Apuntes sobre IA

Sergio de Mingo

2 de febrero de 2026

Contents

1 Reconocimiento de patrones

Hasta ahora, tu red solo ha tenido que distinguir entre 4 combinaciones. Pero, ¿qué pasa cuando la entrada no son dos bits (0 o 1), sino una imagen de 28x28 píxeles de un número de 0 a 9 escrito a mano? El dataset MNIST es el "Hola Mundo" del Deep Learning. Son imágenes de dígitos del 0 al 9. MNIST Es una extensa colección de base de datos que se utiliza ampliamente para el entrenamiento de diversos sistemas de procesamiento de imágenes. La base de datos MNIST consta de 60.000 imágenes de entrenamiento y 10.000 imágenes de prueba. Ahora nuestra entrada X ya no es una matriz de 4×2. Ahora cada imagen se aplana en un vector de 784 números (28 píxeles x 28 píxeles). Si procesamos un *batch* o lote de 64 imágenes, X será [64×784]. La salida Y ya no es un solo valor. Ahora necesitamos saber la probabilidad de que sea un 0, un 1, un 2... hasta el 9. Por tanto, la salida tiene 10 neuronas.

Para procesar las imágenes comenzaremos aplicando un proceso de aplanamiento o *flattering*. Una imagen del dataset MNIST es una cuadrícula de 28×28 píxeles. Cada píxel tiene un valor (normalmente de 0 a 255, que luego normalizamos a [0,1]) que indica qué tan oscuro es ese punto. Como nuestras capas ocultas esperan un vector (una fila de entradas), no podemos meterle un "cuadrado". Tenemos que desenrollar o aplanar la imagen. El proceso es justo un aplanamiento pues tomamos la primera fila de 28 píxeles, luego pegamos la segunda fila a continuación, luego la tercera, y así sucesivamente hasta la fila 28. Pasamos de un tensor de [28,28] a un vector de [784]. Si tenemos un *batch* o lote de, por ejemplo, 100 imágenes, nuestra matriz de entrada X para la red tendrá unas dimensiones de: X=[100×784]. Esto significa que cada fila de tu matriz es una imagen completa pero estirada.

En el XOR usamos la Sigmoide, pero para redes más profundas existe un problema importante con esta función: El gradiente desvanesciente o *Vanishing Gradient*, del que hablaremos a continuación. Otro problema ahora es que en la salida necesitamos varias neuronas, en concreto 10. Así cada una nos devolverá una probabilidad de que el patrón de la entrada es el número asociado a ellas. En el XOR, la salida era un valor entre 0 y 1. En clasificación de números, queremos que la red nos diga: «Estoy un 80% seguro de que es un '5', un 15% de que es un '3' y un 5% de que es un '6'». Para esto usamos la función Softmax en la última capa. Lo que hace es agarrar los valores brutos de salida y convertirlos en una distribución de probabilidad que suma exactamente 100%.

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum e^{x_j}}$$

Otro cambio importante es el cálculo del error. Ahora usaremos la función de la **Entropía cruzada** o *Cross Entropy* de la que hablaremos más adelante. La principal ventaja de esta función es que aumenta la rapidez del aprendizaje de la red. En resumen, esta nueva red tendrá la siguiente estructura:

- **Capa de Entrada** (X): Ya no son 2 bits. Aplanamos la imagen en un vector de 784 neuronas. Pues cada imagen se aplana en un vector de $28x28 = 784$.
- **Capa Oculta**: Digamos que elegimos 128 neuronas para que la red tenga "memoria" suficiente para las formas de los números.
- **Capa de Salida**: Ahora necesitamos 10 neuronas (una para cada dígito del 0 al 9).

El desarrollo matricial ahora nos quedaría de la siguiente forma:

$$Z_1 = X_{batch \times 784} \cdot W_{1(784 \times 128)} + b_{1(128)}$$

$$A_1 = \text{ReLU}(Z_1)$$

$$Z_2 = A_{1(batch \times 128)} \cdot W_{2(128 \times 10)} + b_{2(10)}$$

$$A_2 = \text{Softmax}(Z_2)$$

El primer cambio se aprecia en W_1 donde vemos que tiene 100,352 pesos (784×128). Esto es solo en la primera capa. El aumento de la complejidad es apreciable pues pasamos de tener que reajustar apenas 4 pesos en esta primera capa en el ejemplo anterior a más de 100 mil en este modelo. Vamos ahora a resumir el bucle de entrenamiento igual que hicimos en la red multicapa clásica del ejemplo para el XOR:

```
for epoch in range(epochs):
    # 1. Forward Pass con Softmax
    z1 = np.dot(X_train, W1) + b1
    a1 = relu(z1)  # Cambiamos Sigmoide por ReLU

    z2 = np.dot(a1, W2) + b2
    predicted_output = softmax(z2)

    # 2. El gradiente con la Cross-Entropy
    d_output = predicted_output - y_train

    # 3. Backpropagation
    error_hidden = d_output.dot(W2.T)
    d_hidden = error_hidden * relu_derivative(z1)

    # 4. Actualizacion
    W2 -= learning_rate * a1.T.dot(d_output)
    W1 -= learning_rate * X_train.T.dot(d_hidden)
```

En el XOR, tu salida era un punto. Aquí, tu salida es una distribución. Si la red ve un "3" escrito de forma extraña, la neurona del "3" se encenderá mucho (digamos 0.7), pero la del "8" también podría encenderse un poco (0.2) porque se parecen. El Softmax es el que gestiona esa competencia entre neuronas. El valor que devuelve en `predicted_output` será siempre un vector que sume 1 (por ejemplo `[0.1, 0.0, 0.8, ...]`). El gradiente de la Entropia cruzada es sorprendentemente simple: # (Predicción - Realidad). Si y_train es [0, 0, 1, 0...] (es un '2') # y predicted es [0.1, 0.1, 0.7, 0.1...], el error es la diferencia. La fase de propagación del error hacia atrás es similar aunque ahora usaremos la derivada de ReLU.

1.1 ReLU y el gradiente desvanesciente

El paso de Sigmoide a ReLU (Rectified Linear Unit) es, probablemente, el avance más sencillo pero más importante que permitió que las redes neuronales pasaran de tener 2 capas a tener cientos. Ya se ha explicado anteriormente el problema del gradiente desvanesciente. Si observamos la curva de la sigmoide, vemos que sus valores de entrada están comprimidos entre el 0 y el 1.

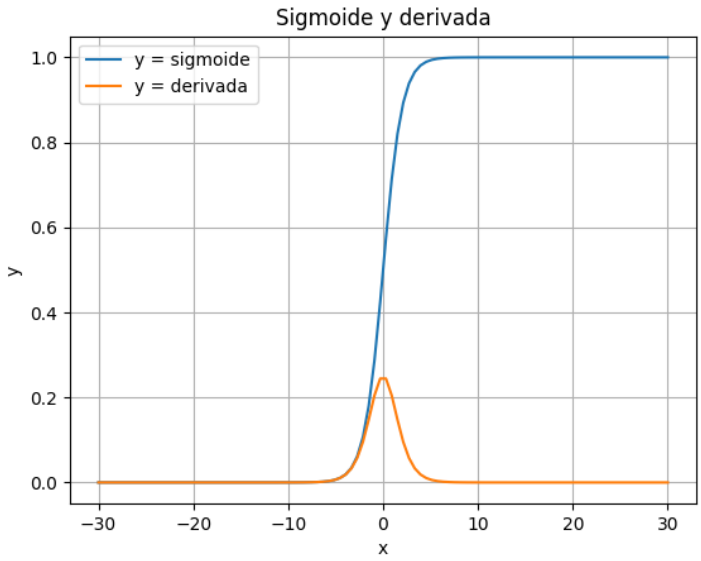

Figure 1: Curva de la sigmoide y su derivada

El problema se encuentra en su derivada cuyo valor máximo es 0.25. Eso es cuando la entrada es 0. Si no lo es, este valor se va rápidamente a 0. En una red con múltiples capas como puede ser la necesaria para MNIST necesitamos aplicar la regla de la cadena con todas ellas y acabamos multiplicando valores muy pequeños (0.25 en el mayor de los casos) o incluso 0. El resultado es que las capas profundas y cercanas a la salida si «aprenden» algo y actualizan sus pesos pero a medida que nos acercamos a las primeras capas, a estas les llega el delta casi reducido a 0 y por lo tanto no ajustarán sus pesos nada, ni se producirá aprendizaje alguno. Para solucionar esto usaremos ReLU, cuya función está formulada anteriormente. ReLU es una función "a trozos" extremadamente simple, y ahí reside su genio:

$$f(x) = \begin{cases} 0 & \text{si } x \leq 0 \\ x & \text{si } x > 0 \end{cases}$$

Su derivada es la clave pues es similar. Para el primer caso es 1 y para el segundo caso es 0. Al multiplicar gradientes en la Regla de la Cadena, multiplicar por 1 no reduce la señal. El error viaja íntegro desde la salida hasta la entrada, sin importar cuántas capas haya por medio. Esto permite que las redes sean mucho más profundas. Además se produce una mejora computacional importante debido a que es una función mucho más sencilla de computar (un simple `if`). ReLU tiene un pequeño riesgo. Si una neurona recibe un golpe de gradiente muy fuerte y sus pesos se vuelven tan negativos que su entrada siempre es < 0, su salida será siempre 0 y su derivada siempre 0. Esa neurona «muere» y deja de aprender para siempre. De ahí que también usemos variantes como Leaky ReLU que deja pasar un poquito de información negativa (0.01x) para que la neurona tenga una oportunidad de «resucitar».

1.2 Propagación del error con entropía cruzada

En la red multicapa usamos para calcular el error que propagábamos la función del Error Cuadrático Medio o MSE. Realmente usábamos su derivada justo en la línea `error = y - predicted_output`. Si vemos la función del error cuadrático medio o E entendemos que su derivada sea la indicada en el código. El $\frac{1}{2}$ se añade por pura conveniencia matemática para que se cancele al derivar:

$$E = \frac{1}{2}(y - \hat{y})^2 \qquad \frac{\partial E}{\partial \hat{y}} = -(y - \hat{y})$$

En esta nueva red para reconocer patrones usaremos una nueva función para calcular el error llamada **Entropía cruzada** o *Cross Entropy*. Su derivada es mucho más agresiva. Si la red está muy segura de que un número es un "3" pero en realidad es un "8", la Cross-Entropy genera un gradiente gigantesco para obligar a la red a cambiar rápido.

Una vez que la red ha procesado ese vector de 784 píxeles a través de las capas y llegamos a la salida, tenemos 10 neuronas (una por cada dígito). Gracias a la función Softmax, estas 10 neuronas nos dan probabilidades ya que convierte números brutos en una «repartición de apuestas» o distribución de probabilidades. Supongamos que le pasamos la imagen de un "3". En ese caso tendremos un vector de salida esperada y que compararemos con el vector de salida predicha u obtenida $\hat{y}$. Para interpretar bien ambos vectores hay que entender que el primer número es la salida de la primera neurona (la asociada al "0"), el segundo el de la segunda neurona o la asociada al "1" y así para todo el vector.

$$y = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]$$

$$\hat{y} = [0.01, 0.02, 0.05, \underline{0.70}, 0.02, 0.10, 0.0, 0.0, 0.10, 0.0]$$

El error (L) se calcula solo mirando la neurona que debería haber acertado. Esto es la cuarta o la correspondiente al número tres con un resultado de 0.70. Como en el vector y casi todos son cero, la fórmula se reduce a:

$$L = -\sum_{i=0}^{9} y_i \cdot \ln(\hat{y}_i) = -\ln(\hat{y}_4)$$

Aunque la fórmula del Softmax es compleja y la de la Entropía Cruzada tiene logaritmos, cuando calculas la derivada para el *Backpropagation* para saber cuánto error enviar atrás, ocurre una simplificación mágica:

$$\frac{\partial L}{\partial Z_2} = \hat{y} - y$$

Es exactamente la misma resta simple usábamos en el XOR anteriormente. Si la red dijo 0.70 para el "3" y la realidad era 1.0, el error es -0.30. Si la red dijo 0.10 para el "5" y la realidad era 0.0, el error es 0.10. Esa diferencia es la que fluye hacia atrás para ajustar los 100,000 pesos de tu red. Vamos ahora a explicar la mejora que obtenemos usando esta función frente al MSE anterior. Cuando usas MSE con una Sigmoide, la red sufre un fenómeno llamado Saturación. Si la red está muy equivocada (por ejemplo, la salida real es 1 pero la red predice 0.001), el valor de la predicción está en la zona plana de la sigmoide. Al calcular el error para el backpropagation, multiplicas por la derivada de la sigmoide:

$$\text{Gradiente} = (y - \hat{y}) \cdot \underbrace{\sigma'(z)}_{\text{¡Casi cero!}}$$

Aunque el error $(y - \hat{y})$ es muy grande (casi 1), al multiplicarlo por una derivada que es casi 0, el gradiente final es diminuto. La red se queda atascada: sabe que está mal, pero no tiene fuerza para moverse porque la pendiente es plana. Cuando usas Cross-Entropy, la función de coste está diseñada específicamente para cancelar esa zona plana de la sigmoide (o softmax). La Cross-Entropy tiene un logaritmo (ln) cuya derivada es $1/x$. Al aplicar la regla de la cadena para obtener el gradiente, el término que "aplastaba" el aprendizaje en la sigmoide desaparece matemáticamente. Mientras que con MSE si la red falla por mucho, el gradiente es pequeño (porque la sigmoide es plana al final) y el aprendizaje es lento al principio, con Cross-Entropy: Si la red falla por mucho, el gradiente es máximo. Cuanto más equivocada está la red, más fuerte es el «latigazo» que la obliga a corregir.