

# Unidad 4 - Arquitecturas de Modelos de Lenguaje

**UNR - TUIA - Procesamiento de Lenguaje Natural**

Docente teoría: Juan Pablo Manson - [jpmanson@gmail.com](mailto:jpmanson@gmail.com) - [LinkedIN](#)

## Introducción

El procesamiento del lenguaje natural (NLP, por sus siglas en inglés) es una disciplina fascinante y desafiante en el cruce entre la lingüística y la inteligencia artificial (IA), dedicada a enseñar a las máquinas a entender, interpretar y generar lenguaje humano. A lo largo de los años, la comunidad científica ha presenciado una evolución notable en las arquitecturas de modelos de lenguaje, que han avanzado en complejidad y eficacia para abordar las tareas de NLP. Este capítulo proporciona una exploración detallada de las arquitecturas de modelos de lenguaje que han marcado hitos significativos en esta trayectoria evolutiva.



Comenzamos con una exposición sobre las Redes Neuronales Recurrentes (RNN), que introdujeron la noción de memoria temporal en el modelado del

lenguaje, permitiendo a las máquinas retener información de los estados anteriores mientras procesan secuencias de texto. Sin embargo, las RNN sufren del desvanecimiento de gradiente, un obstáculo que las Redes Neuronales Recurrentes de Memoria a Largo Plazo (LSTM) lograron superar, ofreciendo una mejora sustancial en la retención de dependencias a largo plazo.

Posteriormente, exploraremos la arquitectura Seq2Seq que, junto con las LSTM, revolucionó las aplicaciones de traducción automática y otras tareas de secuencia a secuencia. La evolución natural llevó al surgimiento del mecanismo de Atención, que permitió a los modelos ponderar diferentes partes de una secuencia de entrada de manera diferencial, proporcionando un enfoque más refinado para capturar dependencias.

Finalmente, abordaremos la llegada de los modelos Transformer, que consolidaron el mecanismo de Atención en su núcleo, desencadenando una revolución en el desempeño de las tareas de NLP. Los Transformers han establecido nuevos estándares de estado del arte, impulsando una serie de innovaciones y modelos derivados que continúan expandiendo el horizonte de lo que es posible en el dominio del procesamiento del lenguaje natural.

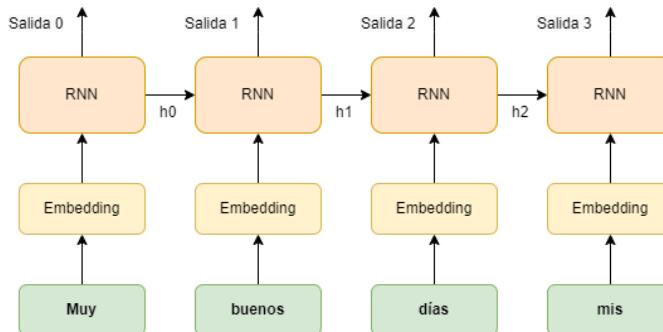
A través de este recorrido, este capítulo aspira a brindar una comprensión profunda de cómo cada una de estas arquitecturas ha contribuido a la evolución del NLP, y cómo han sentado las bases para los desarrollos futuros en esta área intrigante y en rápido cambio de la inteligencia artificial.

## 1. RNN (Redes Neuronales Recurrentes)

Las RNNs fueron introducidas por primera vez en la década de 1980 con la invención de la red Hopfield. Estas son un tipo de red neuronal diseñadas específicamente para procesar secuencias de datos. A diferencia de las redes neuronales tradicionales, las RNN son capaces de recordar información sobre entradas anteriores en la secuencia debido a su estructura recurrente. Este tipo de red es muy útil en tareas como el reconocimiento de voz, la traducción de idiomas, o cualquier otro problema donde los datos tengan un orden o contexto secuencial importante.

Las RNN se componen de una celda (o conjunto de celdas) que procesa un paso temporal a la vez y tiene la capacidad de "recordar" los resultados de los pasos anteriores. Esta característica permite que las RNN manejen secuencias de largo variable, lo cual es fundamental en tareas de lenguaje natural.

En NLP, con una capa de *embeddings*, convertimos las palabras de entrada (en este caso, "Muy", "buenos", "días", "mis") en vectores numéricos densos. Este paso es esencial porque las redes neuronales no pueden trabajar directamente con texto; necesitan números.



Los vectores de *embedding* se alimentan a través de una serie de celdas RNN. En el gráfico, vemos cuatro celdas RNN conectadas en secuencia. Cada celda toma dos entradas:

- El *embedding* de la palabra actual.
- El estado oculto (*h*) de la celda anterior, que contiene la información acumulada de todas las palabras procesadas hasta ese momento.

Esto significa que la primera celda recibe el embedding de la palabra "Muy" y un estado oculto inicial (*h*<sub>0</sub>), que suele estar inicializado a ceros. Luego, genera una salida (Salida 0) y un nuevo estado oculto (*h*<sub>1</sub>), que se pasa a la siguiente celda. El proceso se repite para las siguientes palabras: "buenos", "días" y "mis", donde cada celda RNN actualiza su estado oculto (*h*<sub>1</sub>, *h*<sub>2</sub>, *h*<sub>3</sub>, etc.) basado en la información previa y la palabra actual. Los estados ocultos son la clave de la capacidad de "memoria" de las RNN. Cada estado oculto (*h*) transporta la información sobre todas las palabras anteriores en la secuencia, permitiendo que la red recuerde el contexto de la oración.

Podemos ver cómo con este método nuestras predicciones dependen no solo de la entrada actual, sino también de los datos anteriores. Esta es la razón por la que las RNNs son un modelo adecuado para tratar con secuencias. Algunos problemas ampliamente conocidos resueltos por las RNNs:

- **Dependencias Temporales:** Algunos puntos de datos están relacionados entre sí según su orden. Por ejemplo, en el lenguaje, el significado de una oración puede cambiar drásticamente al reordenar sus palabras. Las RNNs están diseñadas para capturar estas dependencias, lo que las hace perfectas para tareas donde la secuencia de datos importa.
- **Entrada de Longitud Variable:** Las redes neuronales tradicionales funcionan bien cuando las entradas tienen un tamaño fijo, pero ¿qué pasa con oraciones de diferentes longitudes o datos de series temporales con diferentes pasos de tiempo? Las RNNs pueden manejar estos casos, ajustando su memoria interna para cada paso en la secuencia.
- **Análisis de Series Temporales:** Los datos recopilados a lo largo del tiempo, como los precios de las acciones o las lecturas del clima, a menudo exhiben patrones que las RNNs pueden detectar. Pueden aprender a predecir valores futuros basados en puntos de datos pasados.
- **Procesamiento de Lenguaje Natural (NLP):** Los idiomas son secuencias de palabras. Las RNNs destacan en tareas de NLP como traducción de idiomas, análisis de sentimientos y generación de texto. Entienden el contexto y el flujo de palabras, lo que les permite generar oraciones coherentes.
- **Reconocimiento de Voz:** Cuando hablamos, los sonidos que producimos son una secuencia. Las RNNs pueden decodificar esta secuencia de sonidos en texto, permitiendo sistemas de reconocimiento de voz precisos.

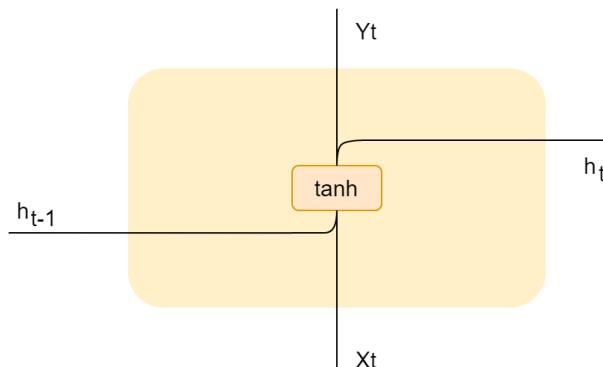
### **Concepto Básico de las RNNs**

Una RNN es como una red neuronal con memoria. Está diseñada para recordar y utilizar la información de los pasos anteriores mientras maneja el paso actual. Esta memoria es lo que hace especiales a las RNNs.

<https://www.youtube.com/embed/aL-EmKuB078?start=133&end=316>

En las redes neuronales tradicionales (redes feedforward), cada entrada se procesa de forma independiente, sin ningún conocimiento del pasado. Pero en las RNNs, la salida en cada paso no solo depende de la entrada actual, sino también de la información de los pasos anteriores. Esto permite que las RNNs

encuentren patrones y comprendan el orden de las cosas en los datos que llegan en una secuencia.



Representación simplificada de una RNN

En una celda RNN básica, la entrada actual y la salida anterior (o estado oculto anterior) se combinan y se pasan a través de una función de activación, generalmente la tangente hiperbólica (tanh), para producir el nuevo estado oculto (y la salida de la celda).

Matemáticamente, esto se puede expresar como:

$$h_t = \tanh(W_x \cdot x_t + W_h \cdot h_{t-1} + b)$$

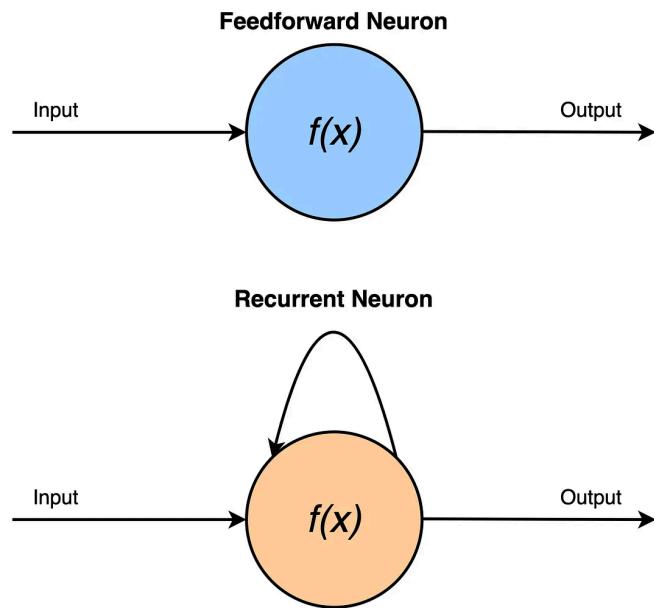
Donde:

- $h_t$  es el estado oculto en el tiempo  $t$ .
- $x_t$  es la entrada en el tiempo  $t$ .
- $W_x$  y  $W_h$  son matrices de pesos para la entrada y el estado oculto, respectivamente.
- $b$  es el término de sesgo.

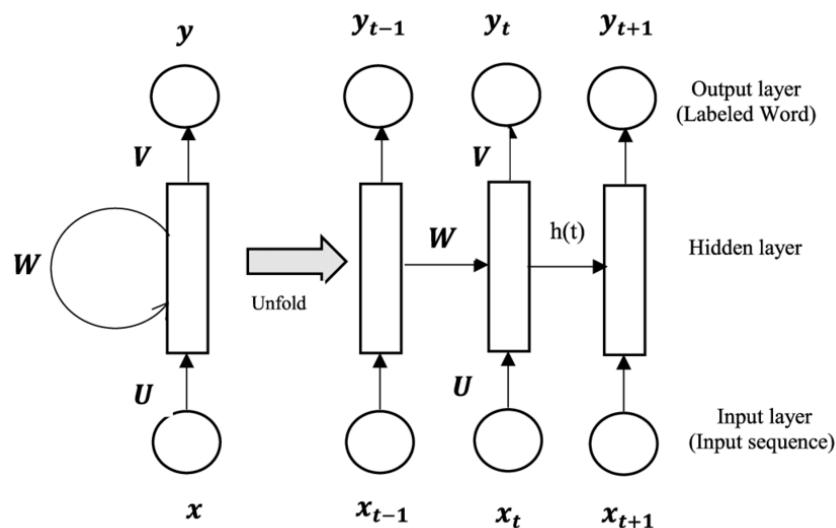
Diferencia con las Redes Neuronales Feedforward: Aquí hay una forma simple de entender la diferencia:

- Red Neuronal Feedforward: Imaginemos mirar cada palabra en una oración sin recordar las anteriores. Es como leer palabras en un orden aleatorio, sin entender el significado general.
- Red Neuronal Recurrente: Pensemos en leer una oración donde recordamos lo que hemos leído anteriormente. Esto nos ayuda a entender cómo se

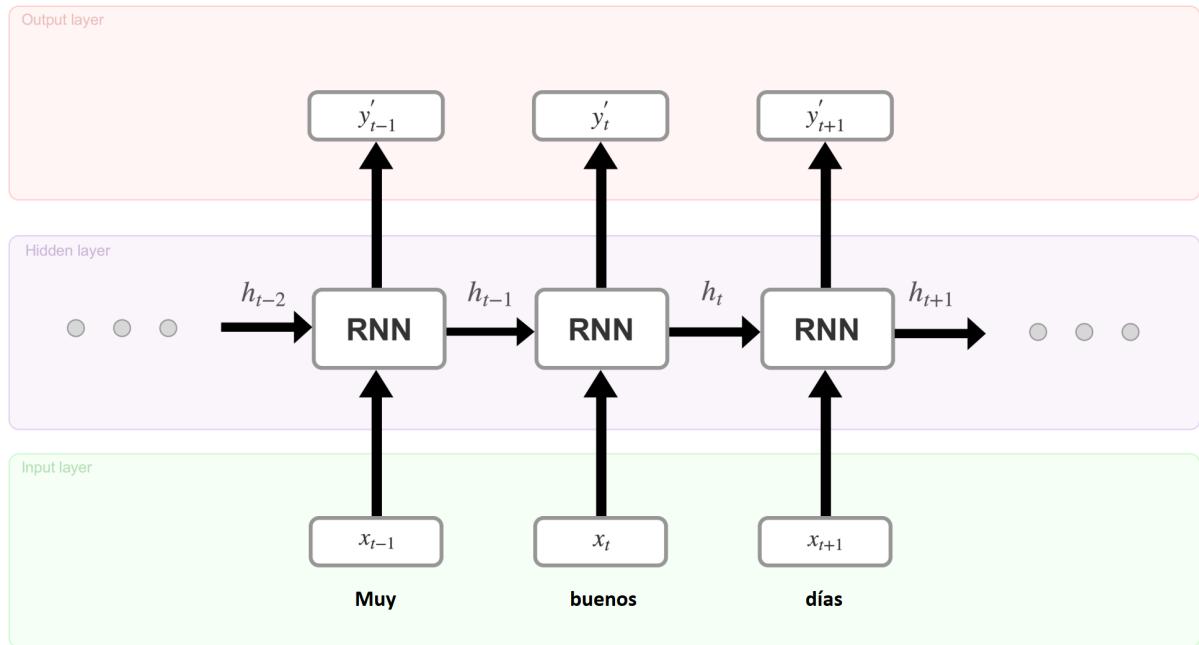
conectan las palabras y forman una oración coherente.



El "unfolding" (despliegue en español) es un concepto utilizado principalmente en el contexto de las Redes Neuronales Recurrentes (RNNs). Se refiere al proceso de representar y visualizar las operaciones temporales de una RNN a lo largo de varios pasos de tiempo:



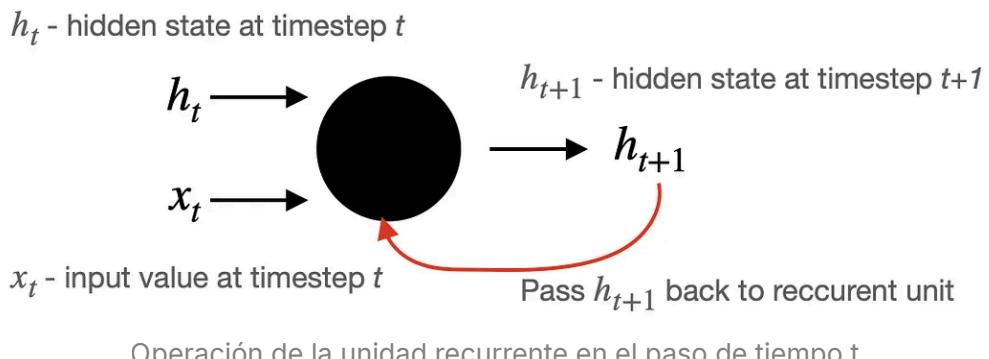
Veamos paso por paso, como funciona una red recurrente.



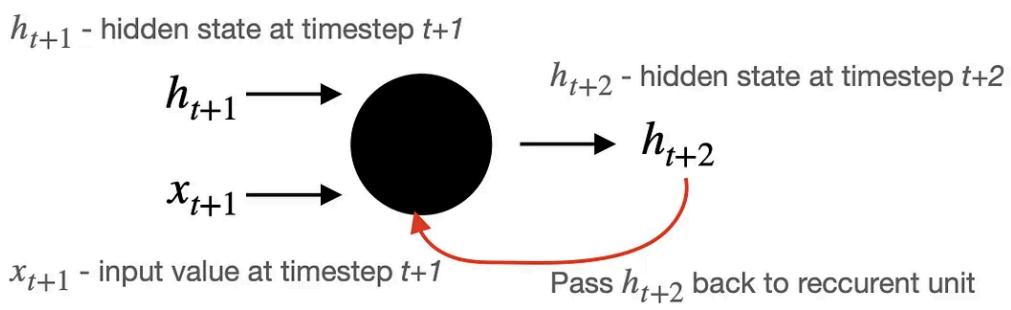
- 1. Codificación de la palabra en el tiempo actual:** La red comienza con la palabra en el paso de tiempo actual (por ejemplo, t-1). Esta palabra es convertida en un vector numérico a través de una capa de *embedding*. Esta capa traduce las palabras en vectores densos que capturan las relaciones semánticas entre palabras. En el paso de tiempo t-1, obtenemos el vector  $x_{t-1}$ , que representa la palabra "Muy" en este ejemplo.
- 2. Procesamiento en la celda RNN:** El vector  $x_{t-1}$ , que es la representación codificada de la palabra "Muy", se alimenta en la celda RNN. La celda RNN también recibe el estado oculto anterior  $h_{t-2}$  (siendo  $h_0$  un vector inicial, típicamente de ceros en el primer paso). Con estas dos entradas, la celda RNN calcula dos salidas:
  - Un nuevo estado oculto  $h_t$ , que es una representación del contexto hasta ese punto en la secuencia.
  - Una salida intermedia  $y'_{t-1}$ , que captura la información procesada en ese paso, pero **aún no es la palabra predicha**.
- 3. Actualización y avance en la secuencia:**
  - Después de procesar el paso de tiempo t-1, el estado oculto actualizado  $h_t$  se pasa al siguiente paso de tiempo t. Aquí, la siguiente palabra "buenos" es convertida en su vector correspondiente  $x_t$  mediante la capa de *embedding*, y el proceso se repite:  $x_t$  y  $h_t$  se introducen en la siguiente celda RNN para generar la siguiente salida  $y_t$  y el nuevo estado oculto  $h_{t+1}$ .

#### 4. Repetición del proceso hasta el final de la secuencia:

- Este proceso continúa para cada palabra en la secuencia. Las celdas RNN siguen actualizando los estados ocultos a medida que avanzan por la secuencia, permitiendo que el modelo mantenga el contexto a lo largo del tiempo.
- Una vez que se alcanza el último paso de la secuencia (en este caso, con la palabra "días"), el estado final  $h_{t+1}$  contiene una representación del contexto completo de la secuencia, que puede utilizarse para hacer predicciones o para tareas posteriores (como decodificar toda la secuencia en una traducción o generar la siguiente palabra de la frase).



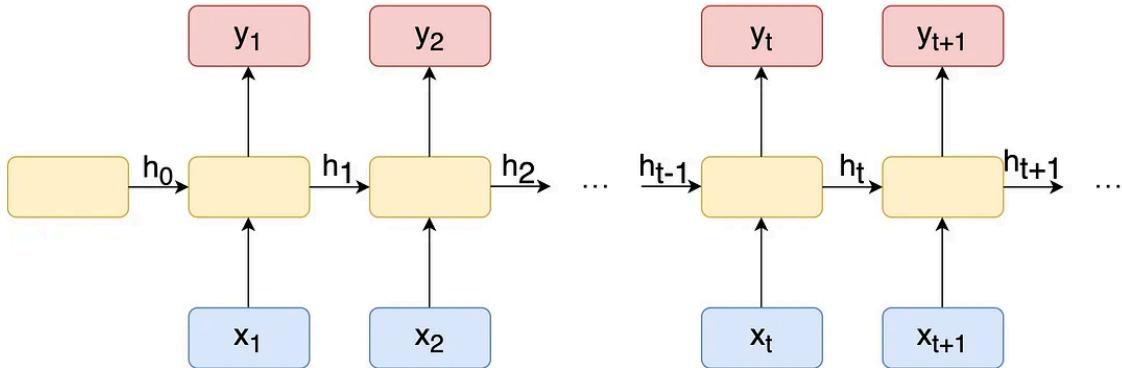
Operación de la unidad recurrente en el paso de tiempo  $t$ .



Operación de la unidad recurrente en el paso de tiempo  $t+1$ .

Podemos ver cómo cada uno de  $\{\dots, h_{t-1}, h_t, h_{t+1}, \dots\}$  contiene información sobre todas las entradas anteriores. El efecto de propagar los estados anteriores hacia la derecha, se llama "Forward Propagation". Esa propagación hace que las RNNs sean buenas para predecir la siguiente unidad en una secuencia.

## Forward Propagation

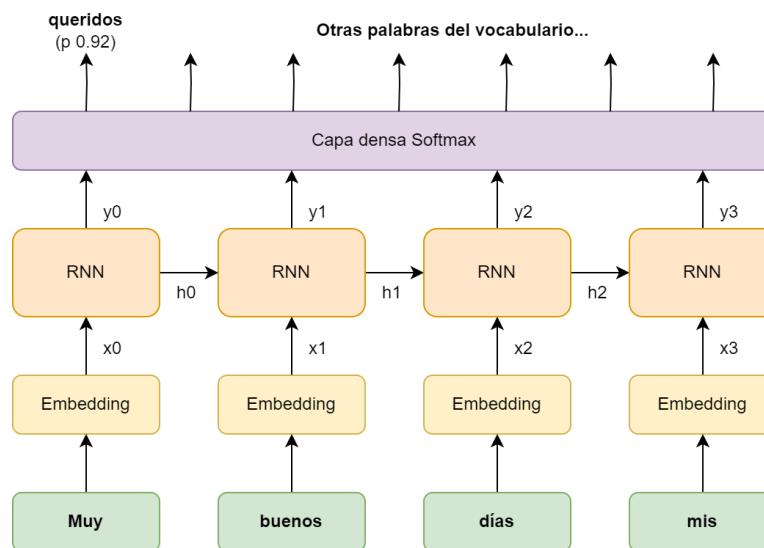


En el ejemplo que veremos a continuación, primero se realiza un preprocesamiento básico del texto, que incluye la tokenización y el padding de las secuencias.



El "padding" es una técnica utilizada en el procesamiento de secuencias para asegurar que todas las secuencias tengan la misma longitud. Esto se logra añadiendo valores adicionales (generalmente ceros) al principio o al final de las secuencias más cortas hasta que alcancen la longitud deseada.

Para que una RNN nos pueda predecir la siguiente palabra, añadiremos una capa densa, que con softmax, nos permite predecir la siguiente palabra. Esta capa de salida depende del vocabulario definido según nuestro conjunto de datos.



Luego, se define un modelo secuencial usando TensorFlow, con una capa de embedding, una capa RNN simple y una capa densa. Finalmente, se compila y se entrena el modelo:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Supongamos que tenemos el siguiente conjunto de datos de texto
data = ["hola cómo estas", "bien y vos", "muy bien gracias", "buen día"]

# Preprocesamiento
tokenizer = Tokenizer()
tokenizer.fit_on_texts(data)
sequences = tokenizer.texts_to_sequences(data)
print('\nsequences:', sequences)

vocab_size = len(tokenizer.word_index) + 1 # Agregamos 1 por el token de cer
max_length = max([len(seq) for seq in sequences])
padded_sequences = pad_sequences(sequences, maxlen=max_length, padding='post')
print('\nvocab_size:', vocab_size)
print('\npadded_sequences', padded_sequences)
print('\ninput_length', max_length)

# Separamos las secuencias en entradas y salidas
X = padded_sequences[:, :-1]
y = padded_sequences[:, -1]
print(f'\nX={X}, y={y}')

# Modelo RNN
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=10, input_length=max_length),
    SimpleRNN(32),
    Dense(units=vocab_size, activation='softmax')
])
```

```

# Compilación y entrenamiento
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.summary()
model.fit(X, y, epochs=200)

# Guardar el modelo en un archivo
model.save("mi_modelo.h5")

```

Y la salida será como la siguiente:

```
sequences: [[2, 3, 4], [1, 5, 6], [7, 1, 8], [9, 10]]
```

```
vocab_size: 11
```

```
padded_sequences [[ 2 3 4]
[ 1 5 6]
[ 7 1 8]
[ 9 10 0]]
```

```
input_length 3
```

```
X=[[ 2 3]
[ 1 5]
[ 7 1]
[ 9 10]], y=[4 6 8 0]
```

```
Model: "sequential_1"
```

---

Layer (type)	Output Shape	Param #
<hr/>		
embedding_1 (Embedding)	(None, 2, 10)	110
simple_rnn_1 (SimpleRNN)	(None, 32)	1376
dense_1 (Dense)	(None, 11)	363
<hr/>		
Total params: 1849 (7.22 KB)		
Trainable params: 1849 (7.22 KB)		

---

```
Non-trainable params: 0 (0.00 Byte)
```

```
Epoch 1/200
```

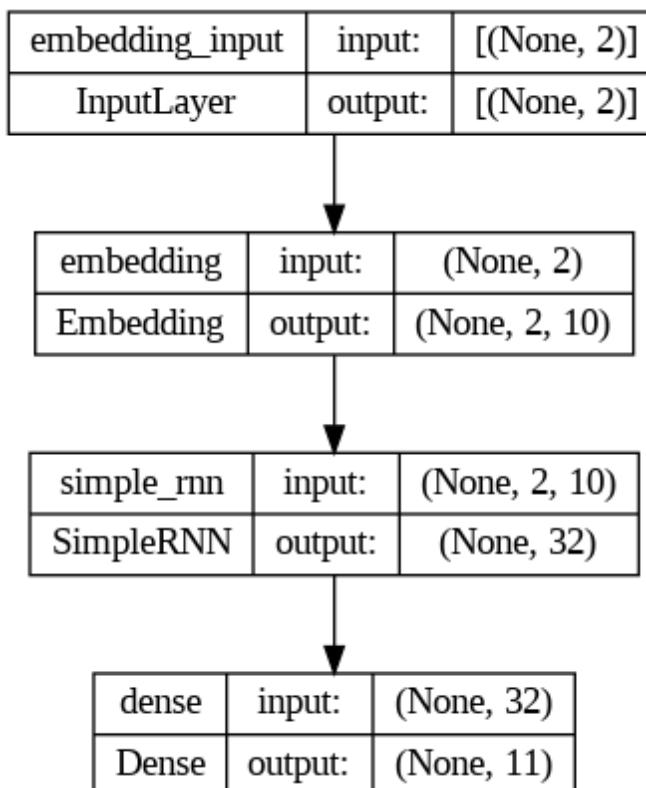
```
1/1 ━━━━━━━━━━ 2s 2s/step - accuracy: 0.0000e+00 - loss: 2.3912
```

```
.....
```

```
Epoch 200/200
```

```
1/1 ━━━━━━━━━━ 0s 57ms/step - accuracy: 1.0000 - loss: 0.0207
```

Gráficamente el modelo queda del siguiente modo:



Allí vemos los parámetros de la red y sus diferentes capas. Una vez que grabamos el modelo, podemos utilizarlo para recuperar más tarde nuestro modelo ya entrenado.

El modelo que creamos, inicialmente consiste en tres capas principales:

`Embedding`, `SimpleRNN`, y `Dense`. A continuación, veamos qué hacen cada una de estas capas:

### Capa Embedding

```
Embedding(input_dim=vocab_size, output_dim=10, input_length=max_length-1
```

- La capa `Embedding` se utiliza para convertir los tokens de entrada en vectores densos de tamaño fijo que pueden ser procesados por la red neuronal. Es una forma de realizar el "word embedding".
- El parámetro `input_dim`: Es el tamaño del vocabulario más 1 (para el índice 0, usualmente reservado para padding o tokens desconocidos). Define cuántos vectores de embedding únicos necesita aprender la capa. El cálculo `vocab_size = len(tokenizer.word_index) + 1` es la forma estándar de calcularlo. La capa internamente tendrá una matriz de pesos de tamaño `(input_dim, output_dim)`. Rango de Índices: Técnicamente, los índices que la capa puede recibir van desde `0` hasta `input_dim - 1`. En nuestro caso, como `tokenizer` asigna índices desde 1, los índices de palabras van de 1 a `input_dim - 1` (que es `len(tokenizer.word_index)`) y el índice 0 se usa para el padding. La capa debe poder manejar todos estos `input_dim` posibles valores.
- El segundo argumento `output_dim = 10` es la cantidad de dimensiones que tendrán nuestros vectores luego del embedding.
- `input_length` es la longitud de las secuencias de entrada. Es igual a la longitud máxima de las secuencias menos uno, porque estamos usando todas las palabras en una secuencia, excepto la última, como entrada para el modelo. El `input_length` determina cuántas veces se "despliega" la celda RNN en el tiempo durante el procesamiento de una secuencia. Sin embargo, es importante notar que este "unfolding" es conceptual y ocurre durante el procesamiento de la secuencia, no es una característica física del modelo.
- El número de parámetros en la capa de Embedding se calcula multiplicando `vocab_size X output_dim = 110`



En el ejemplo de código proporcionado, la red neuronal recurrente (RNN) aprende las representaciones vectoriales de las palabras (embeddings) partiendo completamente desde cero. Esto significa que la capa `Embedding` se inicializa de forma aleatoria y, a través del entrenamiento con el pequeño conjunto de datos ("`hola cómo estas`", etc.), ajusta estos vectores específicamente para la tarea de predecir la siguiente palabra dentro de esas frases particulares. Es un enfoque válido, especialmente útil para adaptar los embeddings a un vocabulario o contexto muy específico si se dispone de suficientes datos de entrenamiento.

Sin embargo, es fundamental destacar que este no es el único método disponible. En muchos escenarios prácticos, especialmente cuando se trabaja con conjuntos de datos más limitados o se busca aprovechar un conocimiento lingüístico más general, se recurre a **embeddings pre-entrenados** (como Word2Vec o GloVe). Estos vectores se han aprendido previamente sobre corpus de texto masivos y pueden usarse para inicializar la capa `Embedding`, transfiriendo ese conocimiento general a la nueva tarea. Luego, se puede optar por mantener estos embeddings fijos o permitir que se ajusten (fine-tuning) durante el entrenamiento del modelo específico.

## Capa SimpleRNN

### SimpleRNN(32)

- `SimpleRNN` es una capa que implementa una Red Neuronal Recurrente (RNN) simple. El argumento `32` se refiere a la dimensionalidad del espacio de salida de la capa, lo que significa que cada salida de la capa RNN en cada paso de tiempo será un vector de 32 dimensiones.
- La capa SimpleRNN procesa la secuencia de embeddings producida por la capa Embedding, manteniendo un estado oculto que se actualiza en cada paso de tiempo.
- La cantidad de parámetros = (`input_dim` + `units` + 1) \* `units`. Donde `input_dim` es la dimensionalidad de la entrada (en este caso, `10`, el tamaño del

embedding), `units` es el número de unidades en la capa SimpleRNN (`32` en este caso) y el `+1` es para el término de sesgo (bias). Por lo tanto:

parámetros =  $(10 + 32 + 1) * 32 = 43 * 32 = 1376$   
Matriz de pesos para la entrada:  $10 * 32 = 320$  parámetros  
Matriz de pesos recurrentes:  $32 * 32 = 1024$  parámetros  
Vector de sesgo: 32 parámetros  
Total:  $320 + 1024 + 32 = 1376$  parámetros

## Capa Dense

La capa densa de salida, usa la activación softmax para generar una distribución de probabilidad sobre el vocabulario, para la predicción de la próxima palabra.

```
Dense(units=vocab_size, activation='softmax')
```

- La capa `Dense` es una capa de neuronas totalmente conectadas.
- `vocab_size` es el número de unidades en la capa, que en este caso es igual al tamaño del vocabulario. Esto significa que la red producirá un vector de salida donde cada elemento representa la probabilidad de que una determinada palabra del vocabulario sea la siguiente palabra en la secuencia. La razón de que la capa `Dense` tenga tantas unidades como el tamaño del vocabulario es que se intenta predecir la probabilidad de cada palabra en el vocabulario como la siguiente palabra en la secuencia.
- `activation='softmax'` es la función de activación que se utiliza para convertir las salidas de la capa en probabilidades que suman 1. Esto es útil para la clasificación multiclase, como predecir la siguiente palabra en una secuencia.
- El número de parámetros en la capa Dense se calcula usando la fórmula:

parámetros =  $(\text{input\_dim} * \text{units}) + \text{units}$   
`input_dim`: dimensionalidad de la entrada (en este caso, el número de unidades en la capa Embedding, que es 32)  
`units`: número de unidades en la capa Dense (en este caso, igual a `vocab_size`, que es 11)  
El `+ units` al final es para los términos de sesgo (bias)

parámetros =  $(32 * 11) + 11 = 352 + 11 = 363$

Matriz de pesos:  $32 * 11 = 352$  parámetros

Vector de sesgo: 11 parámetros

Total:  $352 + 11 = 363$  parámetros

El modelo toma una secuencia de texto como entrada, la convierte en vectores densos con una capa `Embedding`, procesa estos vectores a través de una capa `SimpleRNN`, y finalmente utiliza una capa `Dense` para producir una distribución de probabilidad sobre el vocabulario, indicando la probabilidad de cada palabra de ser la siguiente palabra en la secuencia.

Una vez que tenemos nuestro modelo de RNN entrenado, podemos probar como funciona para predecir la siguiente palabra:

```
import numpy as np

def predict_next_word(model, tokenizer, text, max_length):
    # Convertir el texto a secuencia de números enteros
    sequence = tokenizer.texts_to_sequences([text])[0]

    # Aplicar padding
    sequence = pad_sequences([sequence], maxlen=max_length-1, padding='p')

    # Realizar la predicción
    probabilities = model.predict(sequence)[0]
    predicted_index = np.argmax(probabilities)

    # Mapear el índice predicho a la palabra
    for word, index in tokenizer.word_index.items():
        if index == predicted_index:
            return word
    return None

# Ejemplo de uso
text = "hola cómo"
predicted_word = predict_next_word(model, tokenizer, text, max_length)
print(f"La siguiente palabra predicha después de '{text}' es: {predicted_word}")
```

El modelo que hemos creado, es bastante básico. Podríamos mejorarlo con las siguientes recomendaciones:

- Aumentar la dimensión de la capa de embeddings, puede permitir que el modelo capture relaciones más complejas entre las palabras. Por ejemplo, cambiar la dimensión de 10 a 50 o 100.
- Añadir múltiples capas RNN para hacer que la red sea más profunda. Esto puede ayudar a capturar relaciones más complejas en los datos, pero también puede aumentar el riesgo de sobreajuste.
- Para evitar el sobreajuste, especialmente cuando se aumenta la complejidad del modelo, se pueden añadir técnicas de regularización como el dropout.
- Si es posible, entrenar el modelo con un conjunto de datos más grande. Más datos generalmente conducen a mejores resultados, especialmente con modelos complejos
- Si queremos trabajar con frases más largas, es posible que aumentar el valor de `max_length` para considerar más palabras en la entrada. Sin embargo, secuencias muy largas pueden requerir más memoria y tiempo de entrenamiento.

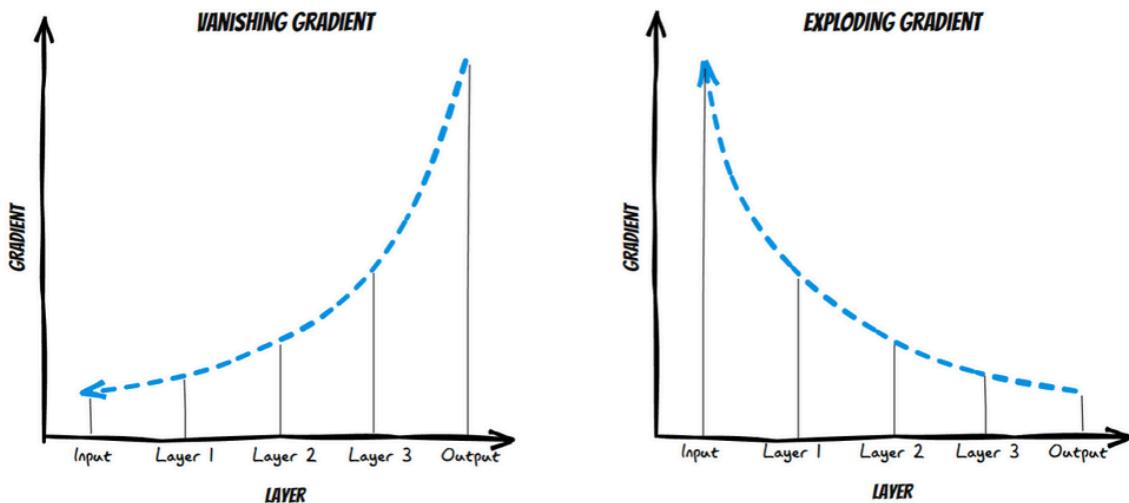
Google Colab

 <https://colab.research.google.com/drive/1ObYfqzWL3ktrfnzVyntb86AOdD7Hxo16?usp=sharing>

Las Redes Neuronales Recurrentes (RNN) son una herramienta valiosa en el procesamiento de secuencias, pero presentan varios problemas y limitaciones:

1. **Problema de las Dependencias a Largo Plazo:** Las RNN tradicionales tienen dificultades para aprender y mantener información de pasos de tiempo muy anteriores en la secuencia. Esto se conoce como el problema de las dependencias a largo plazo. Esencialmente, a medida que la secuencia crece, la influencia de un dato en particular tiende a desaparecer, lo que hace que las RNN sean inefficientes para capturar relaciones en secuencias largas.
2. **Desvanecimiento y Explosión del Gradiente:** Durante el entrenamiento, las RNN pueden enfrentar problemas de desvanecimiento o explosión del gradiente. Esto significa que los gradientes, que se utilizan para actualizar los pesos en la retropropagación, pueden volverse extremadamente

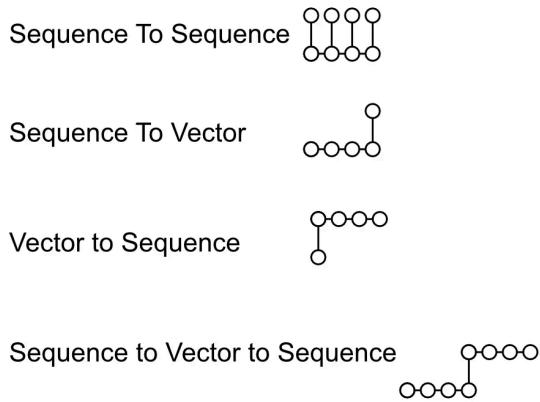
pequeños (desvanecimiento) o extremadamente grandes (explosión). El desvanecimiento del gradiente puede hacer que el entrenamiento sea muy lento o se estanke, mientras que la explosión del gradiente puede hacer que el entrenamiento sea inestable.



3. **Ineficiencia Computacional:** Las RNN necesitan procesar secuencias paso a paso, lo que las hace inherentemente más lentas para entrenar en comparación con las redes feedforward o las redes convolucionales que pueden realizar operaciones en paralelo.
4. **Problema de Sobreajuste:** Al igual que otras redes neuronales, las RNN pueden sobreajustarse, especialmente cuando se tiene un número limitado de datos de entrenamiento. Esto significa que el modelo puede funcionar bien en los datos de entrenamiento pero no generalizar bien a datos no vistos.
5. **Dificultad en la Interpretación:** Las RNN, al igual que otros modelos de aprendizaje profundo, pueden ser consideradas como "cajas negras", lo que significa que puede ser difícil interpretar y entender cómo están tomando decisiones específicas.

### Tipos de redes RNN

Las RNN tienen diferentes variantes basadas en la naturaleza de la entrada y la salida:



## 1. Sequence to Sequence (Secuencia a Secuencia)

- **Descripción:** Toma una secuencia como entrada y produce una secuencia como salida.
- **Ejemplo:** Traducción automática. Dada una secuencia de palabras en un idioma (por ejemplo, inglés), produce una secuencia de palabras en otro idioma (por ejemplo, español).

## 2. Sequence to Vector (Secuencia a Vector)

- **Descripción:** Toma una secuencia como entrada y produce un vector fijo como salida.
- **Ejemplo:** Análisis de sentimientos. Dada una secuencia de palabras (una reseña de película), produce un vector que indica el sentimiento (por ejemplo, positivo o negativo).

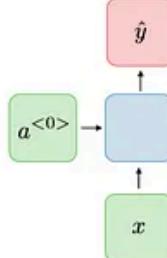
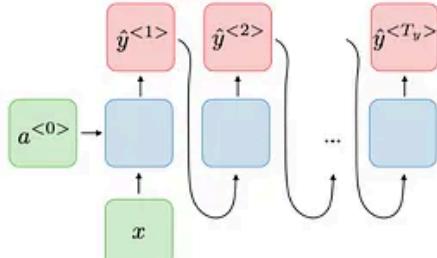
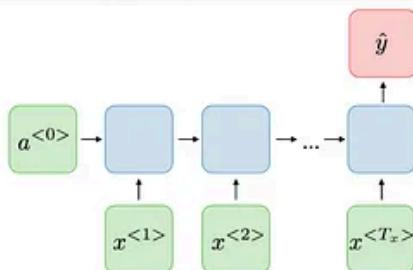
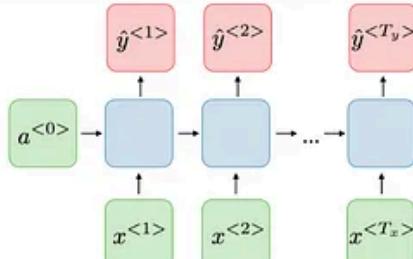
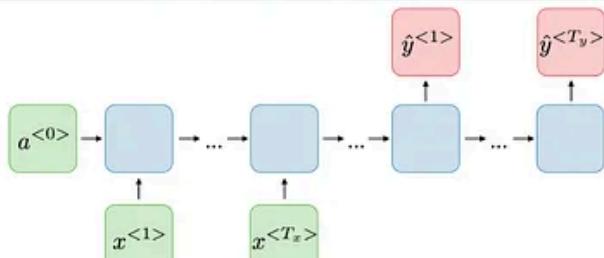
## 3. Vector to Sequence (Vector a Secuencia)

- **Descripción:** Toma un vector fijo como entrada y produce una secuencia como salida.
- **Ejemplo:** Generación de descripciones de imágenes. Dada la representación vectorial de una imagen, produce una secuencia de palabras que describe la imagen.

## 4. Sequence to Vector to Sequence (Secuencia a Vector a Secuencia)

- **Descripción:** Toma una secuencia como entrada, la convierte en un vector (a menudo llamado "vector de contexto" o "estado oculto"), y luego utiliza ese vector para generar una secuencia de salida.
- **Ejemplo:** Resumen automático. Dada una secuencia de palabras (un artículo largo), primero se convierte en una representación vectorial y

Luego se utiliza ese vector para generar una secuencia más corta que resuma el contenido del artículo.

Type of RNN	Illustration	Example
One-to-one $T_x = T_y = 1$		Traditional neural network
One-to-many $T_x = 1, T_y > 1$		Music generation
Many-to-one $T_x > 1, T_y = 1$		Sentiment classification
Many-to-many $T_x = T_y$		Name entity recognition
Many-to-many $T_x \neq T_y$		Machine translation

Artículos interesantes para ampliar información:

<https://towardsdatascience.com/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9>

<https://amitness.com/2020/04/recurrent-layers-keras/>

<https://towardsdatascience.com/animated-rnn-lstm-and-gru-ef124d06cf45>

[https://medium.com/@vipra\\_singh/llm-architectures-explained-rnn-lstm-grus-part-3-c5e1cbfeda1d](https://medium.com/@vipra_singh/llm-architectures-explained-rnn-lstm-grus-part-3-c5e1cbfeda1d)

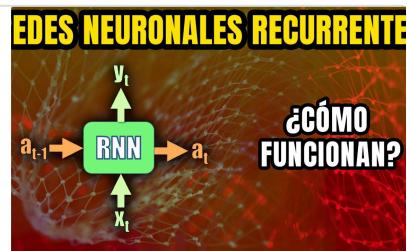
Redes Neuronales Recurrentes: EXPLICACIÓN DETALLADA

🔥🔥 Academia Online 🔥🔥: <https://cursos.codificandobits.com/>

🔥🔥 Asesorías y formación personalizada 🔥🔥:

<https://www.codificandobits.com/servicios/>

➡ [https://www.youtube.com/watch?v=hB4XYst\\_t-I](https://www.youtube.com/watch?v=hB4XYst_t-I)



## 2. LSTM (Long Short Term Memory)

Las LSTM (Long Short-Term Memory) son una variante especializada de las Redes Neuronales Recurrentes (RNN) diseñadas para abordar el problema del desvanecimiento del gradiente, que es común en las RNN tradicionales. Las LSTM fueron introducidas por Sepp Hochreiter y Jürgen Schmidhuber en 1997 y han demostrado ser eficaces en una variedad de tareas de procesamiento de secuencias.

### Características clave de las LSTM:

- Memoria a Largo Plazo:** A diferencia de las RNN tradicionales, que tienden a olvidar la información después de un cierto número de pasos de tiempo, las LSTM están diseñadas para recordar información a lo largo de largas secuencias, lo que les permite capturar dependencias a largo plazo en los datos.
- Estructura de Puertas:** Las LSTM introducen el concepto de "puertas" para regular el flujo de información. Estas puertas determinan qué información se

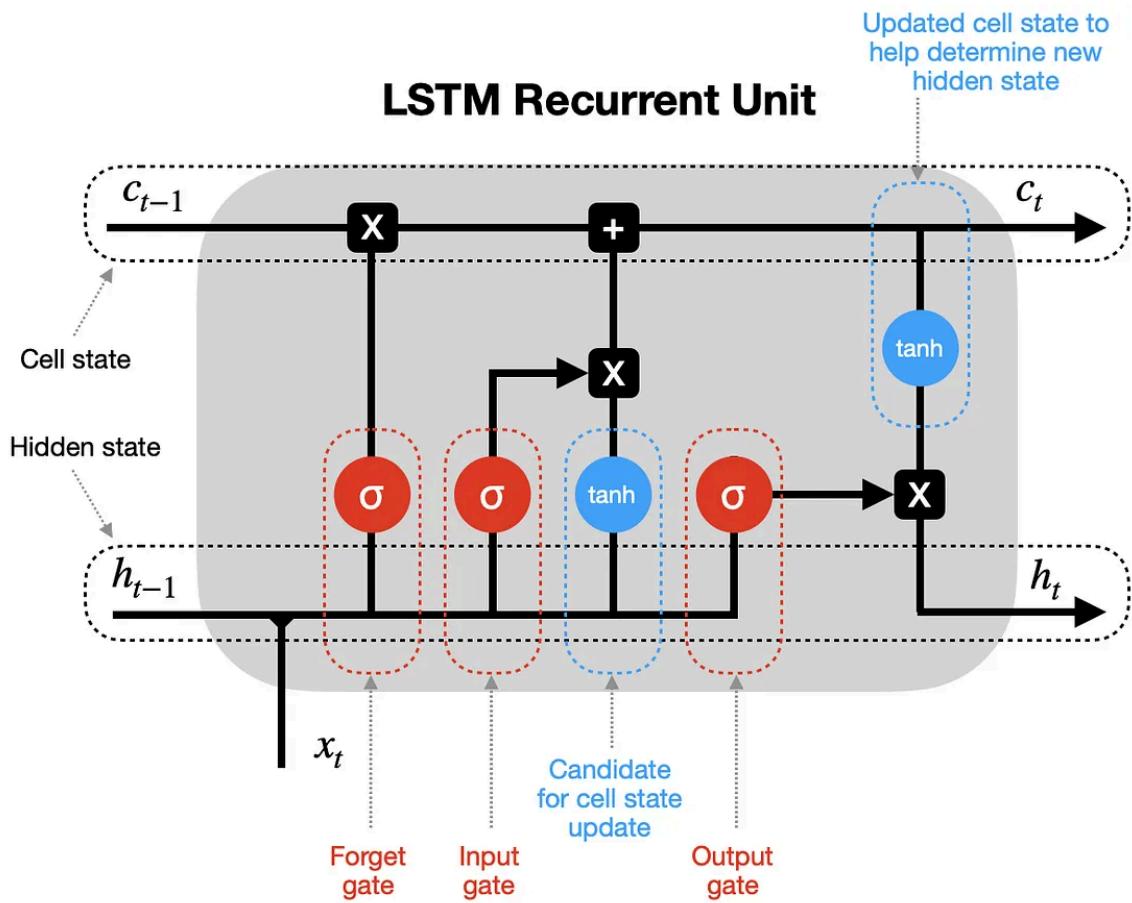
debe mantener o descartar en cada paso de tiempo. Hay tres puertas principales en una LSTM:

- **Puerta de Entrada (Input Gate):** Decide cuánta información nueva se añadirá a la celda de memoria.
- **Puerta de Olvido (Forget Gate):** Decide cuánta información de la celda de memoria se descartará.
- **Puerta de Salida (Output Gate):** Decide cuánta información de la celda de memoria se utilizará para calcular la salida en el paso de tiempo actual.

3. **Celda de Memoria:** Es el componente central de la LSTM que almacena información a lo largo de los pasos de tiempo. La información en la celda de memoria se actualiza en cada paso de tiempo basándose en las decisiones tomadas por las puertas.
4. **Resistencia al Desvanecimiento del Gradiente:** Gracias a su estructura única, las LSTM pueden aprender dependencias a largo plazo sin sufrir significativamente del problema del desvanecimiento del gradiente, que afecta a las RNN tradicionales. Sin embargo, aunque las LSTM son menos susceptibles al desvanecimiento del gradiente en comparación con las RNN tradicionales, todavía pueden ser propensas al problema del gradiente explosivo. La explosión del gradiente puede ocurrir cuando los valores del gradiente se vuelven extremadamente grandes, lo que puede llevar a actualizaciones de peso inestables y, en última instancia, a un modelo que no converge.

La siguientes, es la arquitectura de cada unidad de una red recurrente. La celda se vale de los siguientes elementos, para lograr las capacidades de memoria de corto y largo plazo:

## LSTM Recurrent Unit



$h_{t-1}$  - hidden state at previous timestep t-1 (short-term memory)

$c_{t-1}$  - cell state at previous timestep t-1 (long-term memory)

$x_t$  - input vector at current timestep t

$h_t$  - hidden state at current timestep t

$c_t$  - cell state at current timestep t

**X** - vector pointwise multiplication    **+** - vector pointwise addition

**tanh** - tanh activation function

**(dashed box)** - states

**σ** - sigmoid activation function

**(red dashed box)** - gates

**T** - concatenation of vectors

**(blue dashed box)** - updates

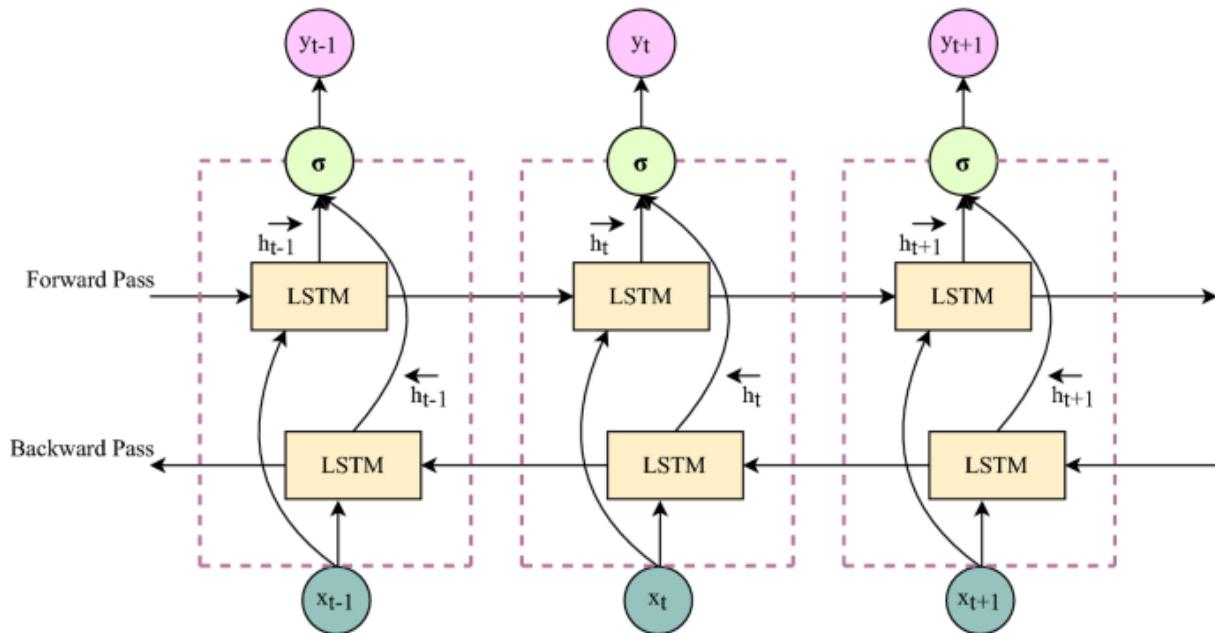
- Estado de Celda ( $C_t$ ):** Representa la memoria a largo plazo de la unidad. Es un camino por el cual la información puede viajar sin alteraciones.
- Estado Oculto ( $h_t$ ):** Representa la memoria a corto plazo o la salida actual de la unidad.

3. **Puerta de Olvido:** Decide qué información del estado de celda debe descartarse o conservarse. Utiliza la función sigmoide para tomar esta decisión, produciendo valores entre 0 (olvidar) y 1 (conservar).
4. **Puerta de Entrada:** Actualiza el estado de la celda con nueva información. Primero, decide qué valores se actualizarán utilizando una función sigmoide. Luego, una capa tanh crea un vector de nuevos valores candidatos. Estos dos resultados se multiplican para actualizar el estado de la celda de largo plazo  $C_t$
5. **Puerta de Salida:** Decide cuál debería ser el próximo estado oculto (salida). Toma la entrada actual y el estado oculto anterior, y después de pasarlo por una función sigmoide, multiplica el resultado con el tanh del estado de la celda para producir el estado oculto de este instante.
6. **Función de Activación tanh:** Produce valores entre -1 y 1 y se utiliza para regular la red.
7. **Función de Activación Sigmoide:** Produce valores entre 0 y 1 y se utiliza principalmente para las puertas en el LSTM, ya que puede producir valores cercanos a 0 o 1, tomando decisiones prácticamente binarias.

## LSTM Bidireccionales

Una **LSTM Bidireccional** es una variante de la red neuronal recurrente LSTM (Long Short-Term Memory) que puede mejorar el rendimiento del modelo en tareas que requieren aprender dependencias de los datos tanto en el pasado como en el futuro.

Una LSTM Bidireccional implica duplicar la primera capa recurrente en la red para que haya dos capas paralelas. Una de las capas recibe la secuencia de entrada tal como está (de principio a fin), mientras que la otra recibe la secuencia de entrada invertida (de fin a principio). Estas dos capas LSTMs leen la entrada de dos direcciones: hacia adelante y hacia atrás (o viceversa). Después de leer la secuencia en ambas direcciones, las representaciones aprendidas por ambas LSTMs se combinan para producir una única representación.



Las LSTMs ya son conocidas por su capacidad para aprender dependencias a largo plazo en los datos. Al agregar una dirección adicional de procesamiento, las LSTMs Bidireccionales pueden potencialmente aprender estas dependencias aún más eficazmente.

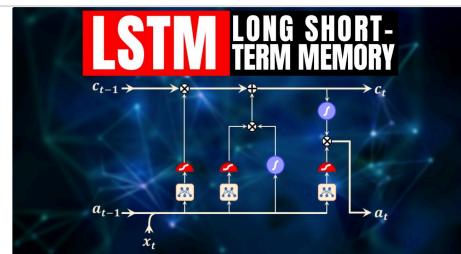
¿Qué es una red LSTM?

🔥🔥 Academia Online 🔥🔥:

<https://cursos.codificandobits.com/>

🔥🔥 Asesorías y formación personalizada 🔥🔥:

➡️ <https://www.youtube.com/watch?v=1BubAvTVBYs>



## Generación de texto

Veamos un ejemplo de una LSTM construida con la librería `tensorflow` y `Keras` para crear un modelo que genere texto:

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Embedding, Bidirectional
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical, get_file
```

```

from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.callbacks import Callback, ModelCheckpoint
import random
import os
import pickle

# Descarga
path = get_file('cien.txt', origin="https://raw.githubusercontent.com/MartaRuiz
corpus = open(path).read().lower()

# Eliminar las líneas que contienen cadenas indeseadas
i = 0
lines = corpus.split('\n')
while i < len(lines):
    if "cien años de soledad" in lines[i]:
        del lines[i:i+3]
    else:
        i += 1
corpus = '\n'.join(lines)

# Preparamos el tokenizador
tokenizer = Tokenizer()
tokenizer.fit_on_texts([corpus])
total_words = len(tokenizer.word_index) + 1

# Guardar el tokenizador
with open('tokenizer.pkl', 'wb') as tokenizer_file:
    pickle.dump(tokenizer, tokenizer_file)

# Crear secuencias de entrada
input_sequences = []
for line in corpus.split('\n'):
    token_list = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(token_list)):
        n_gram_sequence = token_list[:i+1]
        input_sequences.append(n_gram_sequence)

# Calcula el tamaño máximo de las secuencias
max_sequence_len = max([len(seq) for seq in input_sequences])

```

```

# Recortamos el corpus para acelerar la demostración
fraction = 0.10 # Usamos solo el 10% de las secuencias
num_sequences = int(len(input_sequences) * fraction)
input_sequences = input_sequences[:num_sequences]

# Realiza el padding y prepara los datos de entrada del modelo
input_sequences = pad_sequences(input_sequences, maxlen=max_sequence_
X, y = input_sequences[:, :-1], input_sequences[:, -1]
y = to_categorical(y, num_classes=total_words)

# Construcción del modelo
model = Sequential()
model.add(Embedding(total_words, 50, input_length=max_sequence_len-1))
# model.add(Bidirectional(LSTM(150))) # Opción Bidireccional
model.add(LSTM(150))
model.add(Dense(512, activation='relu')) # Capa Dense adicional
model.add(Dense(total_words, activation='softmax'))
optimizer = RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=

# Carga el ultimo modelo entrenado si existe.
try:
    if os.path.exists(f"last_model.h5"):
        print('Cargando parámetros pre-entrenados...')
        model.load_weights(f"last_model.h5")
    except Exception as e:
        print(f"Error al cargar el modelo: {e}")

# Callback para generar texto después de cada época
class GenerateText(Callback):
    def on_epoch_end(self, epoch, logs=None):
        # Selecciona aleatoriamente una secuencia de 5 palabras del corpus
        words = corpus.split()
        start_index = random.randint(0, len(words) - 5)
        seed_text = ' '.join(words[start_index:start_index + 5])
        print(f"\nSeed text: {seed_text}\n")

        next_words = 5

```

```

for _ in range(next_words):
    token_list = tokenizer.texts_to_sequences([seed_text])[0]
    token_list = pad_sequences([token_list], maxlen=max_sequence_len-1,
    predicted = np.argmax(self.model.predict(token_list), axis=-1)
    output_word = tokenizer.index_word[predicted[0]]
    seed_text += " " + output_word
    print(f"\nGenerated Text after epoch {epoch + 1}: {seed_text}\n")

# Entrenamos nuestro modelo con el callback CustomModelCheckpoint
model.summary()
model.fit(X, y, epochs=1000, verbose=1, callbacks=[GenerateText()])

# Guardar el modelo en un archivo
model.save("last_model.h5")
model.summary()

```

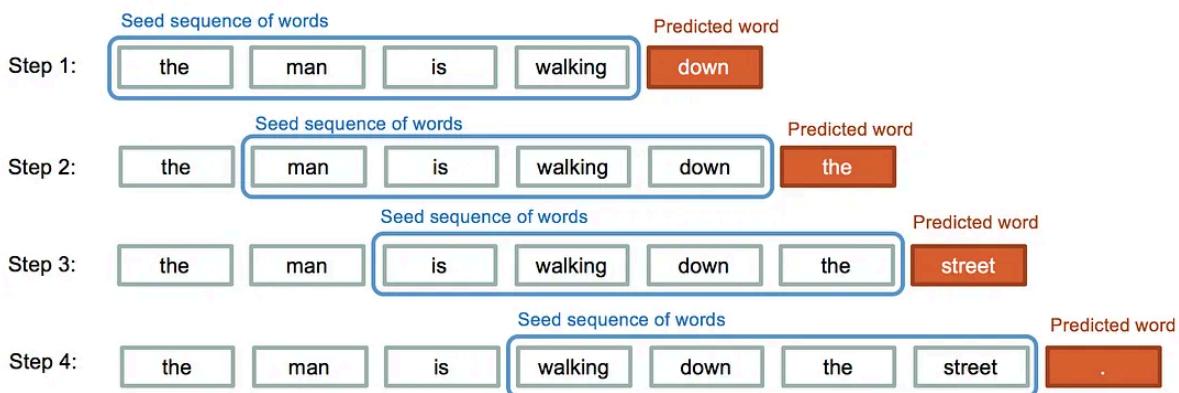
Nuestro modelo queda entonces con la siguiente arquitectura y parámetros:

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
embedding (Embedding)	(None, 23, 50)	789000
Istm (LSTM)	(None, 150)	120600
dense (Dense)	(None, 512)	77312
dense_1 (Dense)	(None, 15780)	8095140
<hr/>		
Total params: 9082052 (34.65 MB)		
Trainable params: 9082052 (34.65 MB)		
Non-trainable params: 0 (0.00 Byte)		

Por cada época ( `epoch` ) de entrenamiento, se ejecuta la función callback `GenerateText()`, para poner a prueba nuestro predictor de palabras. El

procedimiento de predicción de cada palabra, es como se muestra en el gráfico:



Es decir, realizamos la predicción de una palabra, y luego sumamos esa palabra a una ventana deslizante, que nos permite predecir la siguiente palabra. Repitiendo esos pasos, podemos predecir n palabras, partiendo de un texto base.

Con el siguiente script, podemos restaurar el modelo y realizar predicciones:

```
import numpy as np
import pickle
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Cargar el tokenizador (Asumiendo que ya tenemos el archivo tokenizer.pkl)
with open('tokenizer.pkl', 'rb') as tokenizer_file:
    tokenizer = pickle.load(tokenizer_file)
    total_words = len(tokenizer.word_index) + 1
    print(f"total_words: '{total_words}'")

# Cargar el modelo
model = load_model('last_model.h5')
model.summary()

# Obtiene el tamaño máximo de las secuencias de la primer capa del modelo
max_sequence_len = model.layers[0].input_shape[1] + 1

def generate_text(seed_text, next_words, max_sequence_len, tokenizer, mode
```

```

for _ in range(next_words):
    token_list = tokenizer.texts_to_sequences([seed_text])[0]
    token_list = pad_sequences([token_list], maxlen=max_sequence_len-1, padding='post')
    predicted = np.argmax(model.predict(token_list), axis=-1)
    output_word = tokenizer.index_word[predicted[0]]
    seed_text += " " + output_word
return seed_text

# Generar texto
seed_text = "es una"
generated_text = generate_text(seed_text, 3, max_sequence_len, tokenizer, model)
print(generated_text)

```

El ejemplo que hemos visto es bastante básico. Podríamos mejorar el modelo con algunas de estas recomendaciones:

1. **Más datos:** Cuanto más grande sea el corpus, mejor será el modelo para capturar las estructuras lingüísticas y las relaciones entre palabras.
2. **Regularización:**
  - **Dropout:** Añadir capas de Dropout en el modelo para prevenir el sobreajuste.
  - **Regularización L1 y L2:** Estas son técnicas que penalizan ciertos parámetros del modelo si se vuelven demasiado grandes, lo que puede ayudar a prevenir el sobreajuste.
3. **Modelo más profundo:** Añadir más capas LSTM puede ayudar al modelo a capturar relaciones más complejas en los datos. Sin embargo, esto también aumentará el tiempo de entrenamiento y el riesgo de sobreajuste.
4. **Ajuste de hiperparámetros:**
  - **Tasa de aprendizaje:** Experimentar con diferentes tasas de aprendizaje para el optimizador.
  - **Tamaño del batch:** Cambiar el tamaño del batch puede afectar la convergencia del modelo y su capacidad para generalizar.
  - **Optimizadores:** Aunque RMSprop es comúnmente usado con redes LSTM, también se puede probar otros optimizadores como Adam, Adagrad, etc.

5. **Embeddings preentrenados:** En lugar de aprender los embeddings desde cero, se pueden utilizar embeddings preentrenados como Word2Vec o GloVe. Estos embeddings ya han sido entrenados en grandes conjuntos de datos y pueden mejorar la precisión del modelo.
6. **Bidireccional LSTM:** Una LSTM bidireccional procesa la secuencia de entrada desde ambos extremos (adelante y atrás). Esto puede ayudar al modelo a capturar mejor el contexto alrededor de una palabra.
7. **Aumento de datos (Data augmentation):** Al igual que en el procesamiento de imágenes, podemos crear versiones "alteradas" de los datos para aumentar la cantidad de datos de entrenamiento. En NLP, esto podría implicar técnicas como la retrotraducción (traducir una oración a otro idioma y luego volver a traducirla al original).
8. **Entrenamiento por más épocas:** A veces, simplemente entrenar el modelo por más épocas puede mejorar su rendimiento. Sin embargo, es esencial monitorear el rendimiento en un conjunto de validación para evitar el sobreajuste.

Google Colab

🔗 <https://colab.research.google.com/drive/1CZWFiO3pEu7m7pMOFlpM8jnqwXYSoTtq?usp=sharing>



## Las LSTM como clasificadores

Las LSTM no solo son útiles para tareas de secuencia a secuencia (como la generación de texto) sino también para tareas de **clasificación**. Veamos cómo funcionan las LSTM en el contexto de clasificación, utilizando el ejemplo de la clasificación de sentimientos:

Dada una reseña de película, queremos clasificarla como positiva o negativa. El proceso será el siguiente:

1. **Entrada:** La entrada es una secuencia, que en este caso es una reseña de película. Cada palabra de la reseña se convierte en un vector (generalmente utilizando embeddings pre-entrenados o una capa de embedding a entrenar).
2. **Procesamiento con LSTM:** A medida que la LSTM procesa la secuencia palabra por palabra, mantiene un "estado oculto" que captura la

información sobre la secuencia hasta el momento actual. Este estado oculto se actualiza con cada palabra que la LSTM procesa.

3. **Salida:** En lugar de usar la salida de cada paso de tiempo (como haríamos en la generación de texto), solo nos interesa la última salida, que representa la información de toda la secuencia. Esta salida se pasa a una capa densa (fully connected) que tiene una función de activación (por ejemplo, sigmoid para clasificación binaria) para producir una predicción.

#### Ejemplo:

Supongamos que tenemos la reseña: "La película fue increíblemente buena".

1. Cada palabra se convierte en un vector usando embeddings.
2. La LSTM procesa la secuencia palabra por palabra, actualizando su estado oculto en cada paso.
3. Después de procesar la palabra "buena", la LSTM produce una salida que refleja la información de toda la reseña.
4. Esta salida se pasa a una capa densa con activación sigmoid, que produce un valor entre 0 y 1.
5. Si el valor es mayor que 0.5, clasificamos la reseña como positiva; de lo contrario, como negativa.

#### Ventajas de usar LSTM para clasificación:

- **Captura de dependencias a largo plazo:** Las LSTMs son capaces de capturar dependencias a largo plazo en el texto, lo que es crucial para entender el contexto y el sentimiento.
- **Manejo de secuencias de longitud variable:** Las reseñas pueden tener diferentes longitudes, y las LSTMs pueden manejar estas variaciones.
- **Captura de información secuencial:** A diferencia de los modelos tradicionales que tratan las palabras como características independientes, las LSTMs consideran el orden de las palabras, lo que es esencial para entender el significado.

Veamos como queda nuestro ejemplo:

```
# Entrena un modelo LSTM en la tarea de clasificación de sentimientos de IMDB
from keras.datasets import imdb
from keras.layers import LSTM, Dense, Embedding
```

```

from keras.models import Sequential
from keras.preprocessing import sequence

max_features = 20000
 maxlen = 80 # corta textos después de este número de palabras (entre las pa
batch_size = 32

print("Cargando datos...")
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
print(len(x_train), "secuencias de entrenamiento")
print(len(x_test), "secuencias de prueba")

print("Rellenando secuencias (muestras x tiempo)")
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
print("Forma de x_train:", x_train.shape)
print("Forma de x_test:", x_test.shape)

print("Construyendo modelo...")
model = Sequential()
model.add(Embedding(max_features, 128))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation="sigmoid"))

# prueba utilizando diferentes optimizadores y diferentes configuraciones de c
model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accu

print("Entrenando...")
model.fit(
    x_train, y_train, batch_size=batch_size, epochs=15, validation_data=(x_test, y
)
score, acc = model.evaluate(x_test, y_test, batch_size=batch_size)
print("Puntuación de la prueba:", score)
print("Precisión de la prueba:", acc)

```

Luego podemos probar nuestro clasificador de este modo:

```

# Dado que la tokenización y el relleno son específicos del conjunto de dat
# necesitaremos recrear el mismo tokenizador.

```

```

from keras.datasets import imdb
word_to_id = imdb.get_word_index()
# Las palabras en el conjunto de datos de IMDB están indexadas con un
# desplazamiento de 3 porque 0, 1 y 2 están reservados para "padding",
# "start of sequence" y "unknown".
word_to_id = {k: (v + 3) for k, v in word_to_id.items()}
word_to_id["<PAD>"] = 0
word_to_id["<START>"] = 1
word_to_id["<UNK>"] = 2
id_to_word = {value: key for key, value in word_to_id.items()}

def encode_text(text):
    words = text.split()
    ids = [word_to_id.get(word, word_to_id["<UNK>"]) for word in words]
    return sequence.pad_sequences([ids], maxlen=maxlen)

# Función para realizar una inferencia
def predict_sentiment(text):
    encoded_text = encode_text(text)
    prediction = modelo.predict(encoded_text)[0][0]

    if prediction >= 0.5:
        return f"Positivo (Confianza: {prediction:.2f})"
    else:
        return f"Negativo (Confianza: {1 - prediction:.2f})"

# Ejemplo de uso
text = "The movie was amazing and the actors did a great job."
print(predict_sentiment(text))

text = "I did not enjoy the movie. It was boring and predictable."
print(predict_sentiment(text))

```

Y tendremos como resultado:

```

1/1 [=====] - 0s 169ms/step
Positivo (Confianza: 0.98)

```

1/1 [=====] - 0s 22ms/step

Negativo (Confianza: 0.99)

Más información:

<https://towardsdatascience.com/animated-rnn-lstm-and-gru-ef124d06cf45>

Google Colab

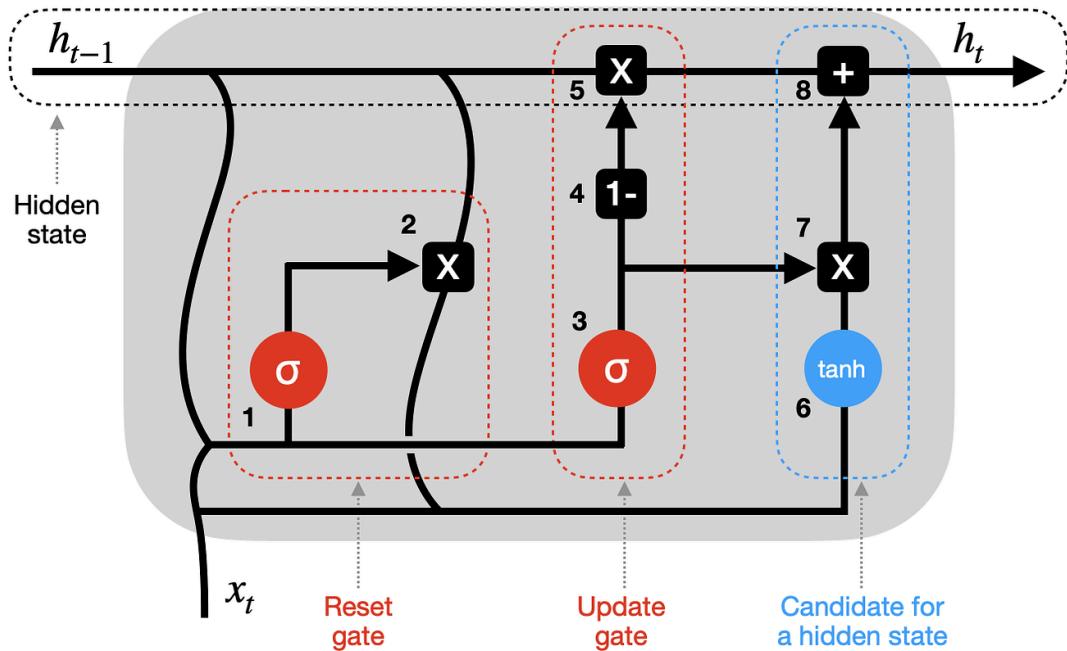
🔗 [https://colab.research.google.com/drive/1IUSfptsjUYTKzxRkkwR-G8-xn7D\\_0Qh3?usp=sharing](https://colab.research.google.com/drive/1IUSfptsjUYTKzxRkkwR-G8-xn7D_0Qh3?usp=sharing)



### 3. Arquitectura GRU

Las Unidades Recurrentes Gated (GRU, por sus siglas en inglés de Gated Recurrent Units) son una variante de las Redes Neuronales Recurrentes (RNN) diseñada para abordar el problema de la desaparición del gradiente, al igual que las Redes Neuronales Recurrentes de Memoria a Largo Plazo (LSTM). Sin embargo, las GRU ofrecen una estructura más simplificada comparada con las LSTM, lo que las hace computacionalmente más eficientes en ciertos casos.

## GRU Recurrent Unit



$h_{t-1}$  - hidden state at previous timestep t-1 (memory)

$x_t$  - input vector at current timestep t

$h_t$  - hidden state at current timestep t

**X** - vector pointwise multiplication    **+** - vector pointwise addition

**tanh** - tanh activation function

**(dashed)** - states

**σ** - sigmoid activation function

**(red dashed)** - gates

**Y** - concatenation of vectors

**(blue dashed)** - updates

Aquí se explica cómo funcionan las GRU:

### 1. Mecanismo de Puertas:

- Las GRU utilizan el concepto de puertas para controlar el flujo de información. En particular, tienen dos puertas: una puerta de actualización (update) y una puerta de reinicio (reset).

### 2. Puerta de Actualización:

- Esta puerta decide qué información del estado anterior se va a mantener y qué información nueva se va a incorporar. Esto ayuda a

determinar la importancia de la información pasada en el contexto actual.

### 3. Puerta de Reinicio:

- La puerta de reinicio decide qué cantidad de información pasada se va a descartar. Esto es útil para deshacerse de la información irrelevante y hacer espacio para la nueva información relevante.

### 4. Actualización del Estado:

- Utilizando las decisiones tomadas por las puertas de actualización y reinicio, las GRU calculan un nuevo estado oculto que se transmite a lo largo de la secuencia y se utiliza para la salida del modelo.

### 5. Eficiencia y Simplificación:

- Las GRU reducen la complejidad de las LSTM fusionando las puertas de entrada y de olvido en una sola puerta de actualización, y eliminando la celda de memoria, lo que resulta en un modelo más simple y más rápido, pero con una capacidad similar para capturar dependencias a largo plazo en los datos secuenciales.

Más información:

<https://towardsdatascience.com/animated-rnn-lstm-and-gru-ef124d06cf45>

## 4. Modelos Sequence-to-Sequence (Seq2Seq). Encoders y Decoders

En el vasto dominio del procesamiento del lenguaje natural (NLP), la tarea de modelar secuencias de datos se ha vuelto crucial para una serie de aplicaciones, incluyendo la traducción automática, la generación de texto y el resumen automático. Tradicionalmente, las Redes Neuronales Recurrentes (RNN) y sus variantes avanzadas como las Redes Neuronales Recurrentes de Memoria a Largo Plazo (LSTM) han sido pilares en la gestión de datos secuenciales, gracias a su capacidad para mantener un estado interno que puede capturar información de los tokens anteriores en una secuencia. Sin embargo, a pesar de su eficacia en capturar dependencias temporales, estas

arquitecturas enfrentan desafíos al lidiar con secuencias largas y al traducir o mapear secuencias de entrada a secuencias de salida de longitud variable.

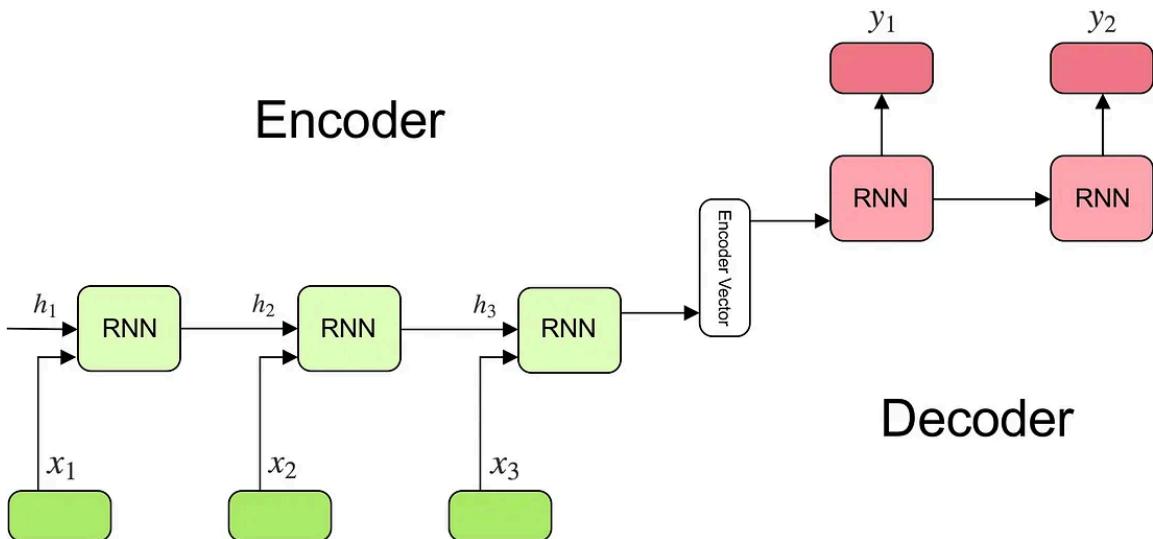
En este contexto, emergen los modelos Secuencia-a-Secuencia (Seq2Seq), ofreciendo un marco más robusto para mapear secuencias de entrada a secuencias de salida. A diferencia de las arquitecturas RNN y LSTM tradicionales que se centran principalmente en procesar secuencias y generar una salida fija, los modelos Seq2Seq están diseñados para manejar tareas donde tanto la entrada como la salida son secuencias, y donde la longitud de la salida puede no ser igual a la longitud de la entrada.

La arquitectura Seq2Seq consta de dos componentes principales: el codificador y el decodificador. El codificador procesa la secuencia de entrada y genera una representación vectorial compacta, mientras que el decodificador toma esta representación y genera una secuencia de salida. Esta estructura bifásica permite que los modelos Seq2Seq manejen eficazmente tareas complejas de mapeo secuencial.

Además, mientras que las arquitecturas RNN y LSTM tradicionales pueden luchar con secuencias de entrada muy largas debido a la desaparición del gradiente y otras limitaciones inherentes, los modelos Seq2Seq, especialmente cuando se les dota de mecanismos de atención, tienen la capacidad de aliviar estos problemas, proporcionando un enfoque más robusto para capturar dependencias a largo plazo y relaciones complejas entre las secuencias de entrada y salida.

La arquitectura de este modelo, surge del paper [Sequence to Sequence Learning with Neural Networks](#) (Ilya Sutskever). Este fue uno de los primeros artículos en presentar el modelo codificador-decodificador (encoder-decoder) para traducción automática y, en general, modelos de secuencia a secuencia. El modelo se aplicó a la traducción del inglés al francés.

Para comprender completamente la lógica subyacente del modelo, repasaremos la siguiente figura:



El modelo consta de 3 partes: codificador, vector intermedio codificador y decodificador.

### Codificador (Encoder)

Una pila de varias unidades recurrentes (células LSTM o GRU para un mejor rendimiento) donde cada una acepta un único elemento de la secuencia de entrada, recopila información para ese elemento y la propaga hacia adelante. En un problema de respuesta a preguntas, la secuencia de entrada es una colección de todas las palabras de la pregunta. Cada palabra se representa como  $x_i$  donde  $i$  es el orden de esa palabra.

Los estados ocultos  $h_i$  se calculan mediante la fórmula:

$$h_t = f(W^{(hh)}h_{t-1} + W^{(hx)}x_t)$$

Esta sencilla fórmula representa el resultado de una red neuronal recurrente ordinaria. Como podemos ver, simplemente aplicamos los pesos apropiados al estado oculto anterior  $h_{(t-1)}$  y al vector de entrada  $x_t$ .

### Vector codificador

Este es el estado oculto final producido a partir de la parte codificadora del modelo. Se calcula utilizando la fórmula anterior. Este vector tiene como objetivo encapsular la información de todos los elementos de entrada para ayudar al decodificador a realizar predicciones precisas.

Actúa como el estado oculto inicial de la parte decodificadora del modelo.

## Decodificador (Decoder)

Una pila de varias unidades recurrentes donde cada una predice una salida  $y_t$  en un paso de tiempo  $t$ . Cada unidad recurrente acepta un estado oculto de la unidad anterior y produce una salida así como su propio estado oculto.

En el problema de preguntas y respuestas, la secuencia de salida es una colección de todas las palabras de la respuesta. Cada palabra se representa como  $y_i$  donde  $i$  es el orden de esa palabra. Cualquier estado oculto  $h_i$  se calcula mediante la fórmula:

$$h_t = f(W^{hh} h_{t-1})$$

Como se puede ver, solo estamos usando el estado oculto anterior para calcular el siguiente.

La salida  $y_t$  en el paso de tiempo  $t$  se calcula usando la fórmula:

$$y_t = \text{softmax}(W^S h_t)$$

Calculamos las salidas utilizando el estado oculto en el paso de tiempo actual junto con el peso respectivo  $W(S)$ . Softmax se utiliza para crear un vector de probabilidad que nos ayudará a determinar el resultado final (por ejemplo, la palabra en el problema de preguntas y respuestas).

El poder de este modelo radica en el hecho de que puede asignar secuencias de diferentes longitudes entre sí. Como podemos ver, las entradas y salidas no están correlacionadas y sus longitudes pueden diferir. Esto abre una gama completamente nueva de problemas que ahora pueden resolverse utilizando dicha arquitectura.

La explicación anterior solo cubre el modelo de secuencia a secuencia más simple y, por lo tanto, no podemos esperar que funcione bien en tareas complejas. La razón es que utilizar un único vector para codificar toda la secuencia de entrada no es capaz de capturar toda la información.

Los modelos Seq2Seq se utilizan principalmente para estas aplicaciones:

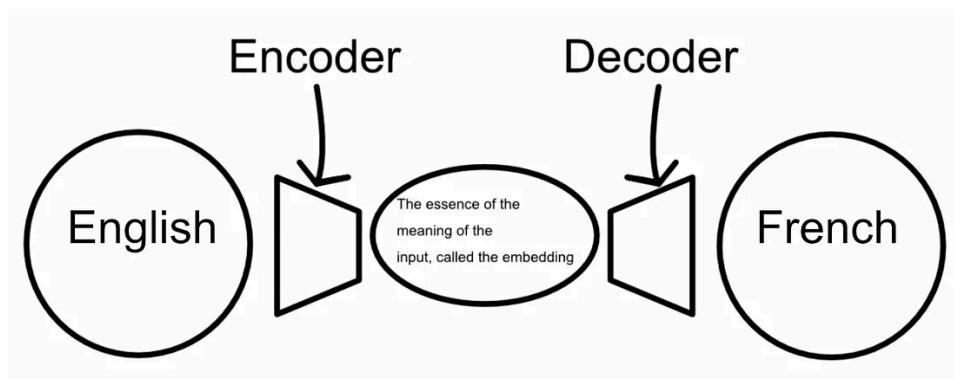
1. **Traducción Automática:** Convertir texto de un idioma a otro.
2. **Resumen Automático:** Generar resúmenes concisos de textos largos.

3. **Respuesta a Preguntas**: Generar respuestas a preguntas basadas en un conjunto de datos dado.
4. **Generación de Texto**: Crear texto coherente basado en alguna entrada dada.
5. **Reconocimiento de Voz**: Transcribir audio a texto.

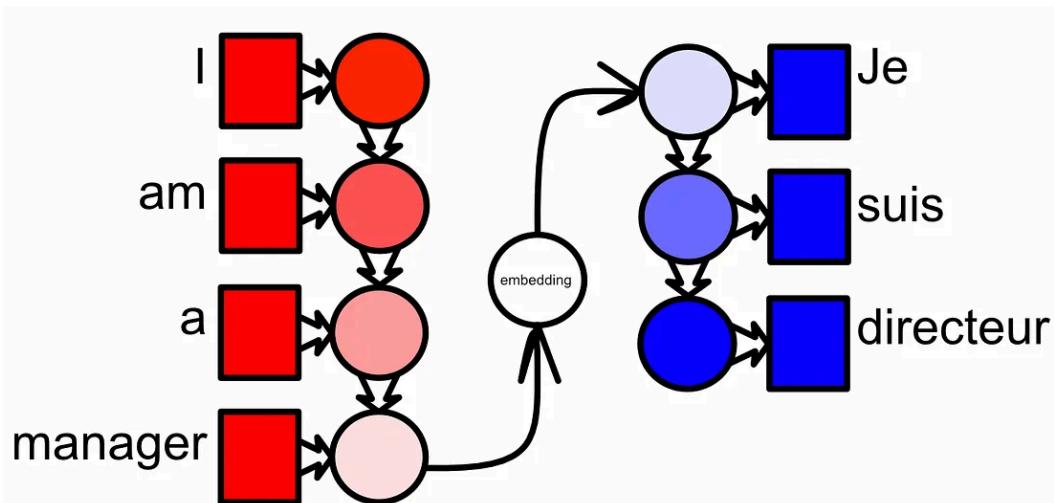
## 5. Mecanismo de Atención (Attention)

El [paper "Attention Is All You Need"](#) introdujo el modelo Transformer, que ha revolucionado el campo del procesamiento del lenguaje natural (NLP) y ha llevado a avances significativos en tareas de NLP.

El mecanismo de atención se popularizó originalmente en [Neural Machine Translation mediante Jointly Learning to Align and Translate \(2014\)](#), que es la referencia guía para esta publicación en particular. Este artículo emplea una arquitectura codificador-decodificador para la traducción del inglés al francés:

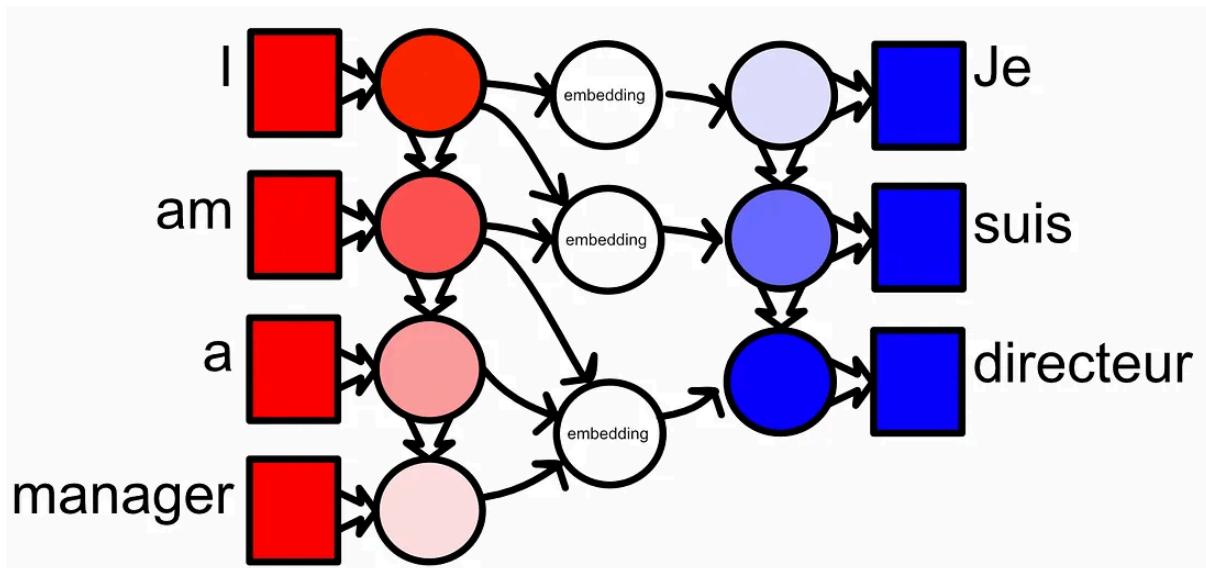


Esta es una arquitectura muy común, pero los detalles exactos pueden cambiar drásticamente de una implementación a otra. Como vimos anteriormente, las Seq2Seq (codificadores-decodificadores de secuencia a secuencia) eran redes recurrentes que podrían incrementalmente "construir" y luego "desconstruir" los embeddings.



Conceptualización del flujo de información de una secuencia simple a una secuencia codificador-descodificador recurrente. El codificador incrusta gradualmente las palabras en inglés, palabra por palabra, en el espacio de incrustación, que luego el decodificador deconstruye. En este diagrama, los círculos representan las incrustaciones en todo el codificador (rojo), el espacio de incrustación intermedio (blanco) y en todo el decodificador (azul). En este caso, esos embeddings son vectores largos y complejos con contenido abstracto que no es fácilmente interpretable por humanos.

Esta idea general, fue lo último en tecnología durante varios años. Sin embargo, un problema con este enfoque es que toda la secuencia de entrada debe incrustarse en el espacio de embeddings, que generalmente es un vector de tamaño fijo. Como resultado, estos modelos pueden olvidar fácilmente el contenido de secuencias demasiado largas. El mecanismo de atención fue diseñado para aliviar el problema de tener que encajar toda la secuencia de entrada en el espacio de incrustación. Lo hace diciéndole al modelo qué entradas están relacionadas con qué salidas. O, en otras palabras, el mecanismo de atención permite que un modelo se centre en partes relevantes de la entrada e ignore el resto.

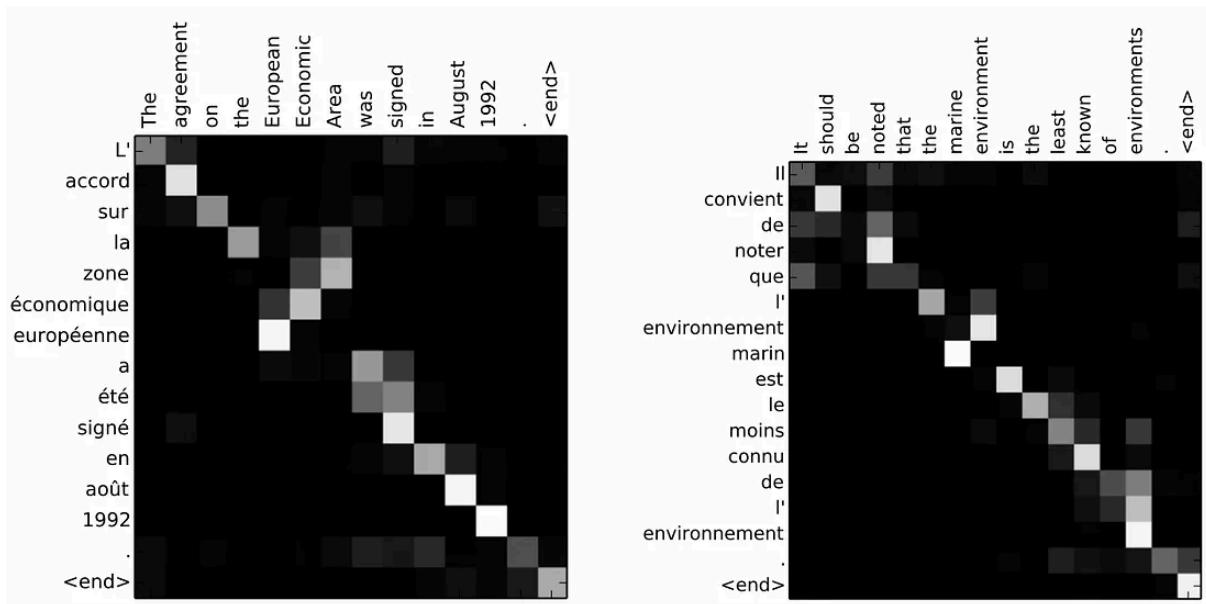


Un ejemplo de cómo sería un proceso de pensamiento basado en la atención. En francés, "je" es exactamente idéntica a la palabra "yo". "Suis" es una conjugación del verbo "être" que es "to be" y se conjuga como "suis" basándose en el sujeto "yo" y el verbo "soy". La elección de "director" está relacionada principalmente con la palabra "gerente", pero también con el contexto en el que se utiliza esa palabra. La elección de qué entradas se relacionan con qué salidas es tarea del mecanismo de atención.

## ¿Cómo funciona la atención?

<https://www.youtube.com/embed/aL-EmKuB078?start=6m44s&end=16m14s>

En la práctica, el mecanismo de atención que analizaremos termina siendo una matriz de puntuaciones, denominada puntuaciones de "alineación". Estas puntuaciones de alineación codifican el grado en que una palabra en una secuencia de entrada se relaciona con una palabra en la secuencia de salida.

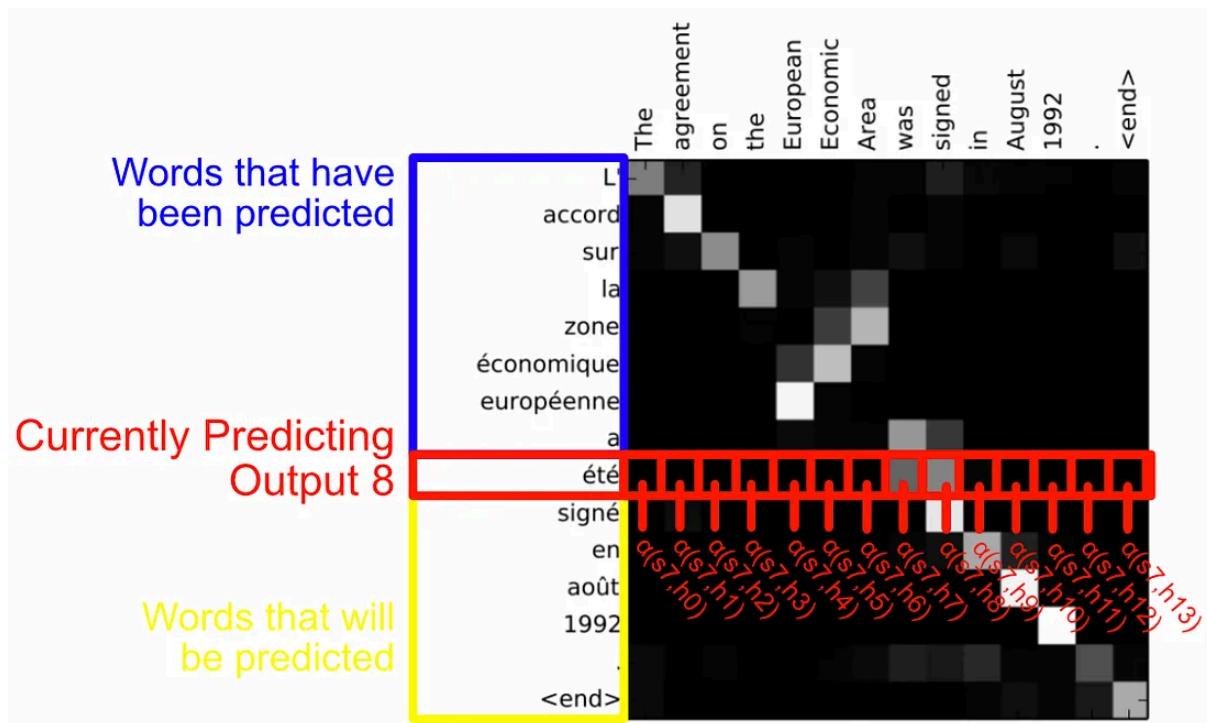


Dos ejemplos de matrices de atención para dos ejemplos diferentes del inglés al francés, de Neural Machine Translation de Jointly Learning to Align and Translate (2014). Este artículo sólo menciona tangencialmente el término "atención" y, de hecho, lo llama "modelo de alineación". El término "atención" parece haberse popularizado más tarde.

El [paper](#) presenta la siguiente función para calcular el puntaje de alineamiento:

$$a(s_{i-1}, h_j) = v_a^\top \tanh(W_a s_{i-1} + U_a h_j)$$

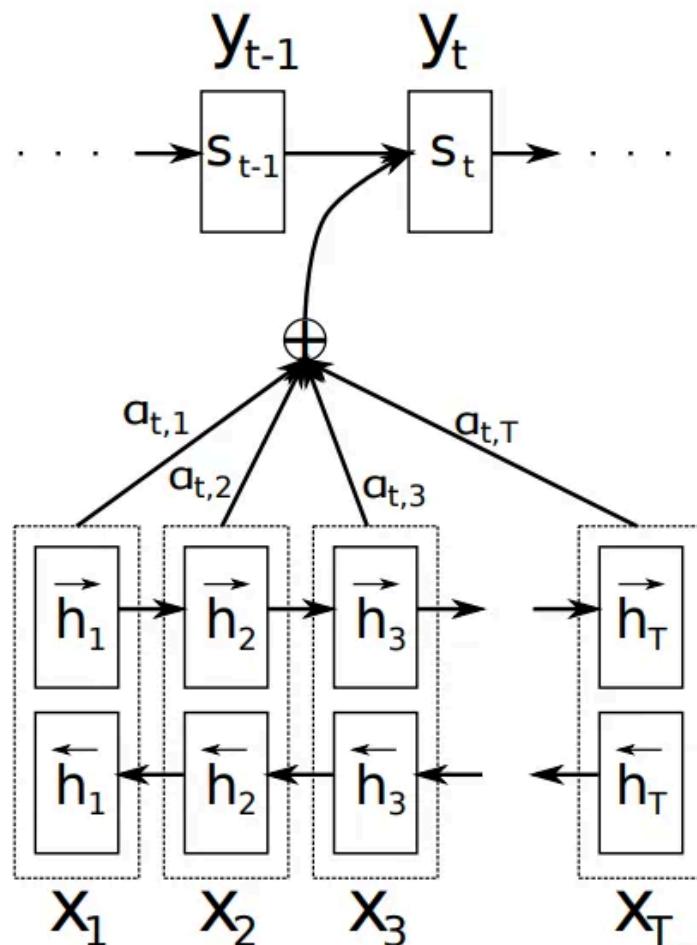
Esta función calcula una puntuación entre la siguiente palabra de salida y una única palabra de entrada, lo que indica qué tan relevante es una palabra de entrada para la salida actual. Esta función se ejecuta en todas las palabras de entrada ( $h_j$ ) para calcular una puntuación de alineación para todas las palabras de entrada dada la salida actual.



Un diagrama conceptual de cómo se calculan las alineaciones para una predicción determinada (palabra 8). La función de alineación se calcula entre la incorporación de salida anterior del decodificador y todas las entradas, para calcular la atención a la salida actual.

Se aplica una función softmax en todas las alineaciones calculadas, convirtiéndolas en una probabilidad. Esto se denomina en la literatura "búsqueda suave" o "alineación suave".

La forma exacta en que se utiliza la atención puede variar de una implementación a otra. En [Neural Machine Translation by Jointly Learning to Align and Translate \(2014\)](#), el mecanismo de atención decide qué embeddings de entrada proporcionar al decodificador.



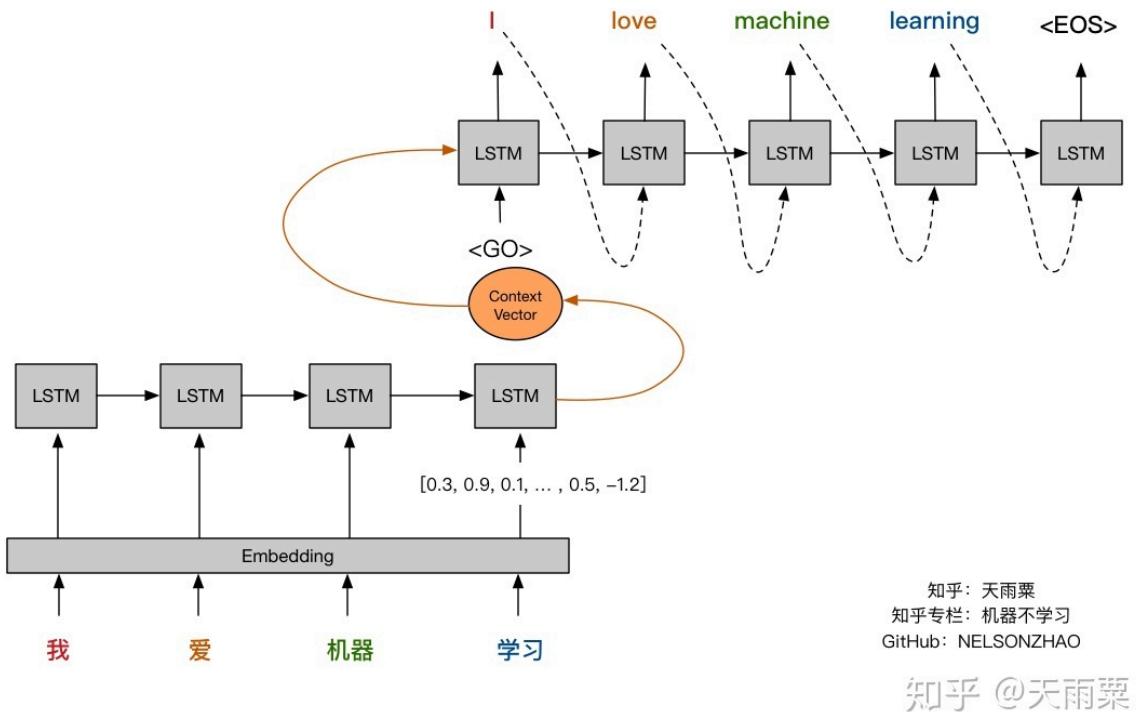
Una ilustración gráfica del modelo propuesto que intenta generar la  $t$ -ésima palabra objetivo  $y_t$  dada una oración fuente ( $x_1, x_2, \dots, x_T$ ). Cada incorporación de entrada se multiplica por su puntuación de alineación respectiva y luego se suman para formar el vector de contexto que se utiliza para el paso de salida del decodificador actual. De la traducción automática neuronal mediante el aprendizaje conjunto de alinear y traducir (2014).

## Resumen de atención aplicada a Seq2Seq

Las redes neuronales recurrentes, particularmente las unidades LSTM (Long Short-Term Memory), han demostrado ser excepcionalmente capaces en tareas que involucran secuencias, como la traducción automática. En el corazón de esta capacidad se encuentra el diseño de los modelos de secuencia a secuencia (seq2seq), que consisten en un codificador que transforma una secuencia de entrada en un vector de contexto, y un decodificador que produce una secuencia de salida basada en ese vector.

En la primera imagen, observamos un modelo seq2seq básico. La entrada se procesa token por token a través de una serie de unidades LSTM que actúan como codificador. El último estado de este codificador se utiliza como vector de

contexto, que se transfiere al decodificador, el cual genera la secuencia de salida.

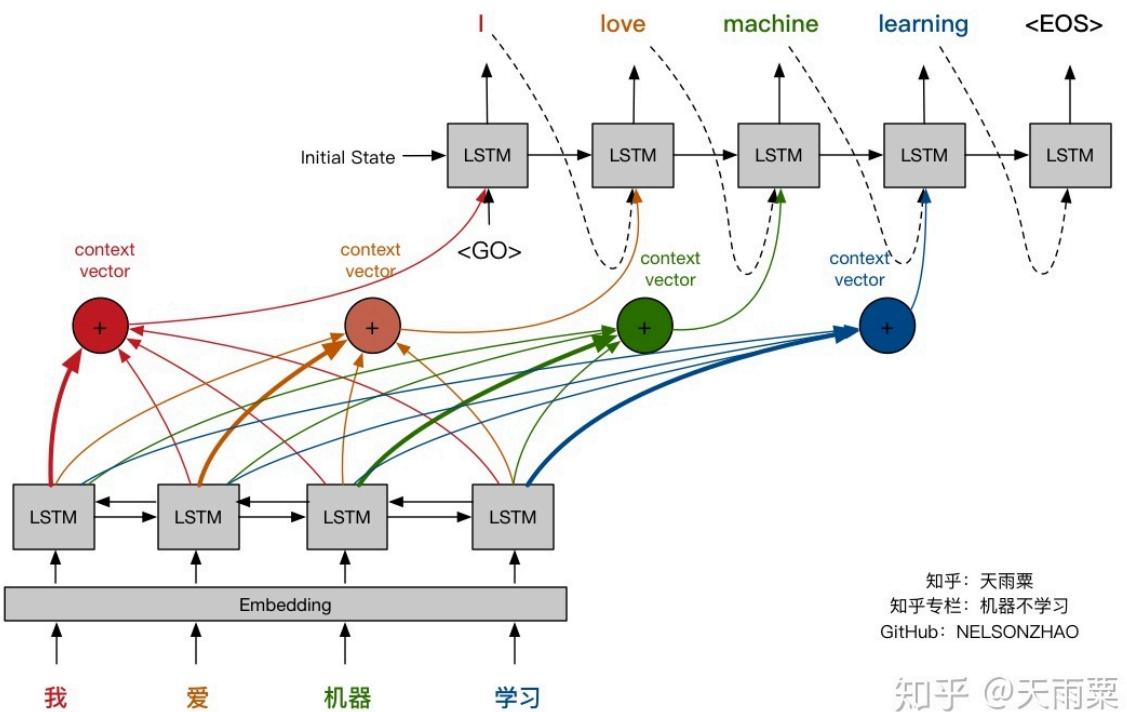


Sin embargo, a medida que las secuencias de entrada y salida se vuelven más largas, es posible que un simple vector de contexto no pueda retener toda la información necesaria.

Aquí es donde entra en juego el mecanismo de atención, como se muestra en la segunda imagen.

El mecanismo de atención permite que el decodificador se "enfoque" en diferentes partes de la secuencia de entrada mientras genera cada token de la secuencia de salida. Esto se visualiza con las conexiones coloridas entre el codificador y el decodificador, representando cómo diferentes partes de la entrada influyen en la producción de tokens específicos en la salida.

Al incorporar atención, los modelos seq2seq pueden manejar mejor las relaciones a largo plazo entre las secuencias de entrada y salida, haciendo que la traducción y otras tareas secuenciales sean más precisas y coherentes.



Mas información:

<https://www.davidsbatista.net/blog/2020/01/25/Attention-seq2seq/>

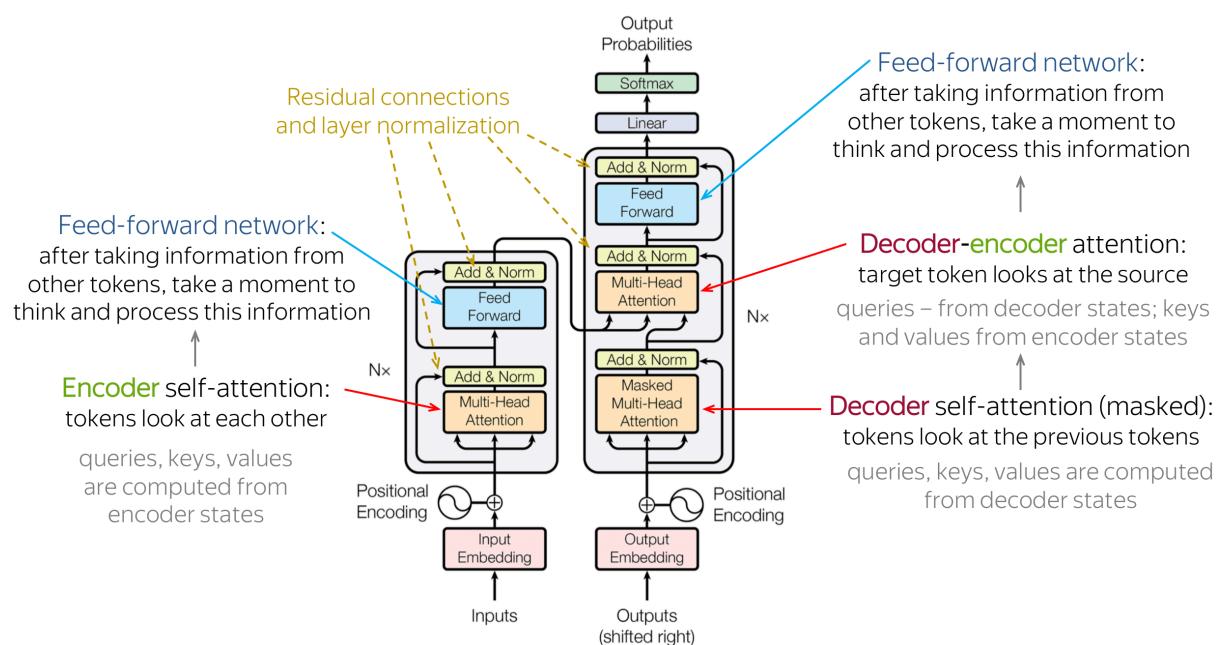
<https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/>

<https://towardsdatascience.com/attention-from-alignment-practically-explained-548ef6588aa4>

## 6. Transformers

En el mundo del procesamiento del lenguaje natural (NLP), la arquitectura de los Transformers ha emergido como un pilar revolucionario. Esta innovadora estructura, introducida en el influyente artículo de 2017 titulado "[Attention Is All You Need](#)" por Vaswani et al., cambió el paradigma de cómo las máquinas

comprenden y generan lenguaje. A diferencia de las técnicas anteriores que se basaban en redes neuronales recurrentes o convolucionales, los Transformers aprovechan el mecanismo conocido como "self-attention" para capturar interacciones en toda una secuencia, independientemente de la distancia entre los elementos. Esta capacidad de capturar relaciones contextuales a largo plazo ha catapultado a los Transformers al centro de NLP, siendo la base de modelos destacados como BERT y GPT. La influencia de este trabajo no puede ser subestimada; ha redefinido las mejores prácticas en NLP, impulsando avances significativos y permitiendo aplicaciones antes consideradas inalcanzables.



Arquitectura de un Transformer, según el paper "Attention is all you need"

## ¿Qué son los Transformers?

Un transformer es un tipo de arquitectura de red neuronal que está bien adaptado para tareas que involucran procesar secuencias como entradas. Quizás el ejemplo más común de una secuencia en este contexto es una oración, a la que podemos considerar como un conjunto ordenado de palabras.

El objetivo de estos modelos es crear una representación numérica para cada elemento dentro de una secuencia; encapsulando información esencial sobre el elemento y su contexto vecino. Las representaciones numéricas resultantes pueden ser transmitidas a redes subsecuentes, las cuales pueden aprovechar esta información para realizar diversas tareas, incluyendo generación y clasificación.

Al crear representaciones tan enriquecidas, estos modelos permiten que las redes subsecuentes comprendan mejor los patrones y relaciones subyacentes en la secuencia de entrada, mejorando su habilidad para generar salidas coherentes y contextualmente relevantes.

La principal ventaja de los transformers radica en su capacidad para manejar dependencias de largo alcance dentro de las secuencias, así como en ser altamente eficientes; capaces de procesar secuencias en paralelo. Esto es particularmente útil para tareas como la traducción automática, análisis de sentimientos y generación de texto.

El **Transformer** se compone principalmente de dos bloques fundamentales: el **bloque encoder** y el **bloque decoder**, los cuales trabajan en conjunto para procesar y transformar secuencias de datos, especialmente en tareas de traducción y generación de lenguaje. A continuación, un resumen del funcionamiento de ambos bloques.

1. **Bloque Encoder:** El bloque **encoder** es responsable de procesar la secuencia de entrada y generar una representación rica que captura la información clave de dicha secuencia. Su estructura básica incluye varios mecanismos clave:
  - a. **Capa de Auto-Atención (Self-Attention):** Esta es la parte central del encoder. La auto-atención permite que cada palabra en la secuencia de entrada se relacione con todas las demás palabras de esa secuencia. Esto significa que el modelo puede captar dependencias largas entre palabras. La auto-atención funciona calculando un conjunto de vectores de **consulta (query)**, **clave (key)** y **valor (value)**, y utilizando estos vectores para calcular una "atención ponderada" entre las palabras.
  - b. **Multi-Head Attention:** En lugar de usar una sola atención, el transformer utiliza múltiples "cabezas de atención" para capturar diferentes aspectos o relaciones en la secuencia. Cada cabeza de atención puede enfocarse en diferentes partes de la secuencia, lo que permite una mejor comprensión de las interdependencias entre las palabras.
  - c. **Feed-Forward Layer:** Despues de la capa de auto-atención, se pasa por una capa completamente conectada (feed-forward), que aplica transformaciones no lineales a cada posición de la secuencia de forma independiente.
  - d. **Normalización y Residuos (Layer Normalization and Residual Connections):** Cada subcapa en el encoder (auto-atención y feed-

forward) está acompañada de una capa de normalización y conexiones residuales. Esto ayuda a estabilizar el entrenamiento y mejorar la eficiencia del modelo.



La salida del encoder de un transformer codifica una representación rica y densa de la secuencia de entrada, donde cada token se representa no solo en función de sí mismo, sino también de su relación con todos los demás tokens de la secuencia. Esta representación incluye información semántica, sintáctica y de contexto posicional, permitiendo al modelo capturar dependencias de largo y corto alcance dentro del texto.

El encoder, al final de su proceso, genera una secuencia de vectores de representación que encapsulan la información de entrada, y esta secuencia es pasada al decoder.

[attachment:4651ed8f-3243-4b8e-8e35-a39b16fab04f:PQLFZoCrH3U6I4Gt.mp4](#)

2. **Bloque Decoder:** El bloque decoder toma la salida del encoder (es decir, la representación generada) y genera la secuencia de salida. Está compuesto por las siguientes capas clave:
  - a. **Masked Multi-Head Self-Attention:** Similar a la capa de auto-atención en el encoder, pero en este caso se aplica un enmascaramiento para evitar que el decoder mire palabras futuras durante la generación de la secuencia. Este enmascaramiento asegura que el modelo genere las palabras una a una en el orden correcto.
  - b. **Multi-Head Attention sobre la Salida del Encoder:** El decoder también tiene una capa de atención que se aplica sobre la salida del encoder. Esto permite que el decoder se "enfoque" en diferentes partes de la secuencia de entrada mientras genera cada palabra en la secuencia de salida.
  - c. **Feed-Forward Layer:** Al igual que en el encoder, el decoder también tiene capas feed-forward que aplican transformaciones no lineales a las

representaciones intermedias.

- d. **Normalización y Conexiones Residuales:** También utiliza normalización y conexiones residuales, de forma similar al encoder, para mejorar la estabilidad y la eficiencia.

Flujo general:

- El encoder procesa la secuencia de entrada en paralelo, generando representaciones vectoriales que capturan relaciones entre las palabras.
- El decoder utiliza estas representaciones junto con la secuencia generada previamente para producir la secuencia de salida de manera autoregresiva, es decir, una palabra a la vez.

## Tokenización

Antes de que un Transformer procese cualquier frase o texto, este debe ser convertido a una forma que la máquina pueda entender. Este proceso se conoce como tokenización.

La tokenización divide el texto en unidades más pequeñas, llamadas tokens. Estos tokens pueden ser tan pequeños como caracteres o tan grandes como palabras. Sin embargo, simplemente dividir el texto en palabras individuales no es suficiente para muchos idiomas y tareas, especialmente cuando enfrentamos desafíos como palabras fuera del vocabulario.

Para abordar este problema, se emplean métodos avanzados de tokenización. Uno de los métodos más populares es el "Byte Pair Encoding" (BPE). BPE funciona combinando iterativamente los pares de caracteres o tokens más frecuentes en un corpus de texto, permitiendo manejar palabras raras o desconocidas y generar un vocabulario dinámico. Al utilizar BPE, es posible descomponer palabras en subpalabras o subtokens, lo que facilita que el modelo maneje una amplia gama de vocabulario sin aumentar excesivamente su tamaño.

Veamos un ejemplo de tokenización:

```
from transformers import BertTokenizer, BertModel  
import torch  
  
# Cargar el tokenizador y modelo  
tokenizer = BertTokenizer.from_pretrained('bert-base-multilingual-uncased')
```

```

model = BertModel.from_pretrained("bert-base-multilingual-uncased")

# Texto de ejemplo
text = "Estamos tratando de entender como funcionan los transformers."

# Tokenizar el texto
encoded_input = tokenizer(text, return_tensors='pt')

# Obtener los tokens IDs
token_ids = encoded_input['input_ids'][0]
print('Token IDs:')
print([int(tok_id) for tok_id in token_ids])

# Decodificar el texto completo para verificar
decoded_text = tokenizer.decode(token_ids)
print('\nDecodificamos los IDs completos:')
print(decoded_text)

# Mostrar cada token y su ID correspondiente
print('\nTokens individuales y sus IDs:')
for token_id in token_ids:
    token = tokenizer.decode([token_id])
    print(f"ID: {int(token_id)}, Token: '{token}'")

# Mostrar las subpalabras usando tokenize() que devuelve las subunidades
print('\nSubpalabras (tokens) usando tokenize():')
tokens = tokenizer.tokenize(text)
print(tokens)

# Para ver la correspondencia entre subpalabras y IDs
print('\nSubpalabras con sus IDs correspondientes:')
for token in tokens:
    id = tokenizer.convert_tokens_to_ids(token)
    print(f"Token: '{token}', ID: {id}")

# Ejecutar el modelo
output = model(**encoded_input)

```

Donde se imprime:

Token IDs:  
[101, 10602, 14000, 21139, 10622, 10102, 66325, 10245, 61562, 10115, 10175, 58263]

Decodificamos los IDs completos:  
[CLS] estamos tratando de entender como funcionan los transformers. [SEP]

Tokens individuales y sus IDs:

ID: 101, Token: '[CLS]'  
ID: 10602, Token: 'esta'  
ID: 14000, Token: '##mos'  
ID: 21139, Token: 'trata'  
ID: 10622, Token: '##ndo'  
ID: 10102, Token: 'de'  
ID: 66325, Token: 'entender'  
ID: 10245, Token: 'como'  
ID: 61562, Token: 'funciona'  
ID: 10115, Token: '##n'  
ID: 10175, Token: 'los'  
ID: 58263, Token: 'transformers'  
ID: 119, Token: ':'  
ID: 102, Token: '[SEP]'

Subpalabras (tokens) usando tokenize():

['esta', '##mos', 'trata', '##ndo', 'de', 'entender', 'como', 'funciona', '##n', 'los', ':']

Subpalabras con sus IDs correspondientes:

Token: 'esta', ID: 10602  
Token: '##mos', ID: 14000  
Token: 'trata', ID: 21139  
Token: '##ndo', ID: 10622  
Token: 'de', ID: 10102  
Token: 'entender', ID: 66325  
Token: 'como', ID: 10245  
Token: 'funciona', ID: 61562  
Token: '##n', ID: 10115  
Token: 'los', ID: 10175

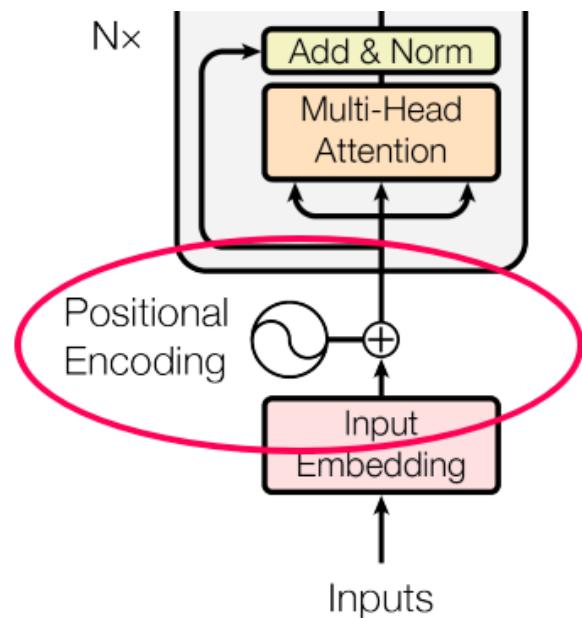
Token: 'transformers', ID: 58263

Token: '!', ID: 119

Los tokens especiales **[CLS]** y **[SEP]** juegan un papel importante en la arquitectura de modelos de lenguaje basados en transformadores, como BERT, para ayudar en el procesamiento y entendimiento de secuencias de texto. El token **[CLS]** (abreviatura de "classification") es un token especial que se inserta al principio de cada secuencia de entrada en modelos como BERT. Sirve como un marcador que el modelo utiliza para generar una representación global o resumida de la secuencia completa. El token **[SEP]** (abreviatura de "separator") es un token especial que se utiliza para indicar el final de una secuencia o para separar dos secuencias de texto diferentes.

### Embeddings y codificación posicional

Una vez que tenemos una secuencia de enteros que representa nuestra entrada, podemos convertirla en embeddings, que son una forma de representar información que puede ser fácilmente procesada por algoritmos de aprendizaje automático; su objetivo es capturar el significado del token que se está codificando en un formato comprimido, representando la información como una secuencia de números, y además captura relaciones semánticas entre las frases o palabras.



Los Transformers proponen un mecanismo novedoso para procesar los embeddings, que es la codificación posicional (Positional Encoding). Esto resulta muy útil para preservar el orden de nuestros tokens.

El codificado posicional, o "positional encoding", es una técnica utilizada en modelos de transformers para dotar a la arquitectura de la capacidad de tener en cuenta el orden o la posición de los tokens en una secuencia. El codificado posicional proporciona la información necesaria sobre la posición de cada token

en una secuencia, permitiendo que el modelo distinga el orden en el que los tokens aparecen.

[https://www.youtube.com/embed/xi94v\\_jl26U?start=2m16s&end=13m46s](https://www.youtube.com/embed/xi94v_jl26U?start=2m16s&end=13m46s)

Para implementar el codificado posicional, se generan valores numéricos para cada posición en la secuencia y se suman a las incrustaciones (embeddings) originales de los tokens. Estos valores son calculados de tal manera que cada posición en la secuencia tiene un codificado posicional único.



Una forma común de hacer esto es mediante funciones sinusoidales, que generan valores que varían de manera periódica. De esta manera, incluso si las secuencias tienen longitudes diferentes, el modelo puede inferir la relación posicional entre los tokens y, por lo tanto, comprender el orden y la estructura de la secuencia.

Veamos un ejemplo conceptual de cómo se hace la codificación posicional, y la inversa, cómo a partir del embedding codificado, podemos obtener la palabra:

```
# !pip install spacy
# !python -m spacy download es_core_news_md

import spacy
import numpy as np

# Cargar el modelo de spacy
nlp = spacy.load("es_core_news_md")

def get_word_embedding(word):
    token = nlp(word)
    return token.vector
```

```

def get_angles(pos, i, d_model):
    angle_rates = 1 / np.power(10000, (2 * (i // 2)) / np.float32(d_model))
    return pos * angle_rates

def positional_encoding(position, d_model):
    angle_rads = get_angles(np.arange(d_model)[np.newaxis, :], position[:, np.newaxis], np.pi / 10000)
    sines = np.sin(angle_rads[:, 0::2])
    cosines = np.cos(angle_rads[:, 1::2])
    pos_encoding = np.concatenate([sines, cosines], axis=-1)
    return pos_encoding

# Frase de ejemplo
sentence = "Lo que mata es la humedad"
words = sentence.split()

# Parámetros
d_model = 300 # Tamaño del embedding de Spacy
max_position = len(words)

# Obtener embeddings de palabras y codificación posicional
word_embeddings = np.array([get_word_embedding(word) for word in words])
pos_encoding = positional_encoding(np.arange(max_position), d_model)

# Combinar embeddings con codificación posicional
combined_embeddings = word_embeddings + pos_encoding

# Función para adivinar la posición basada en el embedding combinado
def guess_position(combined_embedding, words):
    min_diff = float('inf')
    guessed_position = -1
    for idx, word in enumerate(words):
        word_embedding = get_word_embedding(word)
        pos_enc = positional_encoding(np.array([idx]), d_model)[0]
        combined = word_embedding + pos_enc
        diff = np.sum(np.square(combined_embedding - combined))
        if diff < min_diff:
            min_diff = diff
            guessed_position = idx
    return guessed_position

```

```

# Demostración de cómo adivinar la posición de una palabra
for i, word in enumerate(words):
    combined_embedding = combined_embeddings[i]
    guessed_pos = guess_position(combined_embedding, words)
    print(f"Palabra: '{word}', Posición real: {i}, Posición adivinada: {guessed_pos}")

# Ejemplo adicional con una palabra específica
word_to_guess = words[1] # 'que'
combined_embedding_to_guess = combined_embeddings[1]
position = guess_position(combined_embedding_to_guess, words)
print(f"\nLa posición de la palabra '{word_to_guess}' obtenida desde el embedding codificado es {position}\n")

# Imprime:
# Palabra: 'Lo', Posición real: 0, Posición adivinada: 0
# Palabra: 'que', Posición real: 1, Posición adivinada: 1
# Palabra: 'mata', Posición real: 2, Posición adivinada: 2
# Palabra: 'es', Posición real: 3, Posición adivinada: 3
# Palabra: 'la', Posición real: 4, Posición adivinada: 4
# Palabra: 'humedad', Posición real: 5, Posición adivinada: 5
#
# La posición de la palabra 'que' obtenida desde el embedding codificado es 1

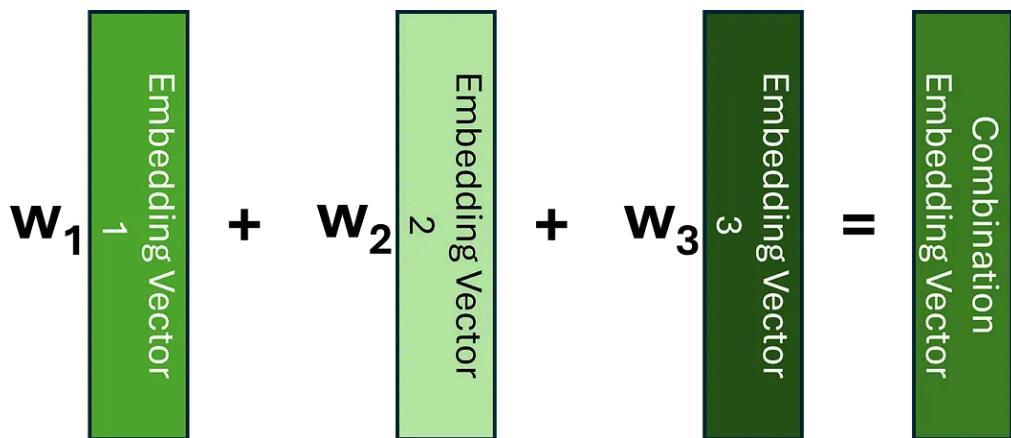
```

## Atención

Quizás el mecanismo más importante utilizado por la arquitectura del transformer es conocido como atención, que permite a la red entender qué partes de la secuencia de entrada son las más relevantes para la tarea dada. Para cada token en la secuencia, el mecanismo de atención identifica qué otros tokens son importantes para comprender el token actual en el contexto dado. Antes de explorar cómo se implementa esto dentro de un transformer, comencemos de manera simple e intentemos entender conceptualmente qué intenta lograr el mecanismo de atención, para construir nuestra intuición.

Una forma de entender la atención es pensar en ella como un método que reemplaza cada embedding de token con un embedding que incluye información sobre sus tokens vecinos; en lugar de usar la misma incrustación para cada token independientemente de su contexto. Si supiéramos qué tokens son relevantes para el token actual, una forma de capturar este contexto sería

crear un promedio ponderado —o, más generalmente, una combinación lineal— de estas incrustaciones.



Mientras procesa una palabra, la Atención permite al modelo concentrarse en otras palabras de la entrada que están estrechamente relacionadas con esa palabra.

Por ejemplo, "Pelota" está estrechamente relacionada con "azul" y "sostiene". Por otro lado, "azul" no está relacionado con "niño".



La arquitectura Transformer utiliza auto-atención (self-attention) relacionando cada palabra en la secuencia de entrada con cada otra palabra.

Por ejemplo, consideremos estas dos frases:

- The cat drank the milk because **it** was hungry.
- The cat drank the milk because **it** was sweet.

En la primera frase, la palabra 'it' se refiere a 'cat', mientras que en la segunda se refiere a 'milk'. Cuando el modelo procesa la palabra 'it', la auto-atención

proporciona al modelo más información sobre su significado para que pueda asociar 'it' con la palabra correcta.



Para permitirle manejar más matices sobre la intención y semántica de la oración, los Transformers incluyen múltiples puntajes (score) de atención para cada palabra. Esto se logra gracias a "Múltiples Cabezas de Atención" (o multi-head attention en inglés). Las múltiples cabezas de atención actúan como diferentes "lentes" o "filtros" que permiten al Transformer ver y capturar diferentes aspectos de los datos simultáneamente.

Por ejemplo, al procesar la palabra "it", el primer puntaje resalta "cat", mientras que el segundo puntaje resalta "hungry". Así que cuando decodifica la palabra "it", por ejemplo, al traducirla a otro idioma, incorporará algún aspecto tanto de "cat" como de "hungry" en la palabra traducida.

The	The	The
cat	cat	cat
drank	drank	drank
the	the	the
milk	milk	milk
because	because	because
it	it	it
was	was	was
hungry	hungry	hungry

Input      Score 1      Score 2

Google Colab

☞ [https://colab.research.google.com/drive/17gRrUxBNFAHsh5ZZugDR\\_9I515PbS1Xp?usp=sharing](https://colab.research.google.com/drive/17gRrUxBNFAHsh5ZZugDR_9I515PbS1Xp?usp=sharing)

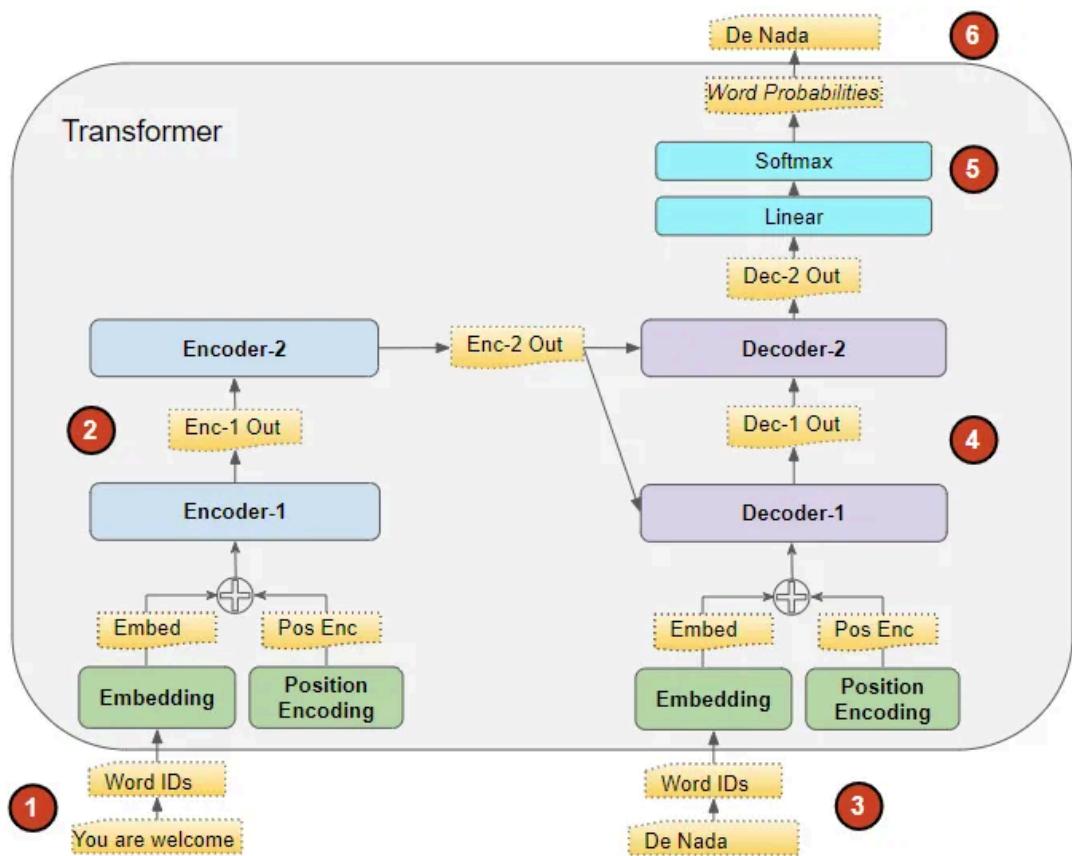


## Entrenando un Transformer

El Transformer funciona de manera ligeramente diferente durante el Entrenamiento y durante la Inferencia. Veamos primero el flujo de datos durante el Entrenamiento. Los datos de entrenamiento constan de dos partes:

- La secuencia fuente o de entrada (por ejemplo, "You are welcome" en inglés, para un problema de traducción)
- La secuencia de destino o objetivo (por ejemplo, "De nada" en español)

El objetivo del Transformer es aprender a producir la secuencia objetivo, utilizando tanto la secuencia de entrada como la secuencia objetivo.



El Transformer procesa los datos de la siguiente manera:

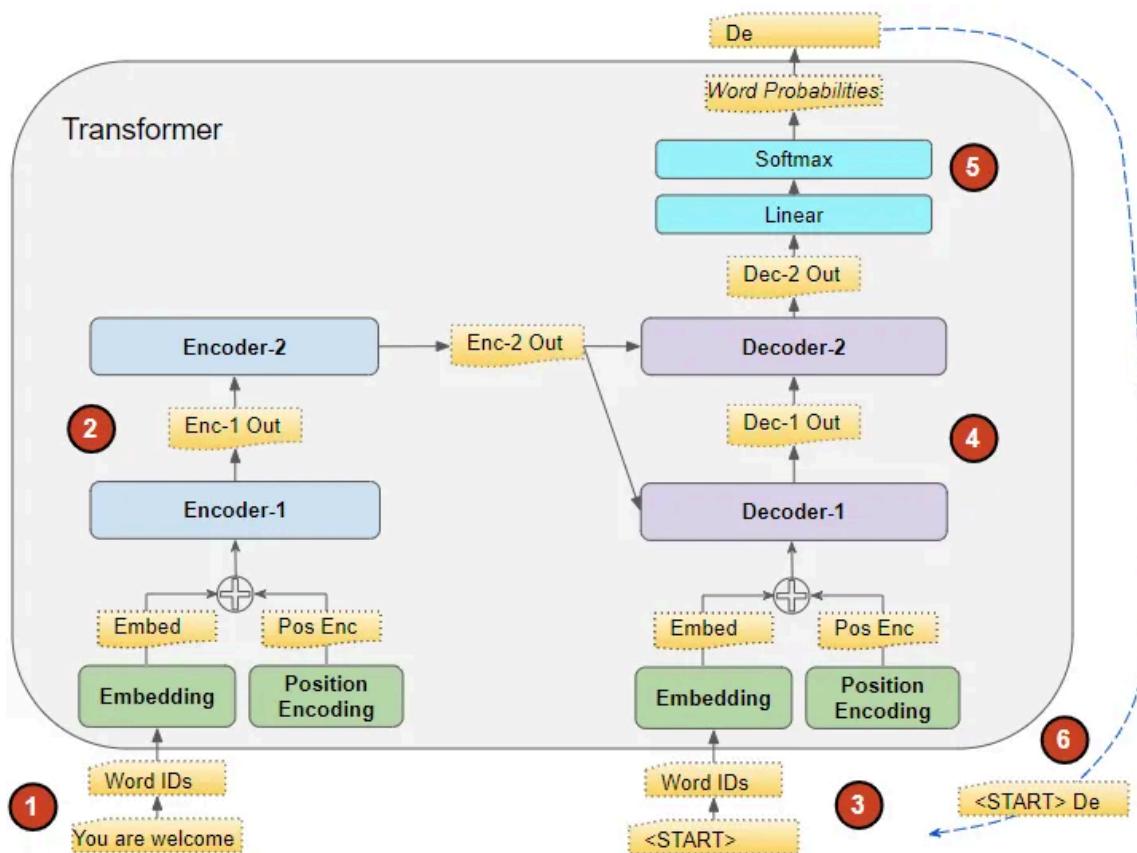
1. La secuencia de entrada se convierte en embeddings (con Codificación de Posición) y se alimenta al Encoder.
2. El conjunto de Encoders procesa esto y produce una representación codificada de la secuencia de entrada.
3. La secuencia objetivo se antepone con un token de inicio de oración, se convierte en embeddings (con Codificación de Posición) y se alimenta al Decoder.
4. El conjunto de Decoders procesa esto junto con la representación codificada del conjunto de Encoders para producir una representación codificada de la secuencia objetivo.
5. La capa de Salida lo convierte en probabilidades de palabras y la secuencia de salida final.
6. La función de Pérdida del Transformer compara esta secuencia de salida con la secuencia objetivo de los datos de entrenamiento. Esta pérdida se utiliza para generar gradientes para entrenar el Transformer durante la retropropagación.

## Inferencia

Durante la Inferencia, solo tenemos la secuencia de entrada y no contamos con la secuencia objetivo para pasar como entrada al Decoder. El objetivo del Transformer es producir la secuencia objetivo solo a partir de la secuencia de entrada.

Por lo tanto, al igual que en un modelo Seq2Seq, generamos la salida en un bucle y alimentamos la secuencia de salida del paso de tiempo anterior al Decoder en el siguiente paso de tiempo hasta que nos encontramos con un token de fin de oración.

La diferencia con el modelo Seq2Seq es que, en cada paso de tiempo, volvemos a alimentar la secuencia de salida generada hasta ese momento, en lugar de solo la última palabra.



Durante la secuencia de inferencia se dan los siguientes pasos:

1. La secuencia de entrada se convierte en embeddings (con Codificación de Posición) y se alimenta al Encoder.
2. El conjunto de Encoders procesa esto y produce una representación codificada de la secuencia de entrada.

3. En lugar de la secuencia objetivo, usamos una secuencia vacía con solo un token de inicio de oración. Esto se convierte en embeddings (con Codificación de Posición) y se alimenta al Decoder.
4. El conjunto de Decoders procesa esto junto con la representación codificada del conjunto de Encoders para producir una representación codificada de la secuencia objetivo.
5. La capa de Salida lo convierte en probabilidades de palabras y produce una secuencia de salida.
6. Tomamos la última palabra de la secuencia de salida como la palabra predicha. Esa palabra se coloca ahora en la segunda posición de nuestra secuencia de entrada del Decoder, que ahora contiene un token de inicio de oración y la primera palabra.
7. Vuelve al paso #3. Como antes, alimenta la nueva secuencia de Decoder en el modelo. Luego toma la segunda palabra de la salida y se añade a la secuencia del Decoder. Repite esto hasta que prediga un token de fin de oración. Tengamos en cuenta que, dado que la secuencia del Encoder no cambia en cada iteración, no tenemos que repetir los pasos #1 y #2 cada vez.

### **Forzado del Maestro (Teacher Forcing)**

El enfoque de alimentar la secuencia objetivo al Decoder durante el entrenamiento se conoce como Forzado del Maestro. ¿Por qué hacemos esto y qué significa ese término?

Durante el entrenamiento, podríamos haber usado el mismo enfoque que se utiliza durante la inferencia. En otras palabras, ejecutar el Transformer en un bucle, tomar la última palabra de la secuencia de salida, añadirla a la entrada del Decoder y alimentarla al Decoder para la próxima iteración. Finalmente, cuando se predice el token de fin de oración, la función de Pérdida compararía la secuencia de salida generada con la secuencia objetivo para entrenar la red.

No solo este bucle haría que el entrenamiento tomara mucho más tiempo, sino que también dificultaría el entrenamiento del modelo. El modelo tendría que predecir la segunda palabra basándose en una primera palabra predicha potencialmente errónea, y así sucesivamente.

En cambio, al alimentar la secuencia objetivo al Decoder, estamos dándole una pista, por así decirlo, al igual que lo haría un Maestro. Aunque predijo una primera palabra errónea, puede usar la primera palabra correcta para predecir la segunda palabra para que esos errores no sigan acumulándose.

Además, el Transformer puede producir todas las palabras en paralelo sin bucle, lo que acelera enormemente el entrenamiento.

Más información:

<https://ig.ft.com/generative-ai/>

## 7. Large Language Models (Grandes Modelos de Lenguaje)

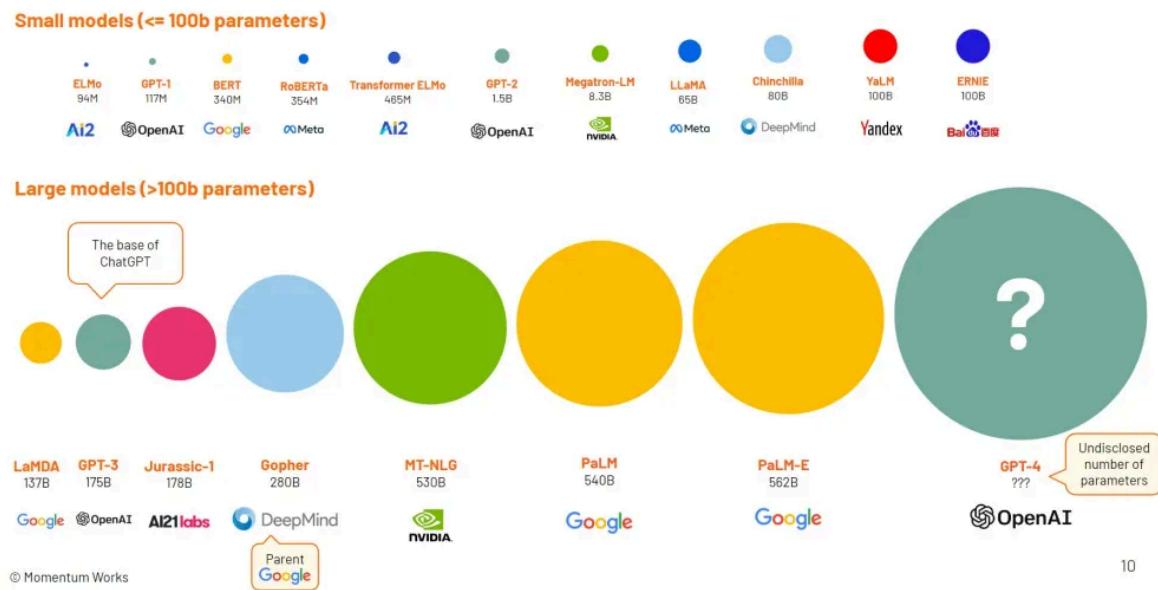
Los Modelos de Lenguaje de Gran Escala (Large Language Models o LLMs) representan una categoría de modelos de aprendizaje profundo diseñados para entender y generar texto humano. Estos modelos se entrena en vastas cantidades de datos textuales y tienen la capacidad de comprender y generar lenguaje de manera coherente, además de desempeñar tareas específicas como traducción, respuesta a preguntas y redacción de texto.

El entrenamiento de estos modelos requiere grandes conjuntos de datos textuales, y el proceso puede ser intensivo en términos de recursos computacionales.

Entre los LLMs notables se incluyen GPT-3 de OpenAI, BERT de Google y RoBERTa de Facebook. Estos modelos han establecido nuevos estándares en varias tareas de procesamiento de lenguaje natural.

Los LLM, se caracterizan por una elevada cantidad de parámetros, y es una forma de medirlos o compararlos entre ellos. En líneas generales, podemos decir que cuanto más grandes son los parámetros aprendidos, más potente en el modelo en tareas de NLP, aunque se está trabajando mucho en la optimización, ya que tamaños de modelos demasiado grandes, sólo pueden entrenarse por grandes compañías que poseen los recursos computacionales para poder hacerlo, y se tornan prohibitivos para aquellos que quieren contar con sus propios modelos.

## Large Language Models are becoming very large indeed



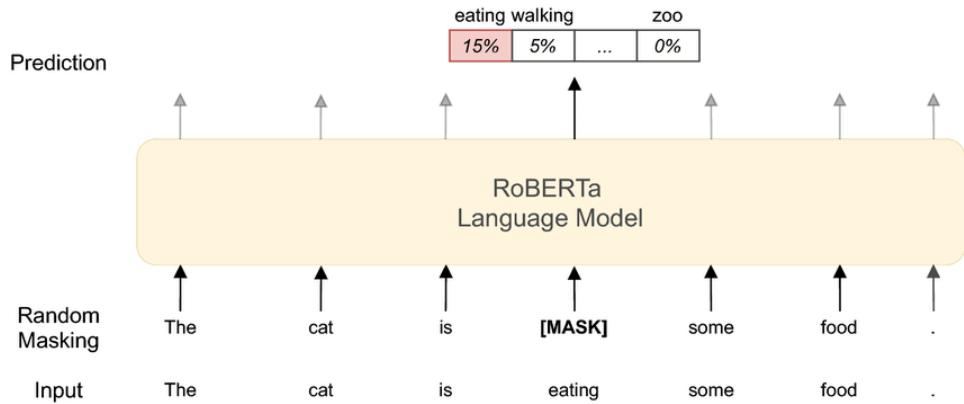
10

Comparativa de tamaños de algunos modelos de lenguaje más populares.

El reciente paper "[Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond](#)" presenta una guía exhaustiva de diferentes modelos de lenguaje. Allí se clasifican principalmente en dos tipos:

### Encoder-Decoder o Encoder-only (Estilo BERT)

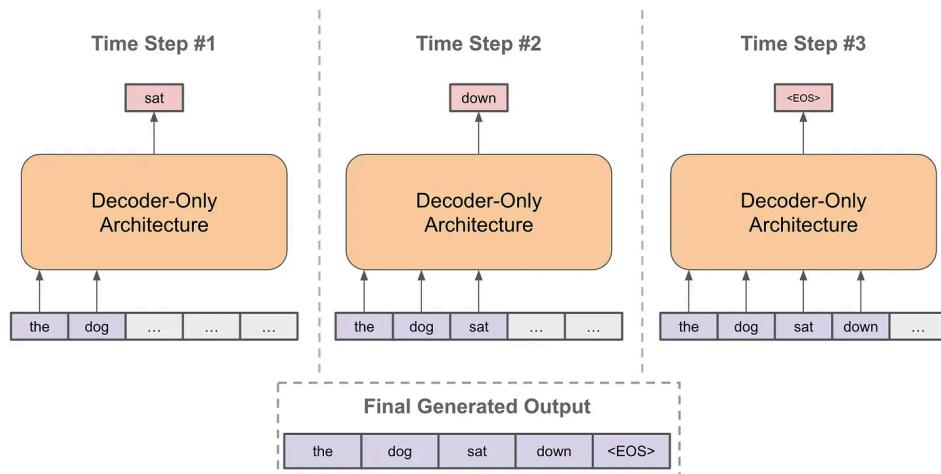
- **Descripción:** Estos modelos utilizan una arquitectura basada en el paradigma de "Masked Language Model" (MLM). La idea es predecir palabras enmascaradas en una oración considerando el contexto circundante. Esta forma de entrenamiento permite que el modelo desarrolle una comprensión más profunda de las relaciones entre palabras y el contexto en el que se usan.
- **Características:**
  - **Entrenamiento:** Se basan en el entrenamiento de "Masked Language Models" (MLMs). En este enfoque, ciertas palabras de una oración se "enmascaran" y el modelo intenta predecir estas palabras enmascaradas basándose en el contexto proporcionado por las palabras no enmascaradas.



- **Tipo de modelo:** Discriminativo.
- **Tarea de pre-entrenamiento:** Predicción de palabras enmascaradas.
- **Ejemplos notables:** BERT, RoBERTa, T5, entre otros. Estos modelos han sido entrenados en grandes corpus de texto utilizando técnicas como la arquitectura Transformer y han logrado resultados de vanguardia en muchas tareas de NLP.
- **Uso:** Estos modelos son especialmente útiles para tareas que requieren una comprensión profunda del lenguaje, como el análisis de sentimientos, la inferencia del lenguaje natural y la respuesta a preguntas.

### Decoder-only (Estilo GPT):

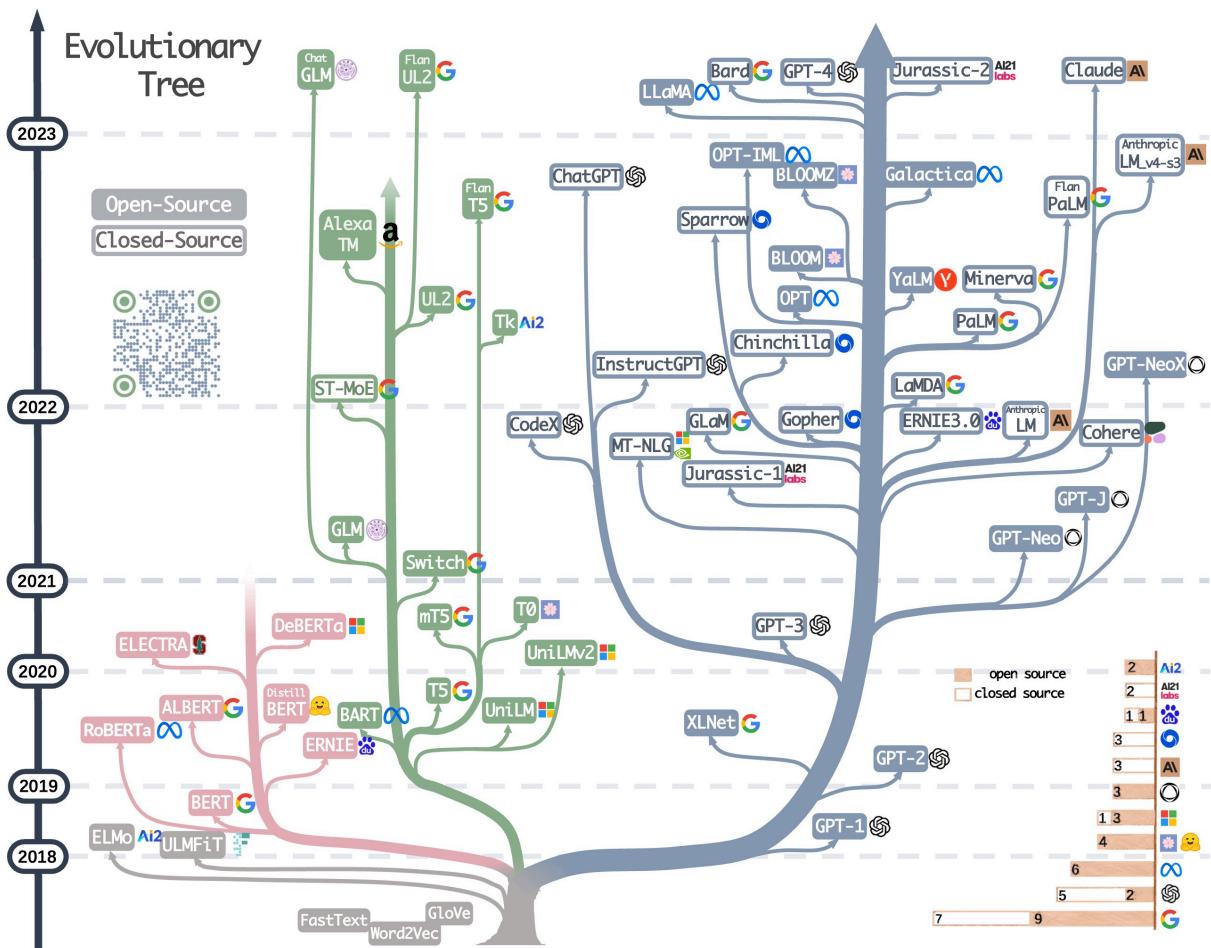
- **Descripción:** Estos modelos utilizan una arquitectura basada en el paradigma de "Autoregressive Language Model". Están entrenados para generar la siguiente palabra en una secuencia dadas las palabras anteriores. Estos modelos han sido ampliamente utilizados para tareas downstream como la generación de texto y la respuesta a preguntas.



- **Características:**

- **Entrenamiento:** Se basan en el entrenamiento de "Autoregressive Language Models". En este enfoque, el modelo intenta predecir la siguiente palabra en una secuencia basándose en las palabras anteriores.
- **Tipo de modelo:** Generativo.
- **Tarea de preentrenamiento:** Predicción de la siguiente palabra.
- **Ejemplos notables:** GPT-3, OPT, PaLM, BLOOM, entre otros. GPT-3, en particular, demostró un rendimiento razonable en configuraciones de pocas (few-shot) o cero muestras (zero-shot) mediante el uso de instrucciones y aprendizaje en contexto.
- **Uso:** Estos modelos son especialmente útiles para tareas que requieren generación de texto, como chatbots, generación de código y tareas que requieren respuestas generativas a preguntas.

Los modelos han evolucionado en los últimos años como si fueran parte de un árbol genealógico, mejorando u optimizando a las generaciones anteriores:

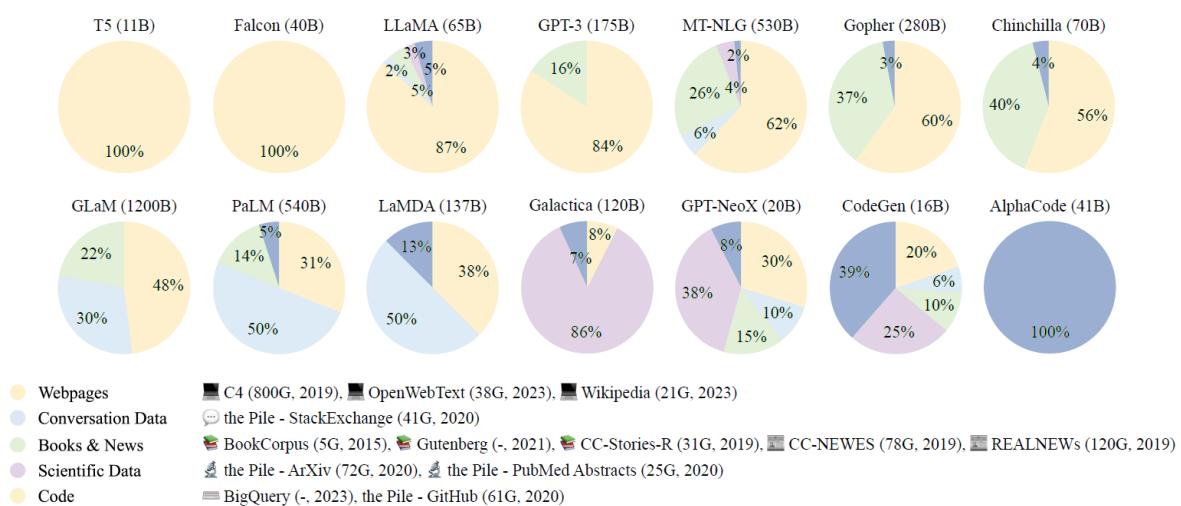


El árbol evolutivo de los LLM modernos rastrea el desarrollo de los modelos lingüísticos en los últimos años y destaca algunos de los modelos más conocidos. Los modelos de la misma rama tienen relaciones más estrechas. Los modelos basados en transformers se muestran en colores distintos del gris: los modelos solo decodificador en la rama azul, los modelos solo codificador en la rama rosa y los modelos codificador-decodificador en la rama verde. La posición vertical de los modelos en la línea de tiempo representa sus fechas de lanzamiento. Los modelos de código abierto están representados por cuadrados sólidos, mientras que los modelos de código cerrado están representados por cuadrados huecos. El gráfico de barras apiladas en la esquina inferior derecha muestra la cantidad de modelos de varias empresas e instituciones.

## Datos de preentrenamiento

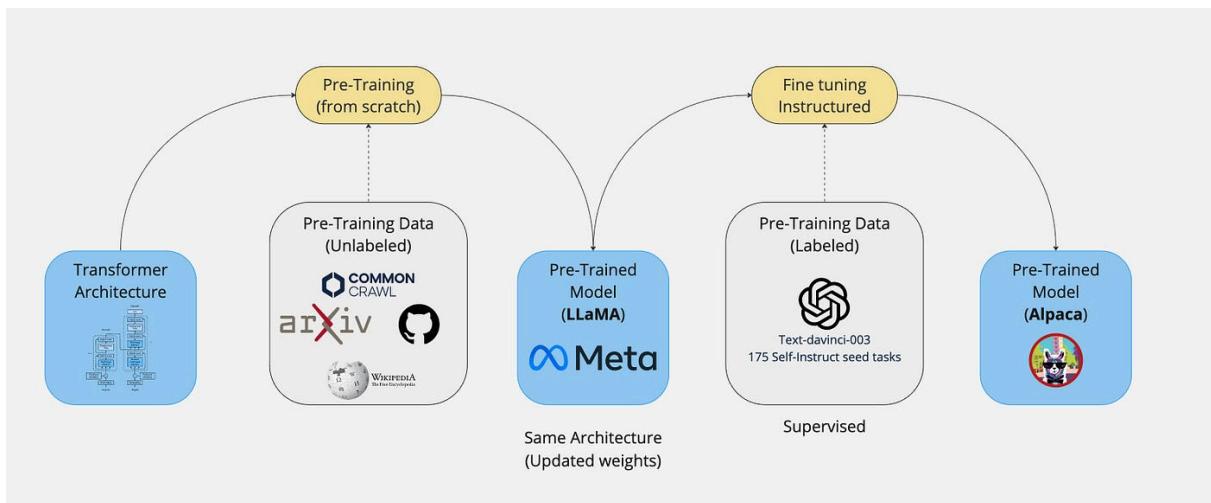
Los datos de preentrenamiento juegan un papel fundamental en el desarrollo de modelos de lenguaje a gran escala. Como base de las notables capacidades de los LLMs, la calidad, cantidad y diversidad de los datos de preentrenamiento influyen significativamente en el rendimiento de los LLMs. Los datos de preentrenamiento comúnmente utilizados constan de una miríada de fuentes de texto, incluyendo libros, artículos y sitios web. Los datos son cuidadosamente seleccionados para asegurar una representación integral del conocimiento

humano, matices lingüísticos y perspectivas culturales. La importancia de los datos de preentrenamiento radica en su capacidad para informar al modelo de lenguaje con una rica comprensión del conocimiento de las palabras, gramática, sintaxis y semántica, así como la habilidad de reconocer el contexto y generar respuestas coherentes. La diversidad de los datos de preentrenamiento también juega un papel crucial en la formación del rendimiento del modelo, y la selección de LLMs depende en gran medida de los componentes de los datos de preentrenamiento. Por ejemplo, PaLM y BLOOM sobresalen en tareas multilingües y traducción automática con una abundancia de datos de preentrenamiento multilingüe. Además, el rendimiento de PaLM en tareas de respuesta a preguntas se mejora al incorporar una cantidad considerable de conversaciones en redes sociales y corpus de libros. Del mismo modo, las capacidades de ejecución y completado de código de GPT-3.5 (code-davinci-002) se potencian con la integración de datos de código en su conjunto de datos de preentrenamiento. En resumen, al seleccionar LLMs para tareas downstream, es recomendable elegir el modelo preentrenado en un campo de datos similar.



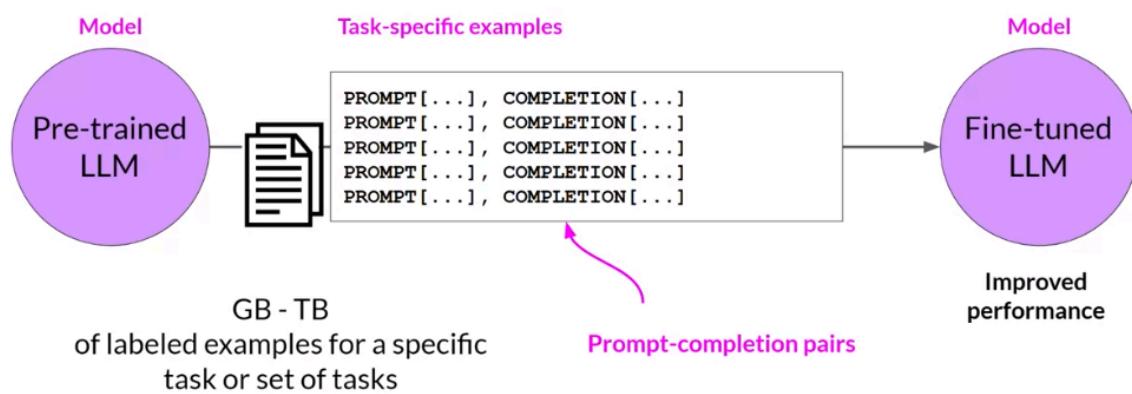
Proporciones de varias fuentes de datos en los datos previos a la capacitación para LLM existentes.

## Finetuning (Ajuste fino)



## LLM fine-tuning at a high level

### LLM fine-tuning



Papers:

[Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond](#)

[A Survey of Large Language Models](#)

[Challenges and Applications of Large Language Models](#)

