Apuntes sobre IA (borrador)

Sergio de Mingo

2 de febrero de 2026

Contents

1 Redes Recurrentes (RNN) y LSTM

Una red neuronal recurrente (RNN) es un tipo de red que contiene conexiones recurrentes, es decir, conexiones en las que la salida de una neurona en un instante se utiliza como entrada para otra neurona en el instante siguiente. Esto permite a las RNNs capturar dependencias temporales y patrones secuenciales. A diferencia de las redes feed-forward tradicionales, donde las operaciones se reducen principalmente a multiplicaciones de matrices y transformaciones mediante funciones de activación, las RNN procesan datos secuenciales manteniendo una memoria oculta (hidden state) que se actualiza en cada paso temporal. Vamos a comenzar a desarrollar el concepto con un problema clásico: la predicción de caracteres. Imagina que le enseñamos a la red la palabra «HOLA». Si le damos la H, debe predecir la O y si le damos la O, debe predecir la L y así sucesivamente. Pero para predecir la A, la red tiene que «recordar» que antes vinieron la H, la O y la L. Si solo viera la L, no sabría si la palabra es «HOLA», «PELO» o «ALTO».

En una CNN, los datos fluyen de entrada a salida y desaparecen. En una RNN, la red tiene un bucle interno. Cada vez que la red procesa una letra, genera dos cosas:

- Una salida: (La predicción de la siguiente letra).
- Un estado oculto (h_t): Una especie de memoria a corto plazo que se vuelve a meter en la red junto con la siguiente letra.

Las RNN básicas tienen un defecto de diseño: son muy buenas recordando lo que pasó hace un segundo, pero pésimas recordando lo que pasó hace diez. Es el problema del desvanecimiento del Gradiente o *vanishing gradient* del que ya hemos hablado anteriormente en la sección **??** pero llevado al extremo. En una CNN o una red multicapa, tienes, por ejemplo, 5 o 10 capas físicas. Pero en una RNN, si estás procesando una frase de 50 palabras, es como si tuvieras una red de 50 capas de profundidad y además hay que tener en cuenta un agravante: la matriz de pesos es la misma en cada paso. En una red normal, cada capa tiene sus propios pesos (W_1, W_2, W_3, ...). En una RNN, aplicas el mismo peso W una y otra vez para cada palabra. Si la frase es muy larga, la información de la primera palabra se diluye tanto que no llega al final. Por eso, después de entender la RNN básica, pasaremos a las LSTM (Long Short-Term Memory), que tienen una estructura especial para decidir qué recuerdos guardar y cuáles borrar.

Para este ejemplo usaremos nombres de ciudades que vamos a intentar predecir. Vamos a construir para ello un diccionario de caracteres. Esto es esencial porque la red no entiende de letras, solo de posiciones en un vector. Este un proceso llamado mapeado o *mapping*.

```
import torch
import numpy as np
```

```
ciudades = ["MADRID", "BARCELONA", "VALENCIA", "SEVILLA", "ZARAGOZA",
            "MALAGA", "MURCIA", "PALMA", "BILBAO", "ALICANTE",
            "CORDOBA", "VALLADOLID", "VIGO", "GIJON", "GRANADA"]

# 1. Crear el vocabulario
alfabeto = sorted(list(set("".join(ciudades) + ".")))
char_to_int = {char: i for i, char in enumerate(alfabeto)}
int_to_char = {i: char for i, char in enumerate(alfabeto)}

n_letras = len(alfabeto)
print(f"Vocabulario: {alfabeto}")
```

Creamos la variable `alfabeto` con todas las letras involucradas en las palabras que queremos predecir (además del punto) y, a partir de ellos, creamos los diccionarios `char_to_int` y `int_to_char`.

Para entrenar la red, necesitamos pares de Entrada (X) y Objetivo (Y). Si la palabra es MADRID.: Cuando ve M, el objetivo es A. Cuando ve A, el objetivo es D. Y así seguiremos sucesivamente hasta que vea D, el objetivo es . (fin). Ahora necesitamos convertir cada palabra a un tensor. Para ello usaremos la siguientes funciones:

```
def palabra_a_tensor(palabra):
    tensor_x = torch.zeros(len(palabra), 1, n_letras)
    for i, char in enumerate(palabra):
        tensor_x[i][0][char_to_int[char]] = 1
    return tensor_x


def palabra_a_objetivo(palabra):
    indices = [char_to_int[char] for char in palabra[1:]]
    indices.append(char_to_int['.'])
    return torch.LongTensor(indices)
```