

Procesamiento del Lenguaje Natural

---

# Tema 8. Modelado neuronal del lenguaje

# Índice

## Esquema

## Ideas clave

8.1. Introducción y objetivos

8.2. Modelos de representación vectorial con word embeddings

8.3. Modelos de lenguaje basados en redes neuronales

8.4. Transformers

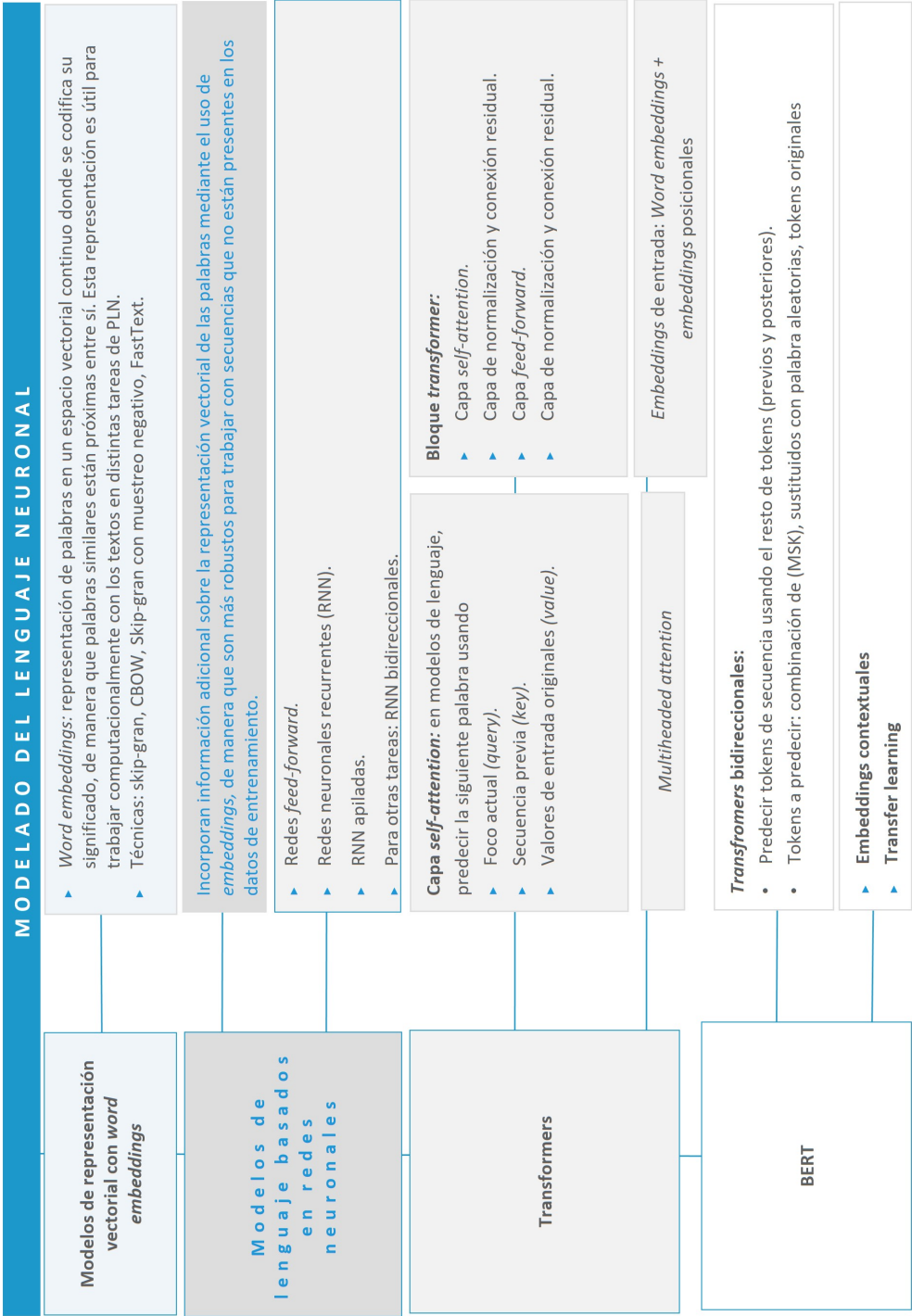
8.5. BERT

8.6. Referencias bibliográficas

## A fondo

Hugging Face

## Test



## 8.1. Introducción y objetivos

A continuación, se abordará el uso de redes neuronales para **construir la representación vectorial** del significado de las palabras mediante el uso de algoritmos como Skip-gram o CBOW. También, se verá el uso de redes neuronales para **abordar tareas de modelado del lenguaje**, como las que se vieron en el tema anterior. Finalmente, se presentará una arquitectura de redes neuronales muy relevante para el PLN, los *transformers*, con los que se pueden llevar a cabo tanto tareas de modelado del lenguaje, como de representación vectorial del significado de las palabras, entre muchas otras. En relación con esto, se hablará del modelo de BERT, y cómo se puede usar para obtener *embeddings* contextuales

### Objetivos

- ▶ Entender el concepto de *word embeddings*, y cómo poder construirlos usando arquitecturas basadas en redes neuronales.
- ▶ Describir cómo se pueden usar arquitecturas de redes neuronales para llevar a cabo tareas de modelado del lenguaje.
- ▶ Describir qué son los *transformers* en el ámbito del procesamiento del lenguaje natural, cómo es su arquitectura, y qué ventajas tienen.
- ▶ Identificar las características de un modelo como BERT, describir su arquitectura, y entender su uso para la obtención de *embeddings* contextuales.

## 8.2. Modelos de representación vectorial con word embeddings

Anteriormente, se estudió cómo los **modelos de representación vectorial** son **representaciones algebraicas de los textos o documentos**, de manera que cada uno de ellos se expresa mediante un vector que recoge su información, de modo que documentos o textos similares tendrán representaciones también similares. Dentro de las distintas técnicas que se pueden usar para construir estas representaciones, se comentó cómo funciona la técnica de **bolsa de palabras** (*Bag of Words*, *BoW*) con la que se obtenía un vector que representa el contenido de ese texto en base a información del texto en sí (representaciones locales), o del texto en sí junto con información del corpus (representaciones globales).

Ahora bien, un problema que se tiene con estas representaciones es el aspecto de la **dispersión** (*sparsity*). Por ejemplo, si se tiene un corpus de  $V=1000$  palabras, y se quiere construir el vector de un texto de diez palabras (sin considerar *stopwords*), se tendrá un vector con 990 componentes a 0 (correspondientes a todas las palabras del vocabulario que no aparecen en ese texto). Este es precisamente **el problema** de la dispersión, el **tener representaciones vectoriales** donde casi todas las componentes estén a 0.

El problema de la dispersión de las representaciones basadas en bolsas de palabras hace referencia a cómo se cómo los vectores que representan los textos tienen muchas de sus componentes a 0.

Además, construir representaciones vectoriales donde se tengan tantos componentes como palabras de un vocabulario puede dar lugar a **espacios vectoriales de mucha dimensión**, lo que se traduciría en tener un gran número de variables. Esto podría dificultar luego el entrenamiento de un modelo de aprendizaje automático (mayor coste de computación y mayor propensión a tener sobreajuste).

Adicionalmente, aunque las representaciones basadas en BoW pueden servir para construir el significado del texto en su conjunto, ¿cómo se podría hacer para representar el significado no sólo del texto sino también de las palabras que lo componen? Si bien se construyen espacios vectoriales donde dos textos similares tendrán vectores cercanos, el **significado de las palabras** no está contemplado dentro de estos espacios.

### *Word Embeddings*

Para solventar el problema de la dispersión, el problema de la representación en espacios vectoriales de gran dimensionalidad, y el de la representación del significado de las palabras, se proponen las **representaciones densas** denominadas **word embeddings**, con las que se asigna un valor numérico a las palabras en base a su semántica. De esta manera, en lugar de representar los textos directamente en un espacio vectorial, **se representan primero las palabras**, de manera que cada una tendrá asociado un vector numérico que recoja su valor semántico en un espacio dimensional de un tamaño predefinido. Este valor semántico (su *embedding*) se puede obtener en función de la relación contextual que cada palabra tenga con el resto de las palabras del corpus.

En la siguiente imagen se muestra el ejemplo clásico de representación con vectores en tres dimensiones. Gracias a que los valores numéricos se infieren por el contexto, se pueden hacer operaciones del siguiente tipo:

$$\text{King} - \text{Man} = \text{Queen} - \text{Woman}$$

Es decir, que los vectores representan el valor semántico de que si a «rey» se le resta la contribución de «hombre» y a «reina» de «mujer» se obtendría un mismo valor que representaría el valor semántico genérico de «monarca».

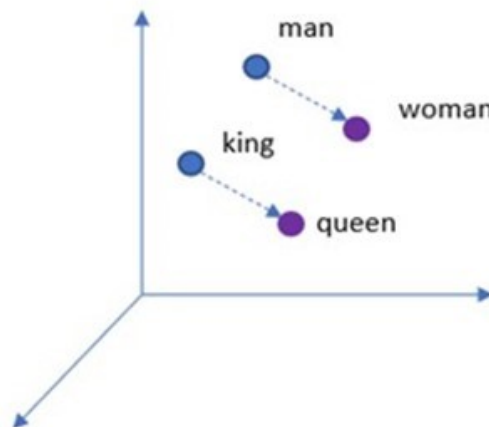


Figura 1 Ejemplo de representación con vectores en tres dimensiones. Fuente: elaboración propia.

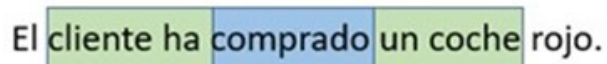
Así, con esta aproximación, cada palabra se representa con un vector de dimensionalidad  $D$ , y cada texto se representa como una matriz  $N \times D$ , con  $N$  el número de palabras. Estos se pueden obtener mediante distintas técnicas, como **Skip-gram** o **CBOW** (*Continuous Bag of Words*).

El término *word embeddings* dentro del PLN hace referencia a la representación de palabras en un espacio vectorial continuo donde se codifica su significado, de manera que palabras similares están próximas entre sí. Esta representación es útil para trabajar computacionalmente con los textos en distintas tareas de PLN.

## Skip-gram

La obtención de estos *embeddings* mediante el algoritmo de Skip-gram consiste en entrenar un modelo que tenga como entrada las distintas palabras del vocabulario, y que trate de predecir el contexto de dichas palabras.

Este contexto se define en función de las **palabras vecinas** que tengan esas palabras de entrada. Las palabras vecinas son las que aparecen en una ventana de un tamaño concreto alrededor de cada una de las palabras del vocabulario.



El cliente ha comprado un coche rojo.

El diagrama muestra la frase "El cliente ha comprado un coche rojo." con tres palabras resaltadas en recuadros: "cliente" (verde), "comprado" (azul) y "un" (verde). Los recuadros de "cliente" y "un" están a la izquierda y a la derecha del recuadro de "comprado", respectivamente, formando una ventana de tamaño 2.

Figura 2. Ejemplo de ventana de tamaño 2 alrededor de la palabra «comprado». Fuente: elaboración propia.

Con ello, se define un conjunto de datos de entrenamiento para entrenar un modelo que trate de **predecir las palabras del contexto de cada palabra del vocabulario**.

(comprado, cliente)  
(comprado, ha)  
(comprado, un)  
(comprado, coche)

Figura 3. Datos de entrenamiento expresados en tuplas con la palabra de entrada (azul) y las palabras del contexto (verde). Fuente: elaboración propia.

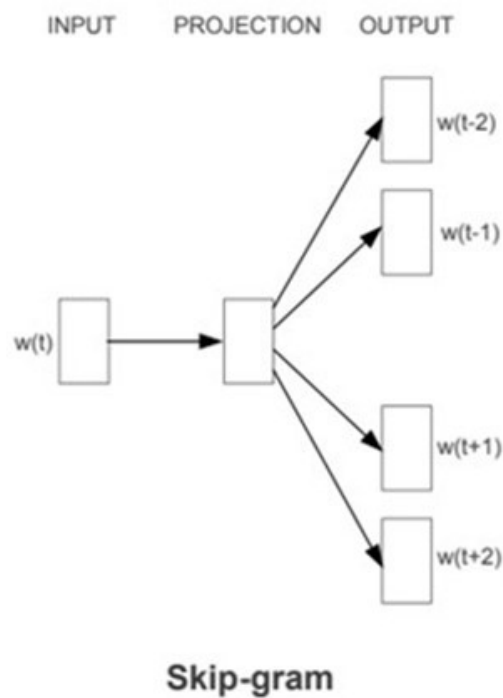


Figura 4. Esquema del algoritmo de Skip-gram para el caso de una ventana de tamaño 2. Fuente: Mikolov et al., 2013).

Así, formalmente, el algoritmo Skip-gram busca obtener lo siguiente:

$$P(w_1, w_2, \dots, w_m | w)$$

Donde

$w_1, w_2, \dots, w_m$  son las distintas palabras del contexto y

$w$  es la palabra de entrada. De esta manera, el modelo de Skip-gram para la obtención de *embeddings* es un modelo **auto-supervisado** (*self-supervised*), ya que es un modelo (dada la palabra de entrada se trata de predecir el contexto) donde los datos de entrenamiento no se obtienen de un proceso de anotación previo, sino que se obtienen automáticamente de los propios datos de entrada.

Una aproximación para construir el modelo de Skip-gram utilizando aprendizaje automático, es la siguiente:

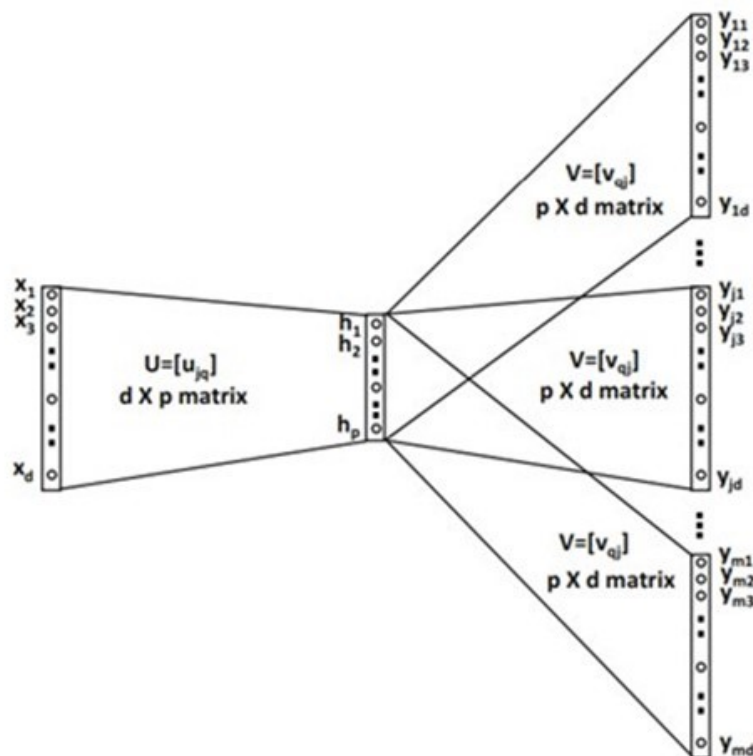


Figura 5. Arquitectura del modelo Skip-gram. Fuente: Mikolov et al., 2013.

Dada una palabra de entrada ( $w_q$ ), se tiene una **red neuronal** con una capa intermedia con la que se trata de predecir si una palabra concreta  $w_j$  pertenece o no su contexto. La entrada del modelo, que corresponde a una palabra concreta del vocabulario (de tamaño  $d$ ), se puede expresar mediante un vector *one-hot encoding*, donde todos los valores sean 0 menos el de la palabra en cuestión. De igual manera, el vector asociado a la salida (la palabra del contexto), también se podrá expresar mediante un vector *one-hot encoding*. Esta arquitectura tiene dos matrices de pesos  $U$  y  $V$ , que serán compartidas para todas las combinaciones de palabras de entrada con respecto a sus respectivas  $m$  palabras de contexto, de manera que se tenga una única matriz  $U$  y una única matriz  $V$  al final del entrenamiento. La matriz  **$U$**  corresponde a la **matriz de embeddings**.

Es decir, que la representación vectorial del significado de las palabras se obtendrá directamente de los pesos de la red neuronal, recogidos en dicha matriz (también denominada *lookup table*).

El valor  $p$  corresponde precisamente a la dimensionalidad del espacio de representación de los vectores (es decir, al número de componentes que estos tengan). Este valor  $p$  es un parámetro que se define a priori.

El valor de salida de la capa oculta se obtendría de la siguiente manera:

$$h_q = \sum_{j=1}^d u_{jq} x_j \quad \forall q \in \{1 \dots p\}$$

Y, con ello, se podría obtener ya la salida del modelo. Ahora bien, como lo que se trata de hacer es predecir si una palabra está o no en el contexto de la palabra de entrada, lo que corresponde a un **problema de clasificación**, la capa de salida tiene incluida una función *softmax*. De esta manera, si la salida de la matriz  $V$  corresponde

a un vector  $z$ , la predicción final quedaría como:

$$\hat{y} = p(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Donde se normaliza la salida para que su valor sea siempre positivo, entre 0 y 1, y corresponda a un valor de probabilidad tal que así la suma de todos los valores de salida sean 1. Esta salida es la que se compararía con el vector de *one-hot encoding* asociado a esa palabra del contexto. De esta manera, ambos se incluirían en una función de coste (como, por ejemplo, la **entropía cruzada**) tal que se busque minimizar:

$$L = - \sum_{i=1}^m \sum_{j=1}^d y_{ij} \log(\hat{y}_{ij})$$

Donde el vector

$\hat{y}$  corresponde a la predicción, y el vector

$y$  al valor objetivo obtenido desde ese *one-hot encoding*.

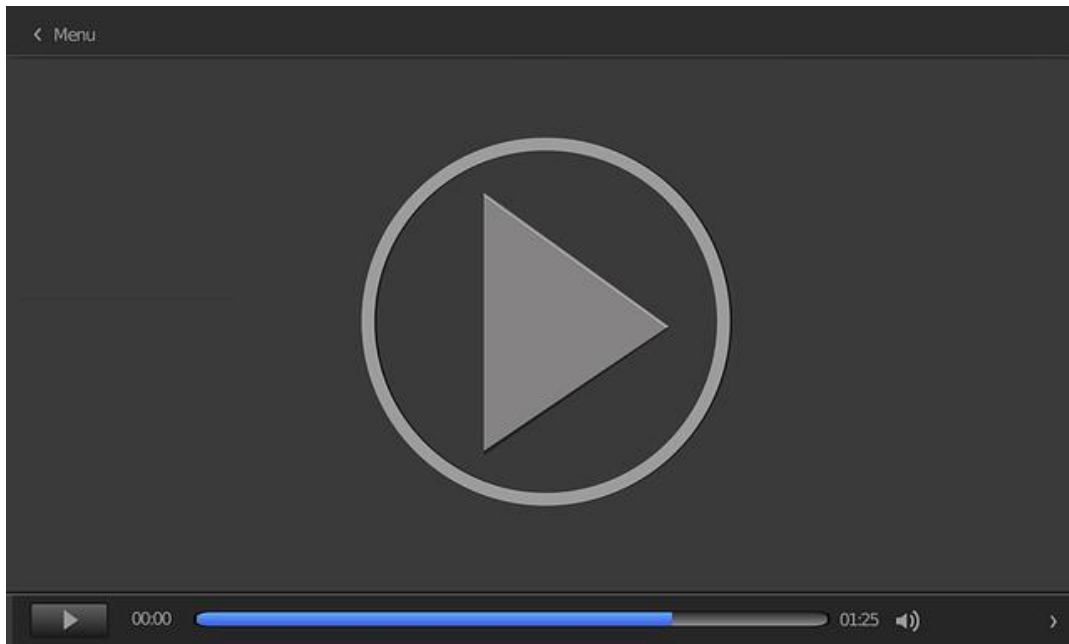
Con esto, se entrenaría el modelo para todas las palabras del vocabulario y para todas las palabras de su contexto, ajustando los pesos mediante técnicas como, por ejemplo, las de **descenso de gradiente**.

Una vez que se tienen las dos matrices finales  $U$  y  $V$ , se puede obtener el *embedding* para una palabra del vocabulario  $d$  obteniendo su vector de entrada y multiplicándolo por la matriz de *embeddings*  $U$  (*lookup table*).

$$\begin{pmatrix} 1 & 0 & \dots & 0 \end{pmatrix} \times \begin{pmatrix} 0.4 & \dots & 2.1 \\ \vdots & \ddots & \vdots \\ -1.3 & \dots & 1.1 \end{pmatrix} = \begin{pmatrix} 0.4 \\ \vdots \\ 2.1 \end{pmatrix}$$

Figura 7. Ejemplo de obtención de los *embeddings* para una palabra de entrada (la primera del vocabulario) dada su *lookup table*. Fuente: elaboración propia.

En el vídeo *Word embeddings y wordvec* se verá cómo el aprendizaje profundo sirve para modelar el significado de las palabras dentro del PLN.



08.01. Word embeddings y word2vec

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=381988c0-5b00-4cbc-9b31-af5b0149ccfa>

### CBOW

El esquema del modelo CBOW es el **inverso al del Skip-gram**. Como entrada se tienen las palabras del contexto de una determinada palabra, y como salida se tiene esa palabra. De esta manera, se plantea un modelo de clasificación que trate de predecir una palabra dado su contexto. Así, la entrada serán tuplas (como en Skip-gram) donde el primer elemento es una palabra del contexto y el segundo la palabra que se quiere predecir.

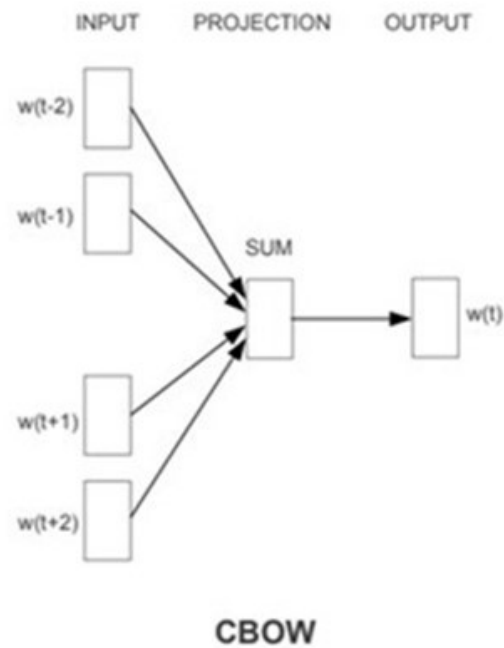


Figura 8: Esquema del algoritmo de CBOW para el caso de una ventana de tamaño 2. Fuente: Mikolov et al., 2013.

Formalmente, sería:

$$P(w|w_1, w_2, \dots, w_n)$$

De manera análoga a cómo era la arquitectura de Skip-gram, CBOW usa un modelo basado en redes neuronales donde se tiene una capa oculta. Así, se tienen dos matrices de pesos,  $U$  y  $V$ , entre la entrada y la capa oculta, y entre la capa oculta y la salida, respectivamente. Esto se ve en la siguiente imagen:

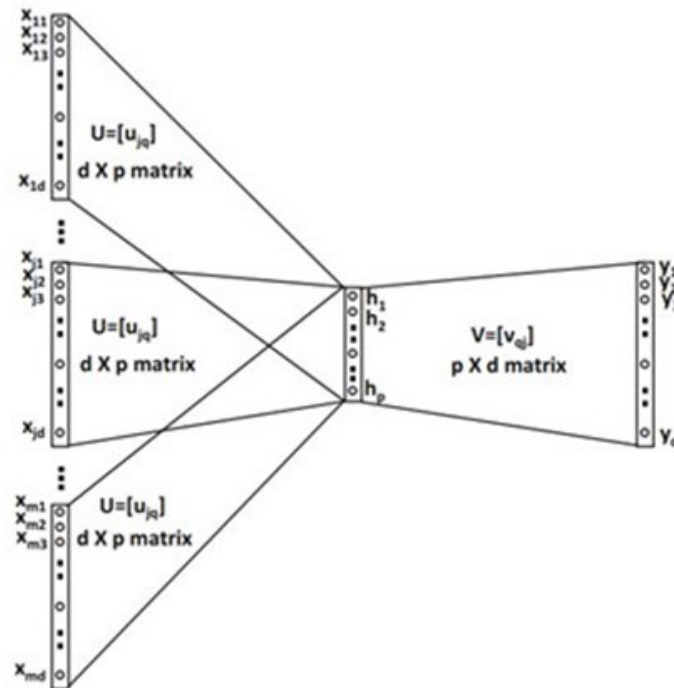


Figura 9. Arquitectura del modelo CBOW. Fuente: Mikolov et al., 2013.

Las palabras de entrada se pueden convertir en vectores binarios con una técnica como *one-hot encoding*, y lo mismo se podrá hacer sobre la palabra de salida. La capa de salida tendrá una función *softmax* para expresar dicha salida como una probabilidad. Igual que en el caso de Skip-gram, las matrices de pesos  $U$  y  $V$  son compartidas entre todas las combinaciones de palabras de contexto y palabras de salida. La matriz  $U$  final después del entrenamiento será la que se use como matriz de *embeddings*.

No obstante, en el modelo de CBOW no es necesario considerar de manera independiente las palabras de entrada del contexto, sino que se puede reducir significativamente el coste computacional considerando ese contexto como un único vector de entrada con el que se busca predecir una determinada palabra.

La manera de hacer esto es hacer *one-hot encoding* sobre todas las palabras del contexto, de manera que de la matriz de *embeddings* se obtengan los valores de los vectores asociados a todas ellas. Luego, sobre esta salida, se aplica una capa de *pooling* que agregue esos valores (por ejemplo, obteniendo el valor medio de esos vectores), y ya con ese valor agregado se obtenga la probabilidad de predicción de la palabra de salida. Un esquema de esta aproximación aparece en la imagen siguiente:

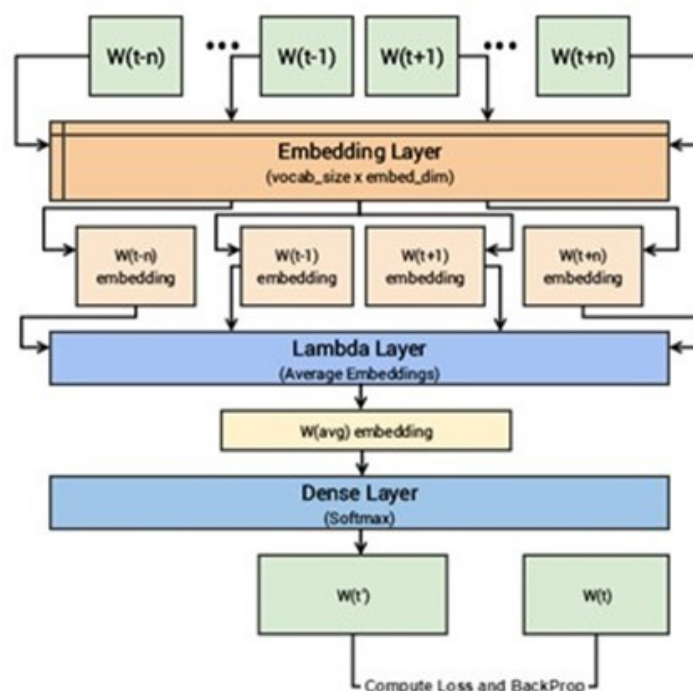


Figura 10. Arquitectura de CBOW donde se agrega el contexto de entrada en un único vector. Fuente: KD Nuggets, 2018.

Comparando Skip-gram con CBOW, aparecen algunas diferencias significativas que hacen que un modelo u otro sean mejores en distintos casos. Con el modelo de CBOW, si se tienen frases como «el agua del mar es clara» y «el agua del mar es prístina», en ambos casos se recibe un mismo contexto y se trata de predecir la palabra más probable para él.

Por este motivo, CBOW da buenos resultados para las representaciones de palabras comunes, pero no tanto para las palabras poco frecuentes, ya que ante un contexto se va a poner el énfasis en predecir las palabras que son más comunes.

Así, será más fácil en el ejemplo anterior representar clara que prístina. En cambio, como con Skip-gram se recibe una palabra y se busca predecir su contexto, para los ejemplos anteriores se tratarían esas palabras, clara y prístina, como palabras independientes, de manera que será más precisa la representación de prístina que con CBOW. Así, Skip-gram da mejores resultados para la **representación de palabras poco frecuentes**.

Otra diferencia significativa entre ambos modelos es que **es más rápido entrenar CBOW que Skip-gram**. El motivo es el siguiente: para el caso del modelo con la capa de *pooling* de CBOW, hay menos combinaciones de entradas-salidas para entrenar el modelo.

### Skip-gram con muestreo negativo

El modelo de Skip-gram visto previamente tienen una fase especialmente costosa a nivel computacional: el cálculo de la capa *softmax*. En esta capa se transforman las salidas de la red neuronal a un vector de probabilidades normalizado sobre todas las palabras del vocabulario, haciéndose este cálculo para todas las palabras de entrada sobre las que se quiere predecir el contexto. Por este motivo, existe una arquitectura de Skip-gram alternativa denominada **Skip-gram con muestreo negativo** (Skip-gram with negative sampling, SGNS).

Con la arquitectura SGNS, en vez de trabajar con todas las palabras del vocabulario en la predicción del contexto, se trabaja con un subconjunto de palabras formado por palabras que aparecen en el contexto de la palabra de entrada, y algunas palabras que no aparecen. Con esto se busca que **el modelo sepa diferenciar qué palabras**

**son del contexto y cuáles no.** Para poder implementar esta arquitectura, se lleva a cabo un cambio: en lugar de plantear una clasificación multiclase, se plantea una **arquitectura de clasificación binaria** usando una función de salida sigmoide, tal que, dada una palabra de entrada y una segunda palabra de entrada, se obtenga la predicción de si esa segunda palabra es del contexto de la primera o no. Esta idea se refleja en la siguiente arquitectura:

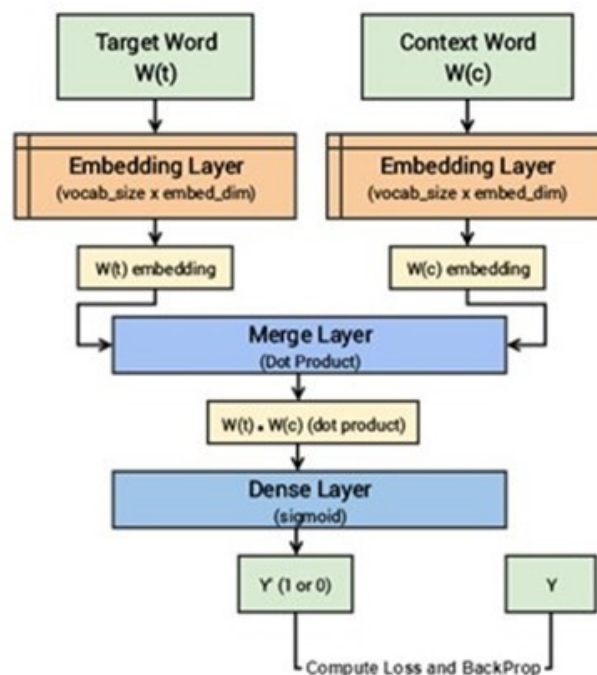


Figura 11. Ejemplo de arquitectura para SGNS. Fuente: KD Nuggets, 2018.

De esta manera, para  $P$  palabras del contexto como entrada y para  $N$  palabras obtenidas de un muestreo que no son del contexto, se obtienen  $P + N$  predicciones para una determinada palabra de entrada, con las que se calcula la siguiente función de coste que se busca optimizar:

$$-\sum_{(i,j) \in \mathcal{P}} \log \left( \frac{1}{1 + \exp(-\bar{u}_i \cdot \bar{v}_j)} \right) - \sum_{(i,j) \in \mathcal{N}} \log \left( \frac{1}{1 + \exp(\bar{u}_i \cdot \bar{v}_j)} \right)$$

Donde el valor de los sumatorios corresponde a la salida de las predicciones del modelo en función de valores de entrada. Así, con esta función se busca que esos valores P sean lo mayor posible (predicciones cercanas a 1) y los N lo más pequeños posibles (cercanos a 0).

### FastText

Uno de los problemas que aparecen con las representaciones vistas hasta ahora de word2vec es **cómo representar palabras desconocidas**. La representación de las palabras se ha obtenido mediante el entrenamiento previo de una red neuronal sobre un corpus de entrenamiento, y una vez hecho esto, se usa la matriz de pesos para obtener esas representaciones.

Ahora bien, podría ocurrir que se quieran obtener representaciones de palabras que no aparecían en el corpus sin tener que reentrenar toda la red neuronal para ello. Esto es lo que trata de solucionar el modelo de **FastText** (Bojanowski et al., 2017) mediante la construcción de representaciones vectoriales no sólo de las palabras, sino también de los distintos n-gramas de una misma palabra. Así, por ejemplo, la palabra «where» con n-gramas de tamaño 3 para sus caracteres se representaría como:

<wh, whe, her, ere, re>

Con ello, se obtienen los *embeddings* con técnicas como Skip-gram para cada uno de esos n-grama constituyentes, de manera que el *embedding* de la palabra se construye luego como la suma de todos esos *embeddings* de los constituyentes. Cuando se quieren representar palabras desconocidas, se representarán en base a la suma de los *embeddings* de los n-gramas constituyentes que sí aparecen en el corpus.

### 8.3. Modelos de lenguaje basados en redes neuronales

Las redes neuronales no son solo útiles para construir modelos de representación vectorial, como es el caso de la obtención de *word embeddings* mediante el uso de las técnicas vistas previamente. También son útiles para la construcción de **modelos de lenguaje** (*language model*, LM). Como ya se comentó en el tema previo, con un modelo del lenguaje se calculan las probabilidades de que desde una secuencia se continúe con un determinado elemento. Ahora bien, para el cálculo de esta probabilidad no se tienen que considerar todos los elementos de la secuencia, sino que se puede aproximar considerándose únicamente los N últimos términos. Formalmente, esto se expresa de la siguiente manera:

$$P(w_t | w_1, \dots, w_{t-1}) \approx P(w_t | w_{t-N+1}, \dots, w_{t-1})$$

Para el caso de  $N = 4$  sería:

$$P(w_t | w_{t-3}, w_{t-2}, w_{t-1})$$

En el tema previo se vio cómo era posible obtener estas probabilidades en base al uso de modelos estadísticos basados en n-gramas. Ahora bien, estos modelos tenían una serie de limitaciones como, por ejemplo, el cálculo de probabilidades cuando se tienen secuencias no vistas en el corpus de entrenamiento. Si, por ejemplo, se trabaja con  $N = 4$  y en el corpus de entrenamiento se tiene una frase como

- *Tengo un perro de mascota*

Cuando se tengan secuencias como «un perro de» se podrá predecir «mascota». Ahora bien, si aparecen frases nuevas como «tengo un gato de» no se sabrá qué palabra podrá seguir a esa secuencia. Este problema se soluciona en gran parte con el uso de **modelos basados en redes neuronales** para la construcción de LM, ya que en ellos se incorpora el concepto de *embeddings* visto previamente, de manera que, aunque el LM no haya visto nunca la palabra «gato» en esa secuencia, a nivel de representación de *embeddings* será similar a «perro» y se podrá estimar que una palabra como «mascota» podría seguir a esa secuencia.

Los modelos de lenguaje basados en redes neuronales incorporan información adicional sobre la representación vectorial de las palabras mediante el uso de *embeddings*, de manera que son más robustos para trabajar con secuencias que no están presentes en los datos de entrenamiento.

### *Redes feed-forward*

La construcción de un LM basado en redes neuronales se puede llevar a cabo mediante el uso de **distintas arquitecturas**. Una primera aproximación es mediante el uso de redes *feed-forward*, como refleja la imagen siguiente.

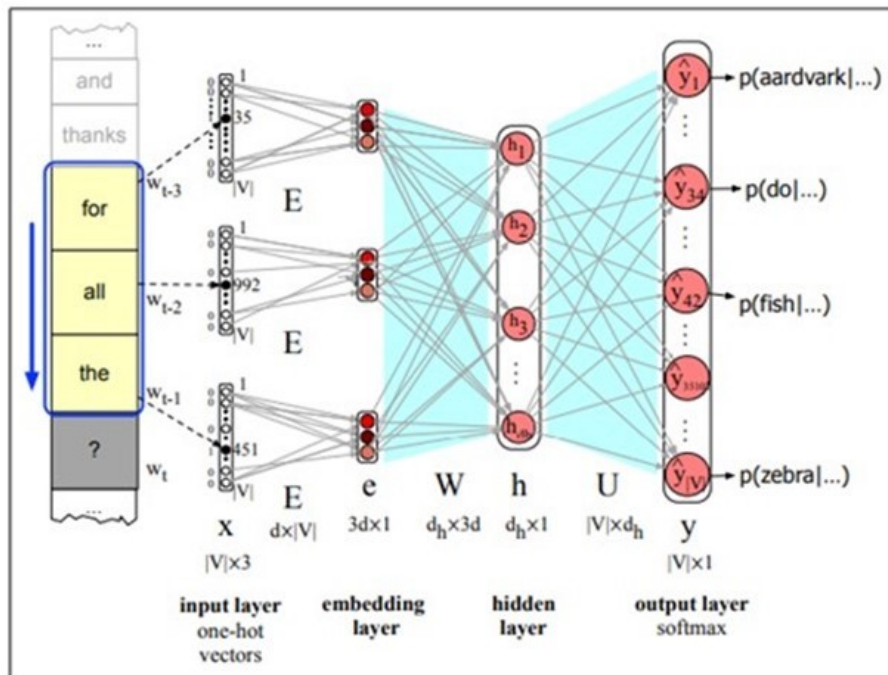


Figura 13. Arquitectura para un LM basado en feed-forward networks. Fuente: Jurafsky, 2021.

Como muestra la imagen, tras definir el tamaño de la secuencia, las palabras que la componen se expresan mediante sus vectores *one-hot encoding*, los cuales se usan para obtener los *embeddings* de esas palabras desde la matriz de pesos  $E$ . Estos vectores de *embeddings* se concatenan para generar un vector  $E$  final que se multiplica por la matriz de pesos  $W$ . Sobre la salida final se aplica la función de activación de la capa de salida (ej., *softmax*) para tener la probabilidad de que cada una de las palabras del vocabulario sigan a esa secuencia de entrada. De este modo, las ecuaciones de la red son:

$$\begin{aligned} \mathbf{e} &= [\mathbf{Ex}_{t-3}; \mathbf{Ex}_{t-2}; \mathbf{Ex}_{t-1}] \\ \mathbf{h} &= \sigma(\mathbf{We} + \mathbf{b}) \\ \mathbf{z} &= \mathbf{Uh} \\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{z}) \end{aligned}$$

Un aspecto importante de esta arquitectura es la matriz de pesos  $E$  desde la que se obtienen los *embeddings*. Para los valores iniciales de esta matriz, en lugar de ser valores aleatorios, se puede usar la matriz de *embeddings* resultante de aplicar alguno de los algoritmos de *word embeddings* vistos en el apartado anterior.

Una vez que se tiene esta arquitectura junto con un corpus de entrenamiento, pasarán por ella distintas secuencias de entrada para tratar de predecir la palabra que sigue tras ellas, y en base a esta información se irán ajustando los pesos del modelo con técnicas como, por ejemplo, **el descenso de gradiente**.

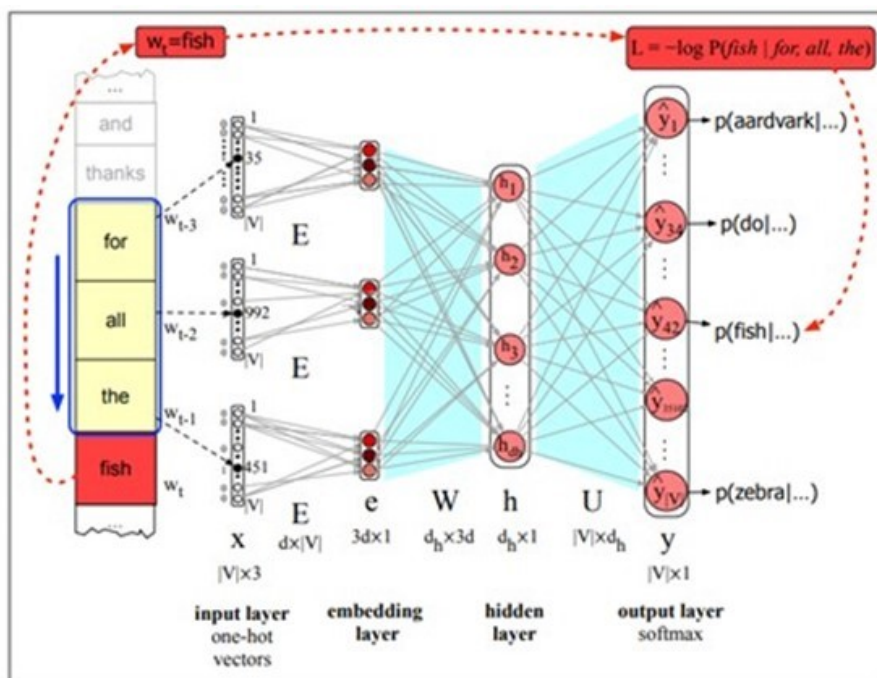


Figura 15. Cálculo de la función de coste en el LM. Fuente: Jurafsky, 2021.

Para completar el estudio de esta sección y ver en detalle las ecuaciones involucradas en el entrenamiento del modelo puedes leer el Capítulo 7 del libro: Jurafsky, D. y Martin, J. H. (2021). *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition and Computational Linguistics*. Prentice-Hall.

<https://web.stanford.edu/~jurafsky/slp3/7.pdf>

### Redes neuronales recurrentes

Además de los modelos *feed-forward* vistos previamente, existen otras arquitecturas de redes neuronales que son útiles para el modelado del lenguaje, como las **redes neuronales recurrentes** (*recurrent neural networks, RNN*).

Las RNN se caracterizan por utilizar no sólo la información de entrada para calcular la salida, sino que también utilizan la información de los estados previos, de manera que son útiles para capturar información secuencial, como es el caso de los textos.

En la siguiente imagen se ve un ejemplo de RNN, donde  $W$  es la matriz de pesos,  $V$  está asociada a la capa oculta, y  $U$  es la matriz de pesos de salida. De esta manera, el valor de salida  $y_t$  para un instante  $t$  no depende sólo de la entrada en ese momento ( $x_t$ ), sino que depende también de la información del estado previo. Esta información se recibe mediante otra matriz de pesos,  $U$ , que conecta la capa oculta del paso anterior con la capa oculta en el paso actual.

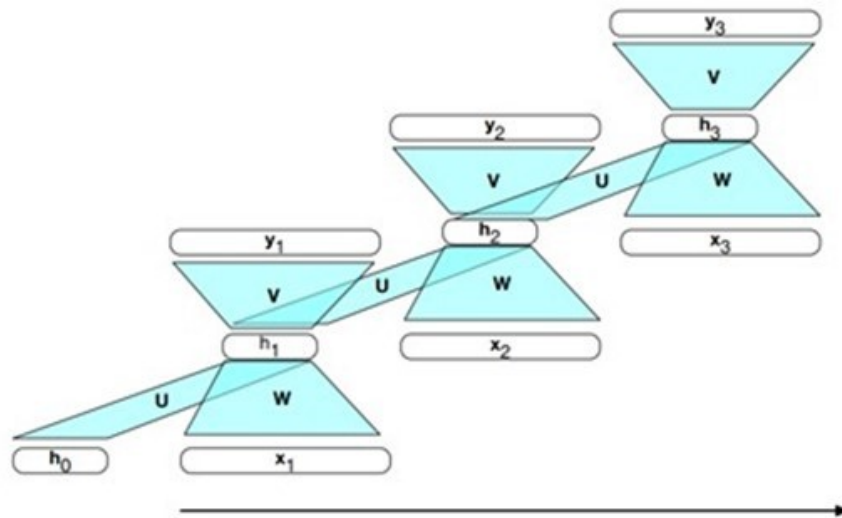


Figura 16. Ejemplo de arquitectura de RNN. Fuente: Jurafsky, 2021.

Trasladando el esquema de las RNN al problema del modelado del lenguaje, se tiene una arquitectura como la que muestra la siguiente imagen. Cada paso de la RNN corresponderá a la predicción de la siguiente palabra de la secuencia, usando información tanto de la última palabra (la palabra de entrada en el estado  $t$ ), como la información previa de la secuencia (recibida desde el estado  $t-1$ , que contiene tanto la información de  $t-1$  como la información previa a ese estado).

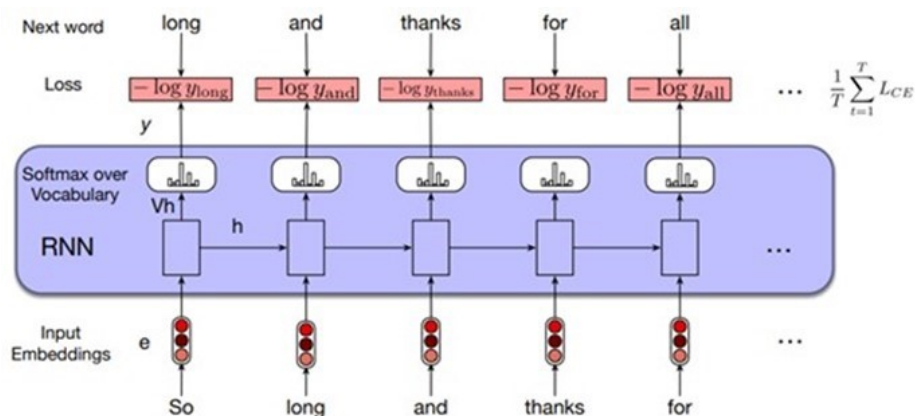


Figura 17. RNN para un LM. Fuente: Jurafsky, 2021.

El esquema de matrices es el mismo que el que se describió para la RNN en general, con una salvedad, y es que aparece una matriz adicional  $E$ , la matriz de

*embeddings*, que convierte el valor del vector de la palabra de entrada (expresada mediante su *one-hot encoding*) en su vector de *embeddings* ( ). Con ello, se tienen las siguientes ecuaciones para el vector de entrada (el *one-hot encoding*) de la palabra de la secuencia en el instante  $t$  (donde  $g$  corresponde a la función de activación de la capa oculta):

$$\begin{aligned} \mathbf{e}_t &= \mathbf{E}\mathbf{x}_t \\ \mathbf{h}_t &= g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \\ \mathbf{y}_t &= \text{softmax}(\mathbf{V}\mathbf{h}_t) \end{aligned}$$

Como ocurría con las *feed-forward networks*, las matrices de pesos  $E$ ,  $V$  y  $U$  son comunes para todas las palabras de la secuencia de entrada. La salida es un vector con las probabilidades de que cada palabra del vocabulario sea la palabra siguiente de la secuencia. De esta manera, correspondería a la probabilidad de que esa palabra fuese la siguiente en la secuencia dada las palabras previas, como se muestra a continuación:

$$P(w_{t+1} = i | w_1, \dots, w_t) = \mathbf{y}_t[i]$$

En la imagen previa también se muestra el cálculo de la función de coste en cada uno de los pasos. Al ser un modelo supervisado (auto-supervisado), se tiene la información de la palabra que realmente sigue a cada secuencia, de manera que se puede comparar con la palabra predicha para obtener el valor de la función de coste y ajustar los pesos. Utilizando la ecuación de la **entropía cruzada** con el valor del vector de salida correcto para la palabra  $w$  e el valor predicho para esa palabra, se tiene el siguiente valor de la función de coste considerando todas las palabras del vocabulario.

$$L_{CE} = - \sum_{w \in V} y_t[w] \log \hat{y}_t[w]$$

Como el vector de salida está expresado también según su *one-hot encoding*, su valor puede ser 0 o 1, de manera que la función de coste se puede simplificar a la siguiente ecuación para calcular el error en el estado  $t$  con respecto a la predicción de la siguiente palabra de la secuencia, .

$$L_{CE}(\hat{y}_t, y_t) = -\log \hat{y}_t[w_{t+1}]$$

Finalmente, en estas arquitecturas existe también un concepto, denominado **weight tying**, con el que se simplifica la estructura de la RNN, reduciendo el número de parámetros que se tienen que ajustar, considerando la matriz de salida  $V$  igual que la de los *embeddings*,  $E$ .

La intuición tras esto es que ambas matrices representan un concepto de *embeddings* similar, y ambas tienen una dimensionalidad que depende del tamaño del espacio de los *embeddings* y del tamaño del vocabulario.

Las ecuaciones quedarían:

$$\begin{aligned} \mathbf{e}_t &= \mathbf{E}\mathbf{x}_t \\ \mathbf{h}_t &= g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \\ \mathbf{y}_t &= \text{softmax}(\mathbf{E}^{\text{intercal}}\mathbf{h}_t) \end{aligned}$$

## RNN apiladas

La arquitectura vista previamente de RNN para LM no es la única posible. En ocasiones, ocurre que el modelado de los datos puede ser muy complejo y esto requiera de arquitecturas que modelicen mejor las relaciones entre dichos datos. Por este motivo, existen también arquitecturas como las **RNN apiladas**, donde no se tienen una única capa oculta, sino que **se tienen distintas capas**, donde cada una de ellas recibe la salida de la capa previa, hasta llegar a la última que es la que generaría la predicción.

Este tipo de aproximaciones pueden dar resultados más precisos que las soluciones de una única capa al tener distintos niveles de abstracción, con los que capturar aspectos diversos de los datos de entrada.

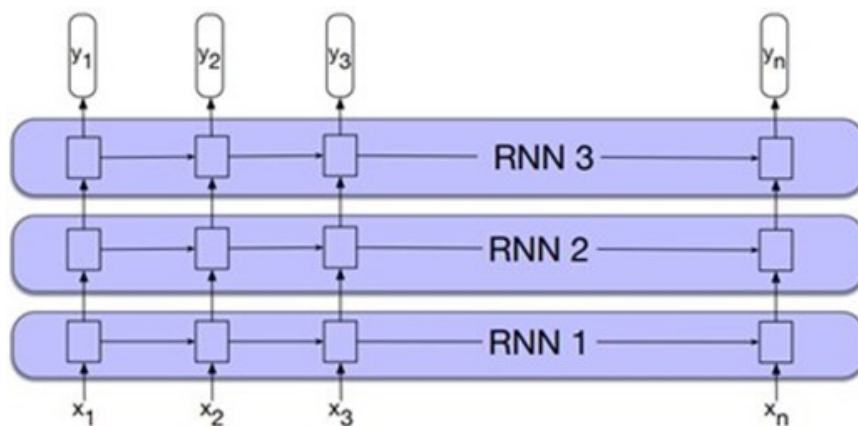


Figura 23. Arquitectura de RNN apiladas. Fuente: Jurafsky, 2021.

## RNN bidireccionales

Las RNN vistas hasta ahora consideran de manera conjunta la información del estado actual y de los estados previos. No obstante, en ocasiones también es interesante **considerar la información posterior** a un determinado estado para tareas como el LM. Por ejemplo, si se quiere predecir la palabra herramienta en la siguiente frase:

- ▶ María buscó el gato entre sus herramientas para poder arreglar el coche.

La información previa para esa palabra «María buscó el gato entre sus» podría no ser tan útil como la información posterior «para poder arreglar el coche» para predecirla.

Así, existen también las RNN bidireccionales, donde cada estado recibe información tanto de la secuencia previa como de la posterior para hacer las predicciones.

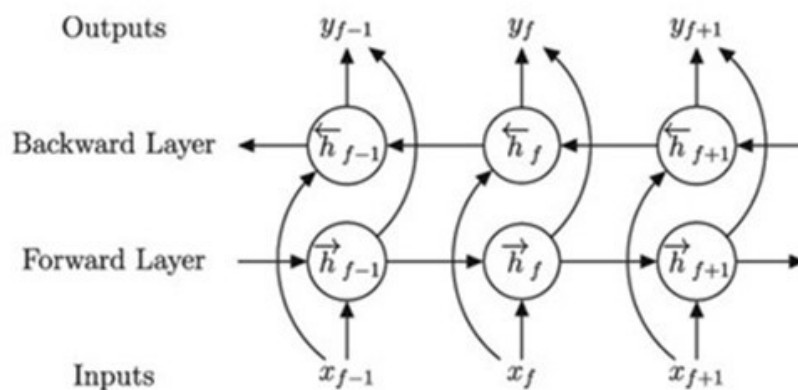


Figura 24. Arquitectura de una RNN bidireccional. Fuente: Graves et al., 2013.

Las ecuaciones asociadas a estas RNN son similares a las vistas hasta ahora, con la salvedad de que se distingue entre dos capas ocultas, las asociadas al procesamiento de la secuencia de *izquierda a derecha*, y las asociadas al procesamiento de la secuencia de *derecha a izquierda*. Para un instante  $f$  son, respectivamente,  $\vec{h}_f$  y  $\overleftarrow{h}_f$ . Con ello, una manera de construir la red es con las siguientes ecuaciones:

$$\begin{aligned}\vec{h}_f &= g(\mathbf{W}_{x\vec{h}}\mathbf{x}_f + \mathbf{W}_{\vec{h}\vec{h}}\vec{h}_{f-1} + \mathbf{b}_{\vec{h}}) \\ \overleftarrow{h}_f &= g(\mathbf{W}_{x\overleftarrow{h}}\mathbf{x}_f + \mathbf{W}_{\overleftarrow{h}\overleftarrow{h}}\overleftarrow{h}_{f-1} + \mathbf{b}_{\overleftarrow{h}}) \\ \mathbf{y}_f &= m(\mathbf{W}_{\vec{h}y}\vec{h}_f + \mathbf{W}_{\overleftarrow{h}y}\overleftarrow{h}_f + \mathbf{b}_y)\end{aligned}$$

Donde  $m$ , que corresponde a la función de activación sobre la capa oculta, puede ser la función *softmax* para el caso del modelado del lenguaje. En estas ecuaciones genéricas se incluyen los términos opcionales de *bias*  $\vec{b}_h, \overleftarrow{b}_h$  y  $b_y$ .

Como se puede observar, el uso de las RNN bidireccionales está condicionado a que esté disponible toda la secuencia del texto al usar tanto la información previa como la información posterior de una palabra.

Por este motivo, no son estructuras que sirvan para todas las tareas de PLN. Por ejemplo, no servirían para un **modelado de lenguaje causal** (donde se trata de predecir la siguiente palabra usando sólo la información previa de la secuencia), o para tareas de PLN como el autocompletado de frases.

## Otras arquitecturas: LSTM

Las arquitecturas de RNN tienen varios problemas que se deben tener en cuenta. Supongamos una frase como:

- Los vuelos que la aerolínea canceló estaban llenos.

En este caso, predecir una palabra como «canceló», en su tiempo verbal adecuado, puede ser factible al tener en su contexto cercano su sustantivo asociado «aerolínea». Ahora bien, la palabra «estaban» está vinculada con *vuelos*, y este sustantivo está muy lejos de su contexto cercano. Esto puede hacer que su predicción sea más complicada ya que, aunque se reciba la información previa de la secuencia, la información relevante está muy distante.

Adicionalmente, un problema común en las RNN es respecto a **retropropagar** (*backpropagate*) el error a lo largo de las distintas etapas para ajustar los pesos. Ocurre que, al ajustar los pesos en base a las derivadas parciales y a la regla de la cadena, el gradiente se irá reduciendo y tomando valores cada vez más pequeños en las etapas más lejanas, no ajustándolos adecuadamente. Este problema se conoce como **desvanecimiento de gradiente**.

Para evitar estos problemas, se han propuesto arquitecturas más complejas, como por ejemplo mediante el uso de unidades ocultas como las **LSTM** (*long short-term memory*).

---

Para completar el estudio de esta sección y ver más información sobre la necesidad del uso de las LSTM y su arquitectura interna, puedes leer el Capítulo 9 del libro Jurafsky, D. y Martin, J. H. (2021). *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition and Computational Linguistics*. Prentice-Hall.  
<https://web.stanford.edu/~jurafsky/slp3/9.pdf>

---

## 8.4. Transformers

Ocurre que, aunque los modelos basados en RNN son útiles para modelar secuencias, como textos (y en particular para LM), tienen a su vez una serie de desventajas para tener en cuenta. Primero, aun usando estructuras avanzadas como las LSTM, no siempre se puede evitar la pérdida de información desde etapas anteriores lejanas. Segundo, la estructura de las RNN dificulta que se pueda procesar su información en paralelo.

Por este motivo, han surgido nuevos tipos de arquitecturas de redes neuronales, como los *transformers*, donde se busca conservar las ventajas de las RNN, solucionando los problemas mencionados.

### *Self-attention*

Los *transformers* se componen de distintos bloques, donde se recibe un vector de datos de entrada

$(x_1, \dots, x_n)$ , y se genera un vector de salida

$(y_1, \dots, y_n)$  del mismo tamaño. Esto se lleva a cabo mediante distintos **bloques**, que se componen de distintas capas de redes neuronales. En concreto destaca una de ellas: la capa de **self-attention**. Esta capa de *self-attention* se describe en la siguiente imagen.

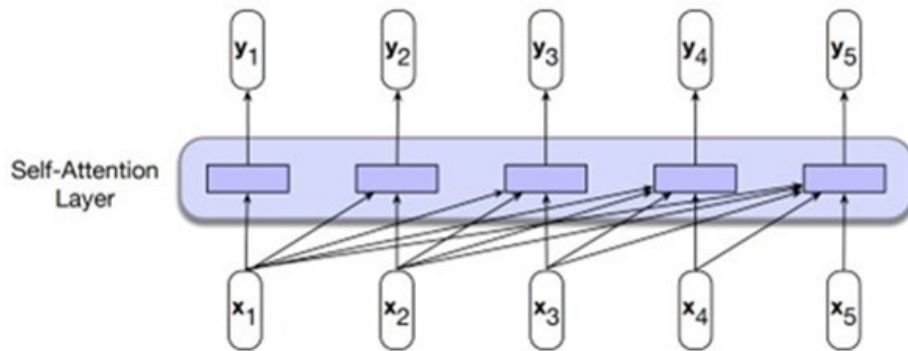


Figura 26. Capa de *self-attention*. Fuente: Jurafsky, 2021.

Como se puede observar, esta capa recibe un vector de entrada  $x$  y genera un vector de salida  $y$  del mismo tamaño. Cada valor

$y_i$  que se predice usa información del valor de entrada actual

$x_i$ , pero también recibe la información de todos los estados anteriores. Estos valores de entrada se expresan mediante un vector de *embeddings*, de manera análoga al resto de modelos vistos hasta ahora. A diferencia de las RNN, cada uno de los valores de salida se trata de manera independiente desde el punto de vista computacional, de manera que se pueden procesar en paralelo.

El cálculo de cada uno de esos valores

$y_i$  se lleva a cabo tal como indica la siguiente imagen, donde se muestra un ejemplo para el cálculo de

$y_3$ .

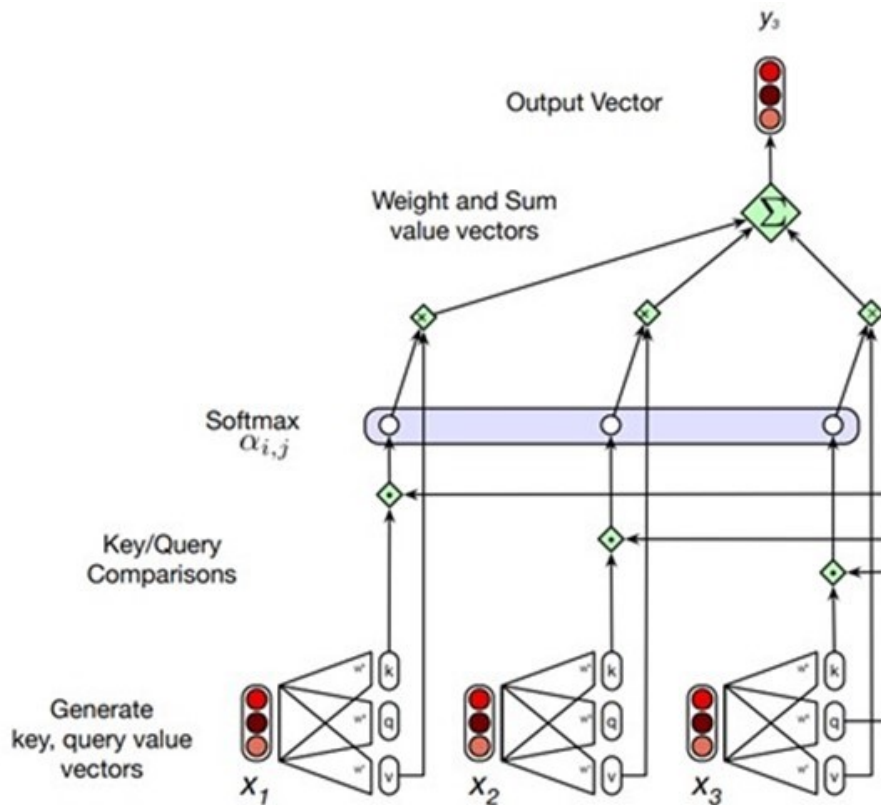


Figura 27. Cálculo de  $y_3$  con el modelo de attention. Fuente: Jurafsky, 2021.

Como se puede observar, para el cálculo se parte de los datos de entrada, se obtiene desde ellos el vector de *embeddings* correspondiente para cada uno, y luego se usan para generar 3 vectores  $q$ ,  $k$  y  $v$ , que, combinados entre sí, generan la salida final. Estos 3 vectores son:

- El **foco actual** del modelo de *attention*, que para una salida  $y_i$  hace referencia a  $x_i$ . Este se denomina **query (q)**.
- Ese foco actual  $q$  es un vector que se va a comparar tanto con el elemento actual como con los elementos previos mediante un producto escalar entre los vectores que da como resultado el valor de su similitud. Así, Las entradas de todos los elementos dan lugar a los vectores **k (key)**, que se multiplican con **q** en cada elemento, dando

como resultado un valor numérico. Todos estos valores concatenados dan lugar a un vector que recibe la capa *softmax* con la que se genera un vector de probabilidades final, con un valor

$\alpha_{i,j}$  por cada elemento de entrada.

- Los valores de entrada también son usados directamente en el cálculo de la salida, en combinación con los resultados de los dos puntos anteriores (**value, v**). Así, cada vector de *embeddings* de entrada se usa para generar un vector  $v$  que se multiplica por la salida de la capa *softmax*. La suma de los elementos de este vector final da como lugar la salida de la capa de *attention* para ese valor

$y_i$ .

Trasladado a ecuaciones,

$y_i$  se obtiene de la siguiente manera, en base a las componentes

$\alpha_{i,j}$  de la salida de la capa *softmax*, y a los vectores

$v_j$  obtenidos desde las entradas de los distintos elementos del modelo:

$$y_i = \sum_{j \leq i} \alpha_{ij} v_j$$

Por otro lado, los productos escalares entre el foco actual

$q_i$  y cada uno de los elementos de entrada expresados mediante un vector

$k_j$  dan un resultado como el siguiente (donde se ha normalizado en función del tamaño del vector  $k$ ):

$$score(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}$$

El cálculo de Q, K y V expresado matricialmente, donde se tienen los vectores para todos los elementos de entrada, son las siguientes:

$$Q = XW^Q$$

$$K = XW^K$$

$$V = XW^V$$

La matriz X corresponde a la matriz de *embeddings*, donde cada fila es el vector de *embeddings* asociado a uno de los elementos de entrada. Multiplicando esta matriz por las matrices

$W^Q$ ,

$W^K$  y

$W^V$  se obtienen, respectivamente, los valores de las matrices Q, K y V. De esta manera, la salida del modelo de *self-attention* se puede expresar de la siguiente manera:

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

Ahora bien, para el cálculo de un determinado valor

$y_i$  sólo se usa la información del elemento actual y de los elementos previos, pero no de los elementos siguientes. Por este motivo, ese producto matricial anterior no se puede realizar directamente así, ya que si no se tendría en cuenta toda la secuencia para cada elemento. Para evitar esto (y poder, entre otras cosas, usar estos modelos para tareas como LM), los elementos de la matriz resultante del producto matricial de  $QK^T$  que van después del foco actual reciben un valor arbitrario que les hace ser nulos, de manera que su información no se use en el cálculo. Esto se ve en la siguiente imagen.

N

$q1 \cdot k1$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$q2 \cdot k1$	$q2 \cdot k2$	$-\infty$	$-\infty$	$-\infty$
$q3 \cdot k1$	$q3 \cdot k2$	$q3 \cdot k3$	$-\infty$	$-\infty$
$q4 \cdot k1$	$q4 \cdot k2$	$q4 \cdot k3$	$q4 \cdot k4$	$-\infty$
$q5 \cdot k1$	$q5 \cdot k2$	$q5 \cdot k3$	$q5 \cdot k4$	$q5 \cdot k5$

N

Figura 28. Ejemplo del caso de la matriz resultante de multiplicar  $Q$  por  $K^T$ , con  $Q$  y  $K$  de dimensión  $N \times d$ . Fuente: Jurafsky, 2021.

## Modelo de *transformers*

Una vez conocido el funcionamiento de la capa *self-attention*, se puede entender el funcionamiento del modelo de *transformers*. El esquema de este modelo se describe en la siguiente imagen:

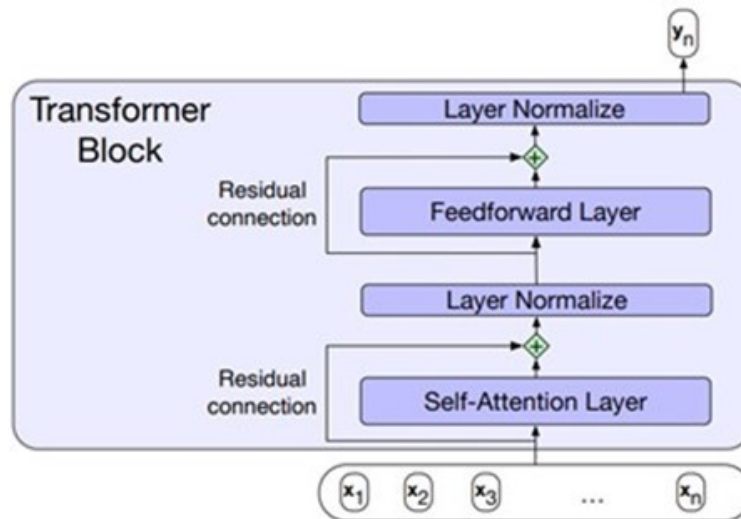


Figura 29. Esquema del modelo de *transformers*. Fuente: Jurafsky, 2021.

En este modelo, se recibe una entrada que pasa por la capa *self-attention* para generar una primera salida intermedia. Esta salida intermedia se combina de nuevo con la información de entrada mediante la **conexión residual**, de manera que el vector de entrada de la capa *self-attention* y la salida de dicha capa se combinan sumando sus valores.

Esta aproximación de la conexión residual es útil para no perder la información de las primeras capas de una red neuronal, recuperándola en fases posteriores al combinarla con la información que sale del propio procesamiento de la red.

Este vector pasa por una **capa de normalización** para asegurar que el vector tiene valores en una escala similar. El vector resultante sirve como entrada de una capa *feed-forward*, que produce un vector de salida que se vuelve a combinar con su entrada mediante otra conexión residual. El vector final se normaliza de nuevo, generando la salida final del modelo.

Expresándolo mediante ecuaciones se tiene:

$$\begin{aligned} \mathbf{z} &= \text{LayerNorm}(\mathbf{x} + \text{SelfAttn}(\mathbf{x})) \\ \mathbf{y} &= \text{LayerNorm}(\mathbf{z} + \text{FFNN}(\mathbf{z})) \end{aligned}$$

La **normalización** se suele hacer mediante el cálculo de la **media** y la **desviación típica del vector**, de manera que, por ejemplo, para un vector  $\mathbf{x}$  sería:

$$\hat{\mathbf{x}} = \frac{(\mathbf{x} - \mu)}{\sigma}$$

Con

$\mu$  la media, y con

$\sigma$  la desviación típica. Además, en lugar de usar directamente este resultado como salida de la capa de normalización, se puede ajustar con unos parámetros

$\gamma$  y

$\beta$ , que también serían parámetros que se ajustarían en la red durante la optimización del error, tal que:

$$\text{LayerNorm} = \gamma \hat{\mathbf{x}} + \beta$$

## Multiheaded Attention

En algunas ocasiones, un modelo simple de *self-attention*, como el que se ha visto previamente, no es suficiente para capturar la complejidad de los datos de entrada.

Por este motivo, existen alternativas más avanzadas, como son los **modelos *multihedaded***, donde, para una misma entrada, se usan distintos modelos *self-attention* (denominados *heads*), y la salida de todos ellos se combina en un único vector para generar con él la salida final. La imagen siguiente muestra esta arquitectura.

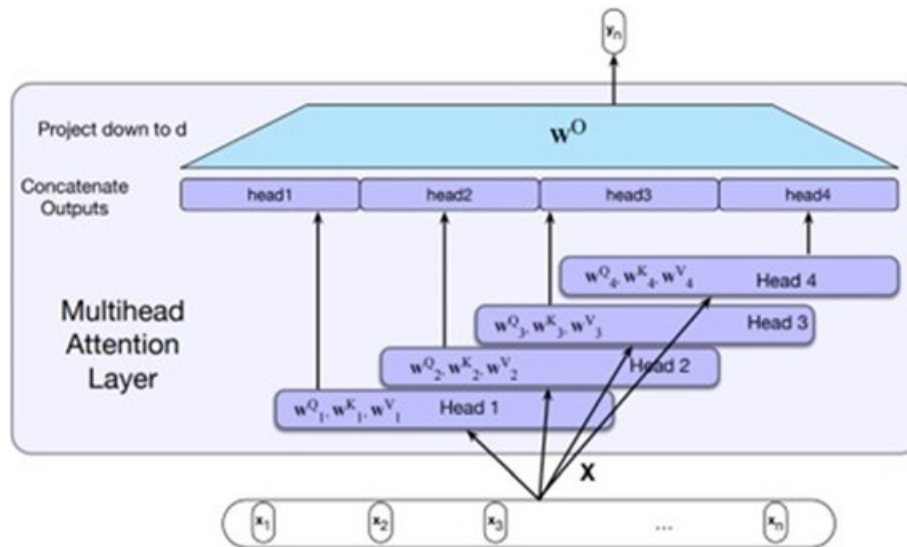


Figura 33. Modelo multiheaded attention con 4 heads. Fuente: Jurafsky, 2021.

De manera genérica, para  $i$  heads, se tienen las siguientes ecuaciones:

$$\begin{aligned}
 \text{MultiHeadAttn}(X) &= (\text{head}_1 \oplus \text{head}_2 \dots \oplus \text{head}_h) W^O \\
 Q &= XW_i^Q ; K = XW_i^K ; V = XW_i^V \\
 \text{head}_i &= \text{SelfAttention}(Q, K, V)
 \end{aligned}$$

## Embeddings posicionales

En los modelos vistos previamente, cuando los datos de entrada son palabras, estas se expresan con su correspondiente vector de *embeddings*, el cual muchas veces toma como valor el resultante de modelos como Skip-gram. Ahora bien, estos vectores representan el valor semántico de la palabra en cuestión, pero no tienen en cuenta otro tipo de información, como **la posición de la palabra dentro de la**

**secuencia.** Por este motivo, existen también los ***embeddings* posicionales**, donde se tienen vectores densos con los que representa la posición de las palabras, y se captura información como que la palabra de la posición 4 es más cercana a la 3 que a la 19.

El uso de *embeddings* posicionales en los modelos de transformers es importante ya que su arquitectura, a diferencia de las RNN, no conserva la información relativa al orden de las palabras (se tiene información de la secuencia previa a una palabra en concreto, pero no del orden de las palabras como tal).

Este problema se solventa con la combinación de los dos *embeddings* para generar el vector de entrada del modelo.

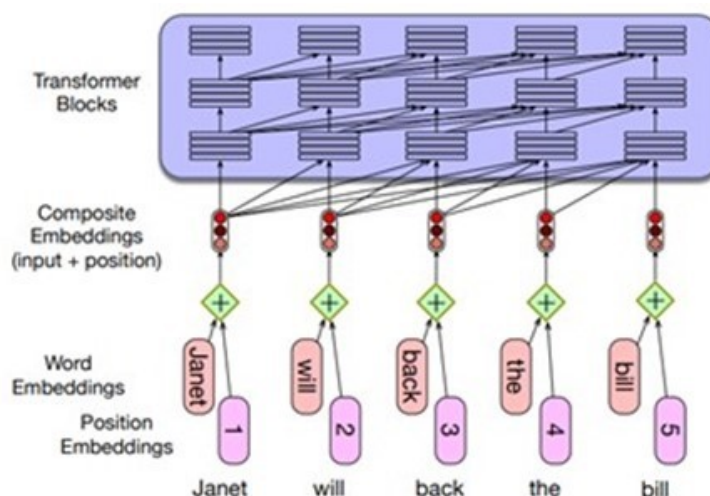


Figura 35. Ejemplo de combinación de *word embeddings* junto con *embeddings* posicionales. Fuente: Jurafsky, 2021.

## Transformers como modelos de lenguaje

Los modelos de *transformers* pueden usarse, como ya se ha comentado, para **problemas de LM**. Para una secuencia de entrada, se usa una capa con el modelo

de *transformers* que recibe como entrada los *embeddings* de la palabra en un estado concreto y de toda la secuencia previa, y con ello se construye un modelo en el que el vector de salida corresponda a la distribución de probabilidad sobre todas las palabras del vocabulario, donde se trate de predecir la palabra que seguirá a la secuencia. Esto se muestra en la imagen siguiente:

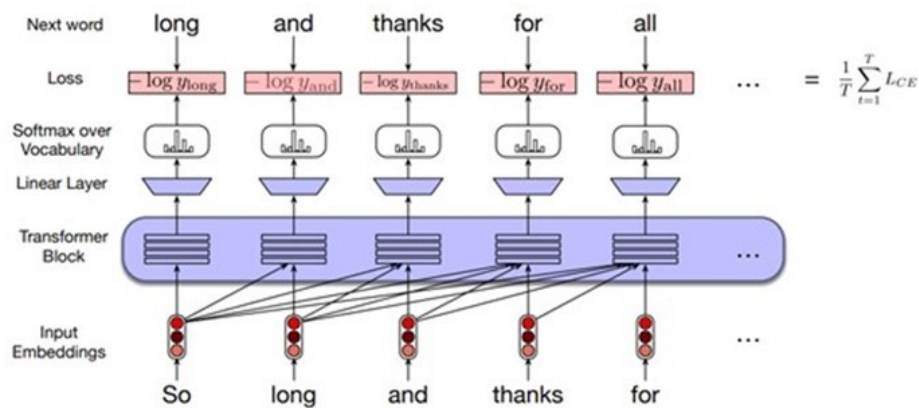


Figura 36. Uso de *transformers* para LM. Fuente: Jurafsky, 2021.

Para completar el estudio de esta sección y ver más información sobre los *transformers*, puedes leer el Capítulo 9 del libro, pág. 17-25: Jurafsky, D. y Martin, J. H. (2021). *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition and Computational Linguistics*. Prentice-Hall. <https://web.stanford.edu/~jurafsky/slp3/9.pdf>

## 8.5. BERT

Los algoritmos que se han visto hasta ahora para generar *word embeddings* tienen en común una cosa: usando la información de un corpus, se generan **representaciones estáticas** de las palabras mediante vectores dentro de un espacio vectorial. Es decir, se busca modelar el significado de las palabras desde una perspectiva general de su uso, pero no desde el contexto concreto de una oración. Ahora bien, en muchas ocasiones es importante considerar el contexto concreto de una palabra para entender su significado. Por ejemplo:

- ▶ María buscó el gato entre sus herramientas para poder arreglar el coche.
- ▶ María buscó a Garfield, su gato doméstico, para darle de comer.

Ambas oraciones comparten una serie de palabras, como es el caso de «gato». Sin embargo, su significado cambia mucho debido al contexto concreto de uso: en el primer caso hace referencia a una herramienta y en el segundo a un animal. Es por este motivo que aparecen los ***embeddings contextuales***, con los que se busca obtener un vector de *embeddings* que represente el significado de las palabras, teniendo en cuenta tanto el conocimiento global (el corpus en general) como el local (el contexto de uso). Una forma de obtener estos *embeddings* es mediante el uso del **modelo BERT** (Bidirectional Encoder Representations from Transformers) (Devlin et al., 2019).

## Transformers bidireccionales

El modelo BERT se basa en el concepto de **transformers bidireccionales**, que tienen en cuenta no sólo el contexto de la secuencia de izquierda a derecha, sino también de derecha a izquierda (como ocurría con las RNN bidireccionales). De esta manera, la capa de *self-attention* sería como muestra la siguiente imagen:

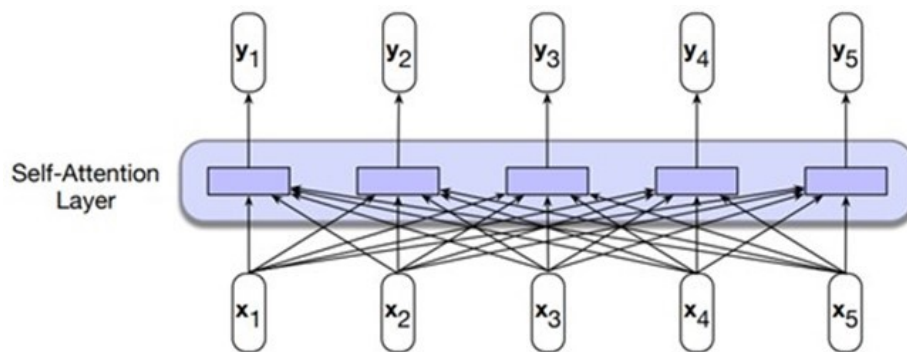


Figura 37. Capa de *self-attention* bidireccional. Fuente: Jurafsky, 2021.

Esta arquitectura no seguiría el esquema causal de izquierda a derecha con el que construir un LM para poder predecir cómo continuará una frase, pero sirve para otras tareas, como la construcción de *embeddings* contextuales. Su construcción es trivial con lo visto hasta ahora: sería el mismo cálculo matricial que en el caso no bidireccional, eliminando el paso de enmascarar la información de la matriz resultante de

$QK^T$  asociada a las palabras siguientes a cada palabra de la secuencia. De esta manera, sería:

N

q1•k1	q1•k2	q1•k3	q1•k4	q1•k5
q2•k1	q2•k2	q2•k3	q2•k4	q2•k5
q3•k1	q3•k2	q3•k3	q3•k4	q3•k5
q4•k1	q4•k2	q4•k3	q4•k4	q4•k5
q5•k1	q5•k2	q5•k3	q5•k4	q5•k5

N

Figura 38. Matriz completa resultante del producto

$QK^T$ . Fuente: Jurafsky, 2021.

Con esto, la arquitectura de BERT sigue el esquema estándar de los *transformers*, con esta particularidad de las capas de *self-attention* bidireccionales y con las siguientes particularidades:}

- En lugar de usar palabras como dato de entrada se usan **subpalabras**. Estas se obtienen mediante un proceso de tokenización con el algoritmo *WordPierce* (Schuster and Nakajima, 2012). Así, en lugar de usar palabras, se usan partes de esas palabras. Un ejemplo de salida de este algoritmo se ve en la siguiente imagen.



Figura 39. Ejemplo de tokenización en subpalabras con WordPiece. Fuente: Google Research, 2021.

- ▶ Las capas ocultas tienen una dimensionalidad de 768.
- ▶ Se usan doce bloques de *transformers*, con doce capas *multihead attention*.

De esta manera, se tiene un modelo de 100M de parámetros.

## Entrenamiento del modelo

Dado que en este caso no se busca construir un modelo que prediga la siguiente palabra de la secuencia (ajustando sus parámetros en función de cómo de cercana sea la distribución de probabilidad a la referencia de la salida), se plantea lo siguiente: omitir algunas palabras de la frase, y que el modelo trate de predecirlas, usando el resto de la secuencia (palabras previas y posteriores). Por ejemplo, se puede partir de la frase:

- ▶ María buscó a Garfield, su gato doméstico, para darle de comer.

Y tener:

- ▶ María buscó a Garfield, su \_\_\_\_\_ doméstico, para darle de comer.

De esta manera, el modelo trataría de predecir la palabra «gato» dado el resto de las palabras de la secuencia. La manera con la que se lleva a cabo esto en BERT es mediante la selección de las frases de entrenamiento, y sobre cada una de ellas, seleccionar un subconjunto de tokens de entrada para usar en el entrenamiento

(tratar de predecirlos usando los tokens conocidos, que sería el resto de ellos). BERT usa un 15 % de los tokens totales para este subconjunto a predecir. Sobre él se hace lo siguiente:

- ▶ El 80 % de ellos se enmascaran cambiándolos por un token [MASK], de manera que el modelo trate de predecir su valor usando la información de los tokens conocidos.
- ▶ El 10 % de ellos se reemplazan por una palabra aleatoria del vocabulario (en base a las probabilidades de los unigramas de los tokens).
- ▶ El 10 % restante de las palabras elegidas para entrenar el modelo se dejan sin cambiar.

El motivo de combinar tokens con [MASK] junto con tokens con palabras no enmascaradas es que el modelo se pueda utilizar para obtener predicciones sobre palabras aun cuando no tengan ese [MASK] (por ejemplo, para cuando se usa para hacer *fine-tuning* en el ámbito de *transfer learning*). Se usan tanto palabras aleatorias como las propias palabras en esas predicciones sin [MASK] para que haya un cierto sesgo en el modelo hacia la palabra en cuestión. Un ejemplo de entrenamiento y predicción con BERT se ve en la imagen siguiente. Como entrada, BERT usa también una combinación de los *embeddings* originales de esos tokens junto con su *embedding* posicional. Estos *embeddings*, de nuevo, se han obtenido previamente a nivel de subpalabras.

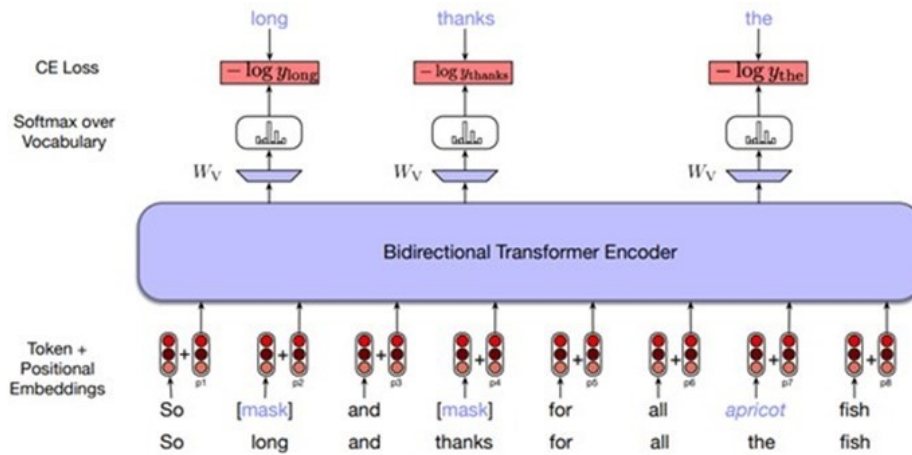


Figura 40. Ejemplo de entrenamiento en BERT. Fuente: Jurafsky, 2021.

Sobre una secuencia concreta, el modelo trata de predecir algunos de los tokens, que pueden estar enmascarados con un token genérico [MSK], o pueden estar reemplazados por una palabra aleatoria. También pueden estar sin cambiar con sus valores originales. En base a la probabilidad predicha sobre el vocabulario frente a la palabra real, se ajustan los valores del modelo.

## Embeddings contextuales

Teniendo el modelo de BERT ya entrenado, se puede utilizar para obtener ***embeddings* contextuales de palabras**. Con estos *embeddings* se obtienen representaciones vectoriales de palabras en función de la secuencia de texto donde aparecen. Por ejemplo, dada una secuencia de tokens

$(x_1, \dots, x_n)$ , si se quiere obtener el *embedding* del token

$x_i$ , esta se puede obtener usando esa secuencia como entrada del modelo, y obteniendo el vector

$y_i$  correspondiente a la última capa. Se puede también usar la salida de distintas capas del modelo con esa entrada, y construir el vector final de *embeddings* como una media de esos vectores.

## *Transfer learning*

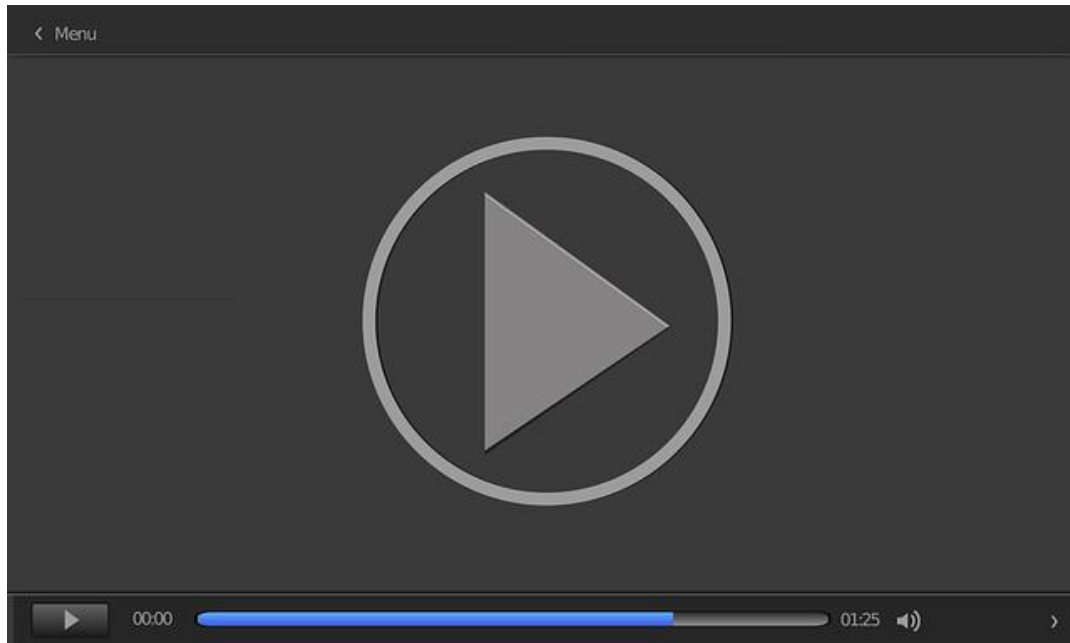
La tarea de entrenar un modelo como BERT es muy costosa computacionalmente, ya que se utilizan corpus de entrada de gran volumen de cara a poder modelar lo mejor posible el conocimiento de una lengua. Por este motivo, es habitual trabajar con modelos de BERT ya pre entrenados, y utilizarlos para distintas tareas de PLN según se necesiten. Así, se puede usar el modelo como entrada de otro proceso, como por ejemplo para obtener directamente los *embeddings* de una oración. Esto se relaciona con el concepto de ***transfer learning***, donde el conocimiento inferido por un modelo en su entrenamiento en una tarea específica se puede trasladar a otra.

---

Para completar el estudio sobre *embeddings* contextuales, BERT y *transfer learning* puedes leer el Capítulo 11 del libro: Jurafsky, D. y Martin, J. H. (2021). *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition and Computational Linguistics*. Prentice-Hall. <https://web.stanford.edu/~jurafsky/slp3/11.pdf>

---

En el vídeo *Transformers y BERT* se verá cómo se puede modelar el lenguaje gracias al aprendizaje profundo, viendo arquitecturas novedosas como es el caso de los *transformers*.



08.02. Transformers y BERT

---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=54618c77-285a-437c-86fc-af5b014a5465>

---

## 8.6. Referencias bibliográficas

Aggarwal, C. (2018). *Machine Learning for Text*. Springer Cham.

Bojanowski, P., Grave E., Joulin A. y Mikolov, T. (2017). *Enriching word vectors with subword information*. *TACL*, 5:135–146

Devlin, J., Chang, M.-W., Lee, K. y Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. *NAACL HLT*.

Graves, A., Jaitly, N., y Mohamed, A. R. (2013, December). *Hybrid speech recognition with deep bidirectional LSTM*. In 2013 IEEE workshop on automatic speech recognition and understanding (pp. 273-278). *IEEE*.

Jurafsky, D. y Martin, J. H. (2021). *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition and Computational Linguistics*. Prentice-Hall.

Schuster, M. y Nakajima, K. (2012). *Japanese and korean voice search*. *ICASSP*.

### Hugging Face

Hugging Face. <https://huggingface.co/>

Puedes ver ejemplos de cómo trabajar con BERT usando Python con la librería Hugging Face y con los tutoriales disponibles en su página web.

1. Indicar cuál de estas afirmaciones es verdadera respecto a los modelos de representación vectorial:

- A. Un modelo de *word embeddings* siempre va a generar vectores de mayor dimensión que uno basado en bolsas de palabras.
- B. Las representaciones vectoriales basadas en bolsas de palabras son densas, mientras que las basadas en *word embeddings* son dispersas.
- C. Con un modelo BoW se puede ver la relación entre palabras, a diferencia de un modelo de *word embeddings*.
- D. Un modelo de *word embeddings* permite ver la similitud entre palabras e incluso hacer operaciones con los vectores que recogen el significado.

2. El modelo de Skip-gram:

- A. Infiere el significado de una palabra usando un modelo que trata de predecir el contexto (palabras vecinas) dada esa palabra.
- B. Infiere el significado de una palabra usando un modelo que trata de predecir la palabra dado el contexto (palabras vecinas) de esa palabra.
- C. Infiere el significado de una palabra construyendo un vector que recoge el número de veces que aparecen algunas palabras en el texto.
- D. Se caracterizan por usar sólo representaciones binarias para indicar las palabras que aparecen en un determinado texto.

3. El modelo de CBOW:
- A. Infiere el significado de una palabra usando un modelo que trata de predecir el contexto (palabras vecinas) dada esa palabra.
  - B. Infiere el significado de una palabra usando un modelo que trata de predecir la palabra dado el contexto (palabras vecinas) de esa palabra.
  - C. Infiere el significado de una palabra construyendo un vector que recoge el número de veces que aparecen algunas palabras en el texto.
  - D. Se caracterizan por usar sólo representaciones binarias para indicar las palabras que aparecen en un determinado texto.
4. ¿Cuál de estos modelos servirá mejor para obtener la representación de palabras desconocidas no presentes en el corpus de entrenamiento?
- A. Skip-gram.
  - B. CBOW.
  - C. FastText.
  - D. TF-IDF.
5. Indica cuál de las siguientes afirmaciones es falsa respecto a la construcción de modelos de lenguaje (LM) con redes neuronales:
- A. Pueden usar la información de los *word embeddings* para representar los tokens de entrada.
  - B. Se construye usando tanto arquitecturas de redes neuronales recurrentes, como de redes *feed-forward*.
  - C. Las redes neuronales recurrentes bidireccionales son útiles para la construcción de LM y para tareas de PLN como el autocompletado de textos.
  - D. Se pueden usar estructuras como las LSTM para evitar perder información de las etapas más lejanas de la secuencia.

6. Indicar cuál de estas afirmaciones es verdadera respecto capa de *self-attention*:
- A. Usa la información del foco actual (*query*, *q*) y de la secuencia hasta una etapa en concreto (*key*, *k*) para construir un vector que sirve como entrada para una capa *softmax*. La salida de esta capa se corresponde a la salida de la capa *self-attention*.
  - B. Usa la información de la secuencia hasta una etapa en concreto (*key*, *k*) para construir un vector que sirve como entrada para una capa *softmax*. La salida de esta capa se combina de nuevo con la información de entrada (*value*, *v*) para obtener la salida de la capa *self-attention*.
  - C. Usa la información del foco actual (*query*, *q*) para construir un vector que sirve como entrada para una capa *softmax*. La salida de esta capa se combina de nuevo con la información de entrada (*value*, *v*) para obtener la salida de la capa *self-attention*.
  - D. Usa la información del foco actual (*query*, *q*) y de la secuencia hasta una etapa en concreto (*key*, *k*) para construir un vector que sirve como entrada para una capa *softmax*. La salida de esta capa se combina de nuevo con la información de entrada (*value*, *v*) para obtener la salida de la capa *self-attention*.
7. ¿Cuál es la estructura de un bloque de *transformers* básico?
- A. Capa *self-attention*, capa de normalización, capa *softmax*, capa normalización.
  - B. Capa *self-attention*, conexión residual, capa de normalización, capa recurrente con LSTM, conexión residual, capa normalización.
  - C. Capa *self-attention*, conexión residual, capa de normalización, capa *feed-forward*, conexión residual, capa normalización.
  - D. Capa *self-attention*, capa de normalización, capa *feed-forward*, conexión residual, capa normalización.

8. Indicar cuál de estas afirmaciones es verdadera sobre los *transformers*:
- A. Se usan *embeddings* contextuales en lugar de *word embeddings* como vectores de entrada para obtener mejores resultados.
  - B. La capa de normalización es siempre estática, y no tiene parámetros que se ajusten durante el entrenamiento del modelo.
  - C. Se pueden usar con una arquitectura *multiheaded*, donde se tienen varias capas *self-attention*, y cada una da una salida que se combina en un único vector final que sirve como salida de esa capa *multiheaded*.
  - D. No es posible usar *transformers* como modelos causales del lenguaje ya que siempre son bidireccionales y tienen por tanto información posterior de la secuencia con respecto a cada palabra.
9. Indicar cuál de estas afirmaciones es verdadera sobre el modelo BERT:
- A. Permite la generación de *embeddings* contextuales, donde se obtiene el *embedding* de una palabra sólo teniendo en cuenta el contexto local de la frase.
  - B. Permite la generación de *embeddings* contextuales, donde se obtiene el *embedding* de una palabra no sólo de manera global, sino teniendo también en cuenta el contexto local de la frase.
  - C. Es habitual entrenar el modelo BERT desde cero cada vez que se vaya a llevar a cabo una tarea de PLN, para tener en cuenta únicamente los textos asociados a dicha tarea.
  - D. BERT se entrena para predecir unos tokens del corpus de entrenamiento dado el resto de tokens, donde todos estos tokens se reemplazan por un token genérico [MSK].

10. Indicar cuál de estas afirmaciones es falsa sobre el modelo BERT:
- A. Usan como entrada tokens que contienen subpalabras, en vez de palabras.
  - B. Usan *transformers* bidireccionales, donde se tiene en cuenta tanto la información previa como posterior de la secuencia.
  - C. Al ser bidireccional sirve para tareas de NLP, como el autocompletado de textos.
  - D. Usa capas de *multiheaded attention*.