9.77899194e-01 -2.37917677e-01  9.71285343e-01
  -6.12936735e-01  7.90131986e-01 -8.67519796e-01  4.97402608e-01
  -9.87655997e-01  1.56638563e-01 -9.83699203e-01 -1.79821849e-01
…
   2.73841977e-07  1.00000000e+00  2.54829672e-07  1.00000000e+00
   2.37137371e-07  1.00000000e+00  2.20673414e-07  1.00000000e+00
   2.05352507e-07  1.00000000e+00  1.91095296e-07  1.00000000e+00
   1.77827943e-07  1.00000000e+00  1.65481708e-07  1.00000000e+00
   1.53992659e-07  1.00000000e+00  1.43301250e-07  1.00000000e+00
   1.33352145e-07  1.00000000e+00  1.24093773e-07  1.00000000e+00
   1.15478201e-07  1.00000000e+00  1.07460785e-07  1.00000000e+00]]
```

```
cosine_similarity(pos(2), pos(10))= [[0.8600013]] cosine_similarity(black, brown)= [[0.9998901]]
```

Añadiendo la codificación posicional al vector de incrustación (embedding)

Si volvemos y tomamos la incrustación de la palabra "black", por ejemplo, y la nombramos y1 = black, estamos listos para agregarla al vector posicional pe(2) que obtuvimos con las funciones de codificación posicional. Obtendremos la codificación posicional pc(black) de la palabra de entrada "black":

$$pc(black) = y_1 + pe(2)$$

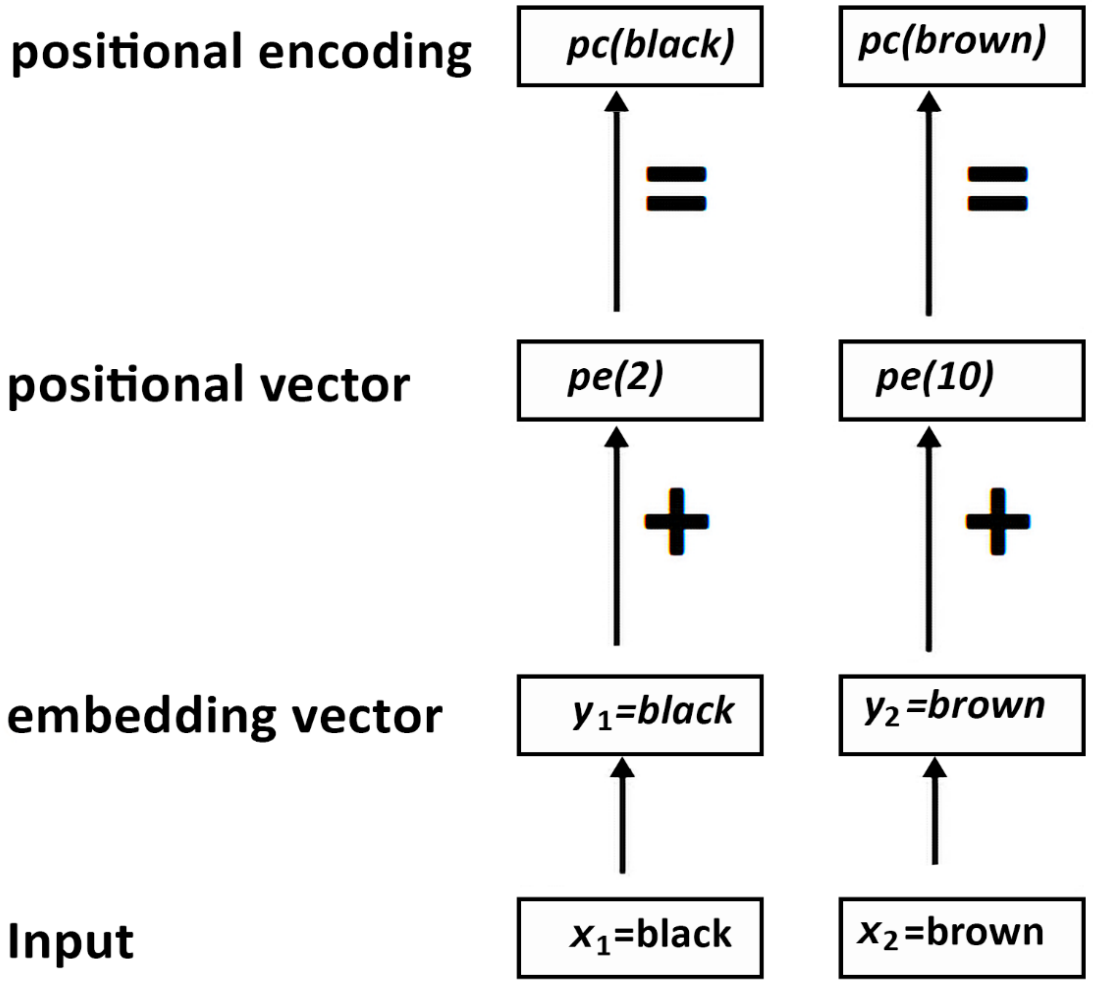


Añadiendo la codificación posicional al vector de incrustación (embedding)

Una de las muchas posibilidades es agregar un valor arbitrario a y1, la incrustación de palabras de "black":

$$y_1 * math.sqrt(d\_model)$$

Ahora podemos agregar el vector posicional al vector de incrustación de la palabra "black", que ambos tienen el mismo tamaño (512).



Añadiendo la codificación posicional al vector de incrustación (embedding)

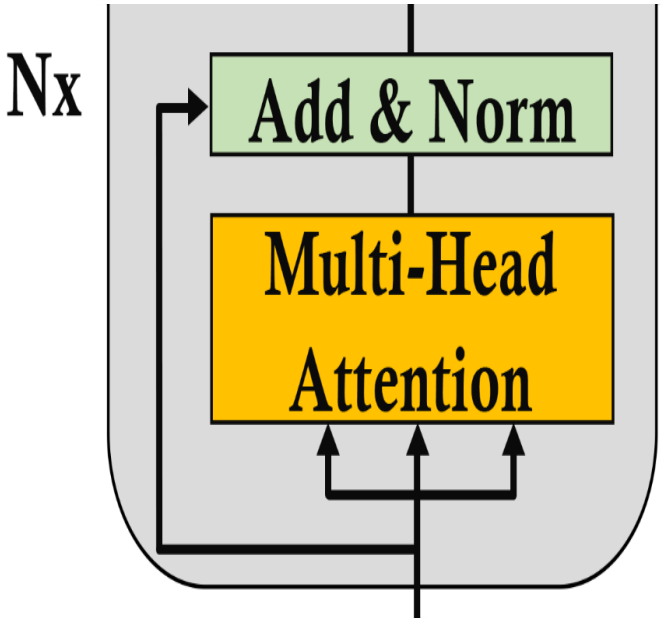
```
for i in range(0, 512,2):
            pe[0][i] = math.sin(pos / (10000 ** ((2 * i)/d_model)))
            pc[0][i] = (y[0][i]*math.sqrt(d_model))+ pe[0][i]

            pe[0][i+1] = math.cos(pos / (10000 ** ((2 * i)/d_model)))
            pc[0][i+1] = (y[0][i+1]*math.sqrt(d_model))+ pe[0][i+1]
```

```
pc(black)=
[[ 9.09297407e-01 -4.16146845e-01  9.58144367e-01 -2.86285430e-01
   9.87046242e-01 -1.60435960e-01  9.99164224e-01 -4.08766568e-02
   …
  4.74274735e-08  1.00000000e+00  4.41346799e-08  1.00000000e+00
   4.10704999e-08  1.00000000e+00  3.82190599e-08  1.00000000e+00
   2.66704294e-08  1.00000000e+00  2.48187551e-08  1.00000000e+00
   2.30956392e-08  1.00000000e+00  2.14921574e-08  1.00000000e+00]]
```

```
[[0.99987495]] word similarity
[[0.8600013]] positional encoding vector similarity
[[0.9627094]] final positional encoding similarity
```

Subcapa 1: Atención múltiple. La arquitectura de la atención múltiple

La subcapa de atención múltiple contiene ocho cabezas y está seguida por una normalización posterior de la capa, que añadirá conexiones residuales a la salida de la subcapa y la normalizará.



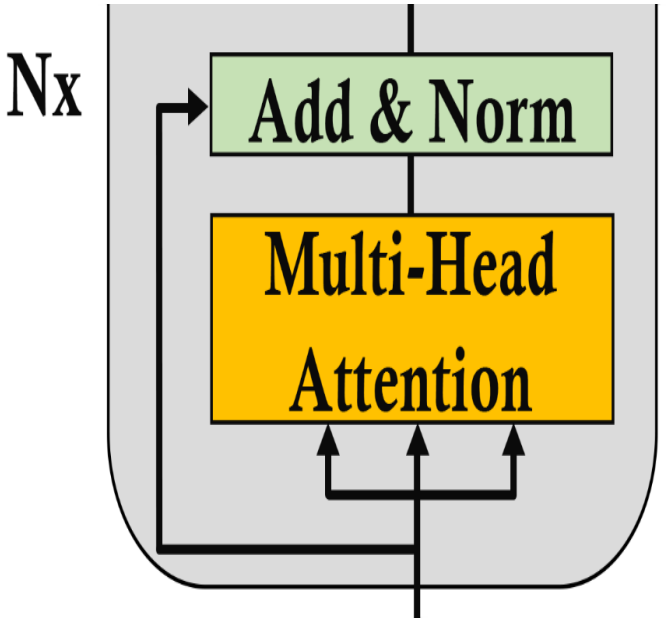
La entrada de la subcapa de atención múltiple de la primera capa de la pila del codificador es un vector que contiene la incrustación y la codificación posicional de cada palabra. Las siguientes capas de la pila no vuelven a comenzar estas operaciones.

Subcapa 1: Atención múltiple. La arquitectura de la atención múltiple

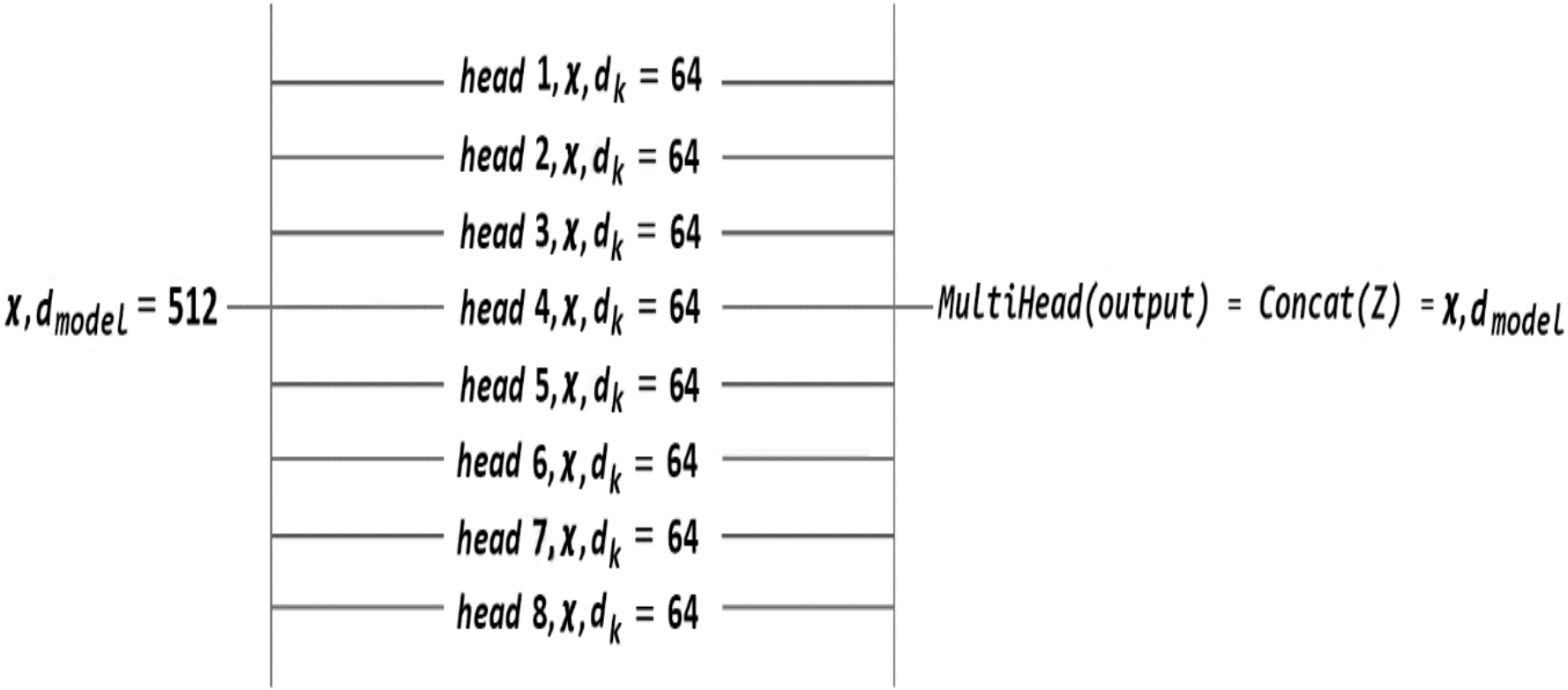
```
Sequence =The cat sat on the rug and it was dry-cleaned.
```

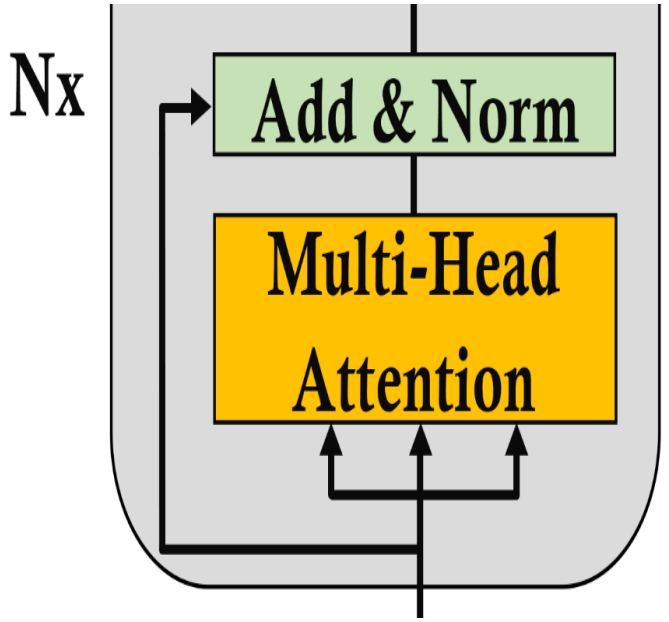
El modelo se entrenará para averiguar si está relacionada con gato o alfombra. Podríamos realizar un cálculo enorme entrenando el modelo utilizando las dimensiones dmodel = 512 tal como están ahora.

Una mejor manera es dividir las dimensiones dmodel = 512 de cada palabra Xn de x (todas las palabras de una secuencia) en 8 bloques de dimensiones dk = 64.



Subcapa 1: Atención múltiple. La arquitectura de la atención múltiple



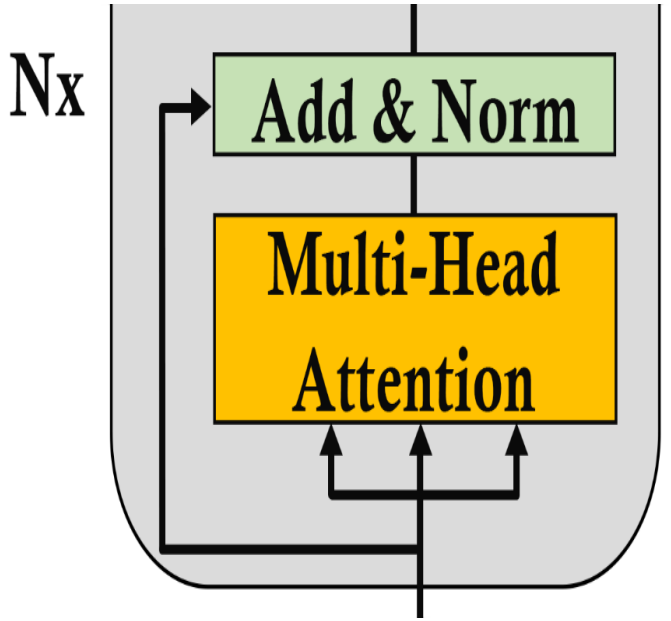


$$Z = (Z_0, Z_1, Z_2, Z_3, Z_4, Z_5, Z_6, Z_7) \qquad MultiHead(output) = Concat(Z_0, Z_1, Z_2, Z_3, Z_4, Z_5, Z_6, Z_7) = x, d_{model}$$

Subcapa 1: Atención múltiple. La arquitectura de la atención múltiple

La atención se define como “Atención de Producto Punto Escalado” (“Scaled Dot-Product Attention”), que se representa en la siguiente ecuación en la que se introducen Q, K y V:

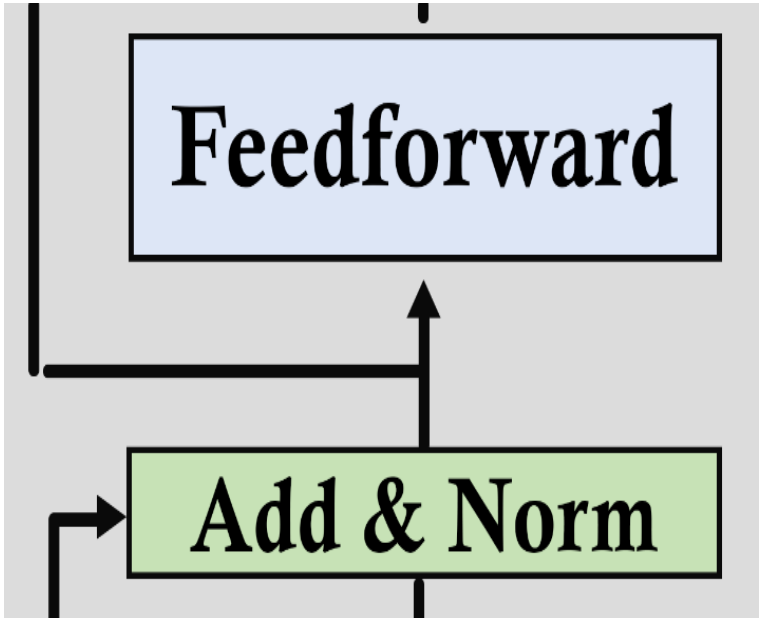
$$Attention(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = softmax\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\boldsymbol{T}}}{\sqrt{\boldsymbol{d_k}}}\right)\boldsymbol{V}$$



Sublayer 2: Red neuronal de avance (Feedforward network)

La subcapa FFN se puede describir de la siguiente manera:

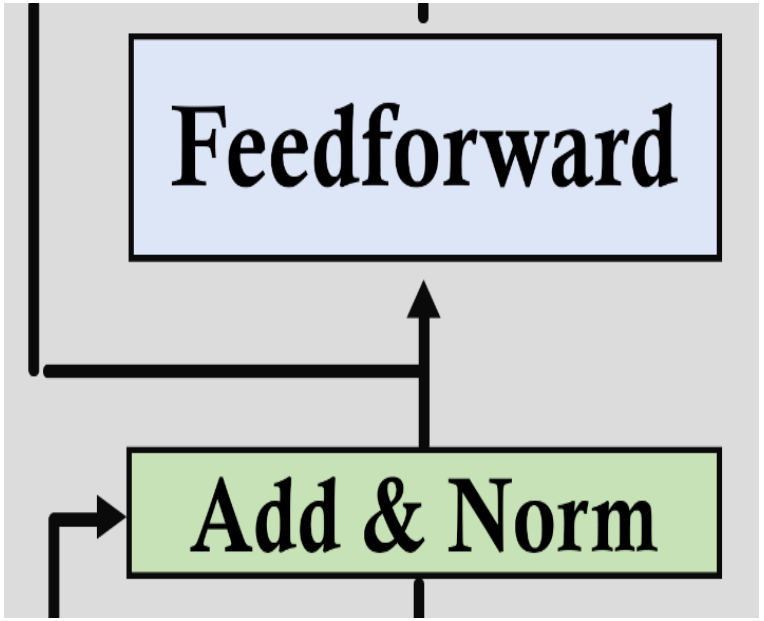
- Las FFN en el codificador y el decodificador son completamente conectadas.
- La FFN es una red por posición. Cada posición se procesa por separado y de manera idéntica.
- La FFN contiene dos capas y aplica una función de activación ReLU.
- La entrada y salida de las capas FFN es dmodel = 512, pero la capa interna es más grande con dff = 2048.
- La FFN se puede ver como realizando dos convoluciones con kernels de tamaño 1.



Sublayer 2: Red neuronal de avance (Feedforward network)

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

La salida de la FFN va al post-LN, como se describió anteriormente. Luego, la salida se envía a la siguiente capa de la pila del codificador y a la capa de atención multi-cabeza de la pila del decodificador.



La pila del decodificador

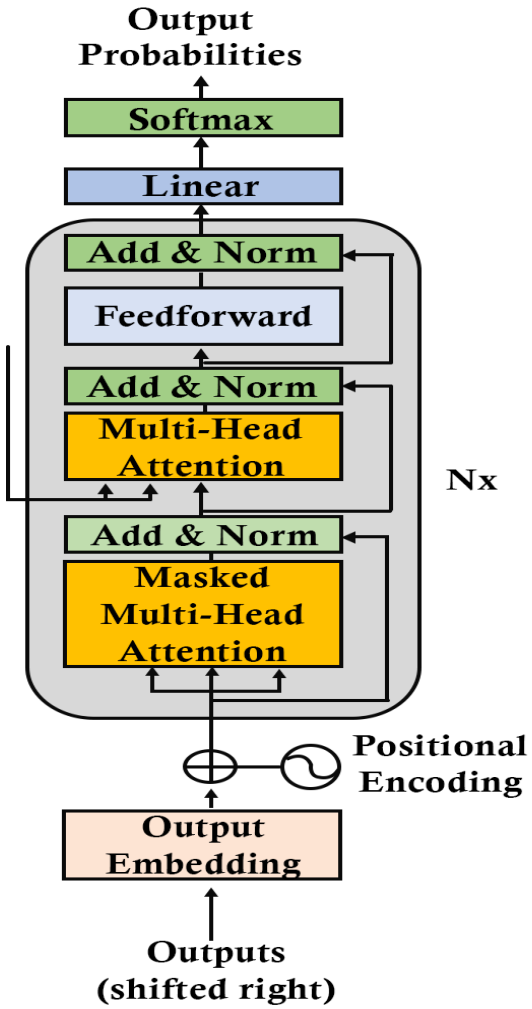
Las capas del decodificador del modelo Transformer son pilas de capas como las capas del codificador.

La estructura de la capa del decodificador permanece igual que la del codificador para todas las N=6 capas del modelo Transformer.

La subcapa FFN tiene la misma estructura que la FFN de la pila del codificador. La normalización de la capa posterior del FFN funciona igual que la normalización de la capa de la pila del codificador.

La capa lineal produce una secuencia de salida con una función lineal que varía según el modelo, pero se basa en el método estándar.

La capa lineal producirá así los siguientes elementos probables de una secuencia, que una función softmax convertirá en el elemento más probable.



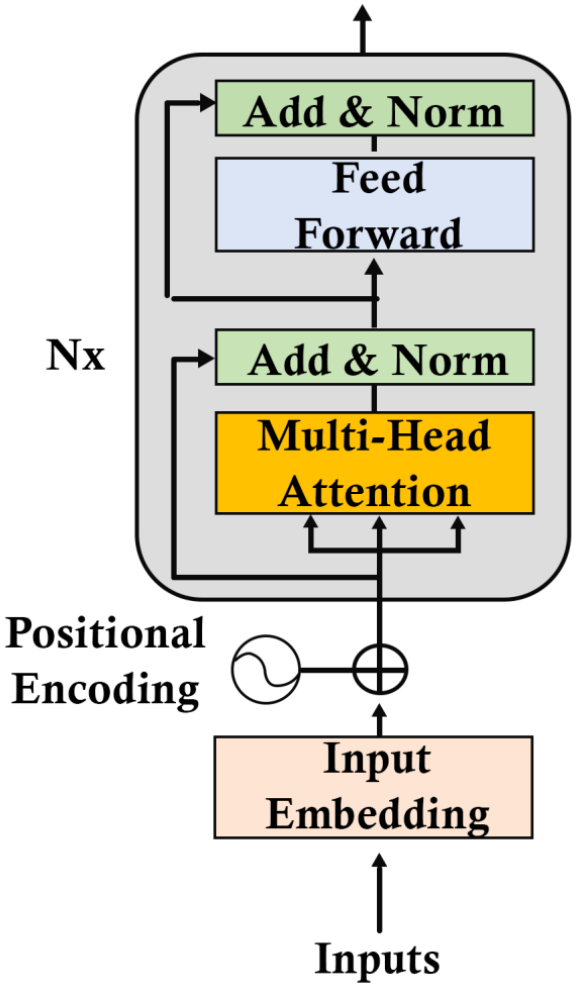
Ajuste fino de modelos BERT

La arquitectura de BERT. La pila del codificador

BERT introduce atención bidireccional en los modelos Transformer.

El modelo BERT no utiliza capas de decodificador.

Un modelo BERT tiene una pila de codificadores pero no tiene pilas de decodificadores.



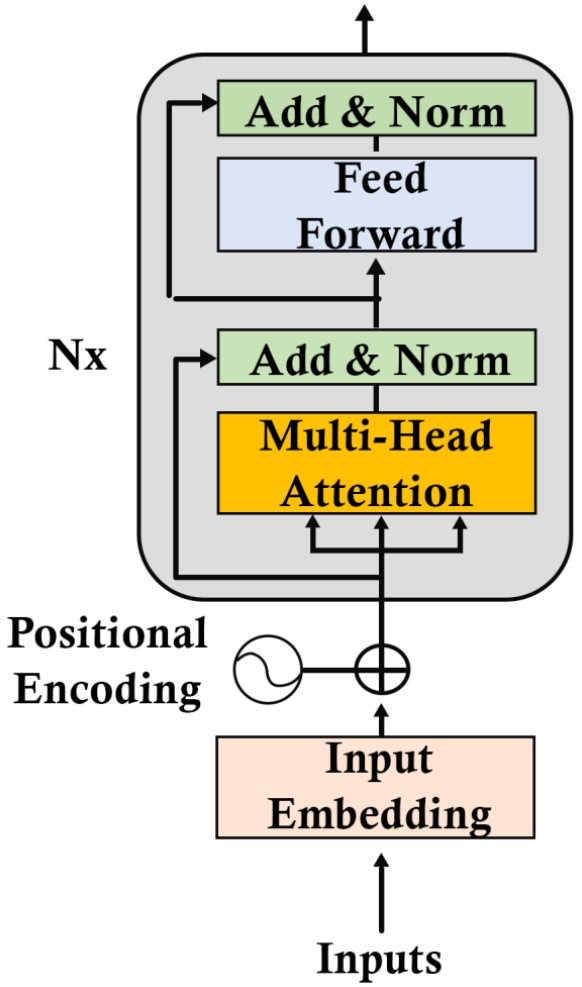
La arquitectura de BERT. La pila del codificador

Las capas del codificador de BERT son más grandes que las del modelo Transformer original.

Se pueden construir dos modelos BERT con las capas del codificador:

- **BERTBASE**, que contiene una pila de N=12 capas de codificador. dmodel = 768, y también puede expresarse como H=768, como en el artículo de BERT. Una subcapa de atención multi-cabeza contiene A=12 cabezas. La dimensión de cada cabeza ZA se mantiene en 64, como en el modelo Transformer original:

$$d_k = \frac{d_{model}}{A} = \frac{768}{12} = 64$$



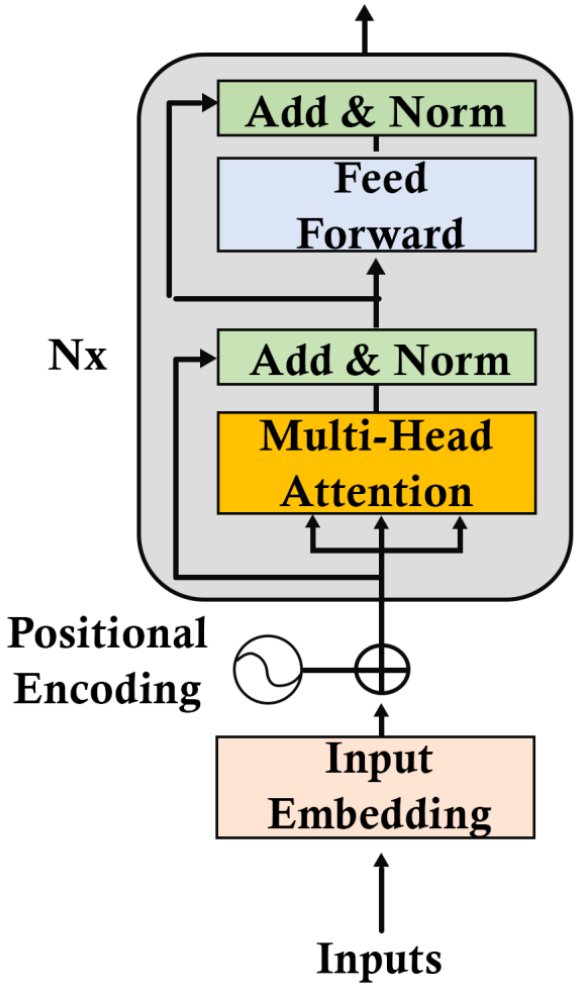
La arquitectura de BERT. La pila del codificador

- La salida de cada subcapa de atención multi-cabeza antes de la concatenación será la salida de las 12 cabezas:

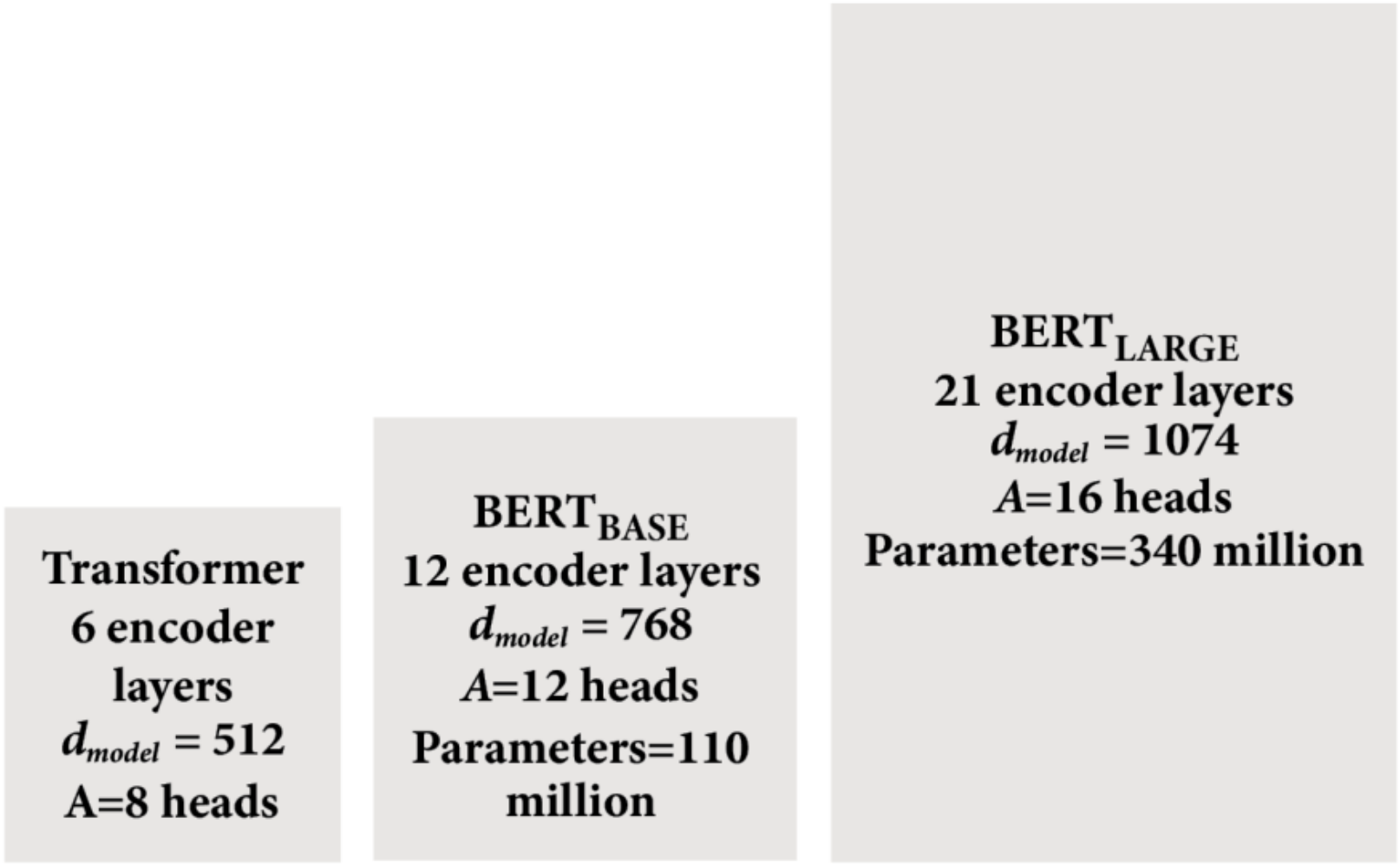
  output_multi-head_attention = {z0, z1, z2, …, z11}

- **BERTLARGE**, que contiene una pila de N=24 capas de codificador. dmodel = 1024. Una subcapa de atención multi-cabeza contiene A=16 cabezas. La dimensión de cada cabeza ZA también se mantiene en 64, como en el modelo Transformer original:

$$d_k = \frac{d_{model}}{A} = \frac{1024}{16} = 64$$



La arquitectura de BERT. La pila del codificador



Preparación del entorno de entrada para el preentrenamiento

Una capa de atención multi-cabeza enmascarada oculta todos los tokens que están más allá de la posición actual.

Por ejemplo, tomemos la siguiente oración:

```
The cat sat on it because it was a nice rug.
```

Si acabamos de llegar a la palabra "it", la entrada del codificador podría ser:

```
The cat sat on it<masked sequence>
```

El modelo se entrenó con dos tareas. El primer método es el Modelado de Lenguaje Enmascarado (MLM). El segundo método es la Predicción de la Siguiente Oración (NSP).

El modelado de lenguaje enmascarado

El modelado de lenguaje enmascarado no requiere entrenar un modelo con una secuencia de palabras visibles seguida de una secuencia enmascarada para predecir.

Una secuencia de entrada potencial podría ser:

```
The cat sat on it because it was a nice rug.
```

El decodificador enmascararía la secuencia de atención después de que el modelo llegara a la palabra "it":

```
The cat sat on it <masked sequence>.
```

Pero el codificador de BERT enmascara un token aleatorio para hacer una predicción:

```
The cat sat on it [MASK] it was a nice rug.
```

El modelado de lenguaje enmascarado

- Sorprender al modelo no enmascarando un solo token en el 10% del conjunto de datos; por ejemplo:

```
The cat sat on it [because] it was a nice rug.
```

- Sorprender al modelo reemplazando el token por un token aleatorio en el 10% del conjunto de datos; por ejemplo:

```
The cat sat on it [often] it was a nice rug.
```

- Reemplazar un token por un token [MASK] en el 80% del conjunto de datos; por ejemplo:

```
The cat sat on it [MASK] it was a nice rug.
```

Predicción de la siguiente oración

El segundo método para entrenar BERT es la Predicción de la Siguiente Oración (NSP).

La entrada contiene dos oraciones.

En el 50% de los casos, la segunda oración es la verdadera segunda oración de un documento.

En el 50% de los casos, la segunda oración fue seleccionada aleatoriamente y no tiene relación con la primera.

Predicción de la siguiente oración

Se añadieron dos nuevos tokens:

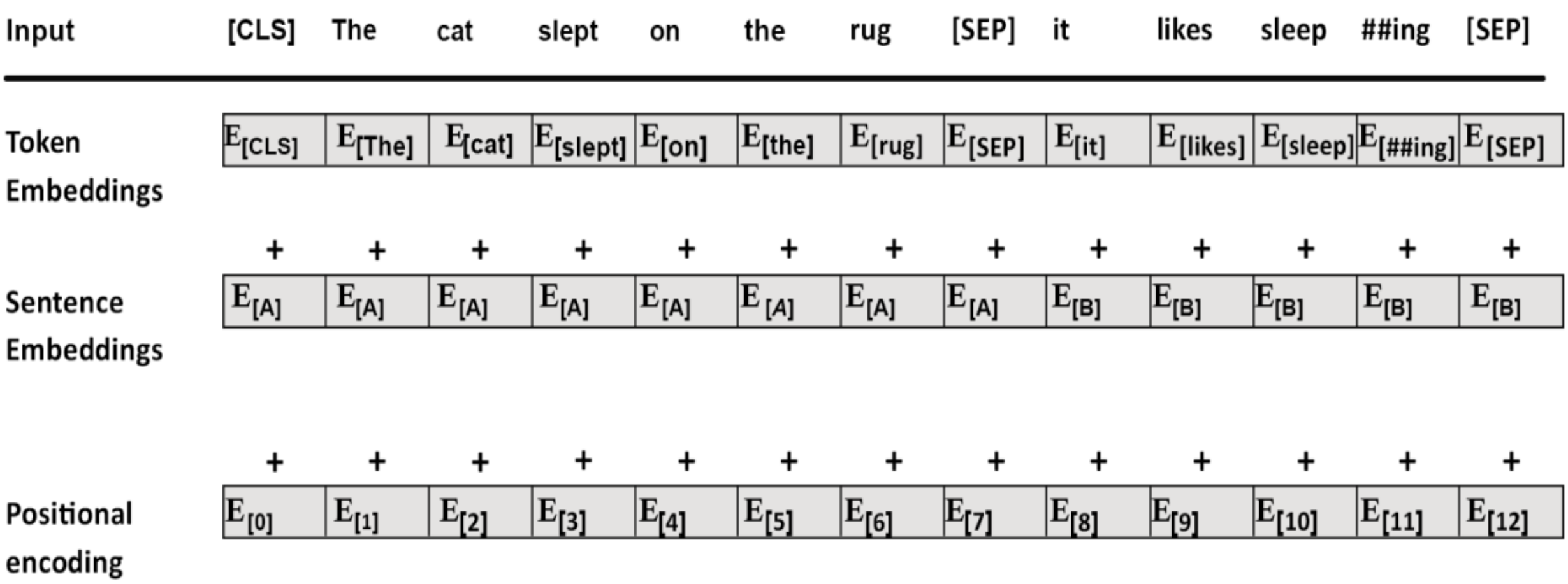
- [CLS] es un token de clasificación binaria añadido al inicio de la primera secuencia para predecir si la segunda secuencia sigue a la primera. Una muestra positiva generalmente es un par de oraciones consecutivas tomadas de un conjunto de datos. Una muestra negativa se crea utilizando secuencias de diferentes documentos.

- [SEP] es un token de separación que señala el final de una secuencia.

```
The cat slept on the rug. It likes sleeping all day.
```

```
[CLS] the cat slept on the rug [SEP] it likes sleep ##ing all day[SEP]
```

Predicción de la siguiente oración



La arquitectura de BERT

- Una secuencia de palabras se descompone en tokens de WordPiece.
- Un token [MASK] reemplazará aleatoriamente los tokens iniciales de las palabras para el entrenamiento de modelado de lenguaje enmascarado.
- Un token de clasificación [CLS] se inserta al principio de una secuencia con fines de clasificación.
- Un token [SEP] separa dos oraciones (segmentos, frases) para el entrenamiento de NSP.
- La incrustación de la oración se agrega a la incrustación de tokens, de modo que la oración A tenga un valor de incrustación de oración diferente al de la oración B.
- La codificación posicional se aprende. No se aplica el método de codificación posicional seno-coseno del Transformer original.

La arquitectura de BERT

Algunas características clave adicionales son:

- BERT utiliza atención bidireccional en sus subcapas de atención multi-cabeza, abriendo vastos horizontes de aprendizaje y comprensión de las relaciones entre tokens.
- BERT introduce escenarios de incrustación no supervisada, preentrenando modelos con texto no etiquetado. Los escenarios no supervisados obligan al modelo a pensar más durante el proceso de aprendizaje de atención multi-cabeza. Esto hace que BERT aprenda cómo se construyen los lenguajes y aplique este conocimiento a tareas posteriores sin necesidad de preentrenar cada vez.
- BERT también utiliza aprendizaje supervisado, cubriendo todos los aspectos en el proceso de preentrenamiento.

BERT:A Two-Step Framework

Step 1 Pretraining

A. Model

$BERT_{BASE}$
12 encoder layers
$d_{model} = 768$
A=12 heads
Parameters=110 million

B. Unlabeled Training Tasks:

- Masked LM.
- Next Sentence Prediction (NSP).

C. Training Data

- BookCorpus (800M words).
- English Wikipedia (2,500M words).

Step 2 Fine-Tuning

A. Initializing the Parameters

The parameters of the downstream model.

B. Labeled Training Tasks

The downstream models are all initialized with the pre-trained parameters.

C. Fine-Tuning

First, each downstream task is initialized with pre-trained parameters.

Then, each downstream task has separate fine-tuned models.

D. Step 2 Fine-Tuning Downstream Tasks

- Natural Language Understanding (GLUE).
- Question Answering (SQUAD v1.1, SQUaAD v2.0).
- Adversarial Generation sentence-pairs (SWAG).

Preentrenando un modelo RoBERTa desde cero

Entrenando un tokenizador y preentrenando un transformador

- Entrenaremos un modelo transformador llamado KantaiBERT utilizando los bloques de construcción proporcionados por Hugging Face para modelos similares a BERT.
- KantaiBERT es un modelo similar a RoBERTa (Enfoque de Preentrenamiento BERT Optimizado de Manera Robusta) basado en la arquitectura de BERT.
- No utiliza la tokenización WordPiece, sino que recurre a la codificación Byte-Pair Encoding (BPE) a nivel de byte. Este método allanó el camino para una amplia variedad de modelos BERT y modelos similares a BERT.
- KantaiBERT, al igual que BERT, será entrenado utilizando el Modelo de Lenguaje Máscara (MLM).

Referencias

- *Bommansani* et al. 2021, *On the Opportunities and Risks of Foundation Models*, https://arxiv.org/abs/2108.07258
- *Chen* et al.,2021, *Evaluating Large Language Models Trained on Code*, https://arxiv.org/abs/2107.03374
- Microsoft AI: https://innovation.microsoft.com/en-us/ai-at-scale
- OpenAI: https://openai.com/
- Google AI: https://ai.google/
- Google Trax: https://github.com/google/trax
- AllenNLP: https://allennlp.org/
- Hugging Face: https://huggingface.co/
- Scikit-learn: https://scikit-learn.org/stable/modules/metrics.html#cosine-similarity

Referencias

- *Ashish Vaswani*, *Noam Shazeer*, *Niki Parmar*, *Jakob Uszkoreit*, *Llion Jones*, *Aidan N. Gomez*, *Lukasz Kaiser*, *Illia Polosukhin*, 2017, *Attention Is All You Need*: https://arxiv.org/ abs/1706.03762
- *Jacob Devlin*, *Ming-Wei Chang*, *Kenton Lee*, and *Kristina Toutanova*, *2018*, *BERT: Pretraining of Deep Bidirectional Transformers for Language Understanding*: https://arxiv.org/ abs/1810.04805
- *Alex Warstadt*, *Amanpreet Singh*, and *Samuel R. Bowman*, 2018, *Neural Network Acceptability Judgments*: https://arxiv.org/abs/1805.12471
- The **Corpus of Linguistic Acceptability** (**CoLA**): https://nyu-mll.github.io/CoLA/

Documentation on Hugging Face models:

- https://huggingface.co/transformers/pretrained_models.html
- https://huggingface.co/transformers/model_doc/bert.html
- https://huggingface.co/transformers/model_doc/roberta.html
- https://huggingface.co/transformers/model_doc/distilbert.html

Referencias

- *Yinhan Liu*, *Myle Ott*, *Naman Goyal*, *Jingfei Du*, *Mandar Joshi*, *Danqi Chen*, *Omer Levy*, *Mike Lewis*, *Luke Zettlemoyer*, and *Veselin Stoyano*, 2019, *RoBERTa: A Robustly Optimized BERT Pretraining Approach*: https://arxiv.org/abs/1907.11692
- Hugging Face Tokenizer documentation: https://huggingface.co/transformers/main_classes/tokenizer.html?highlight=tokenizer
- The Hugging Face reference notebook: https://colab.research.google.com/github/huggingface/blog/blob/master/notebooks/01_how_to_train.ipynb

Transformers for Natural Language Processing

Build, train, and fine-tune deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, and GPT-3