Apuntes sobre inteligencia artificial

Sergio de Mingo

6 de Febrero de 2026

Índice

1. El perceptrón

El Perceptrón (propuesto por Rosenblatt en los 50) es la unidad atómica de la IA. Es básicamente una función matemática que intenta imitar a una neurona biológica. No es más que un clasificador lineal binario. Un clasificador binario es una función que puede decidir si una entrada, representada por un vector de números, pertenece o no a una clase específica.

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

- Entradas (x): Los datos que llegan (ej. valores de píxeles).
- Pesos (w): La "importancia" que la red le da a cada entrada. Aprender en IA es, literalmente, ajustar estos números.
- Bias (b): El sesgo, que permite desplazar la función de activación hacia la izquierda o derecha para ajustarse mejor a los datos. Es como el umbral de un interruptor: Si el bias es muy alto y positivo, la neurona es «entusiasta» y se activa casi siempre. Por contra, si el bias es muy negativo, la neurona es «escéptica» y necesita señales de entrada muy fuertes para activarse.
- Función de activación (f): Decide si la neurona "dispara" una señal o no. Antiguamente usábamos la Heaviside (escalón) o la Sigmoide.

La limitación de este algoritmo es que si dibujamos en un gráfico estos elementos, se deben poder separar con un hiperplano únicamente los elementos «deseados» discriminándolos (separándolos) de los «no deseados». Esto lo veremos mejor en el punto 2, donde estudiaremos una Red Neuronal Clásica (Multilayer Perceptron). Este tipo no es más que muchas de estas neuronas organizadas en capas: una de entrada, una o varias ocultas y una de salida.

1.1. El aprendizaje de un perceptrón

El proceso de aprendizaje de un perceptrón comienza con una fase de **inicialización**, donde el modelo parte de un estado de desconocimiento absoluto, asignando valores aleatorios a sus pesos y un valor inicial (normalmente cero) a su sesgo o bias. Una vez configurado, **el aprendizaje se convierte en un ciclo iterativo que se repite durante varias épocas**. En cada iteración, el perceptrón realiza primero un forward pass o predicción: toma los datos de entrada, los multiplica por sus respectivos pesos, suma el sesgo y pasa el resultado por una función de activación (como la función escalón) para decidir si la neurona debe activarse o no. Inmediatamente después, el modelo compara su predicción con el valor real (y_i) que debería haber obtenido, calculando así el error:

$$Error = y_i - f(w_i \cdot x_i + b)$$

Si existe una discrepancia, entra en juego la regla de aprendizaje, donde el perceptrón ajusta sus pesos internos de forma proporcional a tres factores: la magnitud del error cometido, el valor de la entrada que causó dicho error y, fundamentalmente, la tasa de aprendizaje, que actúa como un regulador de la velocidad del cambio.

$$w_i = w_i + (x_i \times \text{Error} \times \text{Tasa de aprendizaje})$$

Este ajuste busca "mover" la frontera de decisión del perceptrón (la línea recta que separa las clases) para que, en el siguiente intento, la predicción sea más precisa. A través de la repetición constante de este proceso con todos los datos del conjunto de entrenamiento, los pesos convergen gradualmente hacia unos valores óptimos que permiten al modelo clasificar correctamente las entradas, logrando así que la máquina "aprenda" la lógica subyacente de los datos,

1.2. Trabajando con el perceptrón

Para empezar a programar un perceptrón necesitamos establecer su estado interno, tanto los pesos W como el sesgo b. Normalmente estos se inicializarán con valores pequeños. Tras esto haremos el cálculo de la predicción en sí, donde aplicamos la fórmula inicial del perceptrón con la aplicación de la función de activación. Por último, el aprendizaje o el ajuste de los pesos. En este caso esto se ven el método `init()`. Este método recibe además el número de inputs que tendrá el perceptrón para poder generar un array de pesos adecuado. El método `predict()` es básicamente la aplicación de la función de activación para cada componente del vector de entrada `X`. En este caso `Z` es el producto de cada componente del vector por el peso asignado (y al que al final sumamos el sesgo). Como activación usamos la función escalón por lo que el resultado será 1 si $z > 0$ y 0 en cualquier otro caso. El mecanismo fundamental se encuentra en el método `train()`. En este método es donde se irá probando al perceptrón y ajustando sus pesos hasta que el resultado sea el esperado. En este caso probamos 100 veces. En cada prueba realizamos el proceso de aprendizaje/reajuste, teniendo en cuenta que los resultados esperados los tenemos en el vector `y_AND`:

1. Calculamos la predicción para esa entrada: `prediction = self.predict(X[i])`
2. Calculamos el error comparando el resultado con lo esperado: `error = y[i] - prediction`
3. Recalculamos los pesos: `self.weights += error * self.lr * X[i]`
4. Recalculamos el sesgo: `self.bias += error * self.lr`

La magia se produce cuando tras probar el perceptrón con el vector de aprendizaje `y_AND` y viendo que ha sabido hacer una operación AND cambiamos el vector por `y_OR` y aprende a hacer una OR haciéndola correctamente para las mismas entradas.

Siguiendo este razonamiento nos topamos con el primer gran problema del perceptrón. Al intentar implementar este mismo aprendizaje para XOR usando un vector `y_XOR = np.array([0, 1, 1, 0])` vemos que es imposible. Si visualizamos el espacio de entrada como un plano con dos ejes y cuatro puntos (0,0),(0,1),(1,0),(1,1), vemos que en el AND, solo el punto (1,1) es positivo. Puedes dibujar una línea recta que separe ese punto de los otros tres. Lo mismo pasa en el OR donde tres puntos son positivos y solo el (0,0) es negativo. También puedes trazar una línea recta para separarlos.

A continuación se muestra el código del fichero `src/perceptron.py`:

```
class Perceptron:
    def __init__(self, n_inputs, learning_rate=0.1):
        self.weights = np.random.randn(n_inputs)
        self.bias = 0
        self.lr = learning_rate

    def predict(self, x):
        z = np.dot(x, self.weights) + self.bias
        return 1 if z > 0 else 0

    def train(self, X, y, epochs=100):
        for _ in range(epochs):
            for i in range(len(X)):
                prediction = self.predict(X[i])
                error = y[i] - prediction
                self.weights += error * self.lr * X[i]
                self.bias += error * self.lr


X = np.array([[0,0], [0,1], [1,0], [1,1]])
y_AND = np.array([0, 0, 0, 1])
y_OR = np.array([0, 1, 1, 1])

p = Perceptron(n_inputs=2)
p.train(X, y_AND)

print("Probamos que sepa calcular un AND u OR:")
for t in X:
    print(f"Entrada: {t} -> Prediccion: {p.predict(t)}")
```

2. Redes multicapa

Si recordamos el razonamiento del punto anterior donde definíamos en un plano de coordenadas los resultados del perceptrón, si intentamos separar los resultados positivos del XOR vemos que es imposible hacerlo con una sola línea. En el XOR, los puntos positivos (los resultados válidos con los que se obtiene un 1) son (0,1) y (1,0) y los negativos son (0,0) y (1,1). Es imposible separar ambos espacios (de positivos y negativos) con una sola línea recta. Necesitas dos o bien una curva. Esto se explica matemáticamente porque un perceptrón simple es matemáticamente un hiperplano. En 2D, es una recta. Si tus datos no son "linealmente separables", el perceptrón se quedará oscilando para siempre sin encontrar una solución. Para resolver el XOR, necesitamos añadir hacer una pequeña red multicapa:

1. **Capa de entrada** formada por dos neuronas: Recibirá los valores x_1 y x_2
2. **Capa oculta** formada por dos o tres neuronas: Aquí es donde ocurre la magia debido a que estas neuronas crearán nuevas dimensiones.
3. **Capa de salida** formada por una neurona que nos da el resultado final de la operación: 0 o 1.

Para realizar el cálculo ya no multiplicaremos el peso por la entrada en bucle, como haciamos antes. Ahora usaremos multiplicación de matrices. Siendo X la matriz de entrada y W_1 la matriz de pesos de la capa oculta calcularemos $Z_1 = X \cdot W_1 + b_1$ y luego aplicaremos la función de activación. En este caso usaremos la sigmoide. Para la capa de salida calcularemos $Z_2 = A_1 \cdot W_2 + b_2$ y de nuevo activaremos con la función sigmoide. Como nota indicamos que usamos la función sigmoide debido a que es una función no lineal. Si usáramos una función lineal, por muchas capas que pongamos, la red

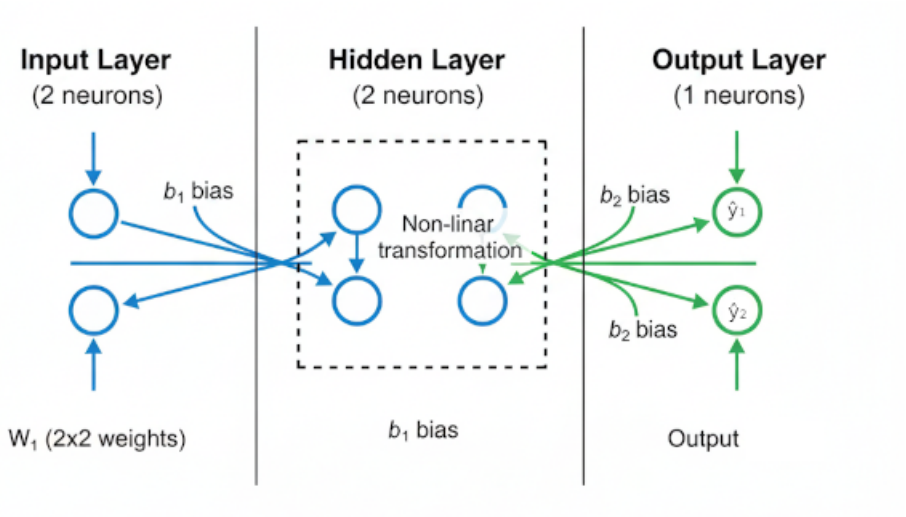

Figura 1: Esquema general de la red multicapa

seguiría siendo una simple combinación lineal (una sola línea recta). La no-linealidad es lo que permite "curvar" el espacio necesario para este caso.

2.1. Aprendizaje en varias capas

El concepto que revolucionó la IA en los 80 fue: ¿Cómo el error se calcula al final, en la salida? ¿cómo sabemos cuánto deben cambiar los pesos de la primera capa? El proceso de aprendizaje ahora se basa en estos tres pasos:

1. El *Forward Pass* o la multiplicación de las matrices y la activación
2. El cálculo del error; ¿Cuánto nos queda para el valor real?
3. El *Backpropagation* donde calculamos el «gradiente» de cada capa usando La Regla de la cadena.
4. Actualizamos los pesos restando el gradiente.

El **Gradiente** es lo que nos dice en qué dirección y con qué fuerza debemos mover el peso para que el error total disminuya lo más rápido posible. En una red neuronal, para saber cómo afecta un peso de la Capa 1 al error final (que se mide en la Salida), tenemos que aplicar **La Regla de La Cadena** a través de todas las capas intermedias. El gradiente que llega a la Capa 1 es el producto de las derivadas de las capas superiores. La derivada es como la velocidad a la que fluye la información del error. Si la derivada es alta (cercana a 1 o mayor): El error viaja con fuerza. El peso w_1 recibe un mensaje claro: *"¡Oye! Te has equivocado mucho, cambia tu valor rápido"*. Hay aprendizaje. Si la derivada es pequeña, el error se va atenuando en cada capa. ¿Por qué usamos derivadas? El objetivo de la red es minimizar una función de Error (o Pérdida). Imagina que el error es una montaña y tú estás en la cima, a ciegas. Quieres bajar al valle (error cero). ¿Cómo sabes hacia dónde dar el paso? Tocando el suelo con el pie para ver la pendiente. Esa pendiente es la derivada. Si la derivada es positiva, el terreno sube; vas hacia atrás. Si es negativa, el terreno baja; vas hacia adelante. El uso de la regla de la cadena es una cuestión de esto anterior. En definitiva una red neuronal es una función compuesta gigante. Si tenemos dos capas, una entrada X, una salida Y y una función de activación σ. La función que resumiría la primera fase sería:

$$Y = \sigma_2 \left(W_2 \cdot \sigma_1 \left(W_1 \cdot X \right) \right)$$

Cuando queremos saber cómo afecta el peso de la primera capa W_1 al error final, tenemos que "deshacer" la función de fuera hacia adentro. La Regla de la Cadena nos dice que **la derivada de una función compuesta es el producto de las derivadas de sus componentes**. Usamos la derivada de la activación porque es la única forma matemática de saber cuánta responsabilidad tiene una neurona específica en el error final. Esto es exactamente lo mismo que el análisis de sensibilidad en ingeniería de sistemas. ¿Cómo afecta una pequeña variación en la entrada a la salida del sistema? La respuesta es siempre la derivada. Si definimos el error como E y recordamos que σ' es la derivada de la función de activación, la fórmula compacta para calcular el gradiente de la primera capa (la más profunda en el Backpropagation) es:

$$\delta_1 = \underbrace{\left(\underbrace{\overbrace{(A_2 - y)}^{\text{Error}} \odot \sigma'(Z_2)}_{\text{Gradiente Salida } (\delta_2)} \cdot W_2^T \right)}_{\text{Error retropropagado}} \odot \sigma'(Z_1)$$

1. El inicio del error: $(A_2 - y)$ calcula cuánto nos alejamos del objetivo.
2. El filtro de la salida: Al multiplicar por $\odot\sigma'(Z_2)$, decidimos cuánta importancia dar a ese error según el estado de la neurona de salida.
3. El salto al pasado: Al multiplicar por W_2^T, proyectamos ese error de la salida hacia las neuronas ocultas usando los mismos pesos (pero transpuestos).
4. El filtro oculto: Finalmente, $\odot\sigma'(Z_1)$ ajusta ese error proyectado según la sensibilidad de la capa oculta.

A continuación se muestra el bucle de aprendizaje de la multicapa implementada de forma completa en `src/multicapa.py`:

```
for epoch in range(epochs):
    # --- FORWARD PASS ---
    hidden_layer_input = np.dot(X, W1) + b1
    hidden_layer_output = sigmoid(hidden_layer_input)

    output_layer_input = np.dot(hidden_layer_output, W2) + b2
    predicted_output = sigmoid(output_layer_input)

    # --- BACKPROPAGATION ---
    error = y - predicted_output
    d_predicted_output = error * sigmoid_derivative(predicted_output)

    error_hidden_layer = d_predicted_output.dot(W2.T)
    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

    # --- ACTUALIZACION DE PESOS (Gradiente Descendente) ---
    W2 += hidden_layer_output.T.dot(d_predicted_output) * learning_rate
    b2 += np.sum(d_predicted_output, axis=0, keepdims=True) * learning_rate
    W1 += X.T.dot(d_hidden_layer) * learning_rate
    b1 += np.sum(d_hidden_layer, axis=0, keepdims=True) * learning_rate
```

Tras la aplicación del *forward pass* y el cálculo de la predicción vemos como empieza la siguiente fase, el *backpropagation*. En el código, la variable `d_predicted_output` ya representa la derivada del error respecto a la salida. A esta variable la llamaremos también **Delta de salida** (δ_{out}). Es básicamente el error ponderado por la pendiente o derivada de la función de activación. Necesitamos

ponderar a la pendiente de la función para ver como de grande o pequeño tiene que ser el ajuste de los pesos. Para seguir propagando este error hacia adentro, ahora calculamos el reajuste en la capa intermedia u «oculta». Para actualizar los pesos de la segunda capa (W_2), aplicamos la regla de la cadena igualmente: la variación o delta del error en este caso depende del delta de salida anterior y de los pesos de esta capa. Esto lo guardamos en `error_hidden_layer` e igual que antes lo ponderamos a la derivada de la función de activación de esa capa.

Tras el cálculo de estas deltas comenzaría la fase de *Actualización de pesos*. La actualización de pesos en la salida la estamos haciendo en `W2 += hidden_layer_output.T.dot(...) ....` Aquí estamos calculando el gradiente para la matriz de pesos completa. Usamos la transpuesta (`.T`) porque queremos relacionar cada neurona oculta con cada error de salida, creando una matriz de ajustes que coincida con las dimensiones de W_2. La actualización de pesos en la entrada sigue la misma lógica pero un paso más atrás. La hacemos en `W1 += X.T.dot(....)`. Aquí, el gradiente depende de la entrada original X y del error que hemos "retropropagado" hasta la capa oculta (`d_hidden_layer`).

Con los sesgos no tenemos una propagación. Cada sesgo se actualiza en base a su propia neurona ya que el sesgo realmente determina qué tan fácil es que la neurona se active, independientemente de la señal de entrada. Para entender bien su actualización pensemos en una ecuación líneal $y = wx + b$. El peso (w) controla la pendiente (la inclinación de la línea) pero el bias (b) controla la intersección con el eje Y (desplaza la línea hacia arriba o hacia abajo). Si solo ajustáramos los pesos, todas nuestras líneas de decisión tendrían que pasar obligatoriamente por el origen (0,0). El bias permite que la línea se mueva libremente por el espacio para rodear los datos donde sea que estén.

2.2. Representación matricial

Vamos a representar las diferentes matrices que más juego tienen en el ejemplo para visualizar mejor todo esto. Partimos de las dos matrices iniciales: la **matriz de datos de entrada** o X, donde la primera columna es la entrada x_1 y la segunda es x_2 y la **matriz de resultados esperados** o y:

$$X = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \qquad y = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

2.2.1. Forward Pass

Ahora empezamos con las matrices más importantes y relacionadas con el proceso de aprendizaje. Primeramente tenemos la **matriz de pesos de la capa oculta** o W_1. Esta matriz conecta las 2 entradas con las 2 neuronas ocultas. Es una matriz de 2×2. Cada columna representa los pesos que llegan a una neurona oculta específica.

$$W_1 = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix}$$

Tras esta también tenemos la **matriz de Salida de la Capa Oculta** o A_1 o `hidden_layer_output`). Es el resultado de multiplicar la entrada X (de 4×2) por W_1. Con esto obtenemos el resultado parcial Z_1 y finalizamos aplicando la función de activación (sigmoide). El resultado es una matriz de 4 filas (ejemplos) y 2 columnas (activaciones de las neuronas ocultas). Cada fila i representa cómo "ve" la capa oculta el ejemplo i del XOR.

$$Z_1 = \left(\begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \right) + (b_1, b_1)$$

$$A_1 = \sigma(Z_1) = \begin{pmatrix} \sigma(z_{11}) & \sigma(z_{12}) \\ \sigma(z_{21}) & \sigma(z_{22}) \\ \sigma(z_{31}) & \sigma(z_{32}) \\ \sigma(z_{41}) & \sigma(z_{42}) \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \\ a_{4,1} & a_{4,2} \end{pmatrix}$$

Vamos ahora con el siguiente paso del aprendizaje. Ahora igual que antes obtenemos la salida de esta capa aplicando como entrada A (calculada anteriormente) con los pesos W_2 y el sesgo apropiado. A esto lo llamamos Z_2 y a esto le aplicaremos la función de activación para obtener A_2 o lo que es lo mismo, la **matriz de predicción final**:

$$Z_2 = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix} \times \begin{pmatrix} w_1' \\ w_2' \end{pmatrix} + \begin{pmatrix} b_2 \\ b_2 \\ b_2 \\ b_2 \end{pmatrix} = \begin{pmatrix} (a_{11} \cdot w_1' + a_{12} \cdot w_2') + b_2 \\ (a_{21} \cdot w_1' + a_{22} \cdot w_2') + b_2 \\ (a_{31} \cdot w_1' + a_{32} \cdot w_2') + b_2 \\ (a_{41} \cdot w_1' + a_{42} \cdot w_2') + b_2 \end{pmatrix}$$

$$A_2 = \sigma(Z_2) = \begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \\ \hat{y}_4 \end{pmatrix}$$

Cada $\hat{y}_i$ es ahora una combinación no lineal de las activaciones ocultas. Si la capa oculta ha hecho bien su trabajo, habrá separado los puntos de tal manera que para el caso (0,1) y (1,0), los valores de a_{i1} y a_{i2} activen con fuerza a W_2, mientras que para (0,0) y (1,1) la combinación resulte en un valor que la sigmoide convierta en casi 0.

2.2.2. Backpropagation o propagación del error

Entramos en la parte más profunda del backpropagation de la IA. Aquí es donde el error viaja hacia el pasado. Primeramente calculamos cuánto error hay en la salida final. Este valor también lo llamaremos **delta de salida** y se representa en código con la variable `d_predicted_output`. Es el producto elemento a elemento (Hadamard product, representado por $\odot$) entre el error residual y la derivada de lo que salió. Esto nos devuelve finalmente una matriz 4×1:

$$\delta_{out} = (y - A_2) \odot \sigma'(Z_2)$$

$$\delta_{out} = \begin{pmatrix} (y_1 - \hat{y}_1) \cdot \sigma'(z_{2,1}) \\ (y_2 - \hat{y}_2) \cdot \sigma'(z_{2,2}) \\ (y_3 - \hat{y}_3) \cdot \sigma'(z_{2,3}) \\ (y_4 - \hat{y}_4) \cdot \sigma'(z_{2,4}) \end{pmatrix} = \begin{pmatrix} \delta_{out,1} \\ \delta_{out,2} \\ \delta_{out,3} \\ \delta_{out,4} \end{pmatrix}$$

Aquí es donde aplicamos la línea: `error_hidden_layer = d_predicted_output.dot(W2.T)`. Como ingeniero, piensa en esto como una proyección inversa: estamos enviando el error de salida de vuelta a través de los mismos cables (W_2) por los que vino.

$$\text{Error}_{hidden} = \delta_{out} \times W_2^T$$

$$\text{Error}_{hidden} = \begin{pmatrix} \delta_{out,1} \\ \delta_{out,2} \\ \delta_{out,3} \\ \delta_{out,4} \end{pmatrix} \times \begin{pmatrix} w_1' & w_2' \end{pmatrix} = \begin{pmatrix} \delta_{out,1} \cdot w_1' & \delta_{out,1} \cdot w_2' \\ \delta_{out,2} \cdot w_1' & \delta_{out,2} \cdot w_2' \\ \delta_{out,3} \cdot w_1' & \delta_{out,3} \cdot w_2' \\ \delta_{out,4} \cdot w_1' & \delta_{out,4} \cdot w_2' \end{pmatrix}$$

El resultado es una matriz de 4×2. Cada columna representa cuánto ruido o error le llegó a cada una de las 2 neuronas ocultas. Finalmente, para saber cuánto debemos cambiar los pesos W_1, necesitamos filtrar ese error por la sensibilidad de la activación de la capa oculta. Es decir, multiplicamos por la derivada de la sigmoide de la capa oculta. Esto se representa en el código con la variable `d_hidden_layer`:

$$\delta_{hidden} = \text{Error}_{hidden} \odot \sigma'(Z_1)$$

$$\delta_{hidden} = \begin{pmatrix} \text{err}_{1,1} \cdot \sigma'(z_{1,1}) & \text{err}_{1,2} \cdot \sigma'(z_{1,2}) \\ \text{err}_{2,1} \cdot \sigma'(z_{2,1}) & \text{err}_{2,2} \cdot \sigma'(z_{2,2}) \\ \text{err}_{3,1} \cdot \sigma'(z_{3,1}) & \text{err}_{3,2} \cdot \sigma'(z_{3,2}) \\ \text{err}_{4,1} \cdot \sigma'(z_{4,1}) & \text{err}_{4,2} \cdot \sigma'(z_{4,2}) \end{pmatrix} = \begin{pmatrix} \delta_{h1,1} & \delta_{h2,1} \\ \delta_{h1,2} & \delta_{h2,2} \\ \delta_{h1,3} & \delta_{h2,3} \\ \delta_{h1,4} & \delta_{h2,4} \end{pmatrix}$$

Mientras que `error_hidden_layer` es el error en bruto que le llega a la capa oculta desde la salida, `d_hidden_layer` es el error ya ponderado por la pendiente de la activación. Si una neurona oculta estaba en la zona plana de la sigmoide (valor muy alto o muy bajo), su derivada será casi 0, y por tanto, su `d_hidden_layer` será casi 0. Dicho de otra manera, si la neurona ya estaba muy convencida de su respuesta (activación saturada), no le importa el error que le mandes porque no va a cambiar sus pesos.

2.2.3. Actualización de pesos

Para completar el ciclo, vamos a ver cómo ese error que ha viajado hacia atrás se convierte finalmente en instrucciones de ajuste para los pesos. En el código, estas son las líneas de Gradiente Descendente. La variable en el código que vamos a desarrollar ahora sería: `W2 += hidden_layer_output.T.dot(...) * learning_rate.` En ella se actualizan los valores de $W2$. Para ajustar esos pesos (recordamos que són los que conectan la capa oculta con la salida), multiplicamos la activación que salió de la capa oculta (A_1^T) y que calculamos en la fase de *Forward Pass* por el delta de salida (δ_{out}) obtenido en la fase de *backpropagation*. En la formulación usaremos η para referirnos al *learning rate*. El resultado es una matriz de $[2 \times 1]$, que encaja perfectamente con las dimensiones de W_2:

$$\Delta W_2 = \eta \cdot \left(\begin{pmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \end{pmatrix} \times \begin{pmatrix} \delta_{out,1} \\ \delta_{out,2} \\ \delta_{out,3} \\ \delta_{out,4} \end{pmatrix} \right)$$

$$\Delta W_2 = \eta \cdot \begin{pmatrix} \sum(a_{i1} \cdot \delta_{out,i}) \\ \sum(a_{i2} \cdot \delta_{out,i}) \end{pmatrix}$$

Ahora hacemos algo similar con los pesos iniciales o W_1. La variable en el código es: `W1 += X.T.dot(d_hidden_layer) * learning_rate`. Aquí es donde resolvemos el misterio del XOR. Usamos la entrada original (X^T) y la multiplicamos por el error que hemos calculado para la capa oculta (δ_{hidden}). El resultado es una matriz de $[2 \times 2]$, lista para sumarse a W_1:

$$\Delta W_1 = \eta \cdot \left(\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \delta_{h1,1} & \delta_{h2,1} \\ \delta_{h1,2} & \delta_{h2,2} \\ \delta_{h1,3} & \delta_{h2,3} \\ \delta_{h1,4} & \delta_{h2,4} \end{pmatrix} \right)$$

$$\Delta W_1 = \eta \cdot \begin{pmatrix} \sum(x_{i1} \cdot \delta_{h1,i}) & \sum(x_{i1} \cdot \delta_{h2,i}) \\ \sum(x_{i2} \cdot \delta_{h1,i}) & \sum(x_{i2} \cdot \delta_{h2,i}) \end{pmatrix}$$

Es importante fijarse en la belleza de la operación $X^T \cdot \delta_{hidden}$. Si la entrada x_{11} fue 0, no importa cuán grande sea el error $\delta_{h1,1}$, el producto será 0. Esto tiene todo el sentido lógico: si una entrada fue cero, no pudo haber contribuido al error final de esa neurona, por lo tanto, no hay razón para cambiar el peso que viene de ella.

Para terminar dejamos aquí una relación entre las variables usadas en el código fuente y su simbología matemática utilizada en el desarrollo:

- `X`: (X) Matriz de Entrada: Los 4 casos del XOR (4×2).
- `y`: (y) Etiquetas Reales: Los resultados deseados (4×1).
- `W1, W2`: (W_1, W_2) Matrices de Pesos: Las conexiones entre capas.
- `b1, b2`: (b_1, b_2) Vectores de Sesgo (Bias): El umbral de activación.
- `hidden_layer_output`: (A_1) Activación Oculta: La salida de la capa intermedia.
- `predicted_output`: (A_2 o $\hat{y}$) Predicción Final: La salida de la red.
- `error`: ($y - A_2$) Error Residual: Diferencia bruta entre realidad y predicción.
- `d_predicted_output`: (δ_2 o Delta 2) Gradiente de Salida: Error de salida multiplicado por la derivada.
- `error_hidden_layer`: (E_{hidden}) Error Retropropagado: El error que "rebota" hacia la capa oculta.
- `d_hidden_layer`: (δ_1 o Delta 1) Gradiente Oculto: El error en la capa oculta listo para ajustar W_1.
- `learning_rate`: (η) Tasa de Aprendizaje: El factor de escala de los ajustes.

2.3. Estudio del error y del aprendizaje

En la gráfica que se muestra a continuación se ve un estudio de como evoluciona el error en las diferentes iteraciones del aprendizaje con diferentes tasas de aprendizaje η. Recordamos que para calcular el error durante el aprendizaje usamos en cada iteración la formula del Error Cuadrático Medio aunque realmente lo que propagábamos era su derivada. Con `square()` elevamos cada miembro del array al cuadrado y con `mean()` calculamos la media de todos los valores del vector de errores. Si se usa una tasa muy grande como $\eta = 10$ el modelo "saltará" tan fuerte que nunca caerá en el valle del error. Verás que los resultados oscilan locamente. Por contra si usamos una tasa muy baja el modelo tardará millones de épocas en aprender. Es como intentar vaciar el océano con una cuchara.

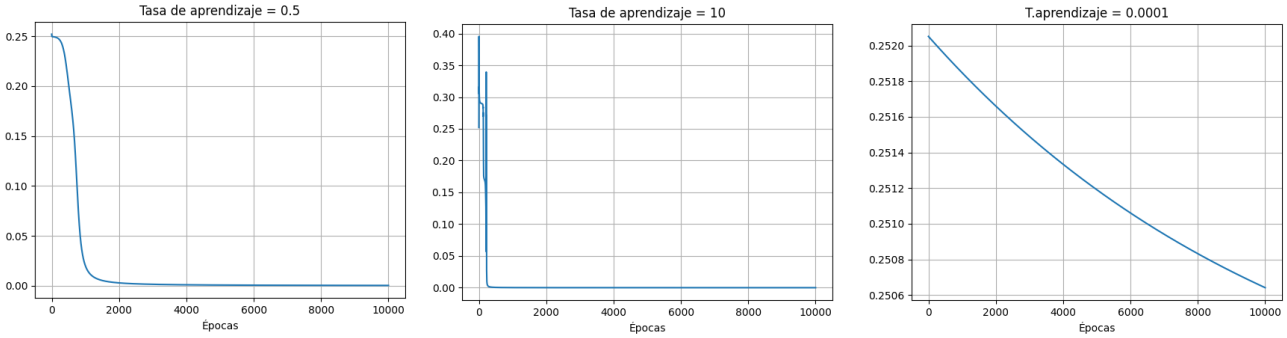

Figura 2: Evolución del error con diferentes tasas de aprendizaje

Usamos una tasa intermedia ($\eta = 0{,}5$). Verás que al principio el error baja muy poco (la red está explorando), luego hay una caída vertical pronunciada (ha encontrado la lógica del XOR) y finalmente se estabiliza cerca de cero. Si el error se queda plano en un valor alto (por ejemplo, 0.25), significa que la red se ha quedado atrapada en un mínimo local. En el XOR, esto suele significar que la red ha decidido que «la mejor opción es decir siempre 0.5», porque no logra entender la lógica no lineal. En estos casos es posible que necesitemos reiniciar la red debido a la mala asignación de pesos aleatoria al comienzo. Como los pesos iniciales son aleatorios, a veces la red empieza en una posición «desafortunada» del mapa matemático. Se pueden utilizar otras técnicas (como Xavier/Glorot) para realizar una preasignación más eficiente que la simplemente aleatoria.

Si el error colapsa hacia cero alrededor de las 1500 iteraciones (como en el caso de usar $\eta = 0{,}5$) en un problema como el XOR, significa que la tasa de aprendizaje está muy bien balanceada. Que la curva baje muy rápido tampoco es siempre un buen síntoma. Por ejemplo, si cayera a las 10 iteraciones: Sospecharíamos de overfitting (sobreajuste) o de que el problema es demasiado simple para la potencia de la red. A veces también puede ocurrir que una caída rápida se detiene en seco antes de llegar a cero. Eso significa que la red «se ha rendido» en un punto que le parece aceptable pero que no es la solución óptima. Para confirmar que esa tasa de 1500 es perfecta, haz esta prueba: fíjate en el final de la curva. Si es plana y suave al final tu tasa es perfecta. La red ha aterrizado suavemente en el mínimo. Si el final de la curva tiene ruido (pequeños dientes de sierra) tu tasa es un poco alta. La red ha llegado al fondo pero está rebotando un poco en las paredes del valle.

3. 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 $4{\times}2$. 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.

3.1. 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.

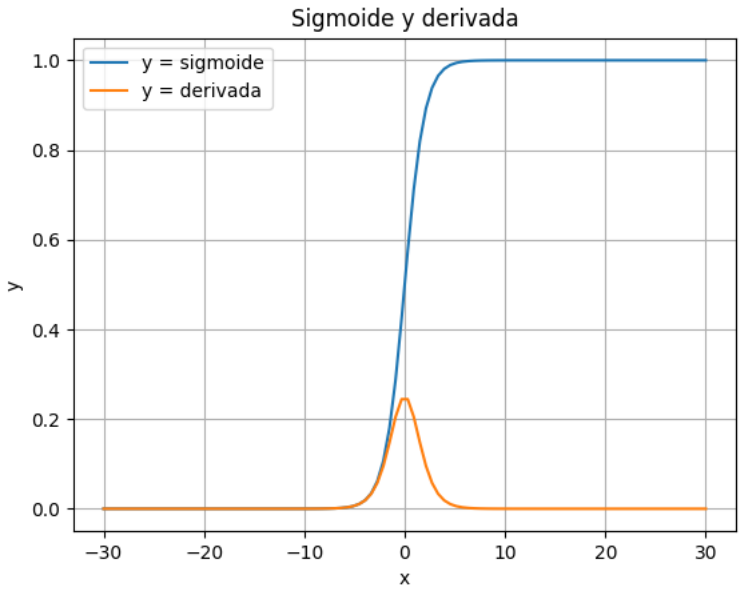

Figura 3: Curva de la sigmoide y su derivada

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».

3.2. 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.

3.3. Arquitectura de red para MNIST

En el XOR solo necesitabas una capa para separar puntos. Para reconocer números (MNIST), la primera capa oculta puede aprender a detectar bordes y líneas, mientras que la segunda capa combina esos bordes para reconocer curvas y círculos (como los de un "8" o un "0"). Vamos a seguir un esquema entonces a 4 capas: $[768 \to 128 \to 64 \to 10]$:

- Entrada (X): [Batch×784] Lote de imágenes aplanadas.
- Capa Oculta 1 (W_1): [784×128]. Transforma píxeles en rasgos básicos.
- Capa Oculta 2 (W_2): [128×64]. Transforma rasgos básicos en formas complejas.
- Capa de Salida (W_3): [64×10]. Clasifica las formas en dígitos del 0 al 9.

Usaremos dos capas debido a que se gana eficiencia en este reparto de tareas. La profundidad suele ser sinónimo de inteligencia en una red neuronal. Esta profundidad permite a la red entender conceptos jerárquicos. El ancho de la red permite recordar muchos detalles específicos. A mayor ancho la red tiene más capacidad de memoria. Respecto al dimensionado de las capas diremos que, mientras que el 128 de la capa de entrada es una obligación física para almacenar el aplanado de las imágenes (explicado en el apartado anterior), el 128 y el 64 de las capas siguientes son una decisión de diseño. Para el 128 no hay

una ley física que marque. Podría ser 100, 200 o 512. Sin embargo, en computación usamos potencias de 2 ($2^7 = 128$) porque las GPUs y CPUs gestionan mucho más rápido la memoria en bloques de este tamaño. Con el 64 pasa lo mismo. Se han diseñado ambas capas como un embudo: Empiezas con 784 detalles brutos (píxeles sueltos, la primera capa los resume en 128 rasgos (líneas, ángulos, bordes), la segunda capa resume esos rasgos en 64 conceptos más complejos (curvas cerradas, cruces) y finalmente todo se reduce a 10 categorías (los dígitos). Podemos jugar a modificar estos números. Si pones números muy pequeños, la red tiene un «cuello de botella». Estás intentando resumir mucha información en muy pocas neuronas. La red será muy rápida, pero probablemente sea menos inteligente y no aprenda bien los detalles. Mientras que si pones números muy grandes la red tendrá una memoria fotográfica increíble. El problema es que será muy lenta de entrenar y, lo más peligroso, podría sufrir de Overfitting, asunto del que hablaremos a continuación. No debemos olvidar además que para que la multiplicación de matrices funcione, la regla es que **la salida de una capa debe ser la entrada de la siguiente**. Resumimos por tanto como queda el proceso matricial o *Forward Pass*:

1. Capa 1: $Z_1 = X \cdot W_1 + b_1 \Rightarrow A_1 = ReLU(Z_1)$
2. Capa 2: $Z_2 = A_1 \cdot W_2 + b_2 \Rightarrow A_2 = ReLU(Z_2)$
3. Capa 3: $Z_3 = A_2 \cdot W_3 + b_3 \Rightarrow A_3 = Softmax(Z_3)$

Ahora el error ahora viajará a través de tres capas por lo que la cadena se alarga y se complica. Para llegar a actualizar W_1 (la capa más alejada de la salida) la señal de error debe propagarse de la siguiente manera: Surge en la salida ($\delta_3 = A_3 - y$), luego pasa por W_3^T para llegar a la Capa 2 y por último pasa por W_2^T para llegar a la Capa 1. Por lo que la fórmula de la Delta 1 (δ_1) compelta sería:

$$\delta_1 = ((\underbrace{(A_3 - y)}_{\delta_3} \cdot W_3^T \odot \mathrm{ReLU}'(Z_2)) \cdot W_2^T) \odot \mathrm{ReLU}'(Z_1)$$

Antes de terminar hablaremos del problema del *overfitting* (en español conocido como sobreajuste) que hemos mencionado anteriormente. Es el enemigo número uno del aprendizaje automático. En términos sencillos, ocurre cuando tu red neuronal tiene tanta memoria que, en lugar de aprender a reconocer los patrones de un número, se memoriza las imágenes exactas de tu set de entrenamiento. Igual que un estudiante que, en lugar de entender las fórmulas de física, se memoriza de memoria las respuestas de los 100 problemas del libro. Si en el examen le pones el problema 101 (aunque sea igual de fácil), suspenderá porque no sabe razonar, solo repetir. Para detectar el overfitting, los ingenieros dividen los datos en dos grupos: Entrenamiento (Training) y Prueba (Testing/Validation). Los primeros son los datos que ve la red y que usa para ajustar los procesos. Los segundos son datos nuevos que la red nunca ha visto. Detectar este sobreajuste puede hacerse cuando el error en el entrenamiento es casi 0. La red parece casi perfecta con lo que conoce. Sin embargo, el error en la prueba con los datos reales es muy alto. Para evitar esto podemos aplicar varias técnicas:

- **Early Stopping** (Parada temprana): Dejamos de entrenar en el momento en que el error de Prueba deja de bajar y empieza a subir, aunque el error de Entrenamiento siga bajando.
- **Dropout** (Abandono): Durante el entrenamiento, «apagamos» aleatoriamente algunas neuronas en cada paso (por ejemplo, el 20 %). Esto obliga a la red a no depender de una sola neurona o camino, forzándola a crear representaciones más robustas y generales.
- **Regularización** (L2): Añadimos una penalización al error si los pesos (W) se vuelven demasiado grandes. Pesos gigantes suelen ser señal de que la red está forzando una curva muy compleja para pasar exactamente por todos los puntos.

3.4. Desarrollo del modelo

Para este desarrollo usaremos PyTorch. Principalmente porque PyTorch hace el Backpropagation automáticamente por ti. Tú solo defines el *Forward*, y él se encarga de las derivadas de la regla de la cadena que tanto nos costó calcular anteriormente. El código está completamente desarrollado en `src/mnist.py`. Necesitamos instalar las librerías de Python `torch`y `torchvision` usando `pip3`.

```
class RedMNIST(nn.Module):
    def __init__(self):
        super(RedMNIST, self).__init__()
        self.flatten = nn.Flatten()
        self.stack = nn.Sequential(
            nn.Linear(784, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 10),
            nn.LogSoftmax(dim=1)
        )

    def forward(self, x):
        x = self.flatten(x)
        return self.stack(x)
```

El primer punto importante del código es la preparación de los datos. De esto se encarga `DataLoader`: En lugar de darle las 60,000 imágenes de golpe, se las damos en *batches* o lotes de 64. El conjunto de todas las imágenes están ubicadas en `trainset`.

Vamos ahora a explicar el código del modelo. En el constructor inicialiremos los parámetros (pesos y sesgos). El método `self.flatten` será el que reciba la imagen y la aplane. de ahí comenzaremos a aplicar toda el flujo de la red que definimos capa a capa el método `nn.Sequential()`. Este método es realmente el cerebro de nuestra red y que hemos explicado en el apartado anterior[1]. Desde fuera del modelo, el punto de inicio será la llamada al método `forward()`. Cuando llamemos a ese método habrá que pasarle un lote de imágenes en el parámetro `x`. Para terminar con el constructor diremos que la función `nn.Linear()` crea automáticamente las matrices de pesos y el vector de bias con las dimensiones correctas. Además, inicializa los pesos de forma inteligente (usando Xavier/Glorot), así que ya no tienes que preocuparte por si empiezan en cero.

```
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=0.003)

for epoch in range(2):
    running_loss = 0.0
    for images, labels in trainloader:
        optimizer.zero_grad()          # Reset de gradientes
        outputs = model(images)        # Forward
        loss = criterion(outputs, labels) # Error
        loss.backward()                # Backpropagation
        optimizer.step()               # Ajuste de pesos
        running_loss += loss.item()
    print(f"Epoca {epoch+1} - Error: {running_loss/len(trainloader):.4f}")
```

Vamos ahora a explicar el proceso de entrenamiento. Para acortar la prueba únicamente realizaremos dos vueltas pero podríamos incrementarlo si quisiéramos. En cada imagen del `trailoader`, además de reiniciar los gradientes, corremos el modelo con la imagen que toque y calculamos el error

[1] Usamos `nn.LogSoftmax` en lugar de `nn.Softmax` porque trabajar con logaritmos de probabilidades es numéricamente más estable. Evita que el ordenador trabaje con números tan pequeños que se conviertan en cero por error

con el método `criterion()`. Este método recibe el vector de salida y el vector asociado a la imagen o `labels` con los resultados que deberíamos haber obtenido. Una vez obtenido este error lo propagaremos hacia atrás con `loss.backward()` y ajustaremos los pesos de toda la red usando `optimizer.step()`. En `running_loss` iremos acumulado los errores de cada lote de 64 imágenes. La idea es que al final de la época calculemos la media dividiendo ese `running_loss` entre el número de lotes que hemos tratado. Como curiosidad técnica diremos que el uso de `loss.item()` provoca que el computador no consuma tanta memoria RAM. La variable loss no es un número normal de Python; es un Tensor de un solo elemento. Este tensor no solo guarda el valor del error (ej. 2.34), sino que también guarda todo el Grafo Computacional (el historial de todas las operaciones matemáticas que llevaron a ese número) para poder hacer el *Backpropagation*. Si hicieramos un `running_loss += loss` estaríamos guardando el tensor completo junto con su historial de cálculos. Al final de la época, tendrías una cadena gigante de miles de operaciones guardadas en tu memoria RAM. Tu ordenador se quedaría sin memoria rápidamente (Memory Leak). De ahí que usemos `loss.item()`, así extraemos del tensor el valor numérico puro (un float), descartando el resto y salvando memoria en cada época.

Por último explicaremos el uso del optimizador Adam. Antes, en redes anteriores hemos usado una tasa de aprendizaje fija o η. Adam es un optimizador con memoria y adaptabilidad. Si el error baja siempre en la misma dirección, Adam "acelera" (como una bola rodando por una colina). A esto le llamamos *Momentum*. Adam también posee una tasa adaptativa. Si una neurona recibe gradientes muy bruscos, Adam le baja la velocidad individualmente para que no se vuelva loca. Si otra neurona casi no se mueve, le sube la velocidad para que aprenda más rápido. Es el estándar de la industria porque hace que casi cualquier red converja mucho más rápido y sin romperse la cabeza ajustando la tasa de aprendizaje manualmente.

3.5. Prácticas propuestas

1. Realiza una implementación de la Red XOR de la sección anterior pero usando PyTorch. (Hecho en `src/multicapa-ptorch.py`
2. Realiza una implementación de una Red multicapa con PyTorch que aprenda a resolver ecuaciones de segundo grado. (Hecho en `src/ecuaciones-ptorch.py` y documentado en `pdf/prac-01-ecuaciones.`
3. Modifica la arquitectura presentada en este apartado añadiendo una tercera capa oculta de 32 neuronas, cambiando el analizador a uno de tipo estocástico (`optim.SGD`) y observa que ocurre con `running_loss`.

4. Redes Neuronales Convolucionales

Anteriormente aplanamos una imagen de 28×28 a 784. Al hacer eso, la red ya no sabe si el píxel 1 está al lado del píxel 2 o del píxel 28. Simplemente ve una lista de números. Piensa en esto: ¿Cómo podrías tú reconocer una cara si te dieran todos los píxeles de una foto mezclados en una bolsa? Sería imposible. Las Redes Neuronales Convolucionales o CNN vienen a resolver precisamente eso. Hasta ahora, las redes veían los píxeles como una lista plana. Si movíamos el dibujo de un "8" un poco a la derecha, la red se confundía porque los píxeles ya no caían en las mismas neuronas de entrada. Las CNN solucionan esto imitando el córtex visual humano. En lugar de conectar cada píxel a una neurona, usamos un filtro. Como si fuera una pequeña rejilla de 3×3 que se desliza sobre la imagen. Este filtro no mira toda la imagen a la vez; mira solo un pequeño trozo, extrae una característica (como una línea vertical o un borde) y se mueve al siguiente píxel. La red ahora no aprende a reconocer «píxeles en tal posición», sino que aprende los valores de ese filtro que detectan rasgos importantes (bordes, curvas, texturas). La estructura maestra de una CNN podría resumirse en los siguiente puntos:

- **Capa de convolución**: Es el motor. Aquí es donde los filtros escanean la imagen. Si aplicas 32 filtros diferentes, obtendrás 32 mapas de características (versiones de la imagen donde resaltan cosas distintas).

- **Capa de pooling**: Es el sintetizador. Su trabajo es reducir el tamaño de la imagen. Toma, por ejemplo, un cuadrado de 2×2 y se queda solo con el valor más alto (el más brillante). Hacemos esto porque si detectamos un borde, no nos importa el píxel exacto, nos importa saber que ahí hay un borde. Esto hace que la red sea invariante a la traslación.
- **Capa totalmente conectada**: Al final de la red, después de que los filtros hayan extraído toda la información, volvemos a usar las capas que ya conoces para tomar la decisión final (¿Es un 8 o es un 3?).

A nivel global podemos resumir las funciones de las capas de la siguiente manera. Mientras que las capas iniciales detectan cosas simples como líneas, puntos y colores, las capas intermedias combinan esas líneas para detectar formas como círculos, cuadrados, ojos, etc. Por último, las capas finales combinan esas formas para detectar objetos complejos como caras, coches, números, etc.

Vamos a empezar la red trabajando con una entrada en forma de matriz de 28x28. Ya no trabajamos con un vector plano como en las redes anteriores. Usaremos para esta primera capa 32 filtros de 3x3. De esa capa salimos con una matriz de 26x26 pues al aplicar filtros de 3x3 y deslizarlos por toda la imagen, cuando llegamos a los bordes al tener que evitar que estos filtros se salgan perdemos el equivalente a un marco de 1 por cada lado de la matriz. De ahí que la matriz de 28x28 se quede en una de 26x26.

En la capa de *pooling* es donde se produce una compresión. Aplicando un Max Pooling de 2x2 a la salida de 26x26 de la capa anterior, el pooling divide la imagen en bloques de 2x2 y se queda con un máximo de uno. Esto reduce a la mitad la matriz con un resultado de 13x13. Resumimos la arquitectura indicando los tensores usados:

1. **Entrada**: [1x28x28] Tensor de 28x28 para un canal de color.
2. **Tras la convolución** con 32 filtros de 3x3: Obtenemos [32x26x26], lo que equivale a 32 versiones distintas de la imagen original, una por filtro usado.
3. **Tras el Pooling** de 2x2: Obtenemos [32x13x13]

Aunque ahora tenemos más datos (32 mapas en lugar de 1), estos datos están jerarquizados. Cada uno de los 32 canales se especializa en algo: el canal 1 puede detectar líneas horizontales, el canal 2 detecta esquinas, etc. Pero antes de llegar a la salida (los 10 números del MNIST), tenemos que volver al mundo que ya conoces. Si al final de las capas de convolución nos queda un bloque de $32 \times 13 \times 13$, lo «aplanamos» multiplicando todas sus dimensiones: $32 \times 13 \times 13 = 5408$ neuronas. Esas 5408 neuronas entrarán en una capa `nn.Linear` clásica. Mostramos a continuación el esquema básico del modelo pero tenemos el ejemplo completo desarrollado para su estudio en `src/mnist-cnn.py`:

```
class NetCNN(nn.Module):
    def __init__(self):
        super(NetCNN, self).__init__()

        # 1. CAPAS CONVOLUCIONALES (Extraccion de rasgos)
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3)
        self.pool = nn.MaxPool2d(2, 2)

        # 2. CAPAS LINEALES (Clasificacion)
        self.fc1 = nn.Linear(5 * 5 * 64, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 5 * 5 * 64)

        x = F.relu(self.fc1(x))
        x = self.fc2(x) # Salida de 10 neuronas

        return F.log_softmax(x, dim=1)
```

Vamos a detallar ahora los aspectos mas interesantes del código. Comenzamos con las dos capas convolucionales: `conv1` y `conv2`. ¿Por qué dos? La respuesta está en la jerarquía de la visión. Imagina que estás intentando reconocer una cara. Con una sola capa (`conv1`), la red solo ve palitos y puntos (bordes). Es imposible reconocer una cara solo sabiendo que hay 100 líneas verticales. Necesitas saber si esas líneas forman un ojo, una nariz o una boca. Ahí es donde entra la `conv2`. La clave de esta segunda capa es que no mira la imagen original. Mira los mapas de características que creó la primera capa. Por ejemplo, mientras que `conv1` detecta rasgos simples (bordes, diagonales, manchas de color), `conv2` mira dónde hay bordes y los combina. Por ejemplo: «Si aquí hay una curva hacia arriba y debajo una línea horizontal, entonces aquí hay un semicírculo». Sin esa segunda capa, tu red sería como alguien que ve las letras de un libro pero es incapaz de juntarlas para formar palabras. Respecto a los canales (el primer parámetro de la función `nn.Conv2d()`) en la `conv1` pusimos 32 filtros, pero en la `conv2` pusimos 64. Al aumentar los canales en la segunda capa, le damos a la red un «vocabulario» mucho más rico para describir lo que está viendo (curvas cerradas, cruces, ángulos agudos, etc.). Estamos aumentando la semántica de la red. Resumiendo, si quitamos la `conv2`, la red intentaría pasar directamente de «detectar bordes» a «decidir si es un 8». Como no ha tenido una capa intermedia para entender qué es un «círculo», le costará muchísimo generalizar. Tendría que compensarlo con una capa lineal (`nn.Linear`) gigantesca y muy ineficiente.

Si vamos a la función `forward()` vemos que tras la primera convolución aplicamos un *pooling* y tras la segunda otro. Al poner capas una tras otra (especialmente con un Pooling de por medio), cada neurona de la capa 2 está viendo un área mucho más grande de la imagen original que una neurona de la capa 1. Una neurona en `conv1` solo mira 3×3 píxeles. No sabe si está en la parte de arriba o de abajo de un número. Tras el *pooling* y la `conv2`, una sola neurona de esa segunda capa puede estar recibiendo información que originalmente cubría un área de 10×10 o más. Por eso se dice que esta segunda capa tiene «perspectiva». Puede entender la geometría del número, no solo los píxeles sueltos. A esto se le llama también campo receptivo.

4.1. Funcionamiento de una capa convolucional

Vamos a describir brevemente la maquinaria dentro de una de estas capas representadas por la función `Conv2d()`. Imagina que tienes la imagen de entrada y sobre ella colocas una rejilla de 3×3 (el filtro). Cada celda de esa rejilla tiene un peso (W). Entonces ocurre una operación matemática llamada

convolución (técnicamente, una correlación cruzada). La convolución es una operación matemática fundamental que combina dos funciones (o señales), f y h para producir una tercera que describe cómo una modifica a la otra. El proceso es más o menos el siguiente:

1. El filtro se sitúa sobre los primeros 9 píxeles.
2. Multiplica cada píxel por su peso correspondiente en el filtro.
3. Suma los 9 resultados y añade el *bias*.
4. Ese resultado único se convierte en un solo píxel de la nueva imagen (el mapa de características).

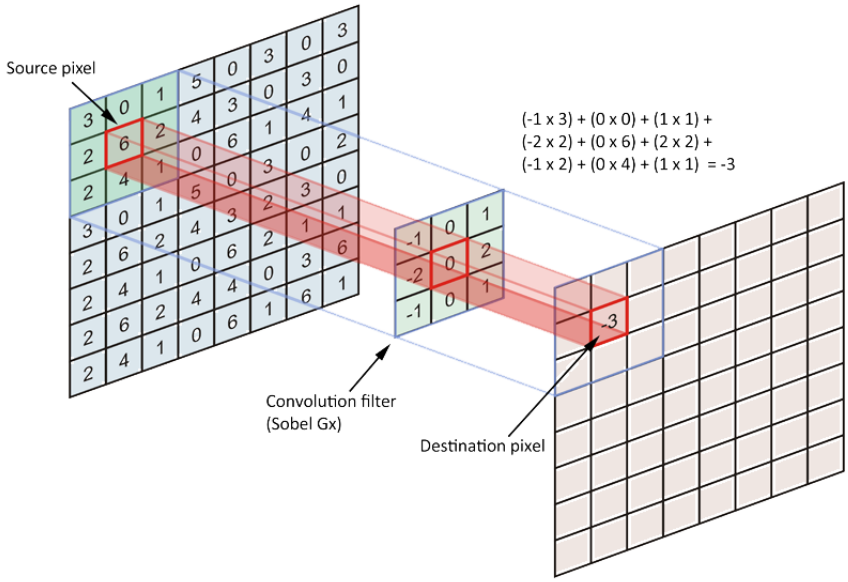

Figura 4: Ejemplo de aplicación de un filtro

La clave es que no hay un diseño previo del filtro, sino que es la red la que lo aprende. Si un filtro tiene pesos negativos en la columna izquierda y positivos en la derecha, cuando pase sobre una zona donde el color cambia de negro a blanco, la suma dará un número muy alto. Acaba de encontrar un borde vertical. Veamos un ejemplo para entender esto último. Tenemos un filtro de 3×3. Vamos a darle esos valores que mencioné: negativos a la izquierda y positivos a la derecha.

$$\text{Filtro} = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

Imagina que el filtro pasa por una zona de la imagen donde todos los píxeles son blancos (valor 255). Multiplicamos cada píxel por su peso y observamos que la suma total es $-765 + 0 + 765 = \mathbf{0}$. Este valor se intepreta como una ausencia de cambios. No hay bordes.

- Columna izquierda: $255 \times (-1) + 255 \times (-1) + 255 \times (-1) = -765$
- Columna central: $255 \times 0 + 255 \times 0 + 255 \times 0 = 0$
- Columna derecha: $255 \times 1 + 255 \times 1 + 255 \times 1 = 765$

Ahora imagina que el filtro está justo encima de un borde. Los píxeles de la izquierda son negros (0) y los de la derecha son blancos (255). Hacemos la misma operación que antes y vemos que la suma

total es mucho mayor: $0+0+765 = \mathbf{765}$. Este valor tan alto lo interpretamos como que la red acaba de detectar un contraste fuerte entre la izquierda y la derecha. En el mapa de características resultante, ese píxel brillará con mucha fuerza.

$$\text{Zona de la imagen} = \begin{bmatrix} 0 & 255 & 255 \\ 0 & 255 & 255 \\ 0 & 255 & 255 \end{bmatrix}$$

- Columna izquierda (negra): $0 \times (-1) + 0 \times (-1) + 0 \times (-1) = 0$
- Columna central: $255 \times 0 + 255 \times 0 + 255 \times 0 = 0$
- Columna derecha (blanca): $255 \times 1 + 255 \times 1 + 255 \times 1 = 765$

El filtro que hemos puesto de ejemplo detecta bordes que pasan de negro a blanco. Si el filtro fuera al revés (positivos a la izquierda y negativos a la derecha), detectaría bordes que pasan de blanco a negro. Si los pesos estuvieran en las filas superiores e inferiores, detectaría bordes horizontales. Si los pesos estuvieran solo en las esquinas, detectaría diagonales. Antaño esos filtros se escribían a mano. Hoy esos filtros se inicializan al azar para que luego la red calcule los errores tras el entrenamiento y realice el *backpropagation* visto anteriormente. Poco a poco, la red ajusta esos pesos poco a poco hasta que, mágicamente, los 32 filtros se han convertido en detectores perfectos de bordes horizontales, verticales, oblicuos, etc. La segunda capa hace lo mismo, pero en lugar de multiplicar píxeles, multiplica los «brillos» (activaciones) de la primera capa. Si el Filtro de Bordes Verticales brilló y el Filtro de Bordes Horizontales también brilló en el mismo sitio, un filtro de la segunda capa diseñado para sumar ambos detectará una esquina.

Pero queda un asunto pendiente. Si tenemos 32 filtros, ¿qué impide que los 32 acaben siendo exactamente iguales si todos buscan minimizar el mismo error? La respuesta es una combinación de caos inicial, especialización y matemáticas de alta dimensión. Cuando creas la red, PyTorch no pone los filtros a cero. Si todos empezaran en cero, todos recibirían el mismo gradiente y, efectivamente, los 32 serían idénticos para siempre. En su lugar, los pesos se inicializan con valores aleatorios muy pequeños. Así cada filtro empieza con una ligera «tendencia» a buscar elementos concretos. Por ejemplo, el Filtro #1 empieza con una ligera tendencia a favor de los píxeles de arriba. El Filtro #2 empieza con una tendencia hacia los de la derecha y así sucesivamente. Como sus puntos de partida son diferentes, cada filtro empieza a "caer" por una ladera distinta de la montaña del error. Es muy difícil que dos filtros que empezaron en lugares distintos acaben convergiendo exactamente al mismo diseño. Además, cuando el sistema intenta mejorar aún más, el algoritmo de optimización (el gradiente) «se da cuenta» de que ya no gana mucho haciendo que el Filtro #2 también detecte bordes verticales. Para reducir el error que queda (el error que el Filtro #1 no puede solucionar), el Filtro #2 se ve empujado a buscar rasgos que el Filtro #1 está ignorando, como bordes horizontales o curvas.

Matemáticamente, el espacio de posibles filtros es inmenso. En un filtro de 3×3 con valores decimales, hay infinitas combinaciones. En redes muy profundas, se utilizan técnicas como la Regularización, que penaliza a la red si los pesos de los filtros son demasiado parecidos entre sí, forzándolos a ser «ortogonales» (matemáticamente independientes). Aunque en nuestra CNN básica para MNIST, el simple hecho de empezar con valores aleatorios y tener un objetivo complejo suele ser suficiente para que cada filtro encuentre su propio nicho ecológico. Aunque antes de terminar con esto hemos de decir que a veces ocurre. En redes gigantescas con miles de filtros, es común que algunos se parezcan. Esto no es necesariamente malo; a veces la red usa esa redundancia como un sistema de seguridad.

4.2. El paso de la convolución a la clasificación

Vamos a explicar brevemente como pasamos la información de las capas de convolución a las capas lineales o lo que es lo mismo, el paso del «Mapa de Características» al «Vector de Clasificación». Mientras que las capas convolucionales mantienen la estructura espacial (una imagen de 28×28 se

convierte en 32 imágenes de 13×13, etc.). Pero las capas finales (Linear) son "ciegas" a la geometría; solo aceptan una lista larga de números. Para entender bien este apartado resumimos el viaje de los datos y sus transformaciones a través de las dos capas convolucionales.

0. Partimos de una imagen original como un cuadrado de 28x28 píxeles.
1. Primera parada: Conv1 (Filtro 3x3). Como vimos, al pasar un filtro de 3x3 sin "padding" (sin añadir bordes extra), perdemos un píxel por cada lado ($28 - 2 = 26$). Ahora tenemos mapas de 26x26.
2. Segunda parada: MaxPool1 (2x2). Ya dijimos antes que el Pooling es un resumidor. Toma bloques de 2x2 y los convierte en 1 solo píxel (el más brillante). Por tanto, divide el tamaño por 2 ($26/2 = 13$. Ahora tendremos mapas son de 13x13.
3. Tercera parada: Conv2 (Filtro 3x3). Volvemos a pasar un filtro de 3x3 sobre esos mapas de 13x13. Volvemos a perder un píxel por cada borde ($13 - 2 = 11$). Ahora tenemos mapas de 11x11.
4. Cuarta parada: MaxPool2 (2x2). Volvemos a dividir por 2. Como 11 es impar, PyTorch por defecto redondea hacia abajo ($11/2 = 5{,}5 \rightarrow 5$). En este punto por tanto, los mapas de características ahora miden 5x5.

Después de todo ese proceso, lo que le queda a la red es una versión minúscula de la imagen original. Pero no es una imagen borrosa; es una destilación de conceptos. Un píxel arriba a la izquierda podría significar: «Aquí hay una curva cerrada». Un píxel en el centro podría significar: »Aquí hay un cruce de líneas». Y como en la última capa convolucional pedimos 64 filtros, lo que tenemos al final es 64 de esos cuadritos de 5x5. Para poder entrar en la capa Linear, necesitamos una fila de neuronas. Así que estiramos (Flatten) todos esos cuadritos y para ello usamos la instrucción: `x = x.view(-1, 64 * 5 * 5)`. Con esta instrucción lo que estamos haciendo es tomar ese «ladrillo» de datos de la última capa de convolución y estirarlo. Donde 64 es el número de filtros (conceptos complejos detectados) y 5x5 es el tamaño de la imagen resumida tras pasar por las convoluciones y los poolings.

Si alguna vez cambias el tamaño del filtro (por ejemplo a 5x5) o añades más capas, ese 5*5*64 cambiará. Si no actualizas el primer número de tu `nn.Linear`, el programa fallará porque la puerta de entrada a la capa final no coincidirá con la cantidad de datos que vienen de las convoluciones.

4.3. El fenómeno del dropout y conclusiones

El Dropout es una de las técnicas más curiosas y efectivas en el Deep Learning. Si las convoluciones son el "cerebro" de la red, el Dropout es su entrenamiento militar. Imagina que tienes un equipo de 10 especialistas trabajando en un proyecto. Si uno de ellos es muy dominante, el resto se vuelve perezoso y deja de aprender, confiando siempre en lo que diga el líder. Si el líder se equivoca, todo el equipo falla. El Dropout consiste en apagar o poner a cero un porcentaje aleatorio de neuronas en cada paso del entrenamiento. Esto obliga a la red a no depender de una sola neurona o de un solo filtro para reconocer un número. Esto provoca que la red desarrolle redundancia. Si apagas la neurona que detecta el «palito vertical del 4», la red se ve forzada a aprender otras formas de identificar el 4. Esto evita el *Overfitting* (que la red se memorice los datos de entrenamiento pero falle con imágenes nuevas).

Tras el entrenamiento y la prueba el colab de Google, el código mostrado en `src/mnist-cnn.py` ha tardado 5 minutos en ejecutarse y ha mostrado los siguientes errores medios para cada época:

```
Epoca 1 - Error medio: 0.2344
Epoca 2 - Error medio: 0.0833
Epoca 3 - Error medio: 0.0632
Epoca 4 - Error medio: 0.0524
Epoca 5 - Error medio: 0.0458
```

Frente a la antigua red MNSIT que no usaba CNN se ve de forma asombrosa que con muchísimas menos épocas de entrenamiento, 5 en este caso frente a las 5000 anteriores llegamos a un error muy bajo igualmente. Mientras que la red lineal sufría para llegar al 90 %, una CNN básica como la nuestra superará el 98 % o 99 % casi sin esfuerzo, porque «entiende» la estructura del número. Cuando la gente habla de Deep Learning, se refiere precisamente a esto. **Al ir añadiendo capas, estamos permitiendo que la red cree conceptos cada vez más abstractos**.

4.4. Prácticas propuestas

1. Desplegar el modelo de MNIST con CNN en una pequeña interfaz web para que puedas dibujar un número con el ratón y que tu red lo adivine en tiempo real.

5. Bibliografía recomendada

- Deep Learning. Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Es el libro de referencia absoluto. Ian Goodfellow es el inventor de las GANs. Los capítulos sobre MLP (Multilayer Perceptrons) y Optimización. Te dará el rigor matemático detrás de por qué la Cross-Entropy y Adam funcionan tan bien.* (https://www.deeplearningbook.org/).
- Neural Networks and Deep Learning. Michael Nielsen. *Un enfoque muy visual. Es un libro online interactivo que explica el Backpropagation mejor que nadie en el mundo. El capítulo 2 es una obra maestra sobre la matemática del error. Si quieres ver diagramas claros de cómo fluyen los pesos, este es tu sitio.* (http://neuralnetworksanddeeplearning.com/).
- Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. Aurélien Géron. *Es el manual de ingeniería más vendido. Aunque el título menciona TensorFlow, los conceptos de arquitectura y pre-procesamiento de datos (como el MNIST) son universales. Fundamentales, las secciones sobre Regulación, Batch Normalization y cómo evitar el Overfitting.*
- Deep Learning with PyTorch. Eli Stevens, Luca Antiga y Thomas Viehmann.