Modelos de Transformers

Víctor Mijangos

Facultad de Ciencias

Lingüística computacional

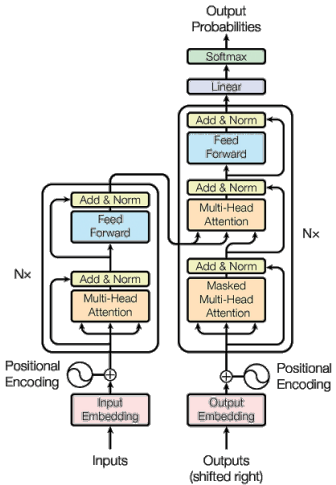
Motivación de los Transformers

Los modelos de atención en RNN mostraron ser una herramienta poderosa. Pero su entrenamiento es costoso, pues las celdas recurrentes **no pueden paralelizarse**.

Por esto, Vaswani et al. (2017) [4] proponen el modelo de **Transformer**, que elimina las celdas recurrentes, pero conserva la atención.

Los Transformers pueden entrenarse de manera paralela, permitiendo crear modelos de mayor capacidad.

Arquitectura de transformers



Los transformers se componen de dos módulos [4]:

- **Encoder:** Contiene a) *embeddings* y de *positional encoding*; b) una sub-capa de *self-attention*; c) sub-capa de suma y normalización; d)sub-capa *FeedForward* con RELu; e) sub-capa de suma y normalización.
- **Decoder:** Cuenta con a) *embeddings* y *encoding de posición*; b) sub-capa de self-attention; c) sub-capa de suma y normalización; d) sub-capa de atención; e) sub-capa de suma y normalización; f) sub-capa *FeedForward*; g) sub-capa de suma y normalización.

La **salida** es una capa lineal con activación Softmax.

Embeddings

La entrada tanto del encoder como del decoder son **embeddings estáticos**. Esto es, un mapeo:

$$w^{(1)}, w^{(2)}, ..., w^{(T)} \mapsto C_{w^{(1)}}, C_{w^{(2)}}, ..., C_{w^{(T)}}$$

Donde $C_{w^{(t)}} \in \mathbb{R}^d$ es vector embedding del token $w^{(t)}$. A d se le llama **dimensión del modelo**.

La representación de entrada se puede escribir como una matriz C que contiene como renglones estos embeddings:

$$\sqrt{d}C = \sqrt{d} \begin{pmatrix} -C_1- \\ -C_2- \\ \vdots \\ -C_T- \end{pmatrix}$$

El factor $\sqrt{d}$ escala los embeddings de entrada.

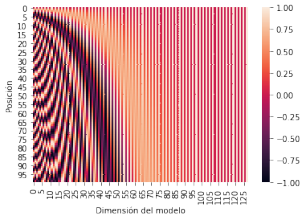
Codificación posicional

Para convertir las representaciones estáticas en representaciones dinámicas, se utiliza la **codificación posicional**: un vector que representa la posición en la que se encuentra el token de la siguiente forma:

$$PE_{2i}(t) = \sin\left(\frac{t}{10000^{2i/d}}\right)$$

$$PE_{2i+1}(t) = \cos\left(\frac{t}{10000^{2i/d}}\right)$$

Done t es la posición, d la dimensión del modelo.



Representación de entrada

La entrada tanto del encoder como del encoder será una matriz X cuyos renglones son las representaciones de cada uno de los tokens de la cadena de entrada:

$$X = \begin{pmatrix} -x_1- \\ -x_2- \\ \vdots \\ -x_T- \end{pmatrix} = \sqrt{d} \begin{pmatrix} -C_1- \\ -C_2- \\ \vdots \\ -C_T- \end{pmatrix} + \begin{pmatrix} -PE(1)- \\ -PE(2)- \\ \vdots \\ -PE(T)- \end{pmatrix}$$

Los vectores $PE(t)$ son la codificación posicional, cuyas entradas están definidas por las funciones anteriores.

Tipos de atención entransformers

Los transformers utilizan diferentes tipos de atención para procesar los datos. Precisamente en la atención radica su innovación. Tenemos tres tipos de atención en los modelos de Transformers:

1. Self-attention o auto-atención: Se enfoca en encontrar pesos de atención entre elmenots de **una misma cadena**.
2. Masked self-attention (auto-atención enmascarada): Igual que la self-attention, pero **enmascara** elementos de la cadena de entrada.
3. Attention: La atención típica que calcula los pesos entre una cadena **de entrada y una de salida**.

Self-attention

Los mecanismos de **self-attention** capturan las dependencias entre las entradas y salidas de una red. En este tipo de atención, los inputs interactúan entre sí para buscar a que poner atención.

Dado un conjunto de entradas, por ejemplo:

$$x^{(t-1)}, x^{(t)}, x^{(t+1)}$$

Cada elemento produce tres representaciones que se obtienen al multiplicar por matrices de parámetros:

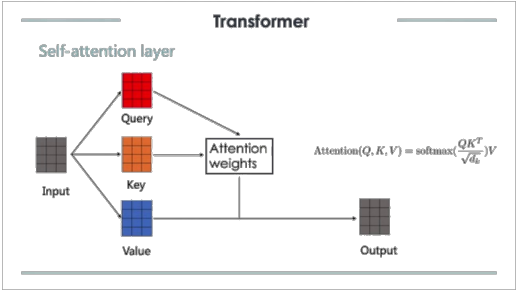
1. Key: $K = \left(K^T x^{(t)}\right)_t$
2. Query: $Q = \left(Q^T x^{(t)}\right)_t$
3. Value: $V = \left(V^T x^{(t)}\right)_t$

Self-attention

Una vez que se han obtenido los valores de Key, Query y Value se aplica el mecanismo de atención (que arroja una matriz de atención):

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

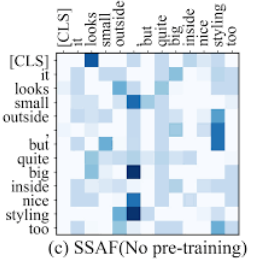
d_k es la dimensión de las matrices.



Self-attention

El mecanismo de self-attention es similar al de la attention en RNNs; su característica es que compara los tokens dentro de una misma cadena y no entre cadenas diferentes. Los pesos de atención son:

$$\alpha_{i,j} = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)_{i,j}$$



(c) SSAF(No pre-training)

Múltiples cabezales

El modelo de Transformer introduce los cabezales (*heads*), que son copias de las sub-capas de atentión y self-attention. Cada cabezal es un mecanismo de atención ponderado por pesos particulares:

$$head_i = Attention(QW^{(Q,i)}, KW^{(K,i)}, VW^{(V,i)})$$

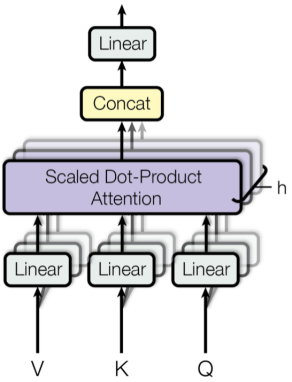
Para obtener la información obtenida por todos los cabezales, se concatenan y se reducen a una matriz con las dimensiones originales:

$$MultiHead(Q, K, V) = [head_1; ...; head_h]W^o \tag{2}$$

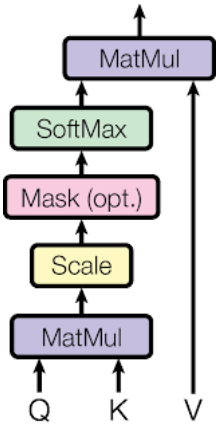
Tal que W^o es una matriz de $h \cdot d_v \times d$.

Mecanismo de MultiHead Attention

Visualmente, el mecanismo de atención multi-cabezal se ve como:



Donde cada cabezal de self-attention o atención es de la forma:



Masked self-attention

En el caso del decoder, que trabaja con la **cadena de salida**, la self-attetnion no se puede aplicar directamente, pues no se conocen los elementos de la cadena.

Para lidiar con esto, se aplica un **enmascaramiento**, que oculta los tokens de la cadena. Se tienen cadenas de la forma:

$$< BOS > w^{(1)} < MASK >< MASK > ...$$

La atención se calcula sobre estas etiquetas para predecir las palabras que ocupan ciertas posiciones.

Attention

En el decoder, se implementa la sub-capa de **atención** entre la cadena de entrada y la de salida. El computo es idéntico al que hemos definido, pero las representaciones K, V y Q cumplen:

- K y V provienen del encoder (son las representaciones de salida de éste)
- Q proviene del decoder, es la representación que proviene de la sub-capa de self-attention anterior.

Por tanto, la ecuación de atención puede quedar como:

$$Attention(Q_{dec}, K_{enc}, V_{enc}) = Softmax\left(\frac{Q_{dec}K_{enc}^T}{\sqrt{d_k}}\right)V_{enc} \quad (3)$$

Normalización

Los modelos de Transformer aplican una conexión residual y normalización entre cada una de las sub-capas.

Esta normalización está basada en Lei Ba et al (2016) [3]. La idea es que los datos tengan una distribución normal estándar:

$$X \sim \mathcal{N}(0, 1)$$

Este tipo de normalización es común en muchas tareas de aprendizaje automático.

Normalización

Para obtener esta normalización, se debe calcular la **media** y la **desviación estándar**.

Para obtener la media tenemos:

$$\hat{\mu} = \frac{1}{|X|} \sum_{x \in X} x$$

Para obtener la desviación estándar usamos la fórmula:

$$\hat{\sigma}^2 = \frac{1}{|X|} \sum_{x \in X} (x - \hat{\mu})^2$$

Suma y normalización

La normalización se obtiene como (a y b son hiperparámetros):

$$LayerNorm(x) = a \odot \frac{x - \hat{\mu}}{\hat{\sigma} + \epsilon} + b \tag{4}$$

En cada sub-capa se aplica entonces la suma y normalización de la siguiente forma:

$$SubLayer(LayerNorm(x)) + x \tag{5}$$

La suma y normalización se aplica entre cada sub-capa tanto en el encoder como en el decoder.

FeedForward

Una vez de aplicar la self-attention en el encoder y la attention en el decoder, se aplica en ambos una sub-capa FeedForward. Esta capa busca:

- Procesar las representaciones obtenidas de los mecanismos de (self-)attention.
- Garantizar la propiedad de aproximador universal de la red.

Estas capas se definen como:

$$\begin{aligned} FFNN(x) &= W^2 \,\text{máx}\{0, W^1 x + b^1\} + b^2 \\ &= W^2 ReLU(W^1 x + b^1) + b^2 \end{aligned}$$

Salida del Transformer

Finalmente, una vez procesado los datos tanto por el encoder como por el decoder, las representaciones finales (salidas del decoder) pasan por una capa lineal y una activación Softmax. Se obtienen las probabilidades:

$$p = Softmax\big(WDecoder\big(Encoder(x)\big) + b\big) \tag{6}$$

Para tareas de clasificación, se puede utilizar:

$$\hat{y} = \arg\max_i p_i$$

Optimización en Transformers

La optimización en los Transformars hace uso del **Optimizador Noam**, este se basa en Adam. Este último se define como:

$$\theta_i \leftarrow \theta_i - \frac{\eta}{\sqrt{\hat{\nu}} + \epsilon}\hat{m} \tag{7}$$

Donde, con $\beta_1, \beta_2 \in [0, 1]$

$$\hat{\nu} = \frac{\nu}{1 - \beta_2}$$
$$\hat{m} = \frac{m}{1 - \beta_1}$$

Y la regla de actualización es:

$$\nu \leftarrow \beta_2 \nu + (1 - \beta_2)\left(\nabla_{\theta_i} R(\theta)\right)^2$$
$$m \leftarrow \beta_1 m + (1 - \beta_1)\nabla_{\theta_i} R(\theta)$$

Optimizador Noam

Además de utilizar Adam para la optimización, los modelos de Transformers utilizan el optimizador Noam, que ajusta el rango de aprendizaje en cada paso r de acuerdo a la regla:

$$\eta_r = \frac{a}{d} \min\{r^{-1/2}, r\omega^{-3/2}\} \tag{8}$$

Donde se tienen los hiperparámetros:

- a, factor de escala; generalmente $a = 2$.
- d, la diemsión del modelo.
- ω, factor de *warmup*; generalmente $\omega = 4000$.

Además, se suele utilizar $\beta_1 = 0{,}9$ y $\beta_2 = 0{,}98$.

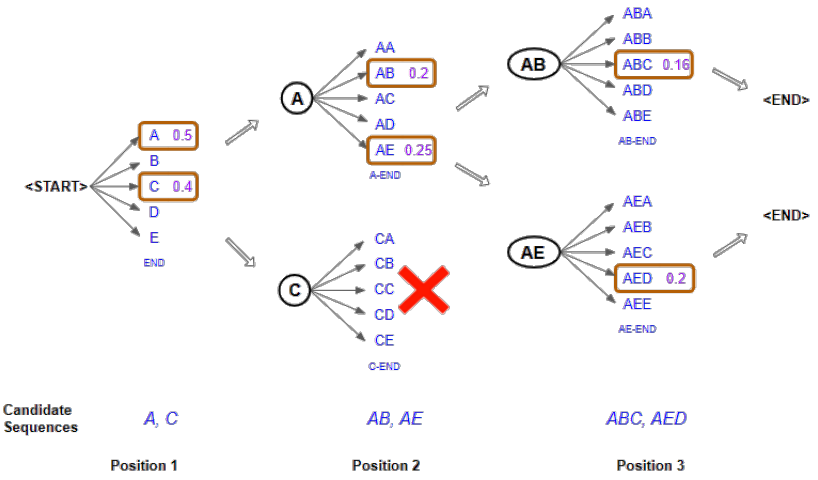
Beam Search

Un método común para obtener cadenas de salida es el de **Beam Search** que puede resumirse como:

1. Seleccionar las k salidas más probables dado *BOS*, y guardarlas. Sean $y_1^{(1)}, y_2^{(1)}, ..., y_k^{(1)}$.
2. En cada estado t de salida, y por cada cadena previa guardada $y_i^{(1)}...y_i^{(t-1)}$, $i = 1, ..., k$, obtener $p(y_i^{(t)}|x, y_i^{(1)}...y_i^{(t-1)})$. Guardar los k argumentos con las probabilidades más altas.
3. Se genera un árbol de transiciones cuya. En cada estado (profundidad del árbol) se toman los k elementos más probables. Se descartan los caminos que no estén entre los k más probables.ç
4. Se toma la cadena más probable de entre los k candidatos.

Cuando $k = 1$, el método se conoce como **Greedy Decoding**.

Beam Search


<START>
A 0.5
B
C 0.4
D
E
END
A
AA
AB 0.2
AC
AD
AE 0.25
A-END
C
CA
CB
CC
CD
CE
C-END
AB
ABA
ABB
ABC 0.16
ABD
ABE
AB-END
AE
AEA
AEB
AEC
AED 0.2
AEE
AE-END
<END>
<END>
Candidate Sequences
A, C
AB, AE
ABC, AED
Position 1
Position 2
Position 3

Modelos del lenguaje enmascarados

Un **modelo del lenguaje enmascarado** es una tupla $\mu = (\Sigma_{MASK}, p)$ con $\Sigma_{MASK} = \Sigma \cup \{< MASK >\}$, y p medida de probabilidad:

$$p(MASK = w^{(t)} | w^{(1)} ... w^{(t-1)} < MASK > w^{(t+1)} ... w^{(T)})$$

Este tipo de modelos del lenguaje toman en cuenta **todo el contexto** para determinar la probabilidad de una palabra.
Para esto enmascaran palabras de la cadena:

«el gato se $< MASK >$ *al ratón»*

BERT

El modelo de BERT (Bidirectional Encoder Representations from Transformers) [2] busca crear **representaciones** de los tokens a partir de un modelo del lenguaje enmascarado.

Consiste en dos pasos:

1. Pre-entrenamiento: Crear representaciones de tokens a partir de un modelo del lenguaje y predicción de siguiente oración.
2. Ajuste fino: Ajustar las representaciones como parte de una tarea específica.

BERT: Modelo del lenguaje enmascarado

El modelo de BERT enmascara el 15 % de una cadena de entrada. Elije tokens de manera aleatoria en base a los tres siguientes casos:

1. Se enmascare la palabra (80 % de las veces):

   la niña $<$ *MASK* $>$ por la ventana

2. Se sustituya por un token aleatorio, por ejemplo «salta» (10 % de las veces):

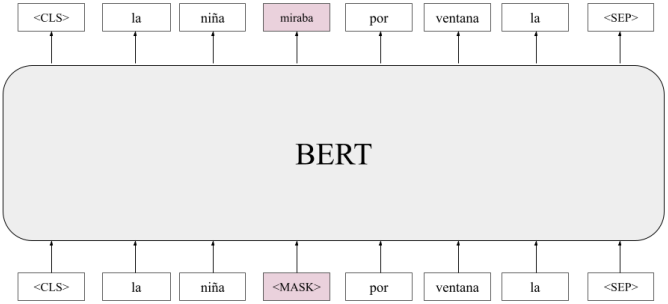
   la niña salta por la ventana

3. Se mantiene el token (10 % de las veces):

   la niña miraba por la ventana

BERT: Modelo del lenguaje enmascarado

Además de la etiqueta de $< MASK >$ que enmascara los tokens, se utilizan $< CLS >$ etiqueta de clase, y $< SEP >$ separador de oraciones. El objetivo es predecir las palabras enmascaradas, lo que crea el modelo del lenguaje:



BERT: Predicción de siguiente oración

Además de pre-entrenar el modelo del lenguaje, BERT aprende un modelo de clasificación. Busca **predecir la oración siguiente**.

BERT toma dos oraciones de entrada, separadas por $< SEP >$. Toma pares de oraciones del corpus como ejemplos positivos. Los ejemplos negativos se crean emparejando oraciones al azar.

1. Si el par de oraciones son consecuentes, se etiqueta como *IsNext*:

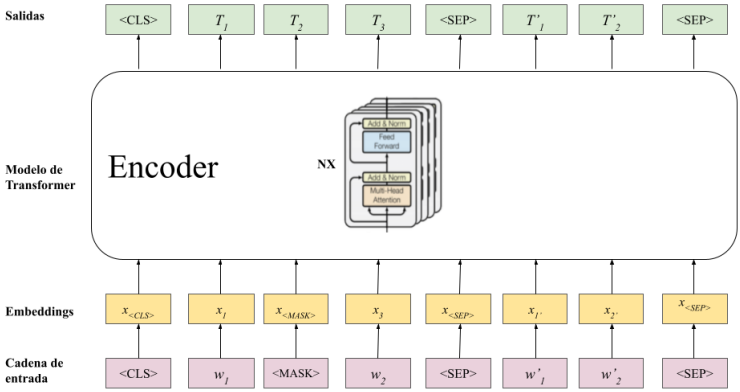
   $< CLS >$ fue a la tienda $< SEP >$ compró un litro de leche $< SEP >$

2. Si el par de oraciones no son consecuentes, se etiqueta como *NotNext*:

   $< CLS >$ fue a la tienda $< SEP >$ miró por la ventana $< SEP >$

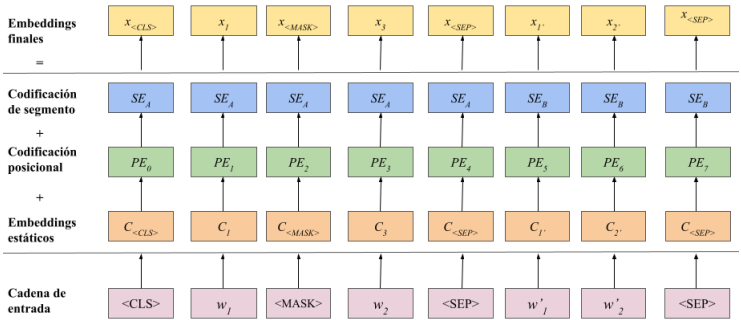
BERT: Arquitectura

El modelo de BERT se basa en el **encoder de un Transformer**; es decir, deja de lado el proceso de decoder. Su salida son las representaciones de este encoder.



BERT: Entradas

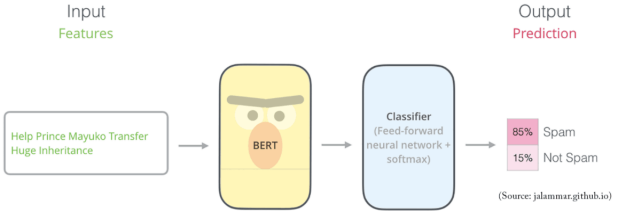
Los embeddings de entrada de BERT incorporan una codificación que indica si el token está en la primera o en la segunda oración. Además de la codificacion posicional y del embedding de token.



BERT: Ajuste fino

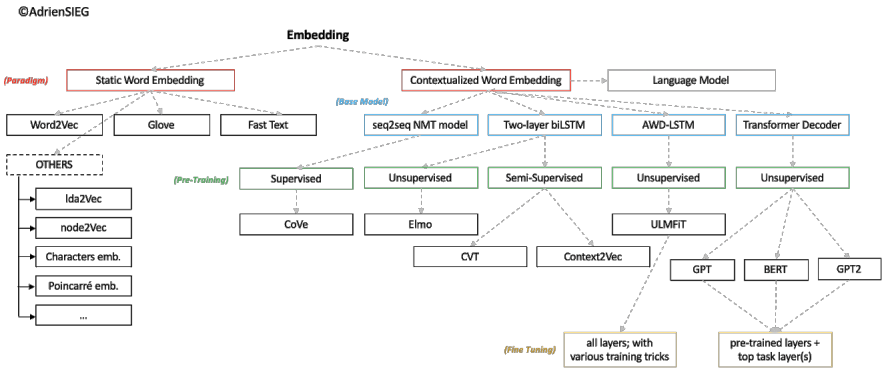
Una vez obtenidas las representaciones de BERT, estas se utilizan para tareas específicas.

El proceso de **ajuste fino** consiste en tomar estas representaciones para inicializar algún modelo de clasificación. De esta forma, la información de las representaciones se incorpora a la tarea.



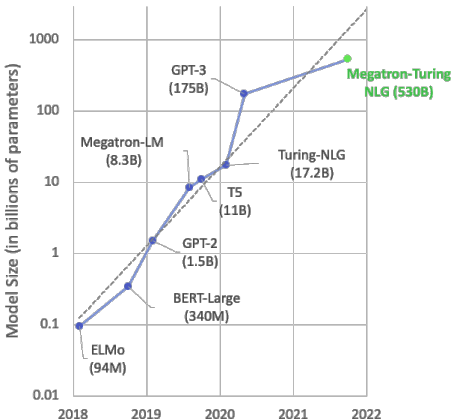
A modo de conclusión

Los transformers han dado pie a nuevas implementaciones para aprender Embeddings y modelos del lenguaje [1]:



A modo de conclusión

Los modelos actuales de NLP, sin embargo, son muy grandes y requieren de una gran capacidad de entrenamiento, muchas veces prohibitiva:



References

Nathan Benaich and Ian Hogarth.
State of ai report 2020, 2020.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova.
Bert: Pre-training of deep bidirectional transformers for language understanding.
arXiv preprint arXiv:1810.04805, 2018.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton.
Layer normalization.
ArXiv e-prints, pages arXiv–1607, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin.
Attention is all you need.
In *Advances in neural information processing systems*, pages 5998–6008, 2017.

The End