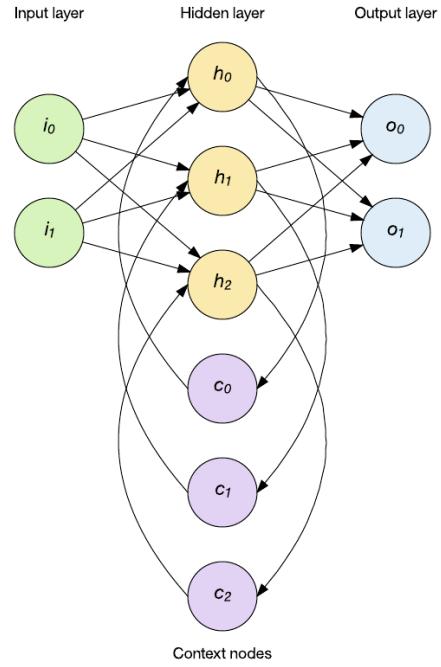


# Unidad 6. Redes neuronales recurrentes



“Programación de Inteligencia Artificial”  
Curso de Especialización en Inteligencia Artificial  
y Big Data

## Contenido

1. Introducción a las Redes Neuronales Recurrentes.....	3
1.1. ¿Por qué necesitamos RNN? .....	3
1.2. Aplicaciones de las RNN .....	3
2. Arquitectura y Funcionamiento de las RNR .....	4
2.1. Estructura de una RNN básica .....	5
2.2. Propagación hacia adelante en una RNN .....	6
2.3. Propagación hacia atrás a través del tiempo.....	6
2.4. Problemas del entrenamiento de RNN .....	6
2.5. Tipos de redes según entradas y salidas .....	7
3. Variantes de la celda RNN .....	9
3.1. RNN básica .....	9
3.2. Long Short-Term Memory (LSTM) .....	9
3.3. Gated Recurrent Unit (GRU).....	11
3.4. Bidirectional RNN .....	13
4. Implementación con Keras/TensorFlow .....	<b>¡Error! Marcador no definido.</b>
4.1. Preprocesamiento de datos para RNN .....	<b>¡Error! Marcador no definido.</b>
5. Casos de Uso y Aplicaciones.....	14
5.1. Generación de texto con RNN .....	14
5.2. Predicción del precio de acciones con LSTM.....	14
5.3. Análisis de sentimientos con modelos recurrentes.....	15
5.4. Aplicaciones en chatbots y asistentes virtuales .....	15
6. Procesamiento del lenguaje natural.....	17
6.1. Problemas habituales en el procesamiento de textos .....	17
6.2. Representación numérica de la entrada .....	17
6.3. Procesamiento del lenguaje en Keras/TensorFlow .....	20
6.4. Otras mejoras aplicables .....	27
7. Optimización y Mejora del Rendimiento.....	28
8. Alternativas a las RNN y tendencias actuales.....	28
8.1. Redes neuronales convolucionales para secuencias.....	28
8.2. Modelos basados en Transformers .....	28
8.3. Comparación entre RNN, LSTM, GRU y Transformers.....	28
8.4. Ejemplo de GPT pequeño en Keras .....	29

# 1. Introducción a las Redes Neuronales Recurrentes

Las redes neuronales recurrentes (RNN, por sus siglas en inglés) son un tipo de red neuronal diseñada para trabajar con datos secuenciales. A diferencia de las redes neuronales **feedforward**, las RNN pueden mantener una memoria interna que les permite procesar secuencias de datos de manera contextual, lo que las hace especialmente útiles para tareas como el modelado de lenguaje, la predicción de series temporales y el reconocimiento de voz.

Mientras que en las redes neuronales **feedforward** la información fluye en una única dirección, desde la capa de entrada hasta la capa de salida, sin retroalimentación. En las redes neuronales recurrentes se permiten conexiones recurrentes que crean ciclos en la red, lo que les permite almacenar información de estados anteriores. Esta memoria interna las hace ideales para tareas secuenciales, ya que cada salida depende no solo de la entrada actual, sino también del contexto previo.

## 1.1. ¿Por qué necesitamos RNN?

Existen múltiples problemas en los que la secuencia de datos es crucial para obtener resultados precisos. Algunos ejemplos incluyen:

- **Análisis de texto y modelado de lenguaje:** en una oración, el significado de una palabra depende del contexto en el que se encuentra.
- **Reconocimiento de voz:** el significado de una palabra hablada depende del contexto fonético y sintáctico.
- **Predicción de series temporales:** para prever el valor de una acción bursátil o la demanda de un producto, es necesario analizar datos pasados.
- **Traducción automática:** un modelo necesita comprender la estructura de la oración en un idioma antes de traducirlo correctamente a otro.

Las redes neuronales tradicionales no pueden manejar bien este tipo de problemas porque no tienen memoria. En contraste, las RNN pueden recordar información previa y utilizarla en su proceso de toma de decisiones, lo que las convierte en una solución ideal para este tipo de aplicaciones.

## 1.2. Aplicaciones de las RNN

Las redes neuronales recurrentes se han aplicado con éxito en una variedad de campos. Algunas de sus aplicaciones más destacadas incluyen:

- A. Procesamiento del lenguaje natural
  - Generación de texto basado en un estilo particular (por ejemplo, escribir un poema o continuar una historia).

- Autocompletado y predicción de palabras en teclados inteligentes.
- B. Análisis y predicción de series temporales
- Pronóstico de ventas y demanda en empresas.
  - Predicción del clima y patrones meteorológicos.
  - Predicción de precios de activos financieros.
  - Aspectos médicos (ver apartado G)
- C. Reconocimiento de voz
- Conversión a texto en asistentes como Siri o Google Assistant.
  - Transcripción automática de audios en reuniones o conferencias.
- D. Generación de texto
- Creación de subtítulos automáticos en videos.
  - Generación automática de noticias o resúmenes de texto.
- E. Traducción automática: sistemas como Google Translate utilizan redes recurrentes para mejorar la precisión en traducciones automáticas.
- F. Análisis de sentimientos: clasificación de opiniones en redes sociales o reseñas de productos para determinar si son positivas o negativas.
- G. Aplicaciones específicas en medicina
- Análisis de electrocardiogramas (ECG) para detectar patrones anormales en los latidos del corazón.
  - Predicción de enfermedades basadas en datos históricos del paciente.

## 2. Arquitectura y Funcionamiento de las RNR

Las **Redes Neuronales Recurrentes (RNN, Recurrent Neural Networks)** son un tipo de red neuronal especializada en trabajar con **datos secuenciales**. A diferencia de las redes neuronales tradicionales (feedforward), las RNN tienen una estructura que permite que la información fluya de forma recurrente dentro de la red, lo que les permite recordar estados anteriores y usarlos para predecir la siguiente salida.

Este tipo de red es especialmente útil en tareas donde el orden de los datos es importante, como el **procesamiento de lenguaje natural (NLP)**, el **reconocimiento de voz** y la **predicción de series temporales**.

## 2.1. Estructura de una RNN básica

La estructura básica de una **Red Neuronal Recurrente (RNN)** es similar a la de una red neuronal tradicional, con la diferencia de que cada neurona no solo recibe la entrada actual, sino también un **estado previo**, que actúa como una memoria de lo que ha ocurrido antes.

**Componentes principales de una RNN:**

1. **Capa de entrada:** recibe los datos secuenciales (pueden ser palabras, valores numéricos, etc.).
2. **Capa oculta recurrente:** Contiene una o más neuronas con conexiones recurrentes que permiten almacenar información de pasos anteriores.
3. **Capa de salida:** Genera la predicción basada en el estado actual de la red.

**Representación matemática de una RNN:**

En una RNN simple, la activación de la neurona en el tiempo  $t$  se define como:

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

Donde:

- $x_t$  es la entrada en el tiempo  $t$ .
- $h_t$  es el estado oculto en el tiempo  $t$ .
- $W_x$  es la matriz de pesos que conecta la entrada con la capa oculta.
- $W_h$  es la matriz de pesos recurrente que conecta el estado oculto anterior  $h_{t-1}$  con el actual.
- $b$  es el sesgo (bias).
- $f$  es la función de activación, generalmente **tanh** o **ReLU**.

La salida  $y_t$  en cada instante de tiempo se obtiene aplicando una transformación al estado oculto actual:

$$y_t = g(W_y h_t + b_y)$$

Donde  $W_y$  es la matriz de pesos que conecta la capa oculta con la salida y  $g$  suele ser una función de activación como softmax (para clasificación) o una función lineal (para regresión).

## 2.2. Propagación hacia adelante en una RNN

El proceso de **propagación hacia adelante (forward pass)** en una RNN ocurre de la siguiente manera:

1. Se recibe la primera entrada  $x_1$  y se calcula el primer estado oculto  $h_1$ .
2. Se utiliza  $h_1$  junto con la siguiente entrada  $x_2$  para calcular  $h_2$ .
3. Este proceso se repite hasta llegar al último estado oculto  $h_T$ .
4. Cada estado oculto puede generar una salida intermedia  $y_t$ , o bien solo la última salida  $y_T$  se utiliza como resultado final.

### Ejemplo práctico:

En un sistema de predicción de texto, si la secuencia de entrada es:

"Hoy hace buen..."

La RNN procesará cada palabra en orden, recordando las anteriores, y generará una predicción para la siguiente palabra:

"Hoy hace buen **tiempo**."

## 2.3. Propagación hacia atrás a través del tiempo

El **entrenamiento** de una RNN sigue el mismo principio de una red neuronal estándar: se minimiza una función de pérdida ajustando los pesos mediante **descenso de gradiente**. Sin embargo, debido a la naturaleza secuencial de las RNN, se usa una versión especial del algoritmo de **backpropagation** llamada **Backpropagation Through Time (BPTT)**.

### Funcionamiento del BPTT:

1. Se calcula la **función de pérdida** después de recorrer toda la secuencia.
2. Se retropropagan los errores **hacia atrás en el tiempo**, desde el último estado oculto  $h_T$  hasta el primero  $h_1$ .
3. Se ajustan los pesos  $W_{hWx}$  y  $W_y$  usando el descenso de gradiente.

## 2.4. Problemas del entrenamiento de RNN

Los problemas habituales en el entrenamiento de redes recurrentes simples pueden ser 2:

### 1. Desvanecimiento del gradiente (Vanishing Gradient)

Cuando se usan funciones de activación como **tanh** o **sigmoid**, los gradientes pueden volverse muy pequeños con cada iteración. Esto hace que

los pesos apenas se actualicen, causando que la red **olvide información a largo plazo**. Para resolverlo se emplean 2 enfoques:

- Usar arquitecturas como **LSTM** o **GRU**, que mitigan este problema gracias a sus puertas de memoria.
- Cambiar la función de activación a **ReLU**, que es menos propensa a este problema.

## 2. Explosión del Gradiente (Exploding Gradient)

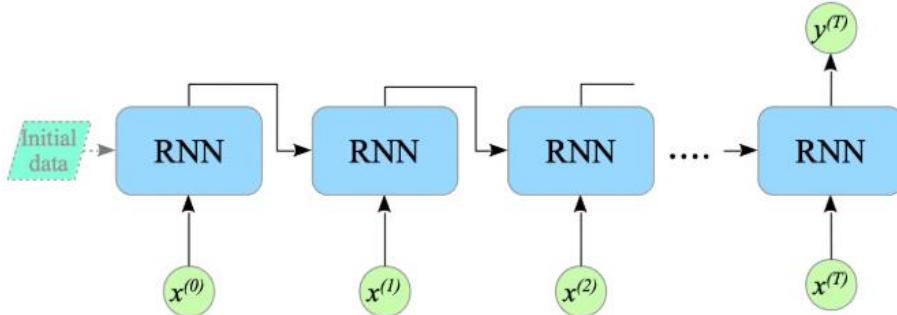
Ocurre cuando los gradientes crecen exponencialmente durante la retropropagación, causando inestabilidad en el entrenamiento. Las posibles soluciones son:

- Aplicar **clipping** al gradiente, limitando su magnitud.
- Usar **regularización** para evitar pesos demasiado grandes.

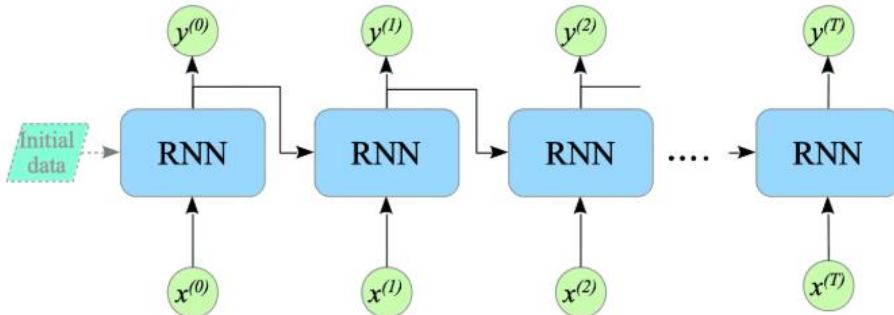
### 2.5. Tipos de redes según entradas y salidas

Las **Redes Neuronales Recurrentes (RNN)** pueden adoptar diferentes arquitecturas dependiendo del formato de sus entradas y salidas. En general, podemos clasificar estas arquitecturas en 3 categorías principales:

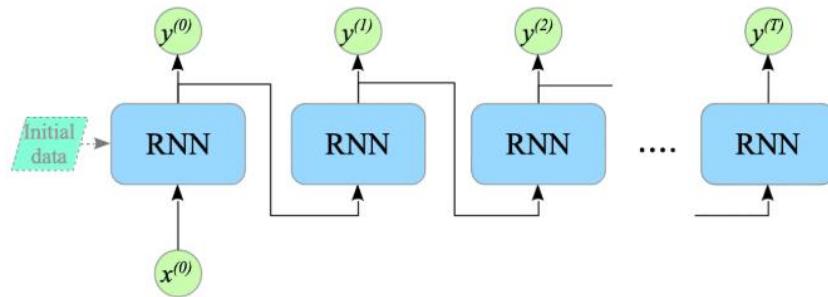
(1) Many-to-One



(2) Many-to-Many



### (3) One-to-Many



### (1) Many-to-One

Esta arquitectura toma una secuencia como entrada y genera otra secuencia como salida. Se usa cuando el número de elementos de entrada y salida pueden ser distintos. Ejemplos: traducción automática (ejemplo: inglés → español) y generación de subtítulos en tiempo real.

### (2) Many-to-Many

Este modelo toma una **secuencia completa** como entrada y genera una única salida. Es útil cuando la clasificación depende del contexto total de la secuencia. Ejemplos: análisis de sentimientos en textos y clasificación de spam en correos electrónicos.

### (3) One-to-Many

Este tipo de arquitectura toma una única entrada y genera múltiples salidas en una secuencia. Se utiliza en tareas donde una sola señal inicial puede dar lugar a una serie de eventos. Ejemplo: generación de texto o música a partir de una nota o palabra inicial.

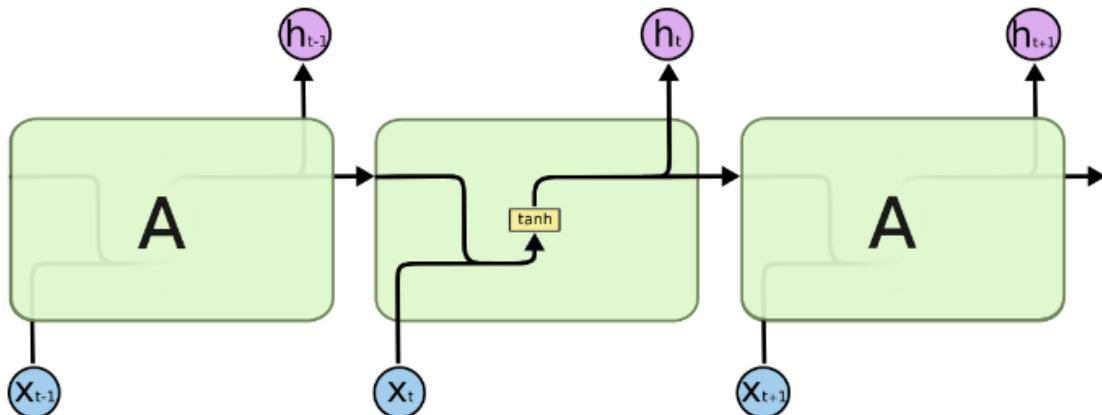
### 3. Variantes de la celda RNN

A pesar de que las Redes Neuronales Recurrentes básicas (RNN) son poderosas para modelar datos secuenciales, presentan limitaciones que han llevado al desarrollo de variantes más avanzadas. En este apartado analizaremos las principales modificaciones y arquitecturas derivadas de las RNN básicas.

#### 3.1. RNN básica

Todas las redes neuronales recurrentes tienen la forma de una cadena de módulos repetitivos. En las redes neuronales recurrentes estándar, este módulo repetitivo tendrá una estructura muy simple, con una sola capa tanh.

- En cada paso de tiempo, la celda recibe una entrada  $x$  (el dato actual) y el estado oculto previo  $h_{t-1}$  (la "memoria" del paso anterior).
- Estos se combinan mediante una transformación lineal con pesos  $W$  (para la entrada) y  $U$  (para el estado oculto), más un sesgo  $b$ .
- La salida se pasa por una función de activación no lineal, típicamente tanh, para producir el nuevo estado oculto  $h_t$ .
- Este  $h_t$  se usa como salida de la celda y se pasa al siguiente paso de tiempo.



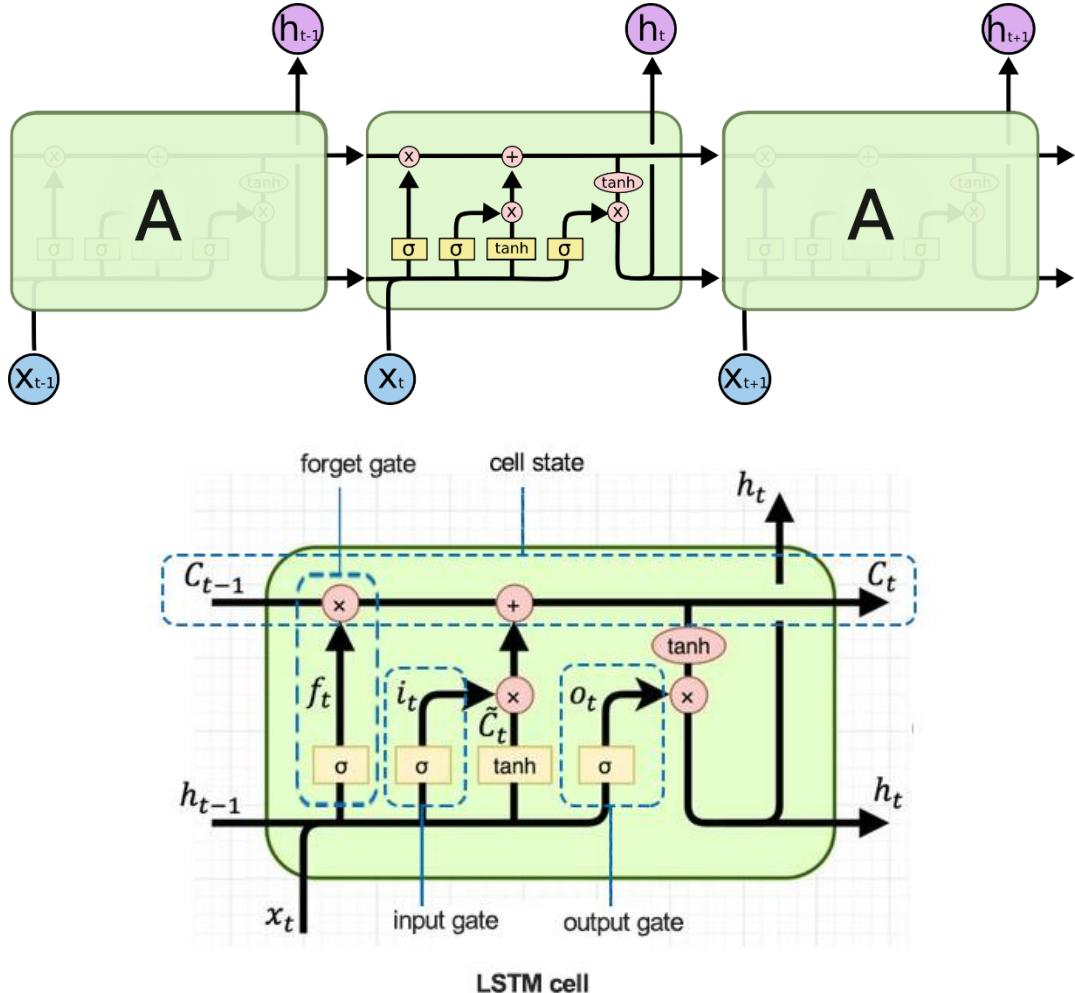
Aunque simple, la RNN estándar sufre del problema de desvanecimiento del gradiente, lo que dificulta aprender dependencias a largo plazo, ya que los gradientes se vuelven muy pequeños durante la retropropagación.

#### 3.2. Long Short-Term Memory (LSTM)

Las redes neuronales Long Short-Term Memory (LSTM) constituyen un tipo especializado de RNNs diseñadas para superar las limitaciones asociadas con la captura de dependencias temporales a largo plazo. A diferencia de las RNN tradicionales, las celdas LSTMs incorporan una arquitectura más compleja,

introduciendo unidades de memoria y mecanismos de puertas para mejorar la gestión de la información a lo largo del tiempo.

Los LSTM también tienen la estructura de cadena que hemos visto en la versión simple de RNN, pero el módulo repetitivo tiene una estructura diferente. En lugar de tener una sola capa de red neuronal, hay cuatro que interactúan de una manera muy especial.



$$i_t = \sigma(x_t U^i + h_{t-1} W^i)$$

$$f_t = \sigma(x_t U^f + h_{t-1} W^f)$$

$$o_t = \sigma(x_t U^o + h_{t-1} W^o)$$

$$\tilde{C}_t = \tanh(x_t U^g + h_{t-1} W^g)$$

$$C_t = \sigma(f_t * C_{t-1} + i_t * \tilde{C}_t)$$

$$h_t = \tanh(C_t) * o_t$$

## Estructura de las LSTMs

Las LSTMs presentan una estructura modular que consta de tres puertas (*gates*) fundamentales: la puerta de olvido (*forget gate*), la puerta de entrada (*input gate*), y la puerta de salida (*output gate*). Estas puertas trabajan en conjunto para regular el flujo de información a través de la unidad de memoria, permitiendo un control más preciso sobre qué información retener y cuál olvidar. Veamos sus componentes más relevantes:

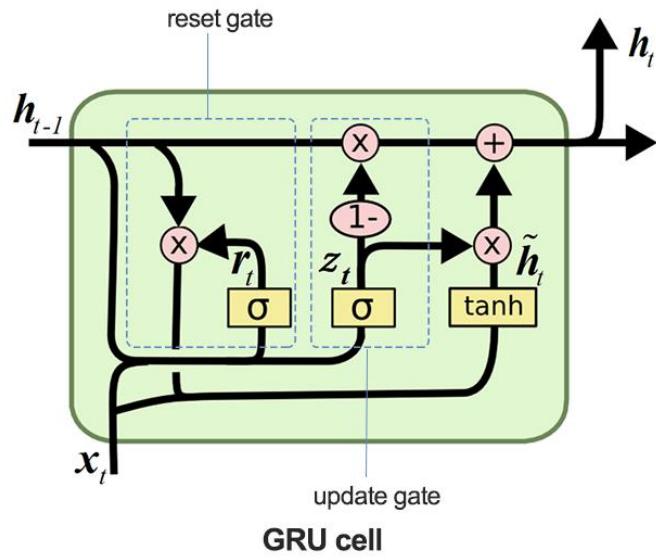
- **Puerta de olvido (*Forget Gate, ft*):** la puerta de olvido determina qué información del estado anterior de la celda debe transferirse. Emite un número entre 0 y 1 para cada número del estado de la celda  $C_{t-1}$ , donde 0 significa olvido completo y 1 significa retención completa.
- **Puerta de entrada (*Input Gate, it*):** esta puerta decide qué valores de la nueva entrada se utilizan para actualizar el estado de la celda. Regula el flujo de nueva información hacia la celda.
- **Puerta de salida (*Output Gate*):** esta puerta determina la salida final para el estado actual de la celda. Decide qué partes del estado de la celda deben salir en función de la entrada  $x_t$  y el estado oculto anterior  $h_{t-1}$ .
- **Memoria candidata ( $\tilde{C}_t$ ):** este componente genera los nuevos valores candidatos que se pueden agregar al estado de la celda. Utiliza la función de activación tanh por lo que los valores están entre -1 y 1.
- **Actualización del estado de la celda ( $C_t$ ):** el estado de la celda se actualiza combinando el estado de la celda anterior y los valores candidatos. La salida de la puerta de olvido controla la contribución del estado de la celda anterior, y la salida de la puerta de entrada controla la contribución de los nuevos valores candidatos.
- **Actualización del estado oculto ( $h_t$ ):** el estado oculto se actualiza según el estado de la celda y la decisión de la puerta de salida. Se utiliza como salida para el paso de tiempo actual y como entrada para el siguiente paso de tiempo.

### 3.3. Gated Recurrent Unit (GRU)

La GRU (Unidad Recurrente con Puertas) es una variante simplificada de la LSTM, con menos parámetros, pero aún efectiva para capturar dependencias temporales. GRU utiliza menos memoria y es más rápido que LSTM, sin embargo, LSTM es más preciso cuando se utilizan conjuntos de datos con secuencias más largas.

Veamos los componentes más destacados:

- **Puerta de actualización:** controla qué parte del estado oculto anterior  $h_{t-1}$  debe trasladarse al siguiente paso temporal. Decide efectivamente el equilibrio entre mantener la información antigua e incorporar información nueva.
- **Puerta de reinicio:** determina cuánto del estado oculto anterior debe olvidarse antes de calcular la nueva activación candidata. Permite que el modelo elimine información irrelevante del pasado.
- **Activación candidata:** genera nuevos valores potenciales para el estado oculto que pueden incorporarse en función de la decisión de la puerta de actualización.
- **Actualización del estado oculto ( $h_t$ ):** se actualiza combinando el estado oculto anterior y el estado oculto candidato. La puerta de actualización controla esta combinación, asegurando que se retiene la información relevante del pasado a la vez que se incorpora nueva información.



$$z_t = \sigma (W_z \cdot [h_{t-1}, x_t])$$

$$r_t = \sigma (W_r \cdot [h_{t-1}, x_t])$$

$$\tilde{h}_t = \tanh (W \cdot [r_t * h_{t-1}, x_t])$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

### 3.4. Bidirectional RNN

Las Redes Neuronales Recurrentes Bidireccionales (BiRNN, Bidirectional RNN) mejoran la capacidad de las RNN tradicionales al procesar la información en dos direcciones:

1. Hacia adelante (del pasado al futuro).
2. Hacia atrás (del futuro al pasado).

Esto permite que la red tenga contexto completo antes de generar una predicción, lo cual es especialmente útil en tareas como el procesamiento de lenguaje natural (NLP), donde una palabra puede depender de las anteriores y de las siguientes.

Una BiRNN está compuesta por dos capas recurrentes:

- Una capa recurrente **hacia adelante** con estados ocultos  $\vec{h}_t$
- Una capa recurrente **hacia atrás** con estados ocultos  $\overleftarrow{h}_t$

La salida final en cada paso de tiempo se obtiene combinando ambos estados ocultos:

$$h_t = [\vec{h}_t, \overleftarrow{h}_t]$$

#### Ventajas

- Captura dependencias a largo plazo en ambas direcciones.
- Mejora la precisión en tareas de procesamiento de lenguaje y reconocimiento de voz.

#### Desventajas

- Mayor costo computacional debido a la duplicación de parámetros.
- No es útil en problemas donde la predicción depende exclusivamente de datos pasados (ejemplo: predicción de series temporales).

## 5. Casos de Uso y Aplicaciones

Las Redes Neuronales Recurrentes (RNN), incluyendo LSTM y GRU, tienen múltiples aplicaciones en el mundo real. Su capacidad para modelar secuencias las hace ideales para tareas en las que el contexto previo es relevante. En este apartado, exploraremos algunos casos de uso destacados.

### 5.1. Generación de texto con RNN

La generación de texto con RNN se basa en entrenar un modelo para que aprenda patrones en secuencias de texto y sea capaz de predecir la siguiente palabra o carácter en función del contexto previo.

Este enfoque se usa en:

- **Generación de escritura creativa** (poesía, cuentos).
- **Asistentes de autocompletado** en editores de texto.
- **Traducción automática** y generación de resúmenes.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Embedding

# Definir el modelo de generación de texto
model = Sequential([
    Embedding(input_dim=5000, output_dim=128, input_length=50),
    LSTM(128, return_sequences=True),
    LSTM(128),
    Dense(5000, activation="softmax")
])

model.compile(loss="categorical_crossentropy", optimizer="adam")
model.summary()
```

### 5.2. Predicción del precio de acciones con LSTM

LSTM es ampliamente utilizada en **predicción de series temporales**, como precios de acciones o ventas. Su capacidad para capturar dependencias a largo plazo la hace ideal para analizar tendencias en datos financieros.

**Aplicaciones:**

- Predicción del **precio de acciones** en la bolsa.
- Estimación de **tendencias de mercado**.
- Predicción de **demandas de productos**.

```

import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

# Crear el modelo LSTM para series temporales
model = Sequential([
    LSTM(50, return_sequences=True, input_shape=(10, 1)),
    LSTM(50),
    Dense(1)
])
model.compile(optimizer="adam", loss="mse")
model.summary()

```

### 5.3. Análisis de sentimientos con modelos recurrentes

El análisis de sentimientos consiste en entrenar un modelo para clasificar textos en categorías como **positivo, negativo o neutral**. Las RNN (especialmente LSTM y GRU) son útiles porque pueden capturar la estructura y el contexto de las frases.

#### Aplicaciones:

- Análisis de **opiniones de clientes** en redes sociales.
- Clasificación de **reseñas** en plataformas como Amazon o TripAdvisor.
- Monitorización de **sentimientos en mercados financieros**.

```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Embedding, Dense

# Modelo de análisis de sentimientos con LSTM
model = Sequential([
    Embedding(input_dim=10000, output_dim=128, input_length=100),
    LSTM(64, return_sequences=True),
    LSTM(64),
    Dense(3, activation="softmax") # 3 clases: positivo, negativo, neutral
])
model.compile(optimizer="adam", loss="categorical_crossentropy",
metrics=["accuracy"])
model.summary()

```

### 5.4. Aplicaciones en chatbots y asistentes virtuales

Los chatbots y asistentes virtuales usan modelos recurrentes para **entender el contexto de una conversación** y responder de manera coherente.

#### Aplicaciones:

- **Atención al cliente** (Amazon Alexa, Google Assistant).
- **Automatización en empresas** (chatbots en páginas web).
- **Soporte técnico automatizado.**

## 6. Procesamiento del lenguaje natural

Una de las aplicaciones más destacadas de las redes neuronales recurrentes es el procesamiento del lenguaje natural. Dentro de este campo, encontramos diversos usos, como sistemas de traducción automática, algoritmos capaces de interpretar el significado general de un texto o incluso modelos avanzados que generan respuestas elaboradas, como ocurre con ChatGPT de OpenAI. En este apartado, exploraremos algunos conceptos fundamentales en los que se basa el procesamiento del lenguaje natural.

### 6.1. Problemas habituales en el procesamiento de textos

El análisis y comprensión del lenguaje escrito presentan múltiples dificultades, entre ellas:

- **Interpretación semántica:** ¿Cómo puede un ordenador comprender el significado de las palabras y trasladarlo a un formato procesable?
- **Orden y estructura de las frases:** La secuencia en la que aparecen las palabras es clave para interpretar correctamente el mensaje. Por ejemplo, las oraciones "Los leones comen gacelas" y "Las gacelas comen leones" contienen las mismas palabras, pero el significado cambia drásticamente según su disposición.
- **Representación numérica:** Las redes neuronales trabajan con datos numéricos, por lo que es fundamental convertir las palabras en valores que puedan ser procesados sin ambigüedades. Este punto será tratado en detalle más adelante.
- **Lenguaje informal y variaciones ortográficas:** El uso de abreviaturas, errores tipográficos o palabras fuera de los diccionarios tradicionales representa un reto adicional en el procesamiento del texto.
- **Segmentación gramatical:** Para una correcta interpretación, es importante identificar la estructura de la oración, separando sus elementos principales (sujeto, verbo, complemento, etc.).

### 6.2. Representación numérica de la entrada

Dado que las redes neuronales solo pueden procesar información en formato numérico, es imprescindible realizar una transformación de las palabras en representaciones numéricas cuantificables. Para ello, existen dos estrategias principales:

- **Codificación (encoding):** Consiste en asignar a cada palabra o letra un valor numérico único, siendo un método sencillo pero limitado.
- **Incrustación de palabras (word embeddings):** Se basa en representar cada palabra mediante un vector numérico, de modo que aquellas con

significados similares tengan representaciones cercanas entre sí en el espacio vectorial.

### 6.2.1. Codificación de palabras (*word encoding*)

Existen dos enfoques principales para la codificación de textos: por caracteres y por palabras.

**Codificación por caracteres:** este método representa cada letra de una frase en una matriz, donde las columnas corresponden a las letras del alfabeto y las filas a las posiciones en la secuencia de texto. En cada fila, se marca con un 1 la letra correspondiente a esa posición. Por ejemplo, para la frase "la caballa", la representación sería:

	a	b	c	d	e	f	g	h	i	j	k	l
l	0	0	0	0	0	0	0	0	0	0	0	1
a	1	0	0	0	0	0	0	0	0	0	0	0
_	0	0	0	0	0	0	0	0	0	0	0	0
c	0	0	1	0	0	0	0	0	0	0	0	0
a	1	0	0	0	0	0	0	0	0	0	0	0
b	0	1	0	0	0	0	0	0	0	0	0	0
a	1	0	0	0	0	0	0	0	0	0	0	0
l	0	0	0	0	0	0	0	0	0	0	0	1
l	0	0	0	0	0	0	0	0	0	0	0	1
a	1	0	0	0	0	0	0	0	0	0	0	0

Aunque este enfoque permite analizar texto letra por letra, tiene algunas desventajas importantes. Requiere una gran cantidad de memoria y no conserva la estructura ni el significado de las palabras, lo que dificulta el análisis semántico. En este caso, la red neuronal solo aprendería patrones en la secuencia de caracteres, sin captar el significado de las palabras o frases completas.

**Codificación por palabras:** en este método, se crea un diccionario de palabras conocidas y se asigna a cada una un identificador numérico único. Por ejemplo:

Palabra	Código
La	1
caballa	2
es	3
bueno	4
comer	5
...	...

Al procesar una frase, se reemplazan sus palabras por sus códigos numéricos. Por ejemplo, la oración "La caballa es buena para comer" podría codificarse como:

[1, 2, 3, 4, 0, 5]

Aquí, el número 0 indica que la palabra "para" no está en el diccionario y, por lo tanto, no se tiene en cuenta. Una limitación de este método es la necesidad de definir el tamaño del diccionario, ya que solo se pueden procesar las palabras incluidas en él.

### 6.2.2. Encaje de palabras (*word embedding*)

El word embedding es una técnica más avanzada que la simple codificación numérica. En lugar de asignar un código único a cada palabra, transforma cada palabra en un vector numérico. Este enfoque permite que palabras con significados similares tengan representaciones vectoriales cercanas en el espacio matemático, lo que ayuda a capturar relaciones semánticas y contextuales entre términos.

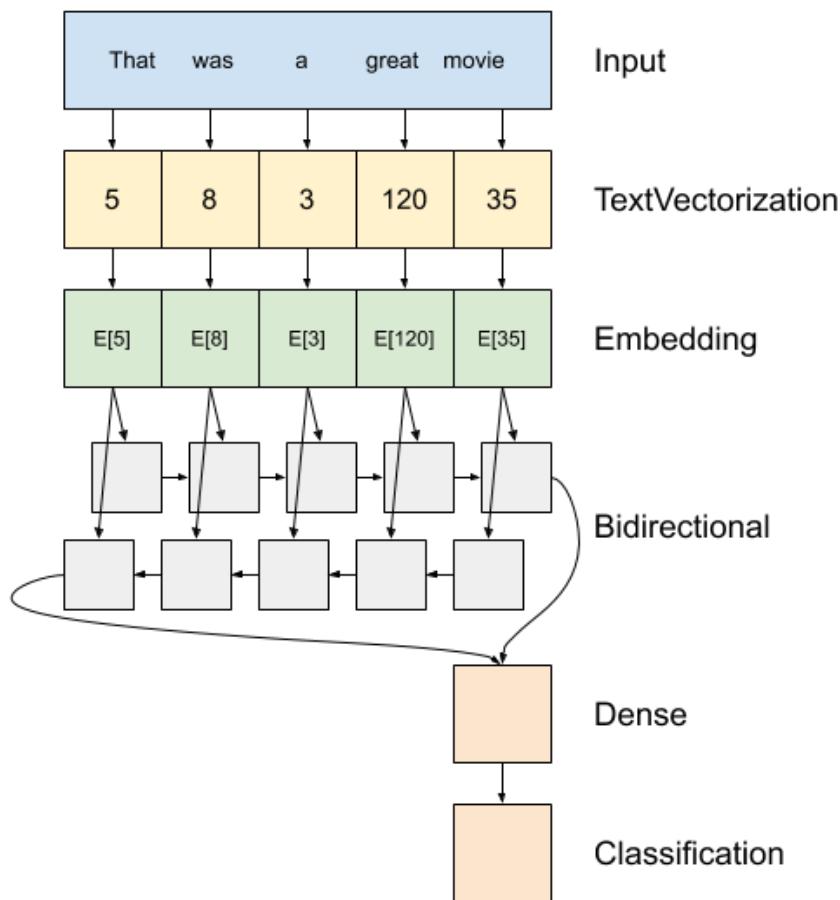
El proceso general es el siguiente:

1. **Codificación:** se asigna un identificador numérico a cada palabra del texto.
2. **Vectorización:** Cada código se transforma en un vector de características que representa el significado de la palabra en un espacio multidimensional.
3. **Procesamiento en la red neuronal:** Los vectores generados se utilizan como entrada en una red recurrente, que procesa la información para generar una salida significativa.

Este método mejora significativamente la comprensión del lenguaje por parte de los modelos de IA, ya que permite reconocer sinónimos, captar relaciones entre palabras y manejar mejor el contexto de una frase.

Al incorporar una etapa adicional después de la codificación de palabras para vectorizarlas, logramos eliminar posibles interpretaciones erróneas derivadas del orden numérico asignado.

Por ejemplo, si cada palabra recibe un código único, la red podría asumir que una palabra con un número más alto tiene mayor relevancia o algún tipo de jerarquía sobre otra con un número más bajo. Además, términos con significados similares, como "triste" y "deprimido", podrían recibir códigos muy diferentes, lo que dificultaría que el modelo establezca relaciones entre ellos.



La vectorización soluciona este problema al eliminar la relación numérica arbitraria y garantizar que palabras con significados cercanos tengan representaciones vectoriales similares. Como es habitual en deep learning, el objetivo es que la propia red neuronal aprenda a generar estas representaciones de manera óptima, identificando patrones y relaciones semánticas de forma autónoma.

### 6.3. Procesamiento del lenguaje en Keras/TensorFlow

Para utilizar redes neuronales recurrentes en tareas de procesamiento del lenguaje natural con Keras/TensorFlow, es fundamental realizar un preprocesamiento adecuado del texto de entrada. Existen diversas estrategias para ello, pero aquí seguiremos un enfoque estructurado en los siguientes pasos:

1. **Extraer todas las palabras del conjunto de entrenamiento** para construir un vocabulario base.
2. **Eliminar las palabras irrelevantes**, conocidas como *stop words*. Estas palabras, como "el", "de" o "y", no aportan significado relevante en el

análisis del texto y pueden descartarse. Existen listas de *stop words* específicas para cada idioma.

3. **Seleccionar las N palabras más representativas**, es decir, aquellas con mayor frecuencia en el conjunto de datos. A cada una se le asignará un código numérico único.

Para automatizar este proceso, podemos utilizar la clase CountVectorizer de *scikit-learn*. Esta herramienta permite especificar la lista de *stop words* y definir el número máximo de palabras que formarán parte del diccionario. Con su método fit\_transform(), podemos transformar un conjunto de textos en una representación numérica, almacenando el diccionario generado en la propiedad vocabulary\_.

```
from sklearn.feature_extraction.text import CountVectorizer

# Definimos un conjunto de textos
textos = [
    "Hola, este es un ejemplo de texto.",
    "Este texto muestra cómo funciona CountVectorizer.",
    "El procesamiento de texto es muy útil en NLP."
]

# Creamos el vectorizador con stop words en español y un máximo de 5 palabras
# en el vocabulario, podemos usar una lista manual para las stop words
vectorizer = CountVectorizer(stop_words="spanish", max_features=5)
X = vectorizer.fit_transform(textos)

# Mostramos el vocabulario generado
print("Vocabulario:", vectorizer.vocabulary_)

# Convertimos los textos en matrices de conteo
print("Matriz de conteo:\n", X.toarray())
```

Además de las palabras incluidas en el vocabulario, es fundamental reservar dos espacios especiales en la codificación numérica:

- **0** para representar palabras desconocidas, es decir, aquellas que no aparecen en el vocabulario generado.
- **1** para indicar palabras de relleno (*padding*), utilizadas para unificar la longitud de las secuencias de entrada.

De este modo, al construir el diccionario de palabras, podemos asignarles índices a partir del número 2, dejando los valores 0 y 1 para estos usos especiales. En Python, podríamos hacerlo de la siguiente manera:

```
# Generamos el diccionario ajustando los índices para reservar 0 y 1
dictionary = {word: i + 2 for i, word in
enumerate(vectorizer.get_feature_names_out())}
dictionary["DESC"] = 0 # Palabras desconocidas
dictionary ["PAD"] = 1 # Padding
```

## Uniendo todo en una función

Vamos a crear una función llamada `get_dictionary` que recibirá como parámetros:

- El array `text` de entradas de texto a procesar
- El array con las *stop words* que queremos tener en cuenta
- El tamaño `N` del vocabulario que queremos construir

La función devolverá el diccionario creado con los textos de entrada.

```
def get_dictionary(text, sw, N):  
    vectorizer = CountVectorizer(stop_words=sw, max_features=N)  
    vectorizer.fit_transform(text)  
    dictionary = vectorizer.vocabulary_  
    dictionary = {word: i + 2 for i, word in  
enumerate(vectorizer.get_feature_names_out())}  
    dictionary['DESC'] = 0  
    dictionary['PAD'] = 1  
  
    return dictionary
```

Esta otra función tomará un texto como entrada y, aplicando el diccionario y la lista de *stop words*, devolverá la secuencia de códigos que representa el texto, incluyendo los símbolos desconocidos y de relleno, para formar una frase de `T` palabras:

```
import re  
...  
  
def text_to_sequence(text, dictionary, stop_words, T):  
    palabras = re.findall(r'\b\w+\b', text.lower()) #Tokenizar  
    palabras = [palabra for palabra in palabras if palabra not in stop_words]  
    secuencia = [dictionary.get(palabra, dictionary['DESC']) for palabra in  
palabras]  
    secuencia = secuencia[:T]  
    secuencia += [dictionary['PAD']] * (T - len(secuencia))  
    return np.array(secuencia)
```

La función `re.findall` del módulo de expresiones regulares nativo de Python (`re`) nos permite separar las palabras de un texto, y obtener en un vector/lista todas las encontradas..

## Ejemplo de red recurrente con *word encoding*

Para poner en práctica todos estos conceptos, utilizaremos un dataset con reseñas de hoteles de la web *TripAdvisor*. En cada línea se tiene el texto completo de la reseña y la valoración numérica (entero de 1 a 5) que ha hecho el usuario en cuestión. Vamos a construir una red recurrente que, con una codificación previa de las palabras, aprenda a adivinar la puntuación que da un cliente a un hotel, dada su reseña.

Comenzaremos nuestro programa cargando las librerías necesarias, junto con las dos funciones que hemos definido previamente, y una lista de *stop words* en inglés:

```
import re
import numpy as np
import pandas as pd
from keras.models import Sequential
from keras.layers import Input, Dense, Dropout, LSTM, Bidirectional
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS # Stopwords
de sklearn

# Parámetros configurables
N = 20000 # Número de palabras del diccionario
T = 100 # Longitud prefijada de la reseña
EPOCHS = 10
NEURONAS = 128

def get_dictionary(text, sw, N):
    ...

def text_to_sequence(text, dictionary, stop_words, T):
    ...

# Usamos la lista de stopwords de sklearn
stop_words = list(ENGLISH_STOP_WORDS)
```

Ahora vamos a leer el CSV de entrada, procesar la columna con las reseñas y convertir cada reseña en una secuencia de  $T = 100$  códigos, según el diccionario que generaremos con las propias reseñas.

```
datos = pd.read_csv('tripadvisor_hotel_reviews.csv')
y = pd.get_dummies(datos['Rating'])
textos = datos['Review']

diccionario = get_dictionary(textos, stop_words, N)

X = np.array(textos.apply(lambda t: text_to_sequence(t, diccionario,
stop_words, T)))
```

Ya estamos en disposición de crear y entrenar la red. Nuestros datos de entrada serán los de la variable X, y la columna objetivo la tenemos almacenada en la variable y (columna *Rating* codificada en *one hot*).

```
modelo = Sequential()
modelo.add(Input((X.shape[1], 1)))
modelo.add(LSTM(units=NEURONAS, return_sequences=True))
modelo.add(Bidirectional(LSTM(units=NEURONAS)))
modelo.add(Dense(units=num_clases, activation='softmax'))

modelo.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
modelo.fit(X, y, validation_split=0.2, epochs=EPOCHS, batch_size=128)
```

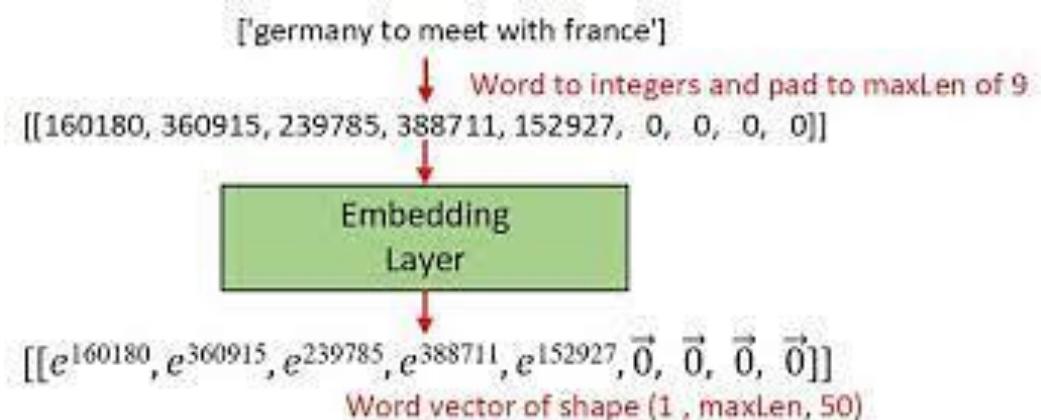
Si entrenamos el modelo, observaremos que la precisión alcanzada ronda el 50%. A primera vista, esto no parece tan malo, considerando que hay cinco categorías posibles y que un modelo aleatorio tendría una probabilidad de acierto del 20%. Sin embargo, el rendimiento de la red aún puede mejorarse significativamente. Uno de los principales inconvenientes proviene de la codificación basada en índices numéricos (word encoding). Esta técnica introduce ciertas limitaciones, como:

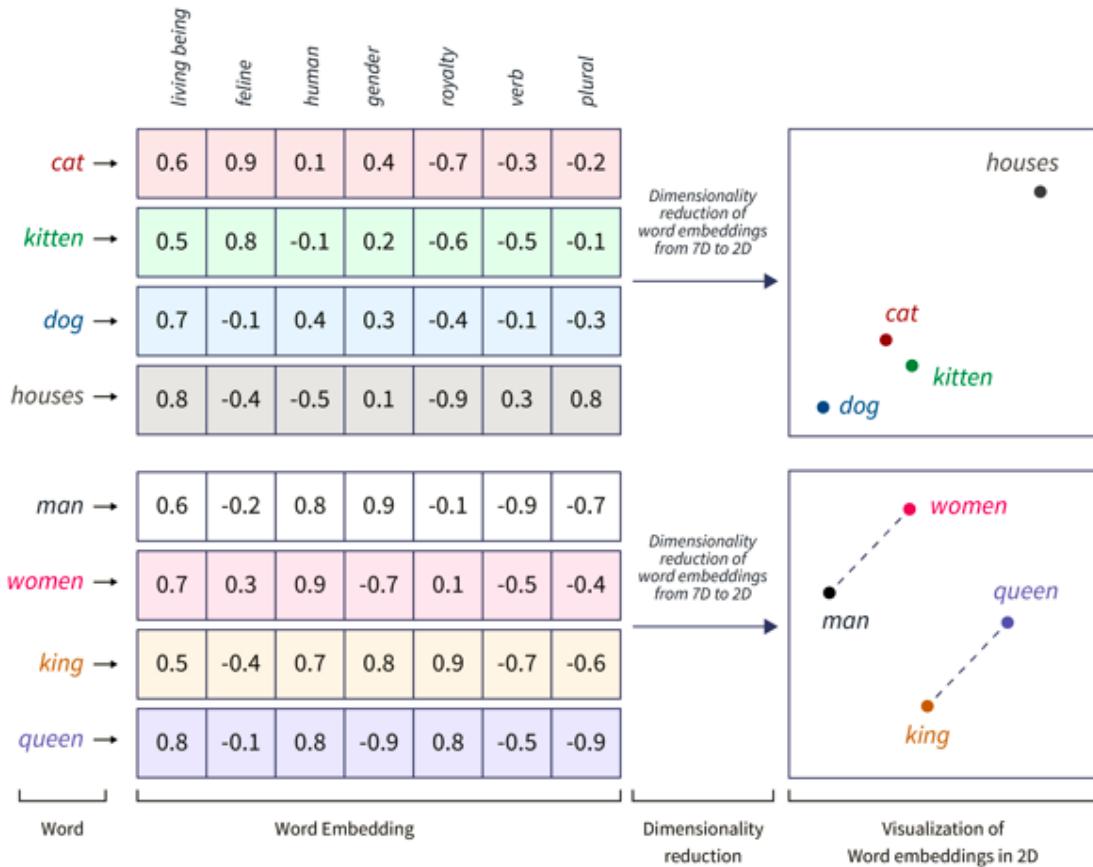
- La red puede interpretar que algunas palabras son "mejores" o "peores" que otras simplemente por el valor numérico asignado.
- Palabras con significados similares, como increíble y maravilloso, pueden recibir códigos muy distintos, lo que dificulta que la red establezca relaciones entre ellas.

Para superar estas limitaciones, utilizaremos el encaje de palabras (word embedding), una técnica que permite representar las palabras mediante vectores numéricos en un espacio semántico, logrando que términos con significados afines tengan representaciones similares. En la siguiente sección exploraremos esta estrategia en detalle.

### Ejemplo de red recurrente con *word embedding*

Para implementar la técnica de *word embedding* en Keras, disponemos de la clase Embedding, ubicada en el módulo keras.layers. Esta capa nos permite transformar secuencias de palabras codificadas en vectores de características con la dimensión que definamos.





## Funcionamiento de la capa Embedding

Esta capa actúa como una tabla de búsqueda, donde cada palabra del vocabulario tiene asociada una representación vectorial aprendida durante el entrenamiento del modelo. Recibe como entrada el código numérico de una palabra y devuelve su representación en forma de vector.

Para configurarla, utilizamos dos parámetros clave:

- **input\_dim:** define el tamaño del vocabulario de entrada, es decir, el número total de palabras consideradas en nuestro diccionario. En los ejemplos previos, este valor se ha representado como  $N$ .
- **output\_dim:** indica la cantidad de dimensiones que tendrá cada vector generado. Este valor es configurable y depende de la complejidad del problema que queremos resolver. Por ejemplo, si  $output\_dim = 100$ , entonces cada palabra se asignará a un vector con 100 elementos.

## Ejemplo práctico

Imaginemos que tenemos un vocabulario de  $N = 5$  palabras y queremos representarlas mediante vectores de tamaño  $D = 5$ . Vamos a construir un modelo

simple en Keras que incorpore una capa de *embedding* para realizar esta conversión.

```
import numpy as np
from keras.layers import Embedding

N = 5 # Tamaño del vocabulario
D = 5 # Dimensión del embedding

embedding_layer = Embedding(input_dim=N, output_dim=D)
entrada = np.array([[4, 3, 1, 1, 3]])
embeddings = embedding_layer(entrada)
print('Representación de {}'.format(str(entrada)))

print(embeddings.numpy())
```

La salida vectorizada que obtendremos para la entrada de las palabras [4, 3, 1, 1, 3] será algo así:

```
Representación de [[4 3 1 1 3]]
[[[-0.03721718  0.04400582 -0.02336024  0.00080264  0.00287908]
 [ 0.03698231  0.04999841  0.02043912 -0.0176939  0.02680326]
 [ 0.02456171  0.01848849 -0.02330163 -0.00816376  0.01585578]
 [ 0.02456171  0.01848849 -0.02330163 -0.00816376  0.01585578]
 [ 0.03698231  0.04999841  0.02043912 -0.0176939  0.02680326]]]
```

Así, la palabra codificada como 4 se representará con el vector [-0.03721718 0.04400582 -0.02336024 0.00080264 0.00287908], y así sucesivamente. Estos vectores se ajustan poco a poco, de forma convencional para el entrenamiento en redes neuronales.

Transformaremos ahora el ejemplo anterior de reseñas de TripAdvisor usando *word embedding*, para comprobar si esta técnica ofrece mejores resultados. En cuanto a cambios importantes, debemos importar la capa Embedding junto a las demás...

```
from keras.layers import Dense, Dropout, LSTM, Bidirectional, Embedding
```

... y declaramos la variable D con el número de dimensiones que queremos que tengan los vectores (podemos hacer varias pruebas con varios valores para dar con uno apropiado).

```
N = 20000
T = 100
D = 128          # Dimensiones de los vectores de embedding
EPOCHS = 10
NEURONAS = 128
```

El siguiente cambio viene en la definición del modelo, añadiendo una capa *Embedding* en la entrada:

```

modelo = Sequential()

# Añadimos 2 unidades más al tamaño para incluir los códigos de
# palabras desconocidas y de padding
modelo.add(Embedding(input_dim=N+2, output_dim=D))
modelo.add(Bidirectional(LSTM(units=NEURONAS, return_sequences=True)))
modelo.add(Bidirectional(LSTM(units=NEURONAS)))
modelo.add(Dense(units=num_clases, activation='softmax'))

modelo.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
modelo.fit(X, y, validation_split=0.2, epochs=EPOCHS, batch_size=128)

```

En este caso es fácil caer en el *overfitting*, con una precisión de más del 90% en el conjunto de entrenamiento y de menos del 60% en el de test. Ajustando el número de capas, el *dropout* y el número de neuronas por capa se puede reducir parcialmente ese *overfitting*, con una precisión cercana al 70%.

## Ejercicio

Transforma la red recurrente del ejercicio anterior en otra que utilice *word embedding* para la misma tarea

### 6.4. Otras mejoras aplicables

El método descrito anteriormente puede ser efectivo en múltiples escenarios de procesamiento del lenguaje natural. No obstante, presenta ciertas limitaciones que conviene abordar. Uno de los principales inconvenientes radica en la estrategia de padding utilizada: si las secuencias de entrada poseen longitudes muy dispares, el relleno uniforme con tokens de padding puede generar ineficiencias en el aprendizaje del modelo. Un exceso de estos valores artificiales podría afectar negativamente la capacidad de la red para extraer patrones significativos.

Para mitigar este problema, existen diversas estrategias que pueden optimizar el tratamiento de las secuencias:

- **Padding adaptativo por lotes:** En lugar de aplicar un padding uniforme a todo el conjunto de datos, se agrupan las secuencias en lotes de tamaño **B**. Para cada lote, se determina la longitud de la secuencia más larga dentro de ese grupo y se aplica padding únicamente hasta ese tamaño. De este modo, se minimiza la cantidad de valores de relleno innecesarios, permitiendo que la red neuronal trabaje con datos más representativos y eficientes.
- **Bucketting o agrupación por longitud:** Esta técnica consiste en ordenar previamente las secuencias según su extensión antes de dividirlas en

lotes. Así, cada lote agrupa textos de longitud similar, reduciendo aún más la necesidad de padding y mejorando la eficiencia del procesamiento.

Combinando ambas estrategias, se consigue que los modelos puedan manejar secuencias de longitud variable sin afectar su rendimiento. Además, estas técnicas permiten que la red neuronal se entrene de manera más eficiente, aprovechando mejor los recursos computacionales y evitando la sobrecarga de información irrelevante.

## 7. Optimización y Mejora del Rendimiento

## 8. Alternativas a las RNN y tendencias actuales

### 8.1. Redes neuronales convolucionales para secuencias

### 8.2. Modelos basados en Transformers

Los **Transformers** han revolucionado el procesamiento de secuencias, desplazando a las RNN y sus variantes (LSTM, GRU) en múltiples tareas. Su arquitectura se basa en el mecanismo de **atención**, permitiendo capturar relaciones entre palabras o elementos sin necesidad de recurrencias.

El artículo "Attention is All You Need" (Vaswani et al., 2017) introdujo el modelo Transformer, marcando un cambio en el aprendizaje de secuencias.

A diferencia de las RNN, que procesan datos en secuencia, los **Transformers** procesan **toda la entrada simultáneamente**. La clave de su éxito radica en el **mecanismo de autoatención (self-attention)**, que permite:

- Considerar **todo el contexto** de una secuencia a la vez.
- Evitar problemas como el **desvanecimiento del gradiente**.
- Permitir una **paralelización total**, acelerando el entrenamiento en GPUs y TPUs.

### 8.3. Comparación entre RNN, LSTM, GRU y Transformers

Característica	RNN	LSTM	GRU	Transformer
----------------	-----	------	-----	-------------

<b>Gestión de dependencias largas</b>	Difícil	Mejor	Mejor	Excelente
<b>Paralelización</b>	No	No	No	Sí
<b>Consumo de memoria</b>	Bajo	Alto	Medio	Alto
<b>Tiempo de entrenamiento</b>	Rápido	Lento	Medio	Medio
<b>Uso en NLP</b>	No recomendado	Bueno	Bueno	Excelente

### Conclusiones:

- Las RNN son útiles para problemas pequeños, pero no escalan bien.
- LSTM y GRU mejoran la retención de información, pero siguen teniendo problemas de paralelización.
- Los Transformers han reemplazado a las RNN en casi todas las tareas de modelado de secuencias.

El futuro del procesamiento de secuencias está dominado por los Transformers. Modelos como BERT, GPT y T5 han cambiado por completo la forma en que abordamos el NLP y la predicción de datos secuenciales.

### 8.4. Ejemplo de GPT pequeño en Keras

Veamos un ejemplo completo de un modelo Transformer, en Keras, inspirado en GPT pero mucho más simple. Se trata de un modelo que puede generar texto basado en una entrada dada.

```
import tensorflow as tf
from tensorflow.keras.layers import Layer, Embedding, Dense,
MultiHeadAttention, LayerNormalization, Dropout
from tensorflow.keras.models import Model
import numpy as np

# ===== 1. BLOQUE TRANSFORMER =====
class TransformerBlock(Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super(TransformerBlock, self).__init__()
        self.att = MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
        self.ffn = tf.keras.Sequential([
            Dense(ff_dim, activation="relu"),
            Dense(embed_dim)
        ])
        self.layernorm1 = LayerNormalization(epsilon=1e-6)
        self.layernorm2 = LayerNormalization(epsilon=1e-6)
        self.dropout1 = Dropout(rate)
        self.dropout2 = Dropout(rate)

    def call(self, inputs, training):
        attn_output = self.att(inputs, inputs) # Autoatención
        attn_output = self.dropout1(attn_output, training)
        ffn_output = self.ffn(attn_output)
        ffn_output = self.dropout2(ffn_output, training)
        output = self.layernorm2(inputs + ffn_output)
        return output
```

```

        out1 = self.layernorm1(inputs + attn_output)
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        return self.layernorm2(out1 + ffn_output)

# ===== 2. MODELO TRANSFORMER =====
class MiniGPT(Model):
    def __init__(self, vocab_size, embed_dim, num_heads, ff_dim, num_blocks,
max_len):
        super(MiniGPT, self).__init__()
        self.embed_dim = embed_dim
        self.embedding = Embedding(vocab_size, embed_dim)
        self.pos_embedding = Embedding(max_len, embed_dim)
        self.transformer_blocks = [TransformerBlock(embed_dim, num_heads,
ff_dim) for _ in range(num_blocks)]
        self.norm = LayerNormalization(epsilon=1e-6)
        self.out_layer = Dense(vocab_size, activation="softmax")

    def call(self, inputs, training):
        positions = tf.range(start=0, limit=tf.shape(inputs)[-1], delta=1)
        embedded_inputs = self.embedding(inputs) +
        self.pos_embedding(positions)

        x = embedded_inputs
        for transformer_block in self.transformer_blocks:
            x = transformer_block(x, training=training)

        x = self.norm(x)
        return self.out_layer(x)

# ===== 3. PREPARAR EL MODELO =====
vocab_size = 10000 # Número de palabras en el vocabulario
embed_dim = 128 # Dimensión de los embeddings
num_heads = 8 # Número de cabezas de atención
ff_dim = 512 # Dimensión de la red feedforward
num_blocks = 4 # Número de bloques Transformer
max_len = 50 # Longitud máxima de la secuencia

# Crear el modelo
transformer = MiniGPT(vocab_size, embed_dim, num_heads, ff_dim, num_blocks,
max_len)
transformer.compile(optimizer="adam", loss="sparse_categorical_crossentropy")
transformer.summary()

# Simulación de datos aleatorios (¡Usar datos reales en la práctica!)
X_train = np.random.randint(0, vocab_size, (1000, max_len)) # 1000 ejemplos
y_train = np.random.randint(0, vocab_size, (1000, max_len))

# Entrenar el modelo
transformer.fit(X_train, y_train, batch_size=32, epochs=10)

## CÓDIGO DEL PROGRAMA DE GENERACIÓN DE TEXTO BASADO EN EL TRANSFORMER

import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# ===== 1. TOKENIZADOR =====

```

```

# Simulación de un vocabulario pequeño (en práctica usar más datos)
vocab_size = 10000
tokenizer = Tokenizer(num_words=vocab_size, oov_token("<OOV>"))
tokenizer.fit_on_texts(["Hola, ¿cómo estás?", "El modelo Transformer genera texto.", "Aprender inteligencia artificial es emocionante."])

# ===== 2. FUNCIÓN PARA GENERAR TEXTO =====
def generar_texto(model, seed_text, max_len=50, num_words=10):

    for _ in range(num_words):
        # Convertir el texto en tokens
        token_list = tokenizer.texts_to_sequences([seed_text])[0]
        token_list = pad_sequences([token_list], maxlen=max_len,
padding='pre')

        # Predecir siguiente palabra
        predicted_probs = model.predict(token_list, verbose=0)[0][-1] # Última palabra
        predicted_index = np.argmax(predicted_probs) # Elegir la palabra con mayor probabilidad

        # Convertir índice en palabra
        for word, index in tokenizer.word_index.items():
            if index == predicted_index:
                seed_text += " " + word
                break

    return seed_text

# ===== 3. PRUEBA DE GENERACIÓN =====
texto_generado = generar_texto(transformer, "Hola, ¿cómo", num_words=10)
print("Texto generado:", texto_generado)

```