Trabajo Práctico 1

1. Entrenamiento de una red de Hopfield (1982)

Carga y conversión de imágenes a formato binario

Se define un conjunto de funciones para cargar las imágenes `.bmp` desde una carpeta y convertirlas a matrices binarias de `0` y `1`, donde `1` representa píxeles blancos y `0` píxeles negros. Estas matrices representan los patrones que serán almacenados en la red de Hopfield.

In [251…

```
from PIL import Image
import os
import numpy as np

def cargar_imagen_en_binario(ruta, invert=False):
    """
    Abre una imagen ya binaria (0 o 255), la convierte a {0,1}.
    - invert: True para 1=negro, False para 1=blanco
    Devuelve list[list[int]] para compatibilidad con tu pipeline actual.
    """
    img = Image.open(ruta).convert("L")  # escala de grises
    arr = np.array(img, dtype=np.uint8)  # (h, w) 0 o 255

    binaria = (arr > 0).astype(np.uint8)  # 1 si 255, 0 si 0
    if invert:
        binaria = 1 - binaria

    h, w = binaria.shape
    print(f"{os.path.basename(ruta)} - tamaño: {w}x{h}")
    return binaria.tolist()

def cargar_patrones_desde_carpeta(carpeta, invert=False):
    """
    Carga .bmp como patrones binarios {0,1}. Mantiene mismos prints y sal
    """
    archivos_bmp = sorted([f for f in os.listdir(carpeta) if f.lower().en
    patrones = []
    for archivo in archivos_bmp:
        ruta_completa = os.path.join(carpeta, archivo)
        patron = cargar_imagen_en_binario(ruta_completa, invert=invert)
        patrones.append(patron)

    if patrones:
        print(f"Se cargaron {len(patrones)} patrones de {len(patrones[0])
    else:
        print("No se encontraron archivos .bmp en la carpeta.")
    return patrones
```

Vectorización y normalización de patrones

En esta etapa se toma cada imagen cargada como una matriz binaria (valores `0` y `1`) y se convierte en un vector unidimensional con valores `-1` y `1`. Esta transformación es necesaria porque la red de Hopfield representa cada patrón como un vector de activaciones, donde los valores deben estar centrados en torno al cero para cumplir con los fundamentos del modelo.

Primero, cada valor `0` se transforma en `-1`, y cada `1` se mantiene. Luego, la matriz bidimensional de la imagen se aplana en un solo vector, lo que permite operar con productos exteriores y aplicar dinámicas de red en forma vectorial.

Este paso asegura que los patrones estén en el formato correcto para entrenar la red y realizar recuperaciones.

In [252…

```
import numpy as np

import numpy as np

def centrar_y_vectorizar_patrones(patrones, invert=False):
    """
    Convierte patrones binarios {0,1} (P,h,w) en vectores {-1,+1} (P,N),
    - invert=True: 1 <-> 0 antes de mapear.
    """
    arr = np.asarray(patrones)
    if arr.size == 0:
        print("Tengo 0 patrones vectorizados de 0 elementos cada uno.")
        return arr.reshape(0, 0).astype(np.int8)

    if arr.ndim == 2:  # un solo patrón
        arr = arr[None, ...]

    if invert:
        arr = 1 - arr

    X = arr.reshape(arr.shape[0], -1)  # (P, N)
    Xpm1 = (X * 2 - 1).astype(np.int8)  # {0,1} -> {-1,+1}

    print(f"Tengo {Xpm1.shape[0]} patrones vectorizados de {Xpm1.shape[1]
    return Xpm1


carpeta_imagenes = "imagenes/50x50"
patrones = cargar_patrones_desde_carpeta(carpeta_imagenes)
patrones_originales = patrones  # si te devuelve listas, no pasa nada
patrones_vectorizados = centrar_y_vectorizar_patrones(patrones)  # -> np.

print("Tengo " + str(len(patrones_vectorizados)) +
      " patrones vectorizados de " + str(len(patrones_vectorizados[0])) +
      " elementos cada uno.")

print(patrones_vectorizados)
```

```
panda.bmp - tamaño: 50x50
perro.bmp - tamaño: 50x50
v.bmp - tamaño: 50x50
Se cargaron 3 patrones de 50 filas y 50 columnas cada uno.
Tengo 3 patrones vectorizados de 2500 elementos cada uno.
Tengo 3 patrones vectorizados de 2500 elementos cada uno.
[[ 1  1  1 ...  1  1  1]
 [-1 -1 -1 ... -1 -1 -1]
 [-1 -1 -1 ... -1 -1 -1]]
```

Inicialización de la matriz de pesos

Antes de entrenar la red de Hopfield, se debe crear la matriz de pesos sinápticos `W`, que define las conexiones entre neuronas. Esta matriz es cuadrada y tiene dimensiones `N × N`, donde `N` es la cantidad total de píxeles (o neuronas) por patrón.

Inicialmente, todos los pesos se establecen en cero, y luego se actualizarán mediante la regla de Hebb durante el entrenamiento. Esta matriz almacena el conocimiento aprendido por la red.

In [253…

```
def inicializar_matriz_pesos(dimension, dtype=np.float32):
    """
    Inicializa una matriz cuadrada de pesos de tamaño NxN con ceros (NumP
    """
    return np.zeros((dimension, dimension), dtype=dtype)

ancho = 50
alto = 50
dimension = ancho * alto  # 2500

pesos = inicializar_matriz_pesos(dimension)
print(f"mis pesos tienen la dimensión: {len(pesos)}x{len(pesos[0])}")
```

```
mis pesos tienen la dimensión: 2500x2500
```

Entrenamiento de la red de Hopfield

La red de Hopfield se entrena utilizando la **regla de Hebb**, una regla de aprendizaje no supervisado que refuerza las conexiones entre neuronas que se activan simultáneamente. En esta implementación, se recorren todos los patrones vectorizados y se calcula la **suma de los productos exteriores** de cada patrón consigo mismo.

El resultado es una matriz de pesos sinápticos que almacena la información de los patrones aprendidos. Finalmente, se eliminan las autoconexiones (valores en la diagonal) ya que una neurona no debe influenciarse a sí misma.

Esta matriz será utilizada para recuperar patrones a partir de entradas ruidosas o incompletas.

In [254…

```
import numpy as np

def entrenar_red_hopfield(patrones_vectorizados, norm='N'):
    """
    Entrena Hopfield con patrones en {-1, +1} (acepta también {0,1} y los
```

```
    Retorna W (N, N) float32, diagonal = 0.

    norm: 'N' -> divide por N (clásico Hopfield)
          'P' -> divide por P (tu versión previa)
    """
    X = np.asarray(patrones_vectorizados)
    if X.ndim == 1:
        X = X[None, :]

    # Asegurar valores en {-1,+1}
    u = np.unique(X)
    uset = set(u.tolist())
    if uset.issubset({0, 1}):
        X = (X.astype(np.int16) * 2 - 1).astype(np.int8)
    elif not uset.issubset({-1, 1}):
        # fallback: >0 -> 1, else -1
        X = np.where(X > 0, 1, -1).astype(np.int8)

    X = X.astype(np.float32, copy=False)
    P, N = X.shape

    W = X.T @ X
    if norm == 'N':
        W /= float(N)
    elif norm == 'P':
        W /= float(P)
    else:
        raise ValueError("norm debe ser 'N' o 'P'.")

    np.fill_diagonal(W, 0.0)
    return W

# Uso igual que antes
pesos = entrenar_red_hopfield(patrones_vectorizados, norm='N')  # o 'P' p
print(f"Matriz de pesos entrenada con tamaño: {len(pesos)}x{len(pesos[0])}
```

```
Matriz de pesos entrenada con tamaño: 2500x2500
```

In [255…

```
def recuperar_patron(patron_inicial, pesos, max_iter=100, rng=None):
    """
    Recupera un patrón aplicando actualización asíncrona hasta converger.

    Parámetros
    ----------
    patron_inicial : array-like (N,)
        Estado inicial en {-1, +1}.
    pesos : np.ndarray (N, N)
        Matriz de pesos (float), idealmente con diagonal en 0.
    max_iter : int
        Máximo de barridos asíncronos completos.
    rng : np.random.Generator | None
        Generador para reproducibilidad. Si None, se crea uno por defecto

    Retorna
    -------
    np.ndarray (N,) dtype=int8 en {-1, +1}
    """
    W = np.asarray(pesos)
    estado = np.asarray(patron_inicial, dtype=np.int8).copy()
    N = estado.size
```

```
    if rng is None:
        rng = np.random.default_rng()

    for _ in range(max_iter):
        cambios = 0
        for i in rng.permutation(N):              # orden aleatorio en ca
            h = W[i].dot(estado)                  # campo local
            nuevo = 1 if h >= 0 else -1
            if estado[i] != nuevo:
                estado[i] = nuevo
                cambios += 1
        if cambios == 0:                          # convergió
            break

    return estado
```

In [256…

```
import numpy as np
import matplotlib.pyplot as plt

def mostrar_comparacion_patron(original, recuperado, ancho, alto, indice=
    ori = np.asarray(original).reshape(alto, ancho)
    rec = np.asarray(recuperado).reshape(alto, ancho)

    # Si están en {-1,+1}, llevamos a {0,1} para visualizar
    if ori.min() < 0 or rec.min() < 0:
        ori_show = (ori + 1) / 2.0
        rec_show = (rec + 1) / 2.0
    else:
        ori_show, rec_show = ori, rec

    fig, axs = plt.subplots(1, 2, figsize=(6, 3))
    axs[0].imshow(ori_show, cmap='gray', vmin=0, vmax=1)
    axs[0].set_title(f"Original")
    axs[0].axis('off')

    axs[1].imshow(rec_show, cmap='gray', vmin=0, vmax=1)
    axs[1].set_title(f"Recuperado")
    axs[1].axis('off')

    plt.suptitle(f"Comparación patrón {indice}")
    plt.tight_layout()
    plt.show()
```

Evaluación de recuperación sin ruido

En este paso se verifica si la red de Hopfield es capaz de recuperar correctamente los patrones originales que fueron utilizados durante el entrenamiento. Para ello, cada patrón se usa como entrada inicial y se aplica la dinámica de la red hasta que converge a un estado estable.

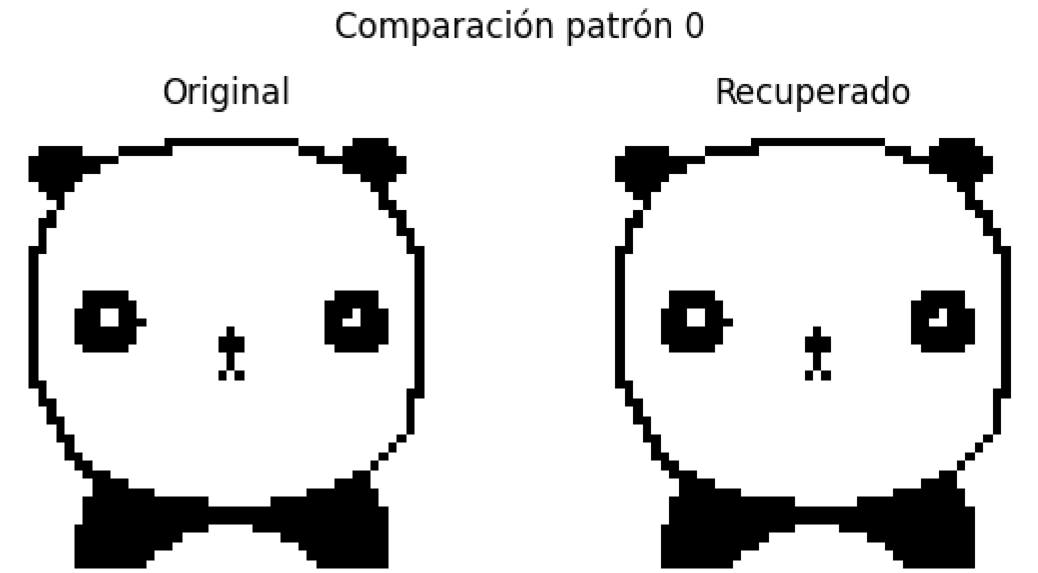
Si el estado final es idéntico al patrón original, se considera que la recuperación fue exitosa. Esto demuestra que la red ha almacenado correctamente los patrones en su memoria asociativa.

In [257…

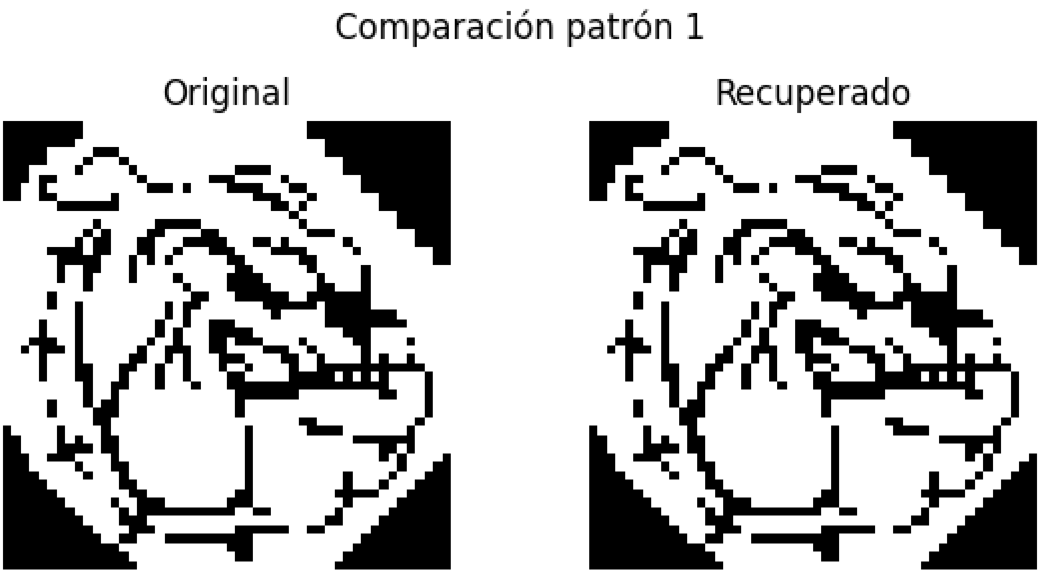
```
for i in range(len(patrones_vectorizados)):
```

```
    patron_original = patrones_vectorizados[i];
    # Recuperación asíncrona (usa tu versión NumPy de recuperar_patron)
    estado_convergido = recuperar_patron(patron_original, pesos, max_iter

    # Comparación correcta para arrays
    if np.array_equal(estado_convergido, patron_original):
        print(f"El patrón {i} fue recuperado correctamente.")
    else:
        print(f"El patrón {i} NO se recuperó correctamente.")

    # Visualización (la función ya soporta arrays)
    mostrar_comparacion_patron(patron_original, estado_convergido, ancho=
```

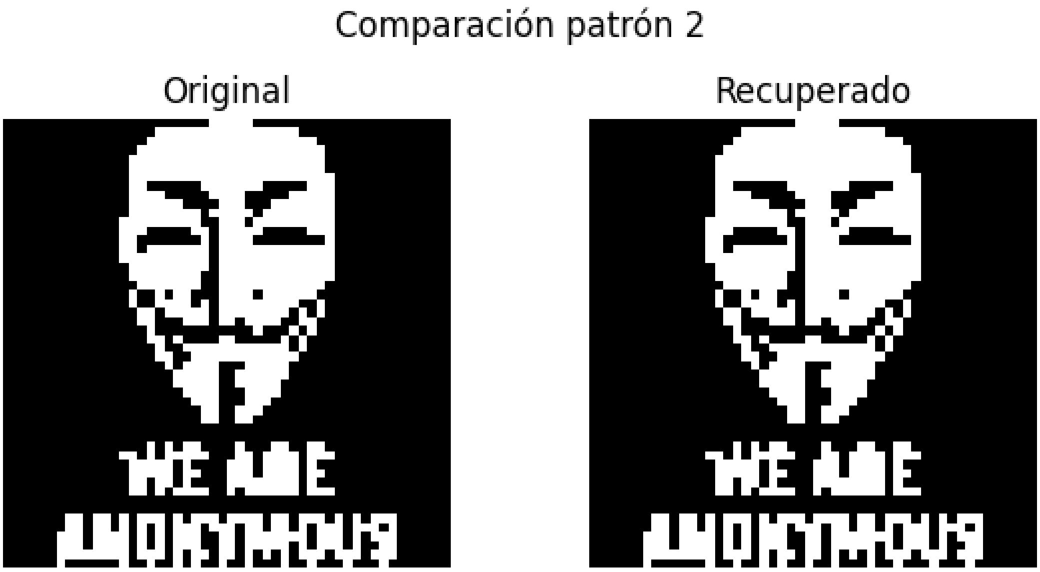
```
El patrón 0 fue recuperado correctamente.
```




```
El patrón 1 fue recuperado correctamente.
```




```
El patrón 2 fue recuperado correctamente.
```



Prueba con imágenes de 60x45

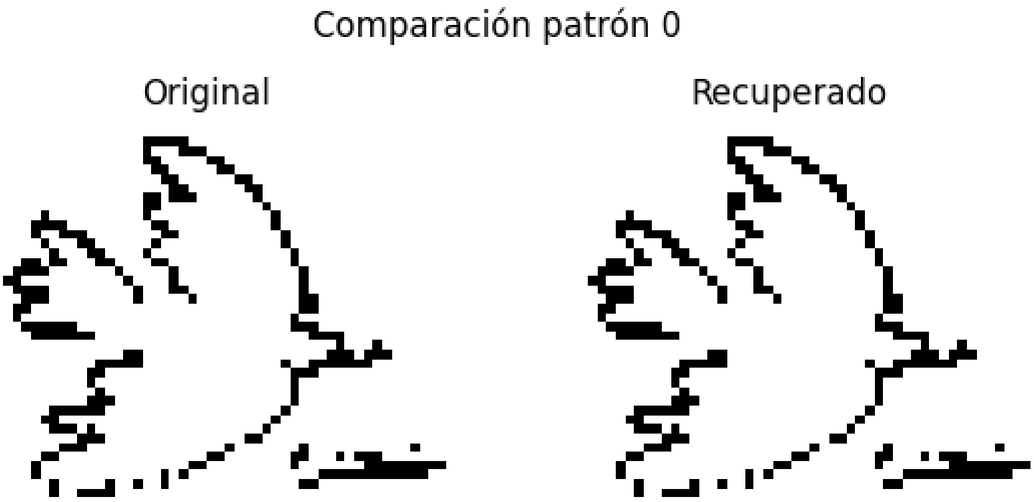
Se repite el procedimiento anterior utilizando un nuevo conjunto de imágenes de tamaño `60x45`. Se cargan, vectorizan y entrenan en la red de Hopfield. Luego, se verifica si la red es capaz de recuperar correctamente los patrones a partir de sí mismos.

Cada comparación muestra el patrón original y su versión recuperada tras aplicar la dinámica de la red. Esto permite verificar que el modelo puede escalar y seguir funcionando correctamente con imágenes más grandes.

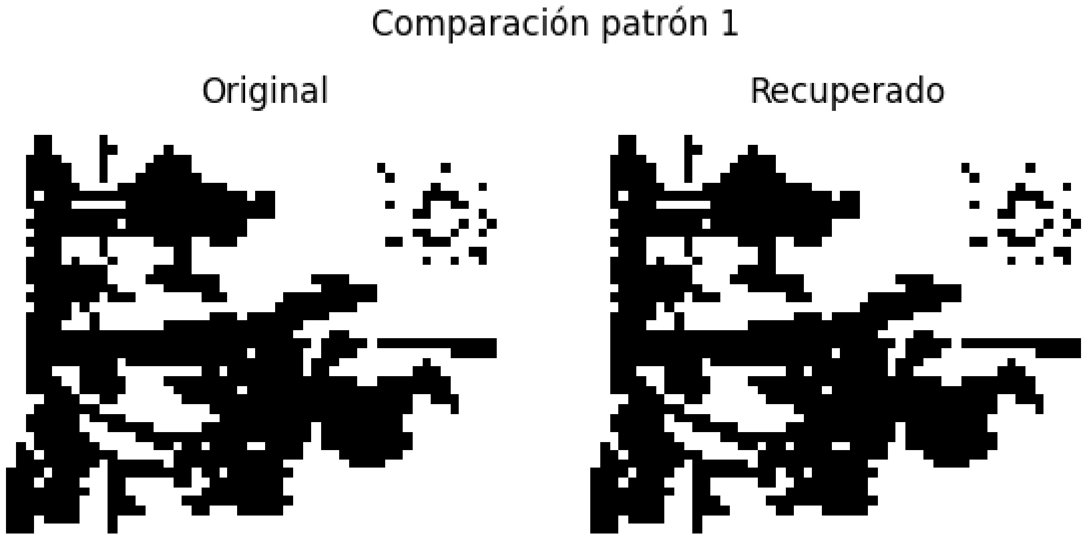
In [258…

```
carpeta_imagenes = "imagenes/60x45"
patrones = cargar_patrones_desde_carpeta(carpeta_imagenes)  # puede devol
patrones_vectorizados_imagenes_grandes = centrar_y_vectorizar_patrones(pa

ancho = 60
alto = 45
dimension = ancho * alto  # 2700

# Entrenamiento (no hace falta inicializar ceros antes)
pesos_imagenes_grandes = entrenar_red_hopfield(patrones_vectorizados_imag

for i in range(patrones_vectorizados_imagenes_grandes.shape[0]):
    patron_original = patrones_vectorizados_imagenes_grandes[i]
    estado_convergido = recuperar_patron(patron_original, pesos_imagenes_

    if np.array_equal(estado_convergido, patron_original):
        print(f"El patrón {i} fue recuperado correctamente.")
    else:
        print(f"El patrón {i} NO se recuperó correctamente.")

    mostrar_comparacion_patron(patron_original, estado_convergido, ancho=
```

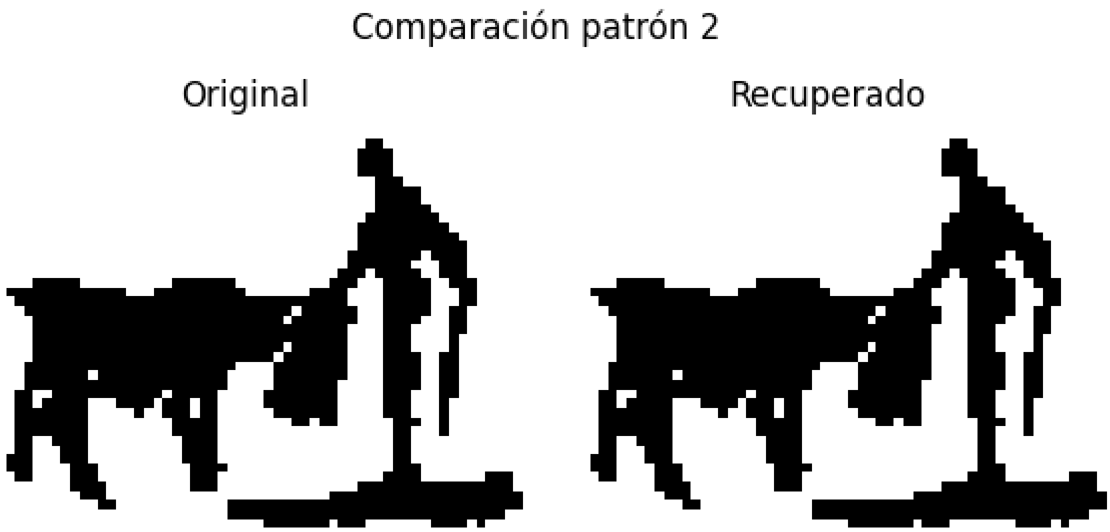
```
paloma.bmp - tamaño: 60x45
quijote.bmp - tamaño: 60x45
torero.bmp - tamaño: 60x45
Se cargaron 3 patrones de 45 filas y 60 columnas cada uno.
Tengo 3 patrones vectorizados de 2700 elementos cada uno.
El patrón 0 fue recuperado correctamente.
```




```
El patrón 1 fue recuperado correctamente.
```




```
El patrón 2 fue recuperado correctamente.
```



Conclusión: entrenamiento con 3 imágenes

Se entrenó la red de Hopfield con un conjunto reducido de 3 imágenes binarias. La red logró recuperar todos los patrones correctamente, lo que demuestra que, con un número limitado de patrones bien diferenciados, el modelo puede almacenar y recordar eficazmente la información sin interferencia.

Agregado de ruido a los patrones

Para evaluar la robustez de la red de Hopfield, se introduce ruido artificial en los patrones antes de presentarlos a la red. La función `agregar_ruido` invierte aleatoriamente un porcentaje de los bits del patrón original (valores `-1` o `1`).

Este proceso simula entradas incompletas o corruptas, y permite comprobar si la red es capaz de recuperar el patrón correcto a pesar del ruido. La proporción de bits alterados se controla mediante el parámetro `porcentaje_ruido`, que puede variar entre `0.0` (sin ruido) y `1.0` (ruido total).

In [259…

```
def agregar_ruido(patron, porcentaje_ruido, rng=None):
    """
    Invierte exactamente floor(n * porcentaje_ruido) bits de un patrón en
    - patron: array-like (N,) o (h,w)
    - porcentaje_ruido: 0..1
    - rng: np.random.Generator opcional para reproducibilidad
    Devuelve: np.ndarray misma forma y dtype=int8
    """
    x = np.asarray(patron)
    shape = x.shape
    x = x.astype(np.int8, copy=True).ravel()       # copia para no modifi

    n = x.size
    k = int(n * float(porcentaje_ruido))
    if k <= 0:
        return x.reshape(shape)

    rng = rng or np.random.default_rng()
    idx = rng.choice(n, size=k, replace=False)     # elige k posiciones ú
    x[idx] *= -1                                   # invierte en bloque (
    return x.reshape(shape)
```

In [260…

```
def contar_diferencias(p1, p2):
    """
    Cuenta cuántos bits difieren entre dos patrones del mismo tamaño.
    Acepta listas o ndarrays; soporta 1D o 2D (p.ej. h×w).
    """
    a = np.asarray(p1)
    b = np.asarray(p2)
    if a.shape != b.shape:
        raise ValueError(f"Formas distintas: {a.shape} vs {b.shape}")
    return int(np.count_nonzero(a != b))
```

Evaluación de la tolerancia al ruido

En esta sección se analiza la capacidad de la red de Hopfield para recuperar correctamente los patrones originales a medida que se introduce ruido. Para ello, se prueba con distintos niveles de ruido (de 0% a 100%) invirtiendo aleatoriamente un porcentaje de los bits en cada patrón.

Se calcula el error promedio de recuperación (proporción de bits incorrectos) para cada nivel de ruido. Finalmente, se grafica la precisión de la red como función del ruido, lo cual permite visualizar su comportamiento frente a distorsiones crecientes.

Un buen modelo debería mantener alta precisión para bajos niveles de ruido, degradándose progresivamente a medida que la distorsión aumenta.

In [261…

```
import numpy as np
import matplotlib.pyplot as plt

def evaluar_robustez_ruido(patrones_vectorizados, pesos, niveles_ruido=No
                           max_iter=10000, modo_ruido='exact', rng=None):
    """
    Evalúa el desempeño de la red de Hopfield ante distintos niveles de r

    Parámetros
    ----------
    patrones_vectorizados : (P,N) array-like en {-1,+1} (o convertible)
    pesos : (N,N) np.ndarray
    niveles_ruido : iterable de floats en [0,1]; por defecto [0.0, 0.1, .
    max_iter : int, iteraciones para la convergencia (recuperación asíncr
    modo_ruido : 'exact' (invierte exactamente floor(p*N) bits por patrón
                 'bernoulli' (cada bit se invierte con probabilidad p; má
    rng : np.random.Generator para reproducibilidad

    Retorna
    -------
    niveles_ruido (list[float]), errores_promedio (list[float])
    """
    if niveles_ruido is None:
        niveles_ruido = [i / 10 for i in range(11)]  # 0.0 ... 1.0

    X = np.asarray(patrones_vectorizados)
    if X.ndim == 1:
        X = X[None, :]
    # Asegurar formato {-1,+1}
    U = set(np.unique(X).tolist())
    if U.issubset({0, 1}):
        X = (X.astype(np.int16)*2 - 1).astype(np.int8)
    elif not U.issubset({-1, 1}):
        X = np.where(X > 0, 1, -1).astype(np.int8)

    P, N = X.shape
    errores_promedio = []
    rng = rng or np.random.default_rng()

    for ruido in niveles_ruido:
        total_errores = 0

        if modo_ruido == 'bernoulli':
            # Aplico ruido por máscara para todos los patrones a la vez (
            mask = rng.random(size=(P, N)) < float(ruido)
            X_ruidoso = X.copy()
            X_ruidoso[mask] *= -1

            # Recupero patrón por patrón (la recuperación asíncrona es po
            for p in range(P):
                estado_convergido = recuperar_patron(X_ruidoso[p], pesos,
```

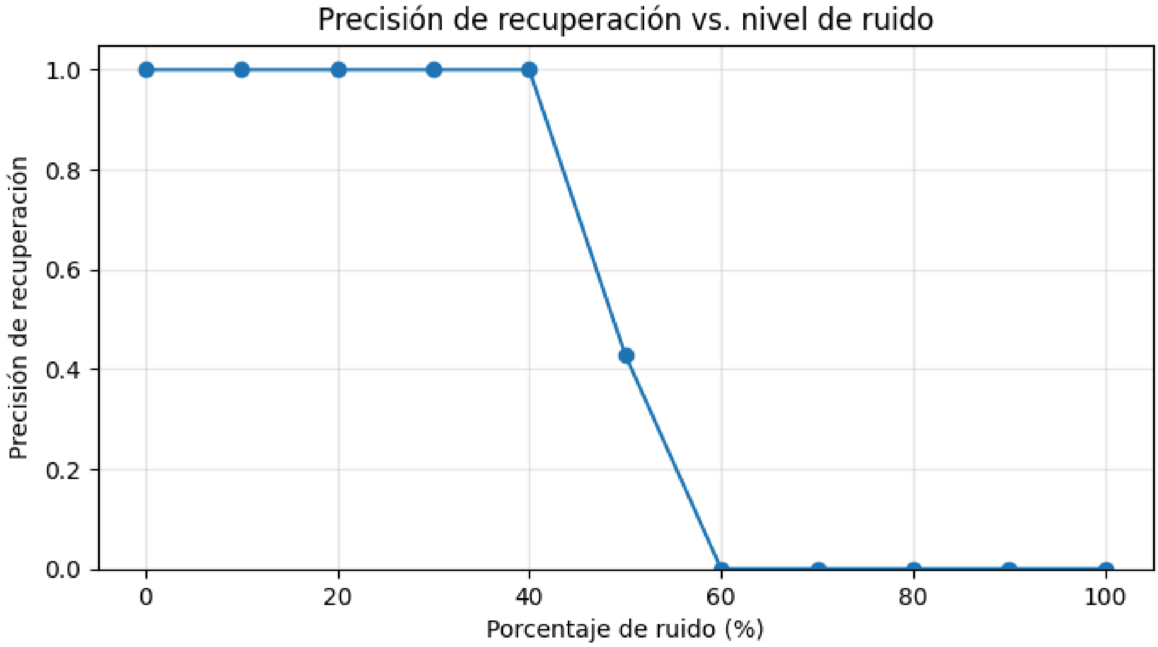
```
                total_errores += int(np.count_nonzero(estado_convergido !

        else:  # 'exact' — mismo comportamiento que tu agregar_ruido (exa
            for p in range(P):
                x_noisy = agregar_ruido(X[p], ruido, rng=rng)  # usa tu v
                estado_convergido = recuperar_patron(x_noisy, pesos, max_
                total_errores += int(np.count_nonzero(estado_convergido !

        promedio = total_errores / (P * N)
        errores_promedio.append(promedio)
        print(f"Ruido {int(ruido*100)}% → Error promedio: {promedio:.4f}"

    return niveles_ruido, errores_promedio


def graficar_precision_vs_ruido(niveles_ruido, errores_promedio):
    """
    Grafica la precisión de recuperación en función del nivel de ruido.
    precisión = 1 - error_promedio
    """
    niveles_ruido = list(niveles_ruido)
    errores_promedio = list(errores_promedio)
    precisiones = [1 - e for e in errores_promedio]

    plt.figure(figsize=(8, 4))
    plt.plot([r * 100 for r in niveles_ruido], precisiones, marker='o', l
    plt.title("Precisión de recuperación vs. nivel de ruido")
    plt.xlabel("Porcentaje de ruido (%)")
    plt.ylabel("Precisión de recuperación")
    plt.ylim(0, 1.05)
    plt.grid(True, alpha=0.3)
    plt.show()
```

Imaganes 50x50

In [262…

```
niveles, errores = evaluar_robustez_ruido(patrones_vectorizados, pesos)
graficar_precision_vs_ruido(niveles, errores)
```

```
Ruido 0% → Error promedio: 0.0000
Ruido 10% → Error promedio: 0.0000
Ruido 20% → Error promedio: 0.0000
Ruido 30% → Error promedio: 0.0000
Ruido 40% → Error promedio: 0.0000
Ruido 50% → Error promedio: 0.5712
Ruido 60% → Error promedio: 1.0000
Ruido 70% → Error promedio: 1.0000
Ruido 80% → Error promedio: 1.0000
Ruido 90% → Error promedio: 1.0000
Ruido 100% → Error promedio: 1.0000
```

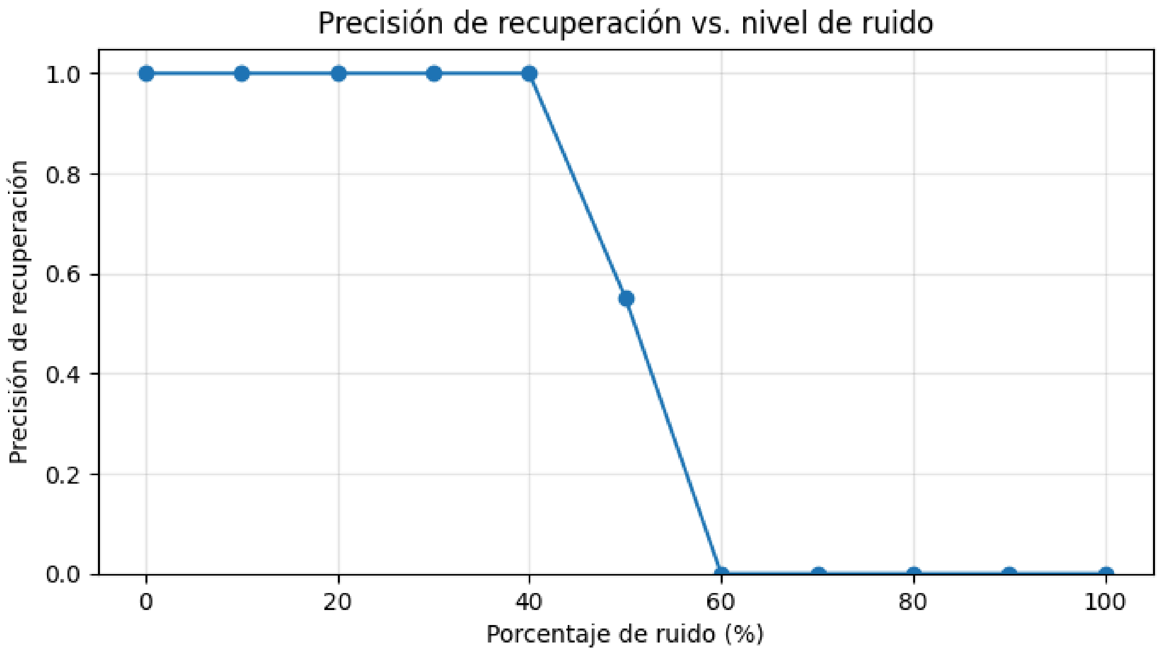


Imagenes 60x45

In [263…

```
niveles, errores = evaluar_robustez_ruido(patrones_vectorizados_imagenes_
graficar_precision_vs_ruido(niveles, errores)
```

```
Ruido 0% → Error promedio: 0.0000
Ruido 10% → Error promedio: 0.0000
Ruido 20% → Error promedio: 0.0000
Ruido 30% → Error promedio: 0.0000
Ruido 40% → Error promedio: 0.0000
Ruido 50% → Error promedio: 0.4478
Ruido 60% → Error promedio: 1.0000
Ruido 70% → Error promedio: 1.0000
Ruido 80% → Error promedio: 1.0000
Ruido 90% → Error promedio: 1.0000
Ruido 100% → Error promedio: 1.0000
```



Punto 3

Evaluación de estados espurios

En esta celda se prueba la existencia de **estados espurios** en la red de Hopfield entrenada:

- **Patrones invertidos**: se invierte el signo de todos los bits de cada patrón entrenado y se comprueba si el estado permanece estable.
- **Combinaciones impares**: se evalúa la estabilidad del estado resultante de aplicar `sign(P0 + P1 + P2)`.

Si estos estados convergen a sí mismos al ser presentados como entrada a la red, se considera que son **estados espurios estables**. Esta es una propiedad conocida de las redes de Hopfield, especialmente cuando se usan múltiples patrones y hay solapamiento entre ellos.

Por último, se evalúa la existencia de estados espurios del tipo **spin-glass**, que son mínimos locales de la energía que **no se parecen a ningún patrón almacenado** ni a combinaciones de ellos.

Para esto, se generan vectores aleatorios de activación (`-1` y `1`) que actúan como entradas completamente nuevas. Si la red converge a esos mismos estados sin haberlos aprendido, se considera que son **estados espurios del tipo spin-glass**.

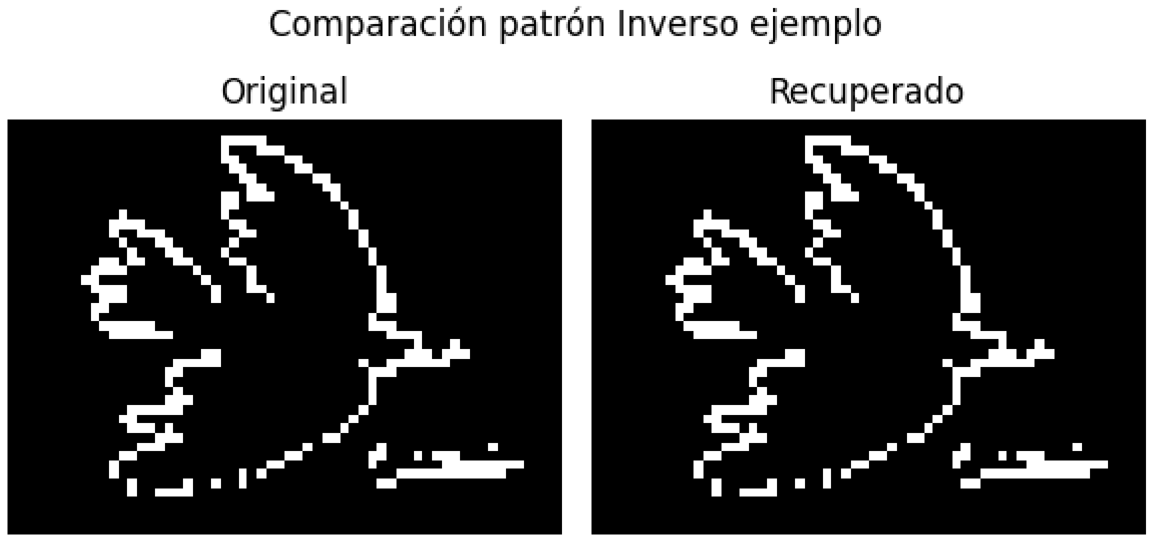
Este fenómeno se vuelve más probable a medida que se entrena la red con un mayor número de patrones, lo cual genera interferencias y reduce la capacidad efectiva de almacenamiento.

In [264…

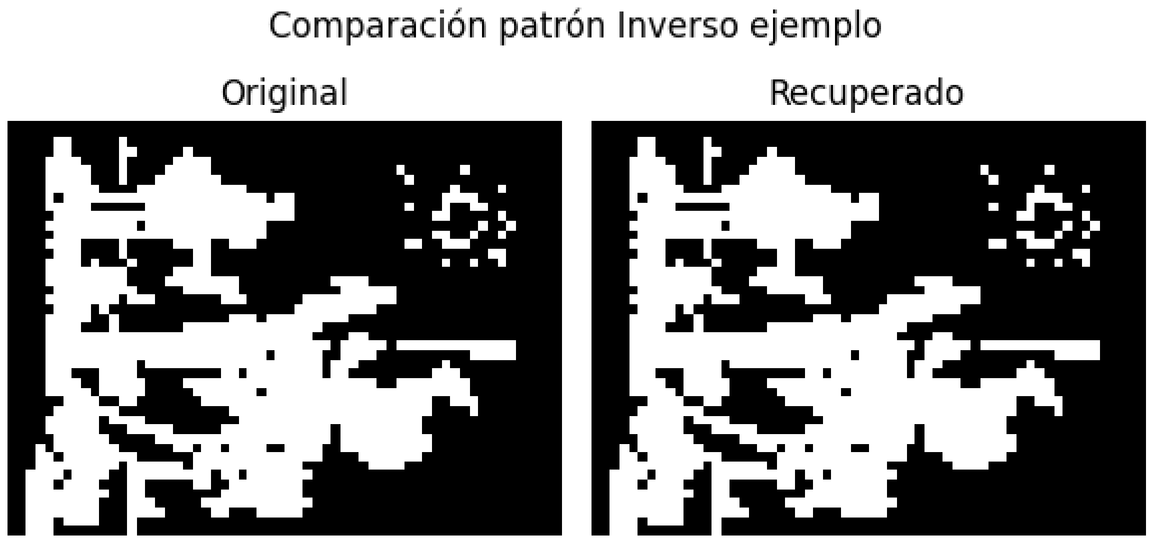
```
carpeta_imagenes = "imagenes/60x45"
patrones = cargar_patrones_desde_carpeta(carpeta_imagenes)                #
patrones_vectorizados = centrar_y_vectorizar_patrones(patrones)           #

ancho, alto = 60, 45
dimension = ancho * alto

# Entrenar la red (vectorizado)
pesos = entrenar_red_hopfield(patrones_vectorizados)                      #

# ---------- ESTADOS ESPURIOS ----------

def es_estable(estado, pesos, max_iter=10000):
    """
    Verifica si un estado es un mínimo estable (converge a sí mismo).
    """
    estado = np.asarray(estado, dtype=np.int8).ravel()
    convergido = recuperar_patron(estado, pesos, max_iter=max_iter)
    return np.array_equal(convergido, estado)

# 1) Inversos de cada patrón
print("\n--- Estados inversos ---")
```

```python
for i in range(patrones_vectorizados.shape[0]):
    patron = patrones_vectorizados[i]
    inverso = (-patron).astype(np.int8, copy=False)
    if es_estable(inverso, pesos):
        print(f"Inverso del patrón {i} es un estado espurio estable.")
    else:
        print(f"Inverso del patrón {i} NO es estable.")
    rec_inv = recuperar_patron(inverso, pesos, max_iter=10000)
    mostrar_comparacion_patron(inverso, rec_inv, ancho, alto, indice="Inv

# 2) Combinación impar de 3 patrones: TODAS las ± (sign(P0 ± P1 ± P2))
print("\n--- Combinación impar de 3 patrones: todas las ± ---")

if patrones_vectorizados.shape[0] >= 3:
    # Elegí los 3 que quieras (acá uso 0,1,2)
    i0, i1, i2 = 0, 1, 2
    P0 = patrones_vectorizados[i0]
    P1 = patrones_vectorizados[i1]
    P2 = patrones_vectorizados[i2]

    # Todas las combinaciones de signos (orden: +++, ++-, +-+, +--, -++,
    S = np.array([
        [ +1, +1, +1],
        [ +1, +1, -1],
        [ +1, -1, +1],
        [ +1, -1, -1],
        [ -1, +1, +1],
        [ -1, +1, -1],
        [ -1, -1, +1],
        [ -1, -1, -1],
    ], dtype=np.int8)

    def etiqueta_signos(row):
        return " ".join([
            ("+P0" if row[0] == 1 else "-P0"),
            ("+P1" if row[1] == 1 else "-P1"),
            ("+P2" if row[2] == 1 else "-P2"),
        ])

    # Apilar y combinar: comb = sign(S @ [P0;P1;P2])
    P_stack = np.stack([P0, P1, P2], axis=0).astype(np.int16)   # (3, N)
    comb_sum = S @ P_stack                                       # (8, N)
    combos = np.where(comb_sum >= 0, 1, -1).astype(np.int8)     # regla:

    # Evaluar estabilidad y MOSTRAR TODAS las combinaciones
    for row, estado in zip(S, combos):
        rec = recuperar_patron(estado, pesos, max_iter=10000)
        estable = np.array_equal(rec, estado)
        print(f"Combinación {etiqueta_signos(row)}: {'ESPURIO' if estable
        # Mostrar lado a lado (init vs recuperado)
        mostrar_comparacion_patron(
            estado, rec, ancho, alto,
            indice=f"{etiqueta_signos(row)} — {'ESPURIO' if estable else
        )
else:
    print("No hay suficientes patrones para probar combinación impar.")
```

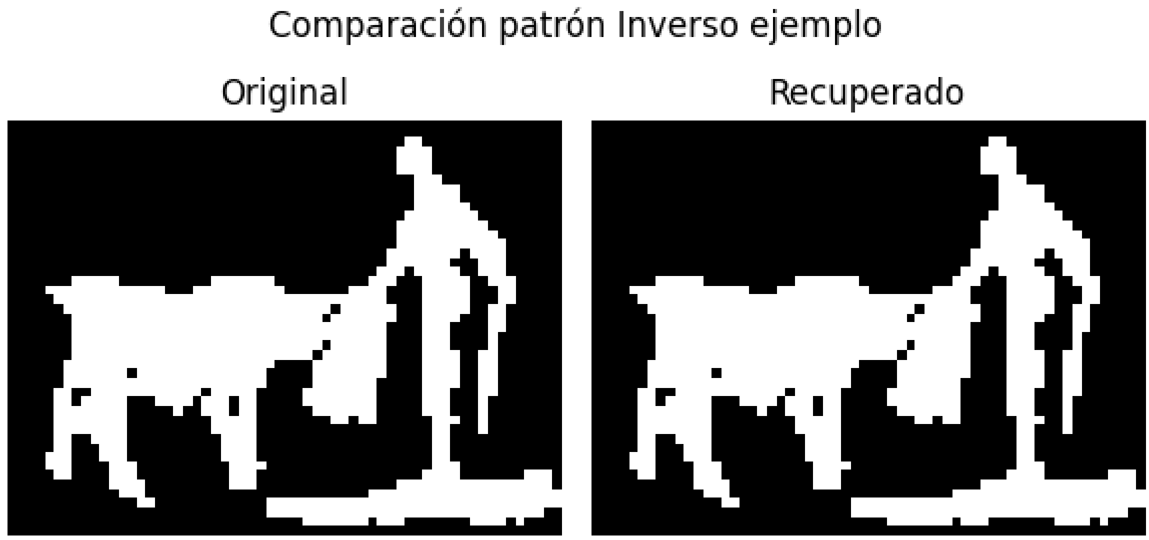
```
paloma.bmp - tamaño: 60x45
quijote.bmp - tamaño: 60x45
torero.bmp - tamaño: 60x45
Se cargaron 3 patrones de 45 filas y 60 columnas cada uno.
Tengo 3 patrones vectorizados de 2700 elementos cada uno.

--- Estados inversos ---
Inverso del patrón 0 es un estado espurio estable.
```



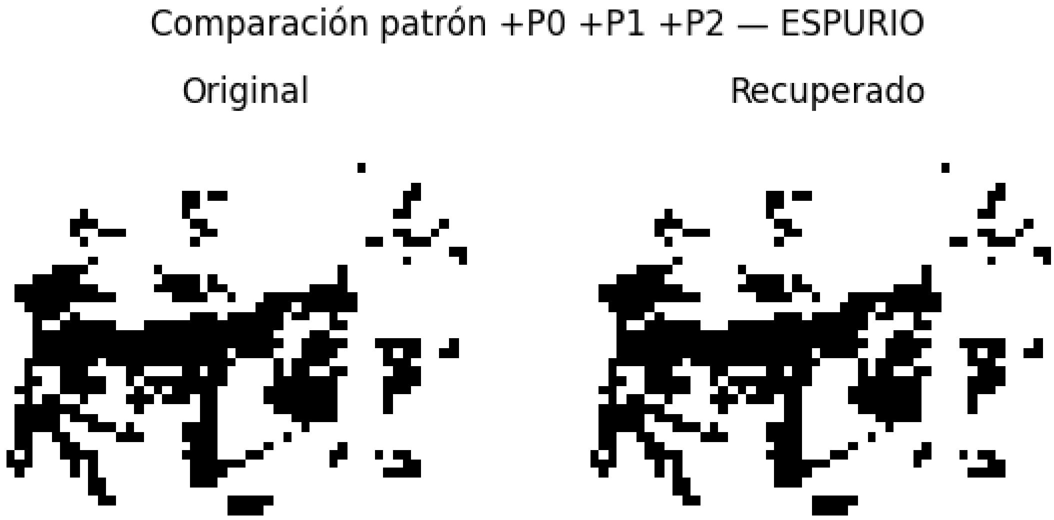

```
Inverso del patrón 1 es un estado espurio estable.
```



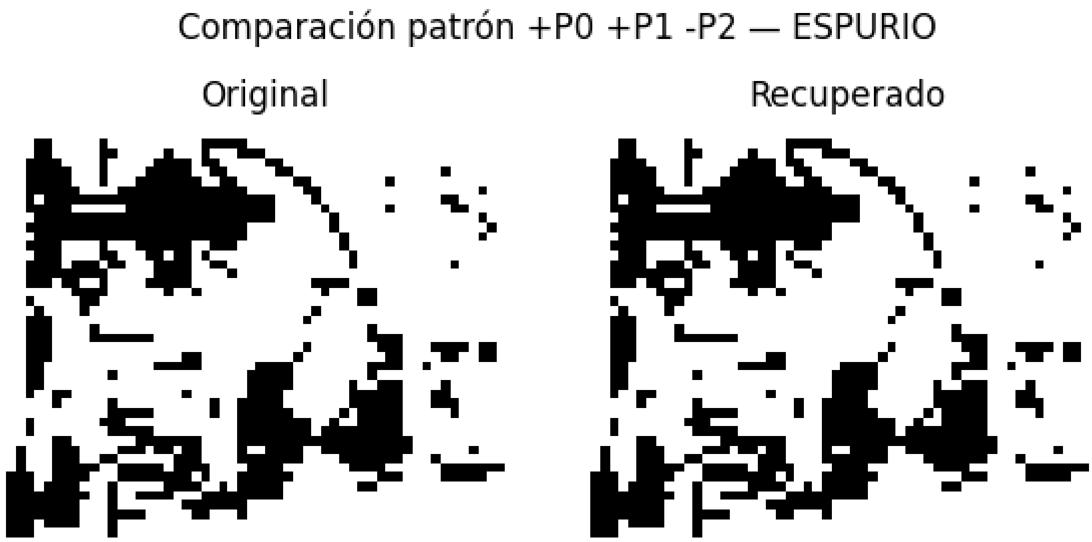

```
Inverso del patrón 2 es un estado espurio estable.
```



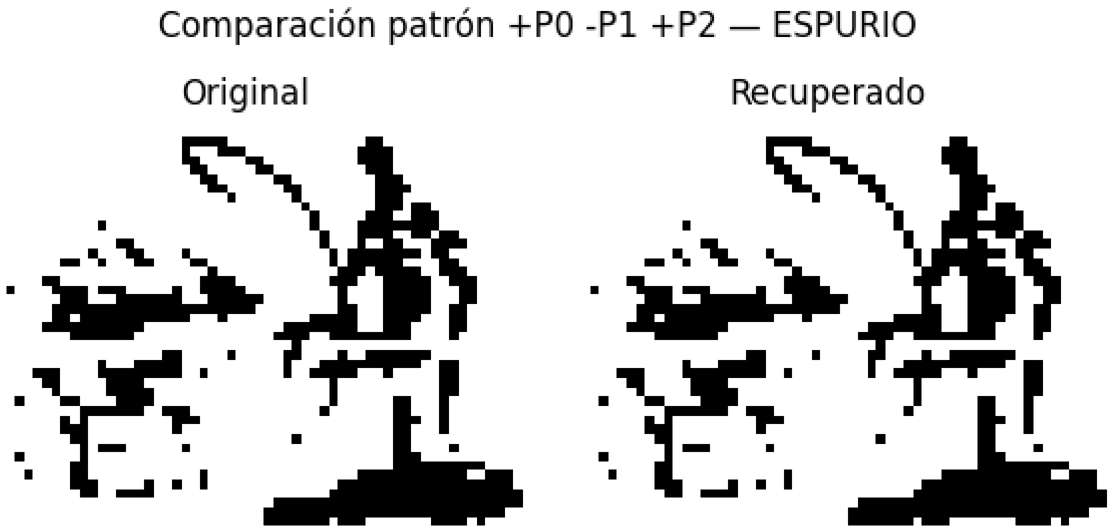

```
--- Combinación impar de 3 patrones: todas las ± ---
Combinación +P0 +P1 +P2: ESPURIO
```



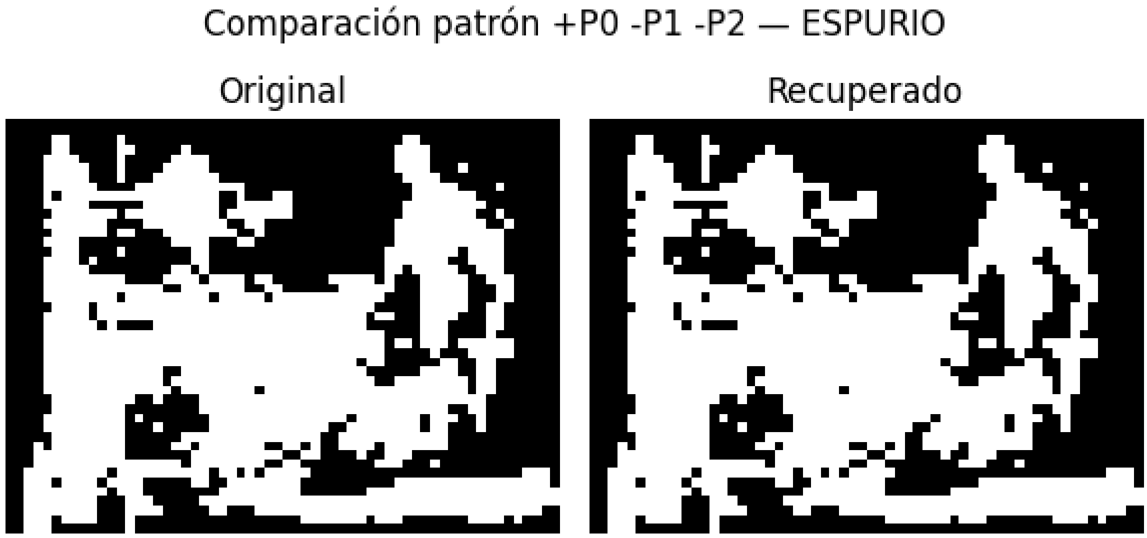

```
Combinación +P0 +P1 -P2: ESPURIO
```



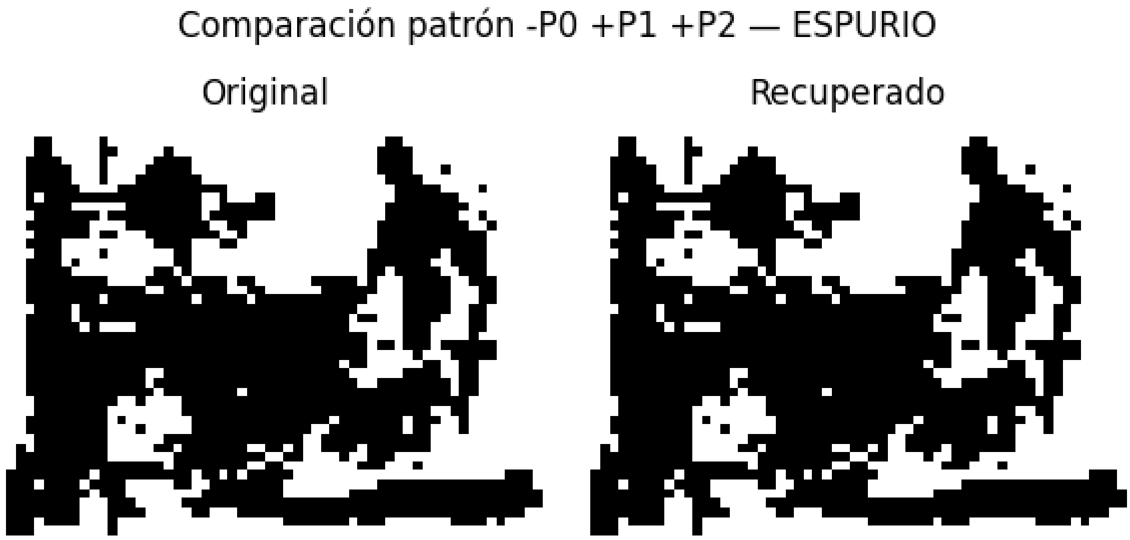

```
Combinación +P0 -P1 +P2: ESPURIO
```



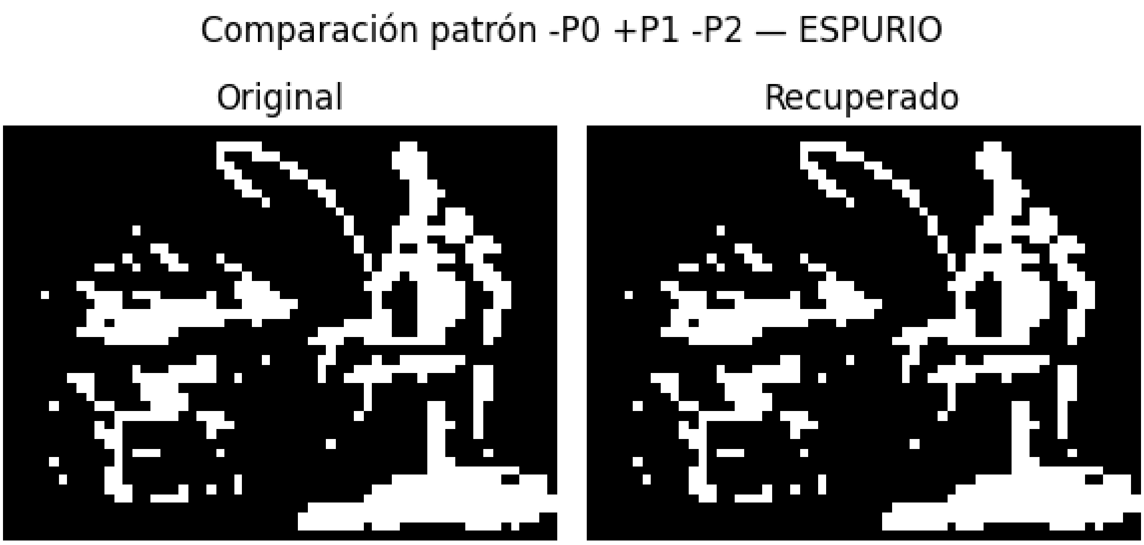
Combinación +P0 -P1 -P2: ESPURIO



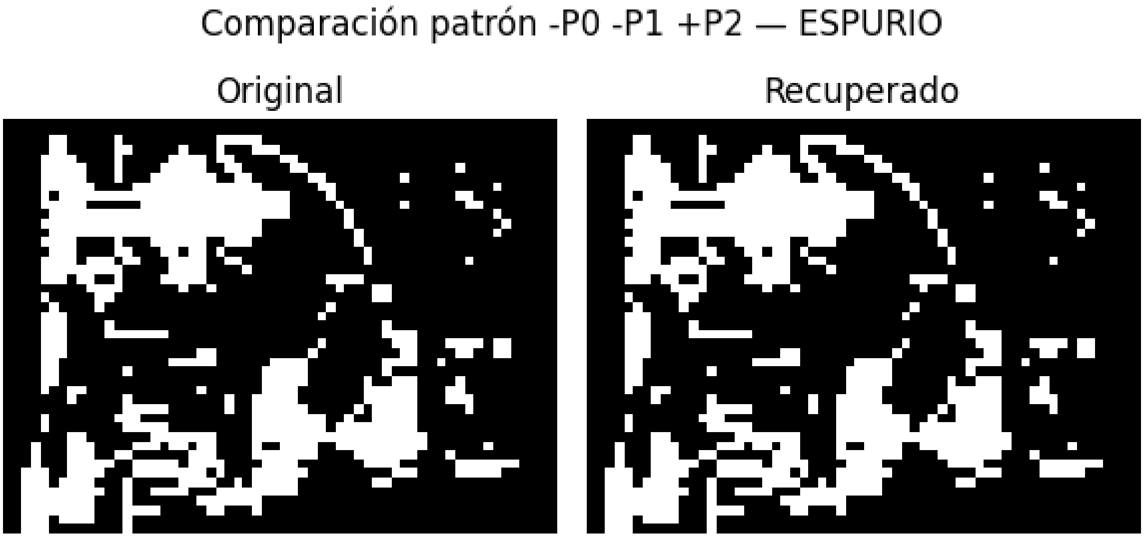
Combinación -P0 +P1 +P2: ESPURIO



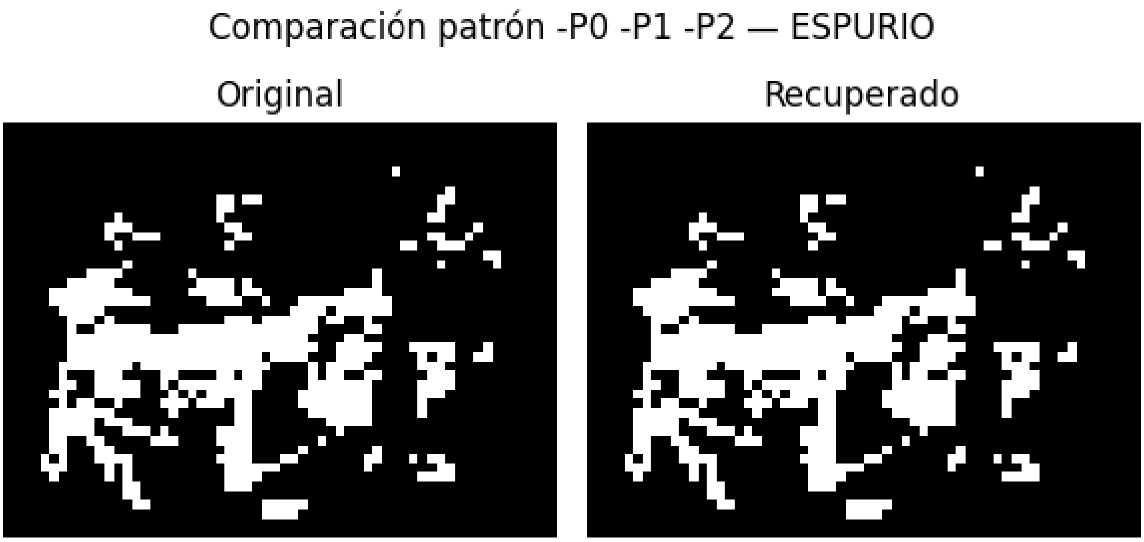
Combinación -P0 +P1 -P2: ESPURIO




```
Combinación -P0 -P1 +P2: ESPURIO
```




```
Combinación -P0 -P1 -P2: ESPURIO
```



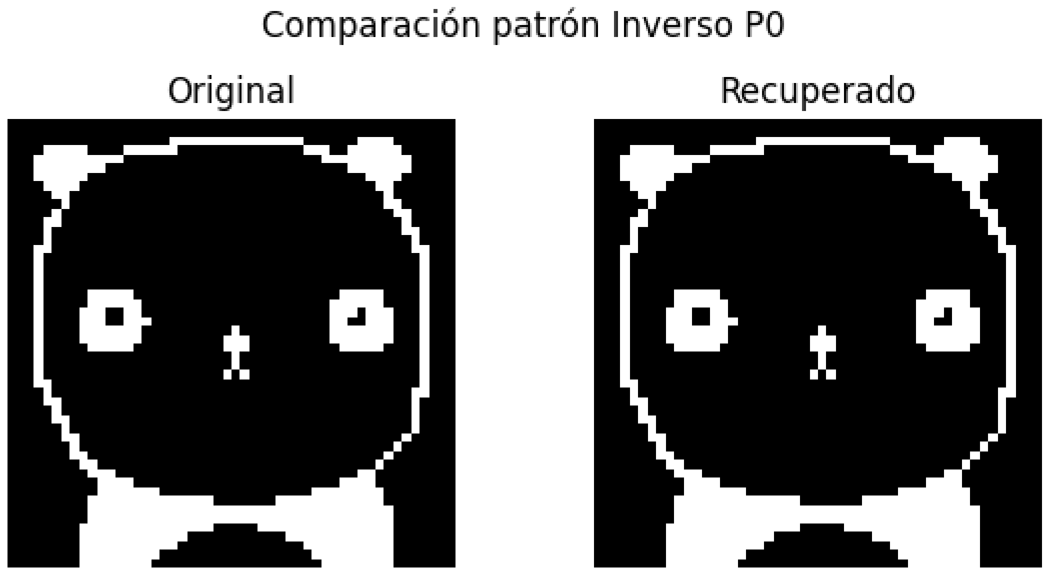
In [265…

```
carpeta_imagenes = "imagenes/50x50"
patrones = cargar_patrones_desde_carpeta(carpeta_imagenes)                 #
patrones_vectorizados = centrar_y_vectorizar_patrones(patrones)            #

ancho, alto = 50, 50
dimension = ancho * alto
```

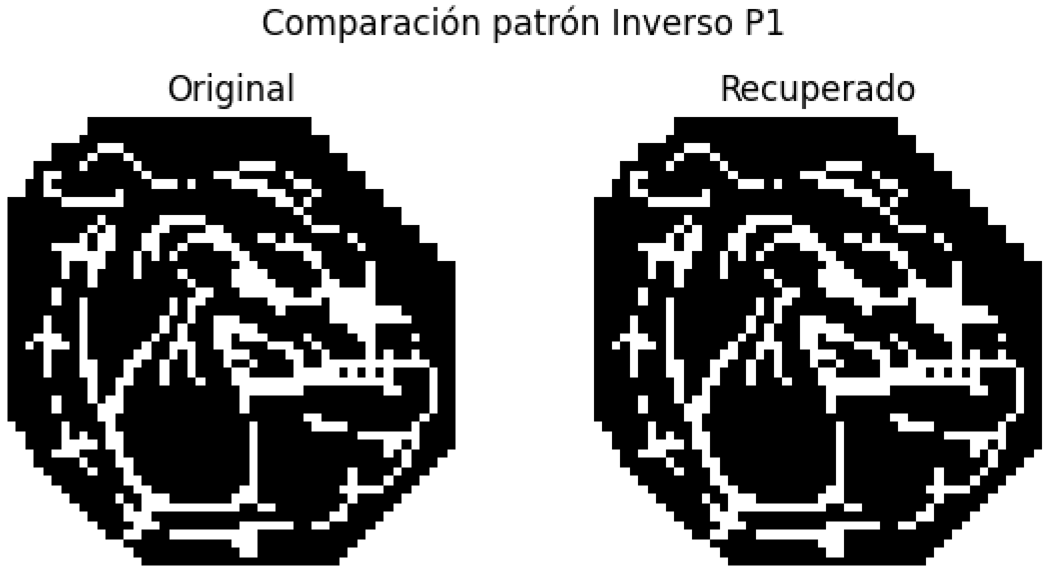
```python
# Entrenar la red
pesos = entrenar_red_hopfield(patrones_vectorizados)                    #

# ---------- ESTADOS ESPURIOS ----------

def es_estable(estado, pesos, max_iter=10000):
    """
    Verifica si un estado es un mínimo estable (converge a sí mismo).
    """
    estado = np.asarray(estado, dtype=np.int8).ravel()
    convergido = recuperar_patron(estado, pesos, max_iter=max_iter)
    return np.array_equal(convergido, estado)

# 1) Inversos de cada patrón
print("\n--- Estados inversos ---")
for i in range(patrones_vectorizados.shape[0]):
    patron = patrones_vectorizados[i]
    inverso = (-patron).astype(np.int8, copy=False)
    if es_estable(inverso, pesos):
        print(f"Inverso del patrón {i} es un estado espurio estable.")
    else:
        print(f"Inverso del patrón {i} NO es estable.")
    rec_inv = recuperar_patron(inverso, pesos, max_iter=10000)
    mostrar_comparacion_patron(inverso, rec_inv, ancho, alto, indice=f"In

# 2) Combinación impar de 3 patrones: TODAS las ± (sign(P0 ± P1 ± P2))
print("\n--- Combinación impar de 3 patrones: todas las ± ---")

if patrones_vectorizados.shape[0] >= 3:
    # Elegí los 3 que quieras (acá uso 0,1,2)
    i0, i1, i2 = 0, 1, 2
    P0 = patrones_vectorizados[i0]
    P1 = patrones_vectorizados[i1]
    P2 = patrones_vectorizados[i2]

    # Todas las combinaciones de signos (orden: +++, ++-, +-+, +--, -++,
    S = np.array([
        [ +1, +1, +1],
        [ +1, +1, -1],
        [ +1, -1, +1],
        [ +1, -1, -1],
        [ -1, +1, +1],
        [ -1, +1, -1],
        [ -1, -1, +1],
        [ -1, -1, -1],
    ], dtype=np.int8)

    def etiqueta_signos(row):
        return " ".join([
            ("+P0" if row[0] == 1 else "-P0"),
            ("+P1" if row[1] == 1 else "-P1"),
            ("+P2" if row[2] == 1 else "-P2"),
        ])

    # Combinar todas: comb = sign(S @ [P0;P1;P2])
    P_stack = np.stack([P0, P1, P2], axis=0).astype(np.int16)   # (3, N)
    comb_sum = S @ P_stack                                       # (8, N)
    combos = np.where(comb_sum >= 0, 1, -1).astype(np.int8)     # regla:
```

```
    # Evaluar y MOSTRAR TODAS las combinaciones
    for row, estado in zip(S, combos):
        rec = recuperar_patron(estado, pesos, max_iter=10000)
        estable = np.array_equal(rec, estado)
        print(f"Combinación {etiqueta_signos(row)}: {'ESPURIO' if estable
        mostrar_comparacion_patron(
            estado, rec, ancho, alto,
            indice=f"{etiqueta_signos(row)} — {'ESPURIO' if estable else
        )
else:
    print("No hay suficientes patrones para probar combinación impar (se

# 3) Estados tipo spin-glass: aleatorios, no relacionados
print("\n--- Estados tipo spin-glass ---")
rng = np.random.default_rng(123)

def generar_estado_aleatorio(n, rng=None):
    rng = rng or np.random.default_rng()
    return np.where(rng.integers(0, 2, size=n, dtype=np.int8)==0, -1, 1).

for i in range(5):  # probar 5 estados aleatorios
    estado_random = generar_estado_aleatorio(dimension, rng=rng)
    if es_estable(estado_random, pesos):
        print(f"Estado aleatorio {i} es un mínimo local estable (posible
    else:
        print(f"Estado aleatorio {i} NO es estable.")
```

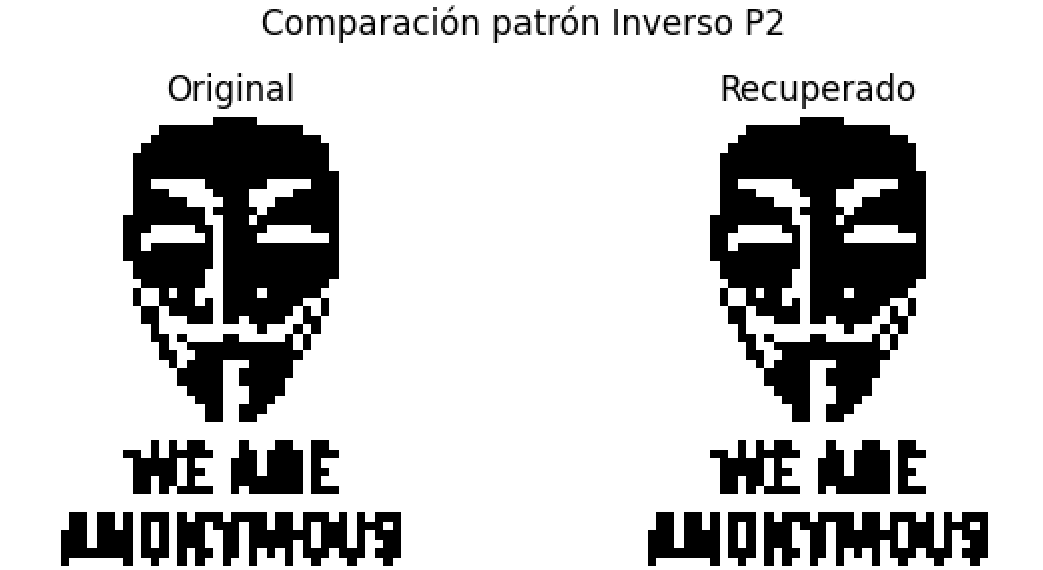
```
panda.bmp - tamaño: 50x50
perro.bmp - tamaño: 50x50
v.bmp - tamaño: 50x50
Se cargaron 3 patrones de 50 filas y 50 columnas cada uno.
Tengo 3 patrones vectorizados de 2500 elementos cada uno.

--- Estados inversos ---
Inverso del patrón 0 es un estado espurio estable.
```



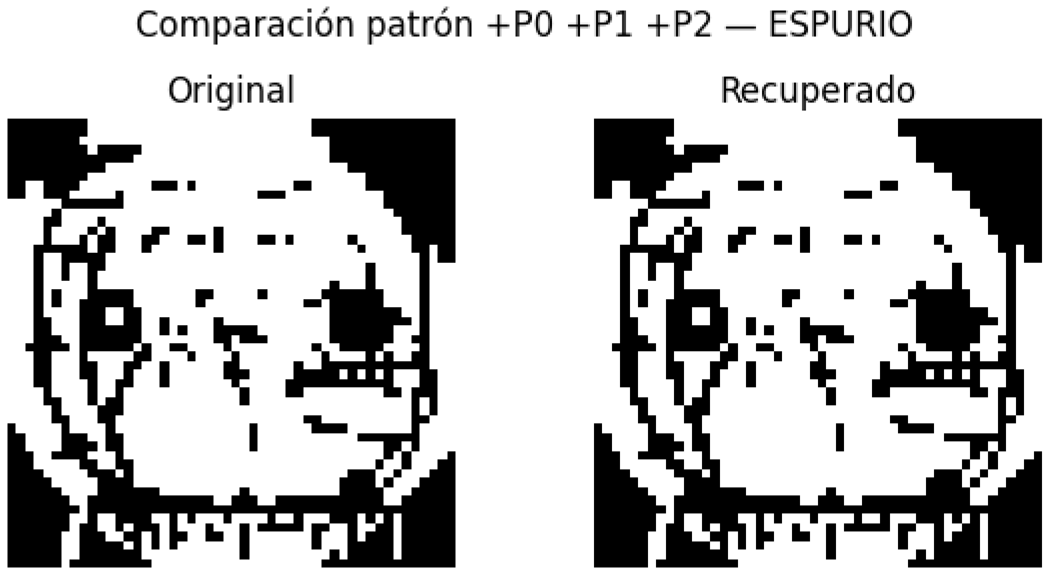

```
Inverso del patrón 1 es un estado espurio estable.
```



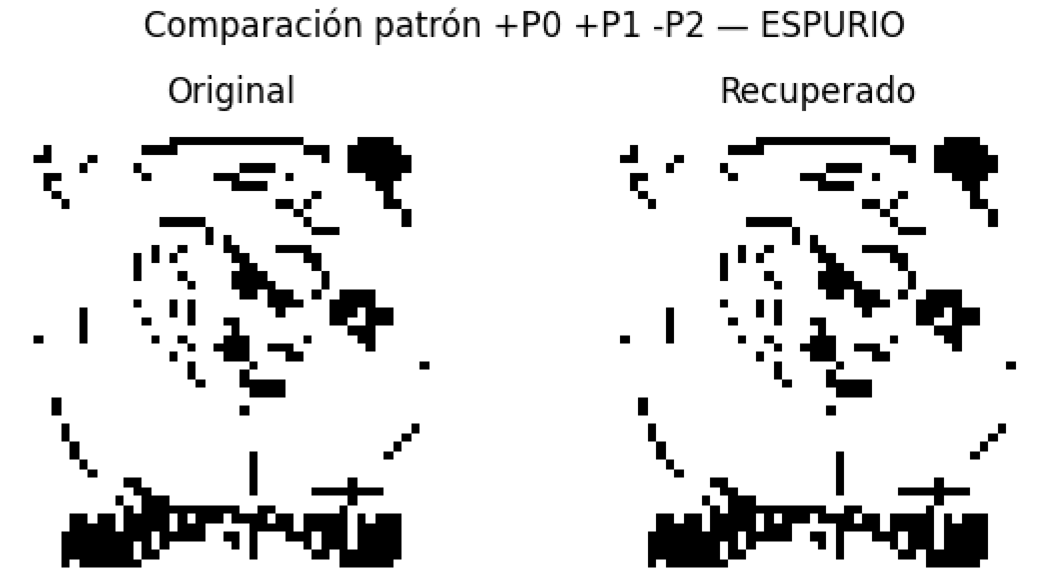

```
Inverso del patrón 2 es un estado espurio estable.
```



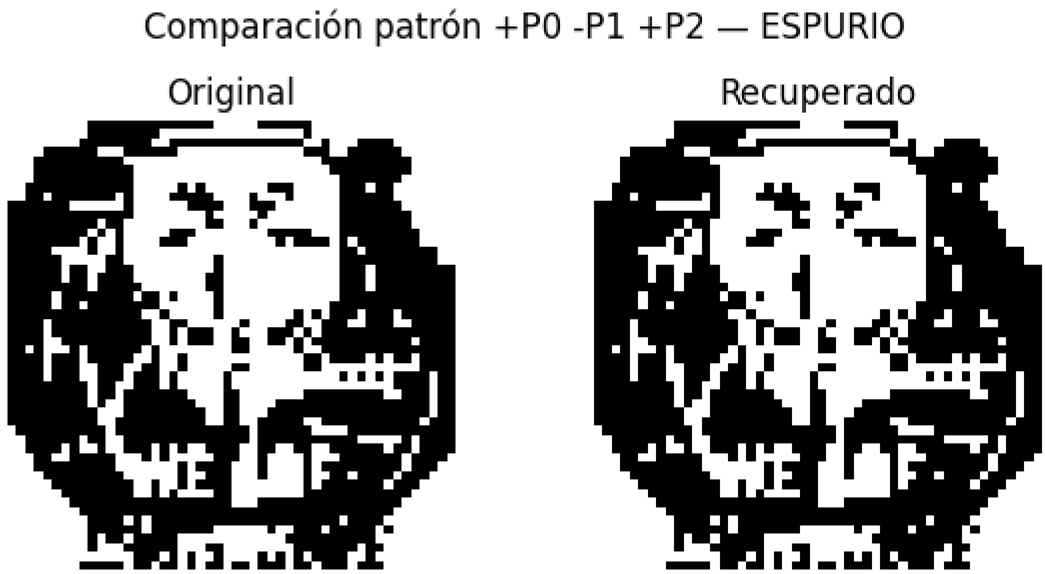

```
--- Combinación impar de 3 patrones: todas las ± ---
Combinación +P0 +P1 +P2: ESPURIO
```



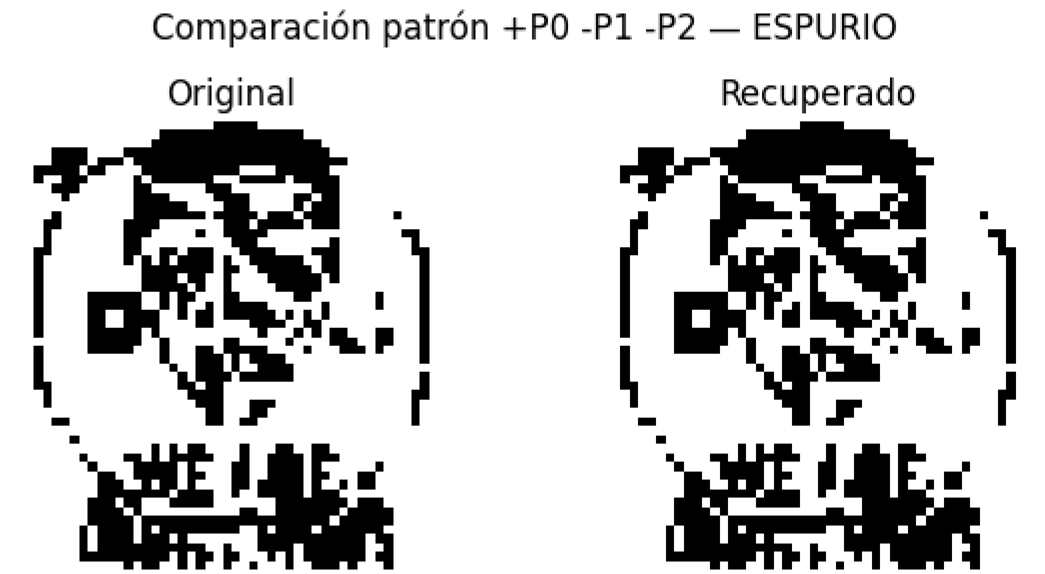

```
Combinación +P0 +P1 -P2: ESPURIO
```



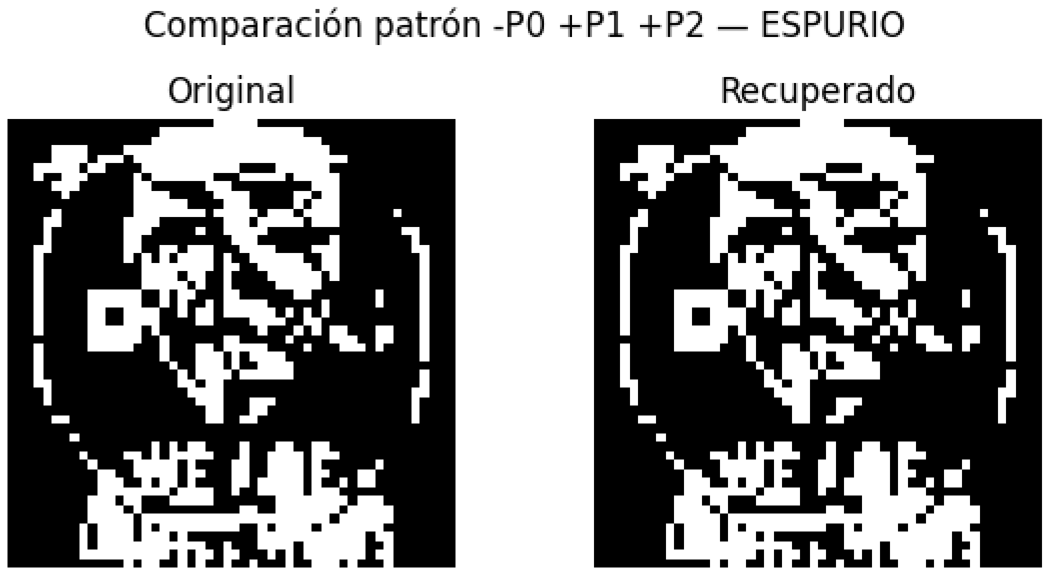

```
Combinación +P0 -P1 +P2: ESPURIO
```



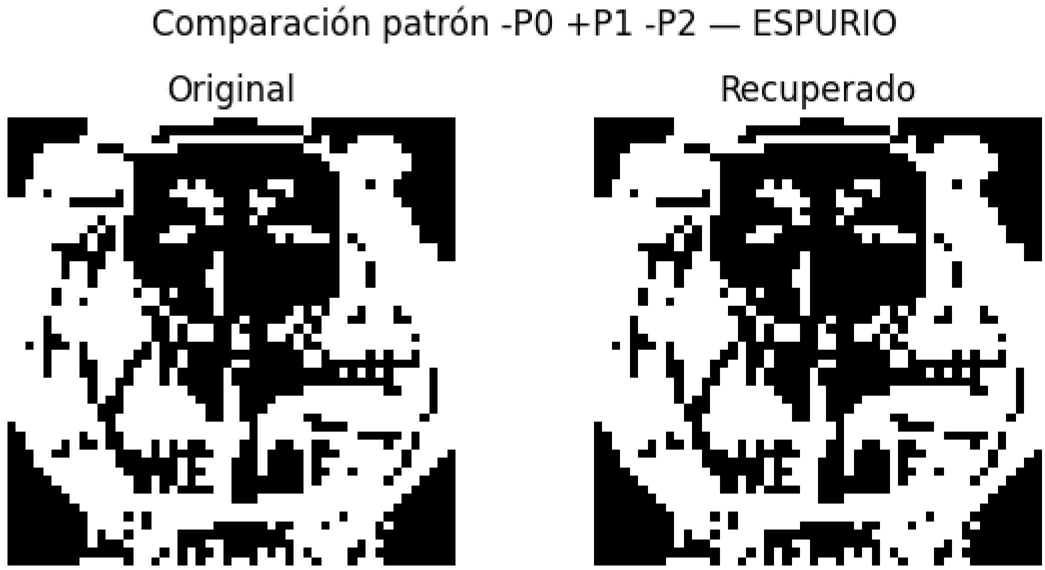

```
Combinación +P0 -P1 -P2: ESPURIO
```



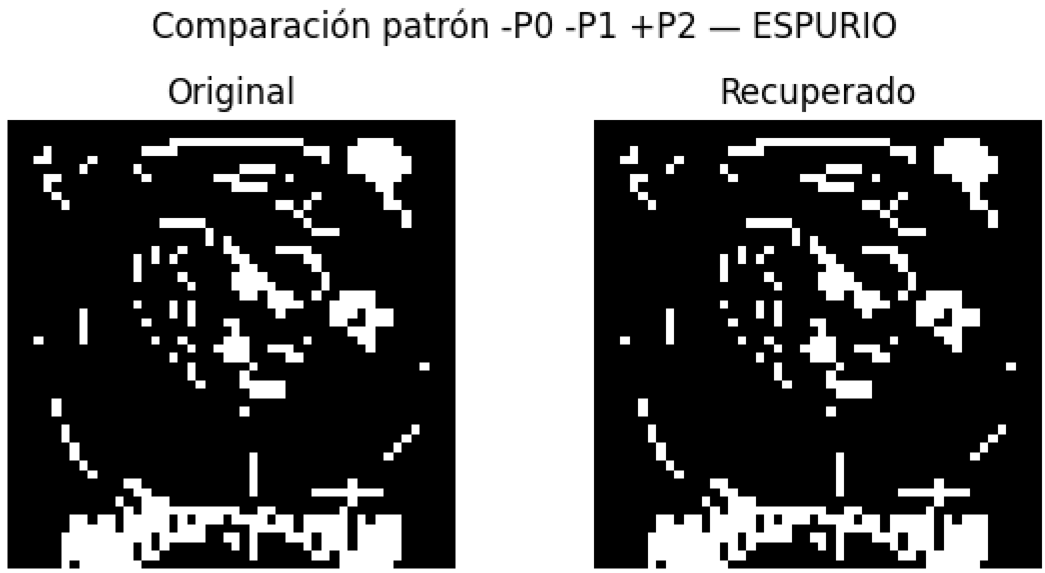

```
Combinación -P0 +P1 +P2: ESPURIO
```



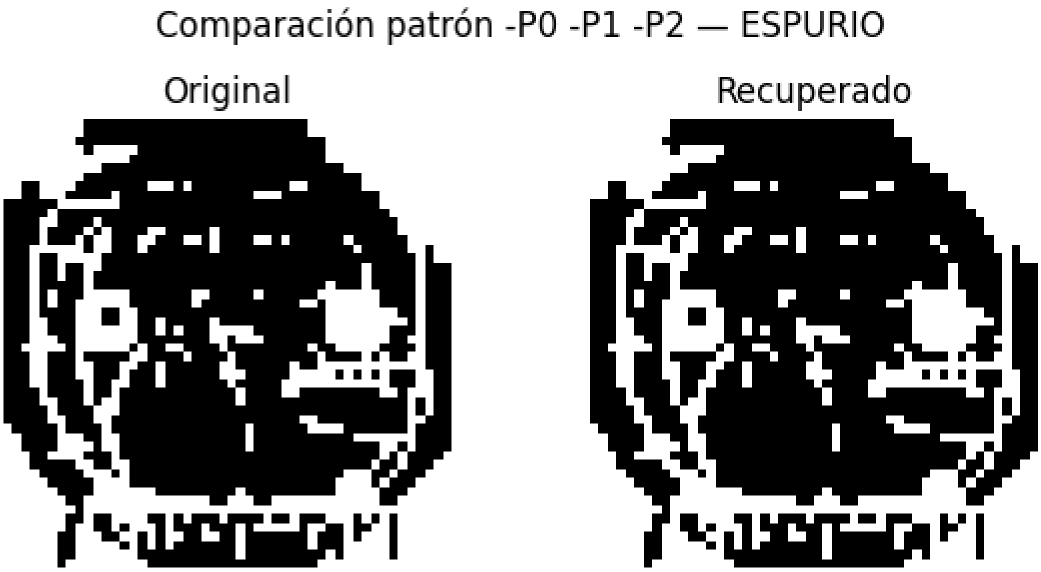

```
Combinación -P0 +P1 -P2: ESPURIO
```




```
Combinación -P0 -P1 +P2: ESPURIO
```




```
Combinación -P0 -P1 -P2: ESPURIO
```




```
--- Estados tipo spin-glass ---
Estado aleatorio 0 NO es estable.
Estado aleatorio 1 NO es estable.
Estado aleatorio 2 NO es estable.
Estado aleatorio 3 NO es estable.
Estado aleatorio 4 NO es estable.
```

Conclusión — Estados espurios (resumen)

- **Inversos**: que la red "recuerde" los **inversos** de tus 6 imágenes es normal.
  Con Hebb, pesos simétricos y umbral 0, si x es fijo entonces $-x$ también lo es (salvo empates).

- **Combinaciones impares**: las configuraciones
  $\operatorname{sign}(\pm P_0 \pm P_1 \pm P_2)$
  aparecen como **estados espurios** clásicos: mínimos locales generados por la superposición de patrones memorizados.

- **Spin-glass**: al muestrear estados aleatorios no encontraste mínimos tipo spin-glass; es **coherente** con tu **baja carga** $\alpha = \frac{P}{N}$ (6 patrones $\ll N$).
  Lejos de la capacidad, es muy improbable toparse con ellos.

Implicancia: la red está en régimen de buena recuperación, pero **hereda** los mínimos inevitables del modelo (inversos y mezclas impares).
No es un bug; es propio de Hopfield con Hebb.

Entrenamiento con las 6 imágenes unificadas

Dado que las redes de Hopfield solo permiten patrones de igual tamaño, se redimensionaron todas las imágenes (3 de `60x45` y 3 de `50x50`) al tamaño común de `50x50`.

Esto permitió unificar los datos y entrenar la red con los 6 patrones simultáneamente.

Luego se verificó si la red era capaz de recuperar cada patrón al ser presentado como entrada. Si todos se recuperan correctamente, significa que la red **almacenó**

exitosamente los 6 patrones. En caso contrario, la aparición de errores indicaría que la capacidad de almacenamiento se vio superada, o que hubo interferencia entre patrones similares.

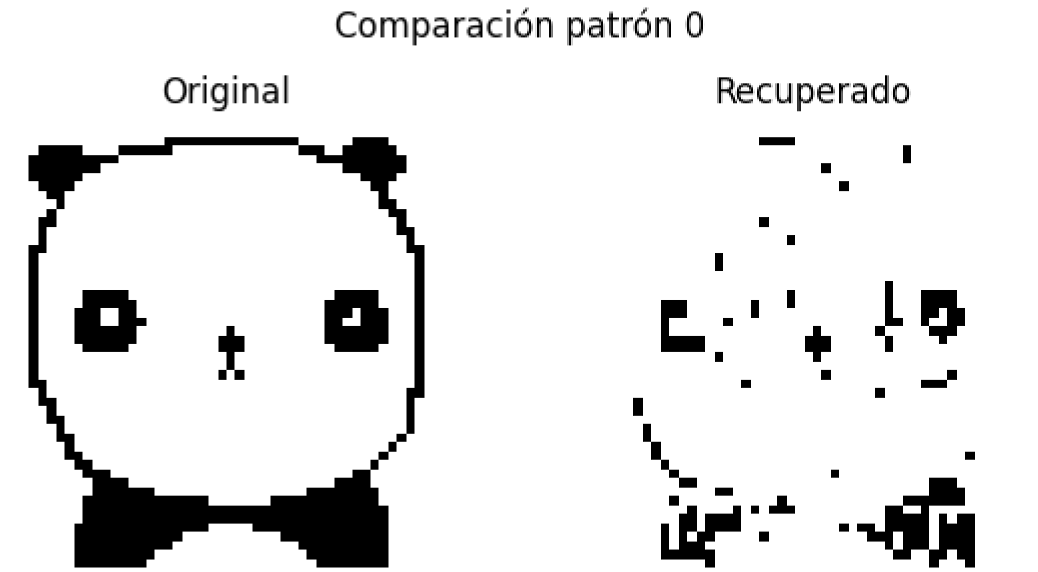
Este experimento permite explorar los límites prácticos de la capacidad de memoria de una red de Hopfield clásica.

In [266…

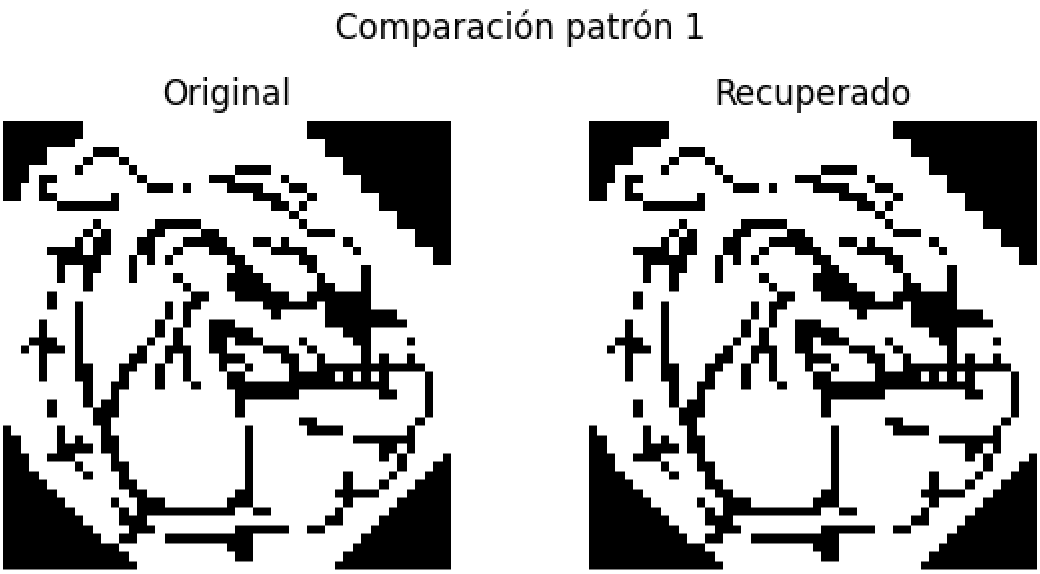
```python
# === Celda 1: carga/entrenamiento + evaluación de recuperación (optimiza
from PIL import Image
import os
import numpy as np

# --- Cargar imágenes 50x50 ---
patrones_50 = cargar_patrones_desde_carpeta("imagenes/50x50")  # si ya te

# --- Cargar imágenes 60x45 redimensionadas a 50x50 (rápido, sin getpixel
def cargar_redimensionadas(carpeta, nuevo_tamaño=(50, 50), threshold=128)
    archivos = sorted([f for f in os.listdir(carpeta) if f.lower().endswi
    mats = []
    for archivo in archivos:
        ruta = os.path.join(carpeta, archivo)
        img = Image.open(ruta).resize(nuevo_tamaño, resample=Image.NEARES
        arr = np.array(img, dtype=np.uint8)                  # 0..255
        binario = (arr >= threshold).astype(np.uint8)        # {0,1}
        print(f"{archivo} - tamaño: {binario.shape[1]}x{binario.shape[0]}
        mats.append(binario)
    if not mats:
        print("No se encontraron .bmp en la carpeta.")
        return []
    print(f"Se cargaron {len(mats)} patrones de {mats[0].shape[0]} filas
    return [m.tolist() for m in mats]  # mantiene tu interfaz (listas)

patrones_60 = cargar_redimensionadas("imagenes/60x45", (50, 50))

# --- Unificamos y vectorizamos ---
# Si querés full NumPy: usa np.stack y pasá todo como array
patrones_unificados = patrones_50 + patrones_60
patrones_vectorizados = centrar_y_vectorizar_patrones(patrones_unificados

# --- Entrenamiento ---
ancho, alto = 50, 50
pesos = entrenar_red_hopfield(patrones_vectorizados)  # versión NumPy

# --- Evaluación + recolección de no perfectos ---
no_perfectos = []   # (i, patron_orig, patron_rec, dif, similitud)
print("\n--- Evaluación de recuperación de los 6 patrones ---")

P, N = patrones_vectorizados.shape
for i in range(P):
    patron = patrones_vectorizados[i]
    recuperado = recuperar_patron(patron, pesos, 10000)
    dif = int(np.count_nonzero(patron != recuperado))
    similitud = 1 - dif / N

    if dif == 0:
        print(f"Patrón {i} fue recuperado exactamente.")
    else:
        print(f"Patrón {i} fue recuperado con similitud del {similitud*10
```

```
        no_perfectos.append((i, patron, recuperado, dif, similitud))

    mostrar_comparacion_patron(patron, recuperado, ancho=ancho, alto=alto
```

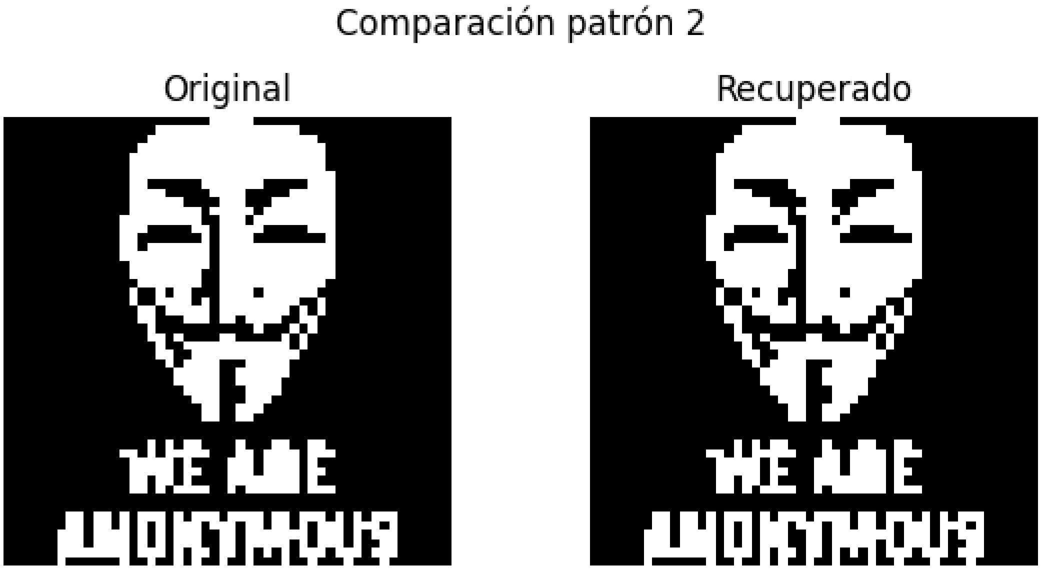
```
panda.bmp - tamaño: 50x50
perro.bmp - tamaño: 50x50
v.bmp - tamaño: 50x50
Se cargaron 3 patrones de 50 filas y 50 columnas cada uno.
paloma.bmp - tamaño: 50x50
quijote.bmp - tamaño: 50x50
torero.bmp - tamaño: 50x50
Se cargaron 3 patrones de 50 filas y 50 columnas cada uno.
Tengo 6 patrones vectorizados de 2500 elementos cada uno.

--- Evaluación de recuperación de los 6 patrones ---
Patrón 0 fue recuperado con similitud del 85.80% (355 bits distintos).
```



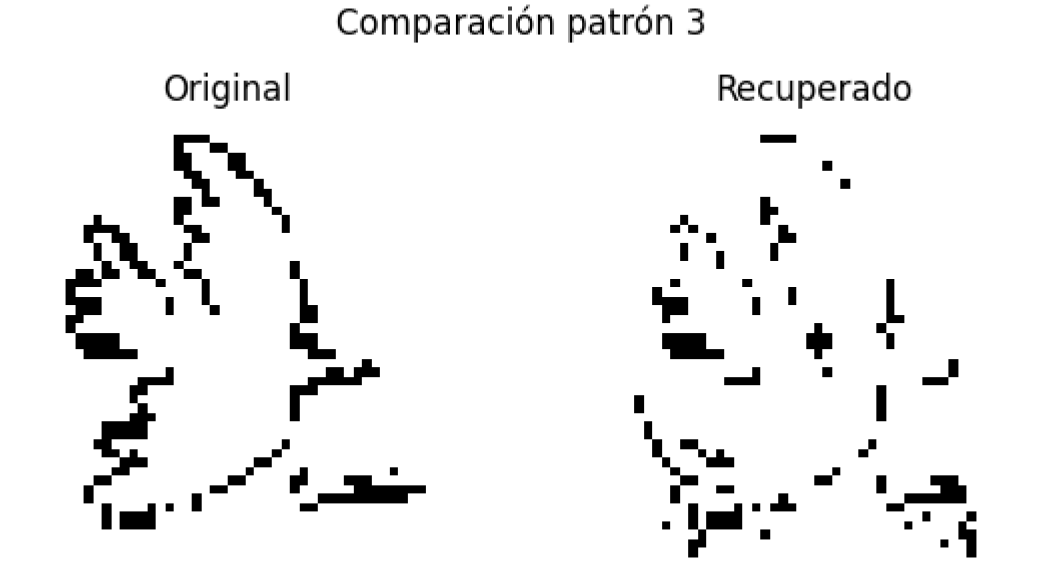

```
Patrón 1 fue recuperado exactamente.
```



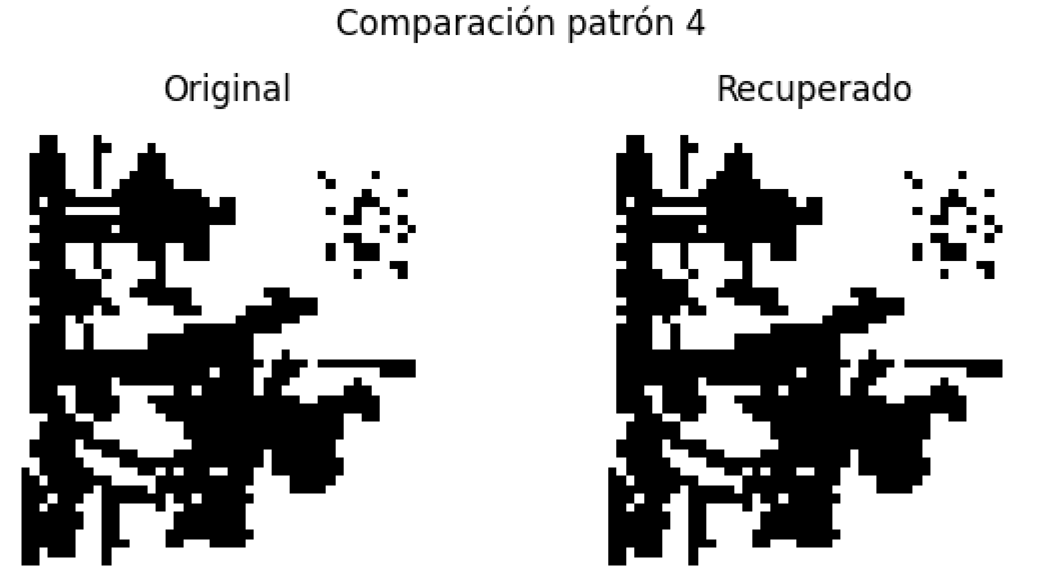

```
Patrón 2 fue recuperado exactamente.
```



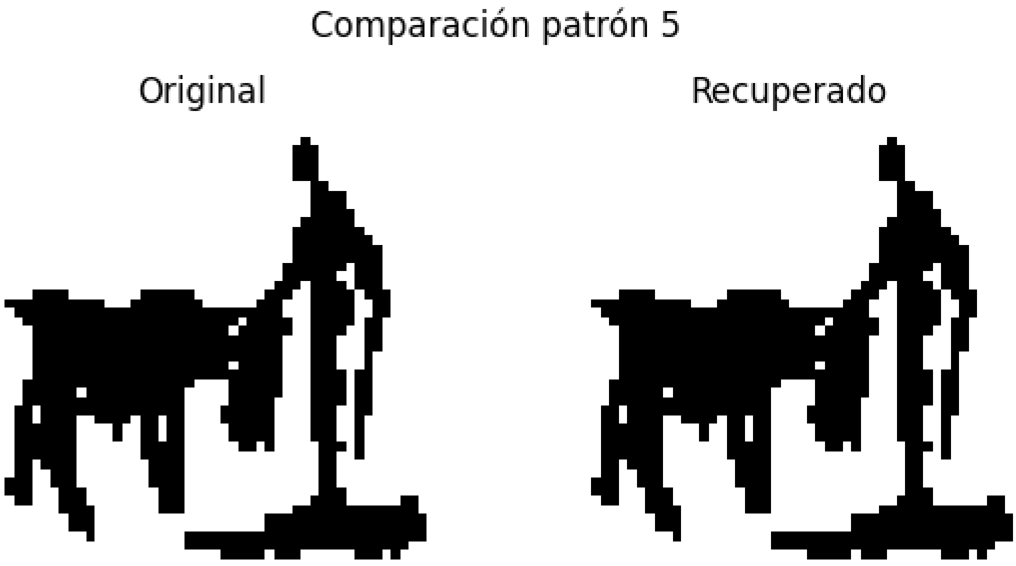

```
Patrón 3 fue recuperado con similitud del 94.32% (142 bits distintos).
```




```
Patrón 4 fue recuperado exactamente.
```




```
Patrón 5 fue recuperado exactamente.
```

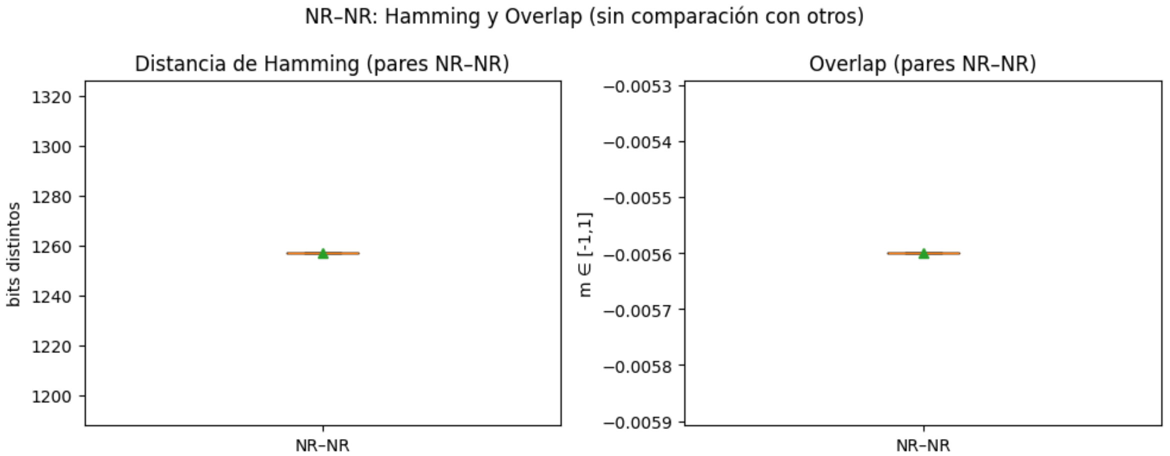


In [267…

```
# === Celda 2-bis (solo NR–NR): Hamming y Overlap
import numpy as np
import matplotlib.pyplot as plt
from itertools import combinations

if 'no_perfectos' not in globals():
    raise RuntimeError("No encuentro 'no_perfectos'. Ejecutá primero la C

if not no_perfectos:
    print("🎉 Todos los patrones se recuperaron al 100%. No hay 'no recup
else:
    # --- Datos base ---
    X_all = np.asarray(patrones_vectorizados)  # (P, N) en {-1,+1}
    P, N = X_all.shape

    # Asegurar codificación en {-1,+1}
    U = set(np.unique(X_all).tolist())
    if U.issubset({0,1}):
        X_all = (X_all.astype(np.int16)*2 - 1).astype(np.int8)
    elif not U.issubset({-1,1}):
        X_all = np.where(X_all > 0, 1, -1).astype(np.int8)

    # Índices del subconjunto "no recuperados al 100%"
    idxs_nr = np.array([i for (i, *_ ) in no_perfectos], dtype=int)
    if idxs_nr.size < 2:
        print(f"Solo hay {idxs_nr.size} patrón(es) NR. No hay pares NR–NR
    else:
        # --- Pares solo dentro de NR ---
        pares_nr = list(combinations(idxs_nr.tolist(), 2))

        H_nr, O_nr = [], []
        for i, j in pares_nr:
            m = float(np.dot(X_all[i], X_all[j]) / N)         # Overlap m
            d = int(round(0.5 * N * (1.0 - m)))               # Hamming d
            O_nr.append(m)
            H_nr.append(d)

        H_nr = np.array(H_nr)
        O_nr = np.array(O_nr)

        # --- Gráficos: solo NR–NR ---
```

```
    fig, axs = plt.subplots(1, 2, figsize=(10, 4))

    axs[0].boxplot([H_nr], labels=['NR–NR'], showmeans=True)
    axs[0].set_title('Distancia de Hamming (pares NR–NR)')
    axs[0].set_ylabel('bits distintos')

    axs[1].boxplot([O_nr], labels=['NR–NR'], showmeans=True)
    axs[1].set_title('Overlap (pares NR–NR)')
    axs[1].set_ylabel('m ∈ [-1,1]')

    plt.suptitle("NR–NR: Hamming y Overlap (sin comparación con otros
    plt.tight_layout()
    plt.show()

    # --- Resumen numérico ---
    print("Resumen (solo NR–NR):")
    print(f"- Pares NR–NR: {H_nr.size}")
    print(f"- Hamming   → mediana: {np.median(H_nr):.0f} | media: {np
    print(f"- Overlap m → mediana: {np.median(O_nr):.3f} | media: {np
```

```
/tmp/ipykernel_145091/1074683677.py:44: MatplotlibDeprecationWarning: The
'labels' parameter of boxplot() has been renamed 'tick_labels' since Matpl
otlib 3.9; support for the old name will be dropped in 3.11.
  axs[0].boxplot([H_nr], labels=['NR–NR'], showmeans=True)
/tmp/ipykernel_145091/1074683677.py:48: MatplotlibDeprecationWarning: The
'labels' parameter of boxplot() has been renamed 'tick_labels' since Matpl
otlib 3.9; support for the old name will be dropped in 3.11.
  axs[1].boxplot([O_nr], labels=['NR–NR'], showmeans=True)
```




```
Resumen (solo NR–NR):
- Pares NR–NR: 1
- Hamming   → mediana: 1257 | media: 1257.0 | IQR: 0
- Overlap m → mediana: -0.006 | media: -0.006 | IQR: 0.000
```

Conclusión: capacidad de almacenamiento y recuperación parcial

En este experimento se entrenaron **6 imágenes** redimensionadas a `50x50` píxeles ($N = 2500$ neuronas).
La **capacidad teórica** de una red de Hopfield es:

$$0.138 \times N \ \approx \ 0.138 \times 2500 \ \approx \ 345 \text{ patrones}$$

Sin embargo, la **capacidad práctica** suele ser mucho menor cuando los patrones no son aleatorios.

Resultados principales

- La red **recuperó exactamente 4 de los 6 patrones**.
- Los otros **2 patrones (i=0 e i=3)** se recuperaron de forma **parcial**:
  - Patrón 0: **85.8% de similitud** con el original.
  - Patrón 3: **94.3% de similitud** con el original.
- El análisis de los pares **NR–NR** (no recuperados) arrojó:
  - **Distancia de Hamming ≈ 1257 bits** (≈ 50% distintos, 50% iguales).
  - **Overlap ≈ -0.006**, prácticamente nulo → indica que, en promedio, no hay correlación lineal fuerte entre ellos.

Interpretación

- Aunque entre sí los patrones NR no muestran gran correlación (overlap cercano a 0), cada uno mantiene **similitudes significativas con otros patrones del conjunto**, en especial entre el patrón 0 y el patrón 3 original (≈ 75% coincidencia, overlap ≈ 0.8–0.9).
- Esta correlación provoca que ambos patrones compartan un **mínimo de energía común** en el paisaje de la red: en lugar de converger limpiamente a su estado original, caen en un **atractor espurio mixto**.
- El **redimensionamiento a 50x50** y la **estructura visual de las imágenes** (bordes, simetrías, regiones homogéneas) hacen que los patrones estén lejos de ser aleatorios, favoreciendo la interferencia.
- La **regla de Hebb** no discrimina entre características relevantes o redundantes: cualquier solapamiento fuerte entre dos patrones puede inducir confusión.

Conclusión general

Aunque la red tenía capacidad teórica suficiente para almacenar cientos de patrones (~345), la **capacidad práctica efectiva se redujo drásticamente por la correlación entre imágenes**.
Incluso con solo 6 patrones, **2 de ellos no pudieron recuperarse exactamente** y terminaron en estados espurios influenciados por otro patrón del conjunto.
Esto confirma que en aplicaciones reales con datos estructurados (no aleatorios), la **interferencia entre patrones similares es la principal limitación de la memoria de Hopfield**.

Generar patrones pseudo-aleatorios

Vamos a definir una función que crea **P** patrones pseudo-aleatorios de dimensión **N**:

- Por defecto los devuelve en **{-1, +1}** (`valores='pm1'`), que es el formato clásico para Hopfield.
- Acepta `seed` para tener **reproducibilidad**.

In [268…
```
def generar_patrones_aleatorios(N, P, valores='pm1', seed=None, return_ty
    """
    Genera P patrones pseudo-aleatorios de dimensión N usando NumPy (vect

    Parámetros
    ----------
    N : int
        Dimensión (número de neuronas).
    P : int
        Cantidad de patrones a generar.
    valores : {'pm1', '01'}
        - 'pm1' -> valores en {-1, +1} (Hopfield clásico).
        - '01'  -> valores en {0, 1}.
    seed : int | None
        Semilla para reproducibilidad (usa np.random.default_rng).
    return_type : {'ndarray', 'list'}
        Tipo de retorno. 'ndarray' (por defecto) o lista de listas.
    dtype : np.dtype
        Tipo de dato del array devuelto (por defecto np.int8).

    Retorna
    -------
    np.ndarray shape (P, N) o lista de listas, según return_type.
    """
    rng = np.random.default_rng(seed)

    if valores == 'pm1':
        # Generar en {0,1} y mapear a {-1,+1}: X = 2*B - 1
        B = rng.integers(0, 2, size=(P, N), dtype=np.int8)
        X = (B * 2 - 1).astype(dtype, copy=False)
    elif valores == '01':
        X = rng.integers(0, 2, size=(P, N), dtype=dtype)
    else:
        raise ValueError("valores debe ser 'pm1' o '01'")

    if return_type == 'list':
        return X.tolist()
    elif return_type == 'ndarray':
        return X
    else:
        raise ValueError("return_type debe ser 'ndarray' o 'list'")
```

Verificación empírica de capacidad (Hopfield '82)

Objetivo. Para un tamaño de red N (empezamos con $N = 30 \times 30 = 900$):

1. Generar P patrones pseudo-aleatorios en $\{-1, +1\}$.
2. Entrenar con Hebb clásico (diagonal nula), normalizando por N.
3. Hacer **una** iteración sincrónica: $\hat{\mathbf{x}} = \operatorname{sign}(W\,\mathbf{x})$.
4. Medir el error medio:

$$\text{error} = \frac{1}{PN} \sum_{p=1}^{P} \sum_{i=1}^{N} 1\left[\hat{x}_i^{(p)} \neq x_i^{(p)}\right]$$

5. Aumentar P de a pasos y registrar, para cada umbral P_{error} de la tabla, el **máximo** P/N tal que el error promedio $\leq P_{\text{error}}$.

Tabla objetivo (una iteración sincrónica):

P_{error}	$p_{\max}/N$
0.001	0.105
0.0036	0.138
0.01	0.185
0.05	0.37
0.1	0.61

Vamos a estimar esa curva empíricamente promediando varios *trials* por cada P.

In [269…

```
import numpy as np

# --- Helpers ---
def signo_binario(A):
    S = np.sign(A, dtype=np.float32)
    S[S == 0] = 1
    return S.astype(np.int8, copy=False)

def experimento_simple(
    N, P_values, trials=3,
    targets=(0.001, 0.0036, 0.01, 0.05, 0.1),
    norm='N',
    patrones_source=None,
    seed_base=1234
):
    """
    Experimento Hopfield (1 paso sincrónico) para estimar capacidad.
    - Acepta patrones externos via `patrones_source` o usa aleatorios por

    Parámetros
    ----------
    N : int
        Número de neuronas (dimensión del patrón).
    P_values : iterable[int]
        Conjunto de cantidades de patrones a probar.
    trials : int
        Repeticiones por cada P para promediar error.
    targets : iterable[float]
        Umbrales de error (P_error) para calcular p_max = P/N.
    norm : {'N','P'}
        Normalización de entrenar_red_hopfield.
    patrones_source : None | callable | dict
        - None: usa generar_patrones_aleatorios(N,P,seed, valores='pm1',
        - callable: patrones_source(N, P, trial) -> ndarray (P,N) en {-1,
        - dict: patrones_source[P] -> ndarray (P,N) o lista de ndarrays p
    seed_base : int
        Base para semillas reproducibles.

    Retorna
    -------
```

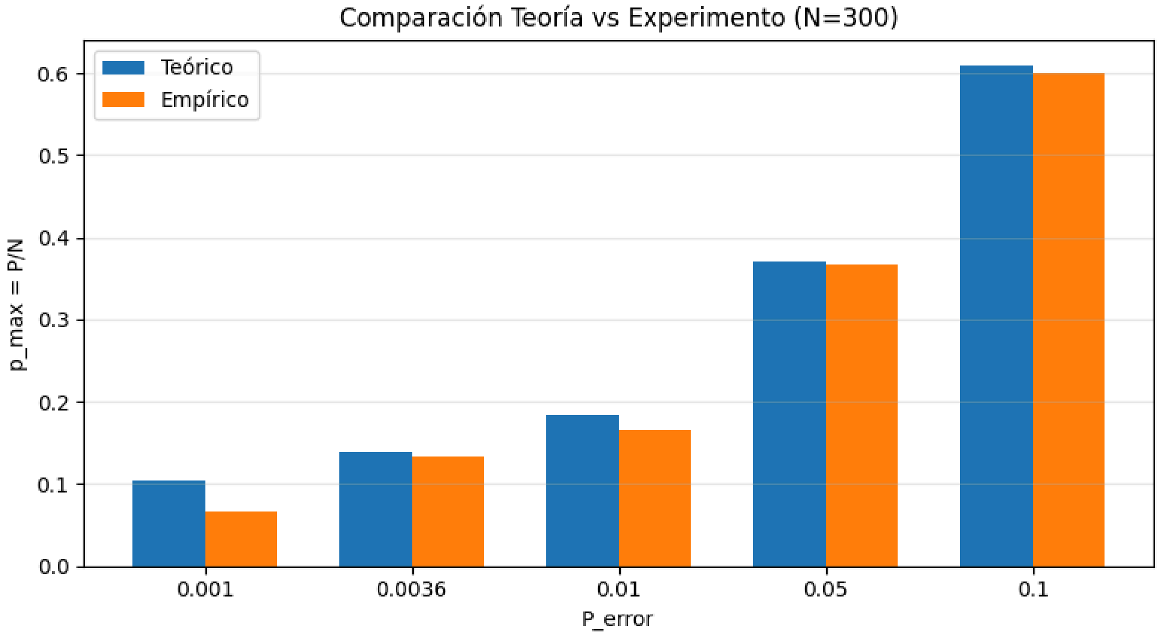
```
    list[tuple(float, float|None, float|None)]
        Lista de (target, p_emp=P/N máximo con err<=target, err@p_emp).
    """
    def _to_pm1(X):
        X = np.asarray(X)
        U = set(np.unique(X).tolist())
        if U.issubset({0, 1}):
            X = (X.astype(np.int8) * 2 - 1)
        elif not U.issubset({-1, 1}):
            X = np.where(X > 0, 1, -1).astype(np.int8)
        return X.astype(np.int8, copy=False)

    def _get_patrones(N, P, trial, seed):
        if callable(patrones_source):
            X0 = patrones_source(N, P, trial)
        elif isinstance(patrones_source, dict):
            pool = patrones_source.get(P, None)
            if pool is None:
                raise KeyError(f"No hay patrones para P={P} en patrones_s
            X0 = pool[trial % len(pool)] if isinstance(pool, (list, tuple
        else:
            X0 = generar_patrones_aleatorios(
                N, P, seed=seed, valores='pm1', return_type='ndarray', dt
            )
        X0 = _to_pm1(X0)
        if X0.shape != (P, N):
            raise ValueError(f"Shape esperado ({P},{N}) y obtuve {X0.shap
        return X0

    mean_errors = []
    for P in P_values:
        errs = []
        for t in range(trials):
            seed = seed_base + 1000 * P + t
            X0 = _get_patrones(N, P, trial=t, seed=seed)
            W = entrenar_red_hopfield(X0, norm=norm)   # Debe existir en
            X_new = signo_binario(X0 @ W)              # Debe existir en
            errores_bits = np.count_nonzero(X_new != X0)
            errs.append(errores_bits / (P * N))
        mean_errors.append(float(np.mean(errs)))

    resultados = []
    for target in targets:
        last_ok = None
        for P, err in zip(P_values, mean_errors):
            if err <= target:
                last_ok = (P, err)
        if last_ok:
            P_ok, err_ok = last_ok
            resultados.append((target, P_ok / N, err_ok))
        else:
            resultados.append((target, None, None))
    return resultados
```

In [270…

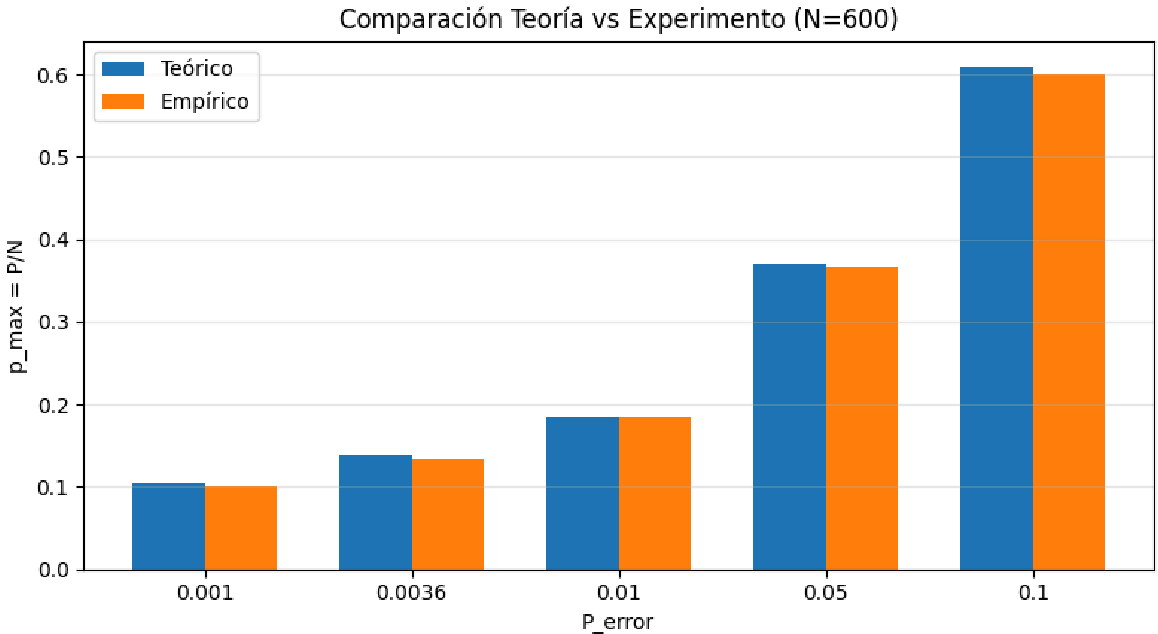
```
# --- Tabla teórica Hopfield (1982, 1 paso sincrónico) ---
THEORETICAL = {
    0.001: 0.105,
    0.0036: 0.138,
    0.01: 0.185,
    0.05: 0.37,
```

```python
    0.1: 0.61,
}

# --- Probar para distintos tamaños de red ---
P_range = {
    300:  list(range(10, 500, 10)),
    600:  list(range(10, 800, 10)),
    900:  list(range(10, 1000, 10)),
    1600: list(range(20, 2000, 20)),
    2500: list(range(20, 1000, 20))
}

Ns = list(P_range.keys())  # usar las mismas claves

def plot_teo_vs_emp(N, resultados, theoretical):
    """
    Grafica barras lado a lado: p_teo vs p_emp para cada P_error.
    Un gráfico por N.
    """
    # Ordenar por P_error tal como vienen en resultados
    labels = [f"{t:.4g}" for (t, _, _) in resultados]
    p_teo = [theoretical[t] for (t, _, _) in resultados]
    p_emp = [pe if pe is not None else np.nan for (_, pe, _) in resultado

    x = np.arange(len(labels))
    width = 0.35

    plt.figure(figsize=(8, 4.5))
    plt.bar(x - width/2, p_teo, width, label="Teórico")
    plt.bar(x + width/2, p_emp, width, label="Empírico")
    plt.xticks(x, labels)
    plt.ylabel("p_max = P/N")
    plt.xlabel("P_error")
    plt.title(f"Comparación Teoría vs Experimento (N={N})")
    plt.legend()
    plt.grid(axis="y", alpha=0.3)
    plt.tight_layout()
    plt.show()

for N in Ns:
    resultados = experimento_simple(N, P_range[N], trials=3, targets=tupl
    # Resumen textual (opcional)
    print(f"\n=== Resultados empíricos (N={N}) ===")
    for tgt, p_emp, err in resultados:
        p_teo = THEORETICAL[tgt]
        if p_emp is None:
            print(f"  - P_error={tgt:.4g}:  p_emp=--   | p_teo={p_teo:.3f
        else:
            delta = p_emp - p_teo
            print(f"  - P_error={tgt:.4g}:  p_emp={p_emp:.3f}  | p_teo={p
    # Gráfico comparativo correcto
    plot_teo_vs_emp(N, resultados, THEORETICAL)
```

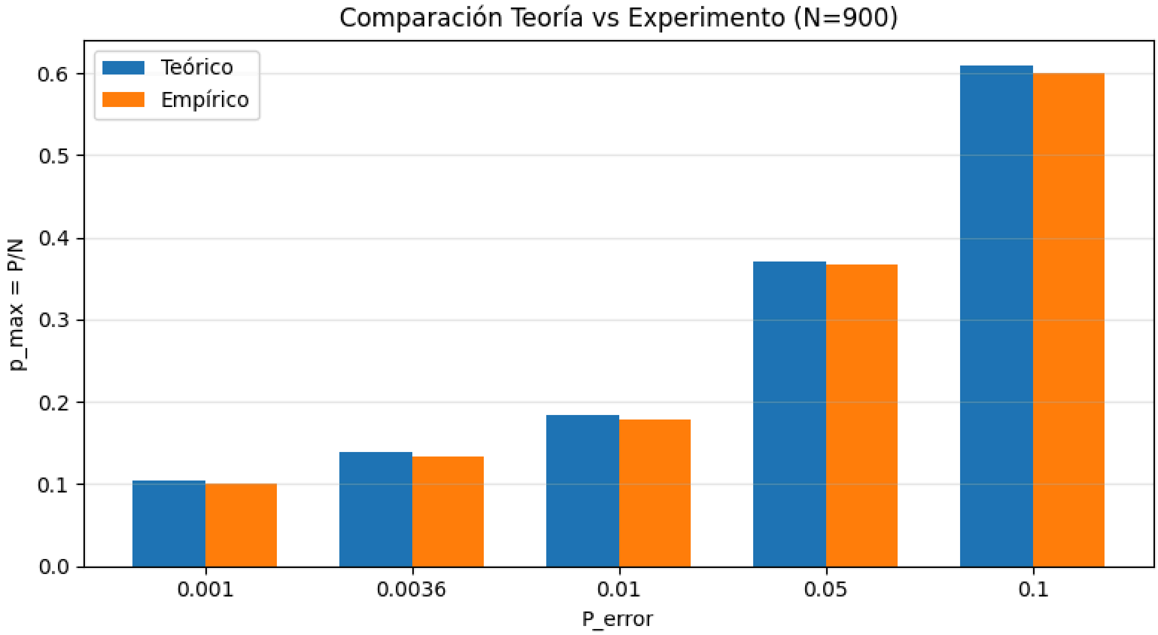
```
=== Resultados empíricos (N=300) ===
  - P_error=0.001:  p_emp=0.067  | p_teo=0.105  | Δ=-0.038
  - P_error=0.0036:  p_emp=0.133  | p_teo=0.138  | Δ=-0.005
  - P_error=0.01:  p_emp=0.167  | p_teo=0.185  | Δ=-0.018
  - P_error=0.05:  p_emp=0.367  | p_teo=0.370  | Δ=-0.003
  - P_error=0.1:  p_emp=0.600  | p_teo=0.610  | Δ=-0.010
```



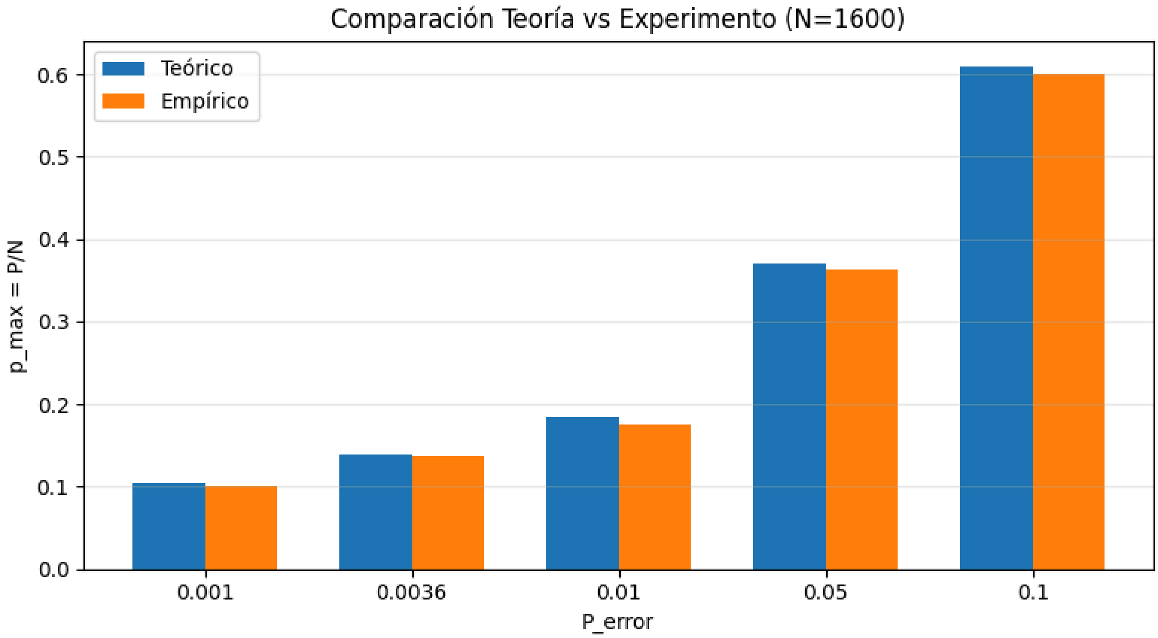

```
=== Resultados empíricos (N=600) ===
  - P_error=0.001:  p_emp=0.100  | p_teo=0.105  | Δ=-0.005
  - P_error=0.0036:  p_emp=0.133  | p_teo=0.138  | Δ=-0.005
  - P_error=0.01:  p_emp=0.183  | p_teo=0.185  | Δ=-0.002
  - P_error=0.05:  p_emp=0.367  | p_teo=0.370  | Δ=-0.003
  - P_error=0.1:  p_emp=0.600  | p_teo=0.610  | Δ=-0.010
```



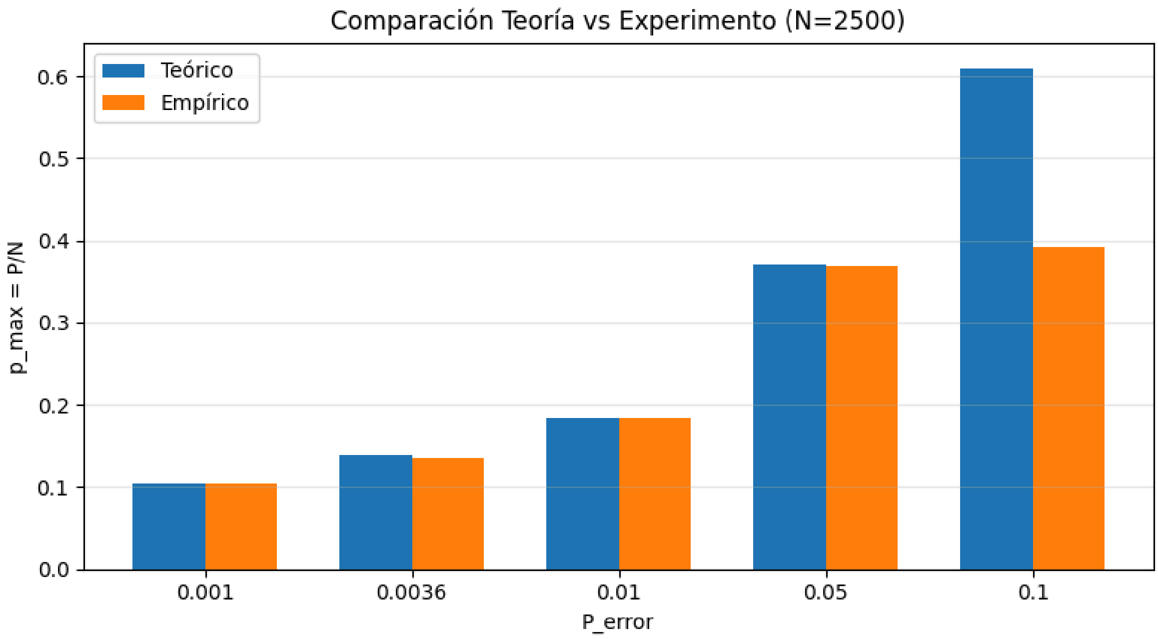

```
=== Resultados empíricos (N=900) ===
  - P_error=0.001:  p_emp=0.100  | p_teo=0.105  | Δ=-0.005
  - P_error=0.0036:  p_emp=0.133  | p_teo=0.138  | Δ=-0.005
  - P_error=0.01:  p_emp=0.178  | p_teo=0.185  | Δ=-0.007
  - P_error=0.05:  p_emp=0.367  | p_teo=0.370  | Δ=-0.003
  - P_error=0.1:  p_emp=0.600  | p_teo=0.610  | Δ=-0.010
```




```
=== Resultados empíricos (N=1600) ===
  - P_error=0.001:  p_emp=0.100  | p_teo=0.105  | Δ=-0.005
  - P_error=0.0036:  p_emp=0.138  | p_teo=0.138  | Δ=-0.001
  - P_error=0.01:  p_emp=0.175  | p_teo=0.185  | Δ=-0.010
  - P_error=0.05:  p_emp=0.362  | p_teo=0.370  | Δ=-0.008
  - P_error=0.1:  p_emp=0.600  | p_teo=0.610  | Δ=-0.010
```




```
=== Resultados empíricos (N=2500) ===
  - P_error=0.001:  p_emp=0.104  | p_teo=0.105  | Δ=-0.001
  - P_error=0.0036:  p_emp=0.136  | p_teo=0.138  | Δ=-0.002
  - P_error=0.01:  p_emp=0.184  | p_teo=0.185  | Δ=-0.001
  - P_error=0.05:  p_emp=0.368  | p_teo=0.370  | Δ=-0.002
  - P_error=0.1:  p_emp=0.392  | p_teo=0.610  | Δ=-0.218
```



Conclusiones sobre la capacidad empírica vs teórica en Hopfield

- En todos los tamaños de red analizados ($N = 300, 600, 900, 1600, 2500$) la red Hopfield muestra un **muy buen ajuste con los valores teóricos** en los **umbrales bajos de error**:
  - Para $P_{\text{error}} = 0.001, 0.0036, 0.01$ las diferencias empíricas Δ son menores a 0.02 en todos los casos.
  - Esto confirma que la predicción teórica de Hopfield (1982) describe con gran precisión la capacidad en el régimen de error casi nulo.
- En el **umbral medio** $P_{\text{error}} = 0.05$ los resultados empíricos también se mantienen muy cerca de la teoría ($\Delta < 0.01$), incluso para tamaños pequeños de red.
  → La red tolera cargas de hasta $p \approx 0.37$ sin desviarse de lo esperado.
- La diferencia más notable aparece en el **umbral alto** $P_{\text{error}} = 0.1$:
  - Para $N = 300, 600, 900, 1600$ se alcanzó $p_{\text{emp}} \approx 0.60$, muy cercano al valor teórico de 0.61.
  - Sin embargo, para $N = 2500$ el máximo p_{emp} fue solo **0.392**, lo que indica una caída prematura de la capacidad.
  - Esto sugiere que en redes más grandes aparecen **efectos de dinámica y estados espurios** que limitan la recuperación cuando se permite un error alto, alejándose de la predicción teórica.

Generación de patrones correlacionados en Hopfield (explicación)

Idea general

Queremos P **patrones** de longitud N en $\{-1, +1\}$ con una **correlación promedio** (overlap medio entre pares) controlada por un parámetro $\rho \in [0, 1]$.

Usamos un modelo *prototipo + flips*:

1. Creamos un **patrón prototipo** $g \in \{-1, +1\}^N$.
2. Cada patrón $x^{(p)}$ se obtiene **copiando** g bit a bit con probabilidad q y **flipping** (cambiando el signo) con probabilidad $1-q$, de forma independiente por bit.

¿Por qué funciona?

Para dos patrones cualesquiera $a \neq b$ y una coordenada i:

- $x_i^{(a)} = g_i$ con prob. q, y $x_i^{(a)} = -g_i$ con prob. $1-q$.
- Lo mismo para $x_i^{(b)}$, de modo independiente.

Entonces:

- $x_i^{(a)} x_i^{(b)} = +1$ si **ambos** copiaron o **ambos** flipperaron: prob. $q^2 + (1-q)^2$.
- $x_i^{(a)} x_i^{(b)} = -1$ si uno copió y el otro flipperó: prob. $2q(1-q)$.

La **esperanza** del producto en una coordenada es:

$$\mathbb{E}\left[x_i^{(a)} x_i^{(b)}\right] = \left(q^2 + (1-q)^2\right) - 2q(1-q) = (2q-1)^2.$$

El **overlap** entre dos patrones es:

$$m(x^{(a)}, x^{(b)}) = \frac{1}{N} \sum_{i=1}^{N} x_i^{(a)} x_i^{(b)}.$$

Por independencia entre bits y para N grande, el **overlap promedio** entre pares tiende a:

$$\mathbb{E}\left[m(x^{(a)}, x^{(b)})\right] = (2q-1)^2.$$

Si queremos fijar una **correlación objetivo** $\rho \in [0, 1]$, imponemos:

$$(2q-1)^2 = \rho \quad \Longrightarrow \quad q = \frac{1+\sqrt{\rho}}{2},$$

y por ende la probabilidad de **flip** es $p_{\text{flip}} = 1 - q = \frac{1-\sqrt{\rho}}{2}$.

Intuición de ρ

- $\rho = 1 \Rightarrow q = 1$: todos los patrones son **idénticos** al prototipo.
- $\rho = 0 \Rightarrow q = \frac{1}{2}$: cada bit copia/flippea al **50%**, patrones **independientes** entre sí (overlap nulo en promedio).
- Valores intermedios de ρ producen familias de patrones **más o menos parecidos** al prototipo.

Detalles prácticos

- **Dominio**: $\rho \in [0, 1]$.
- **Salida**: por defecto en $\{-1, +1\}$. Si preferís $\{0, 1\}$, basta mapear con $(x + 1)/2$.
- **Exactitud empírica**: la correlación observada entre pares fluctúa alrededor de ρ y **converge** al valor deseado cuando N crece (ley de los grandes números).
- **Complejidad**: $O(PN)$, todo vectorizado en NumPy.

Verificación rápida (overlap medio)

Para verificar, calculá el **overlap medio entre todos los pares**:

$$\bar{m} = \frac{1}{P(P-1)} \sum_{a \neq b} \frac{x^{(a)} \cdot x^{(b)}}{N},$$

que debería estar **cerca de** ρ (mejor cuanto mayor sea N).

In [271…

```
def generar_patrones_correlacionados(N, P, rho, seed=None, valores='pm1',
                                     return_type='ndarray', dtype=np.int8
    """
    Genera P patrones de N bits en {-1,+1} con correlación promedio objet

    Modelo:
      - Se crea un prototipo g ∈ {-1,+1}^N.
      - Cada patrón copia g bit a bit con prob q y lo invierte con prob (
      - Con q = (1 + sqrt(rho))/2 se tiene E[x^(a)_i * x^(b)_i] = rho (pa

    Parámetros
    ----------
    N : int
        Número de neuronas (longitud del patrón).
    P : int
        Cantidad de patrones a generar.
    rho : float in [0,1]
        Correlación objetivo (overlap medio entre patrones). 0=independie
    seed : int | None
        Semilla para reproducibilidad (np.random.default_rng).
    valores : {'pm1','01'}
        Formato de salida: 'pm1' -> {-1,+1}, '01' -> {0,1}.
    return_type : {'ndarray','list'}
        Tipo de retorno.
    dtype : np.dtype
        Tipo del array (por defecto np.int8).

    Retorna
    -------
    np.ndarray shape (P,N) o list[list[int]]
        Patrones con la correlación promedio deseada en esperanza.

    Notas
    -----
    - La correlación empírica por pares ≈ rho cuando N es grande (ley de
    - Para rho=1 todos los patrones son idénticos; para rho=0 son indepen
    """
    if not (0.0 <= rho <= 1.0):
        raise ValueError("rho debe estar en [0,1].")
```

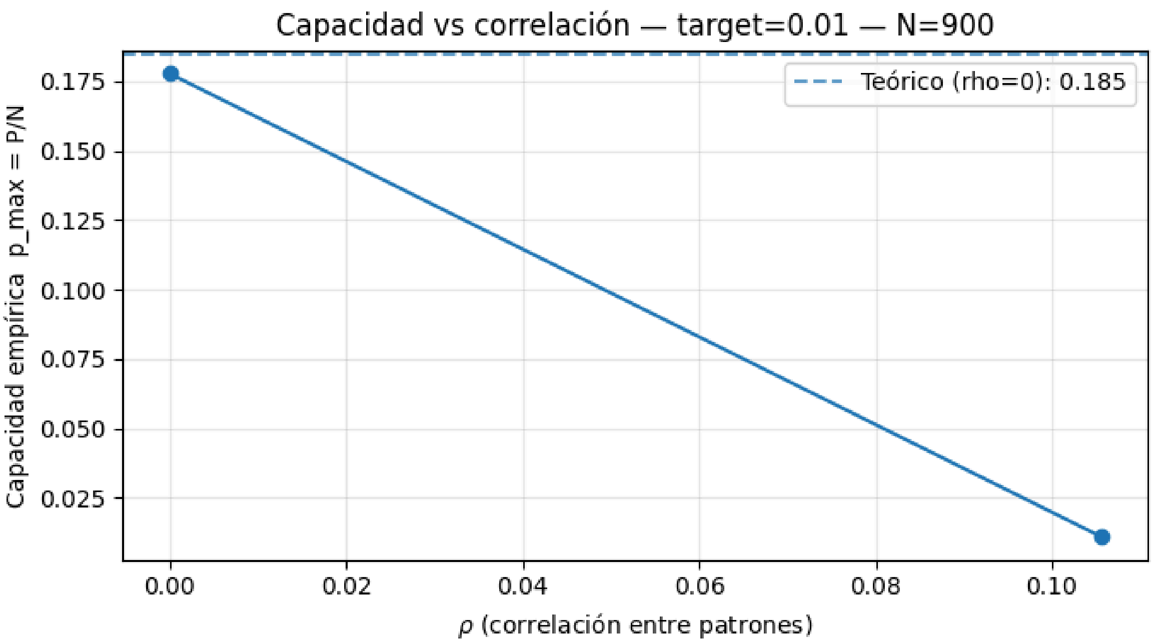
```
    rng = np.random.default_rng(seed)

    # Prototipo base g en {-1,+1}
    g = rng.integers(0, 2, size=N, dtype=np.int8)
    g = (g * 2 - 1).astype(np.int8)  # {-1,+1}

    # Probabilidad de NO copiar el bit del prototipo (flip)
    # Con q = (1 + sqrt(rho))/2 -> p_flip = 1 - q
    q = (1.0 + np.sqrt(rho)) / 2.0
    p_flip = 1.0 - q  # en [0, 0.5]

    # Matriz de flips para P patrones y N bits: True=flip (-1), False=cop
    flips = rng.random((P, N)) < p_flip
    S = np.where(flips, -1, 1).astype(np.int8)  # matriz de signos

    # Construcción de patrones: X[p, i] = g[i] * S[p, i]
    X = S * g  # broadcast de g sobre filas

    if valores == '01':
        X = ((X + 1) // 2).astype(dtype, copy=False)  # {-1,+1} -> {0,1}
    else:  # 'pm1'
        X = X.astype(dtype, copy=False)

    return X if return_type == 'ndarray' else X.tolist()


# --- (Opcional) helper para verificar correlación empírica ---
def overlap_medio_pares(X):
    """
    Overlap medio entre todos los pares de patrones (excluye diagonal).
    X en {-1,+1}. Devuelve promedio de (x_a · x_b)/N para a≠b.
    """
    X = np.asarray(X, dtype=np.int8)
    if set(np.unique(X).tolist()) == {0,1}:
        X = (X * 2 - 1).astype(np.int8)
    P, N = X.shape
    M = (X @ X.T) / float(N)        # matriz de overlaps
    m = (np.sum(M) - np.trace(M)) / (P*(P-1))  # media off-diagonal
    return float(m)
```

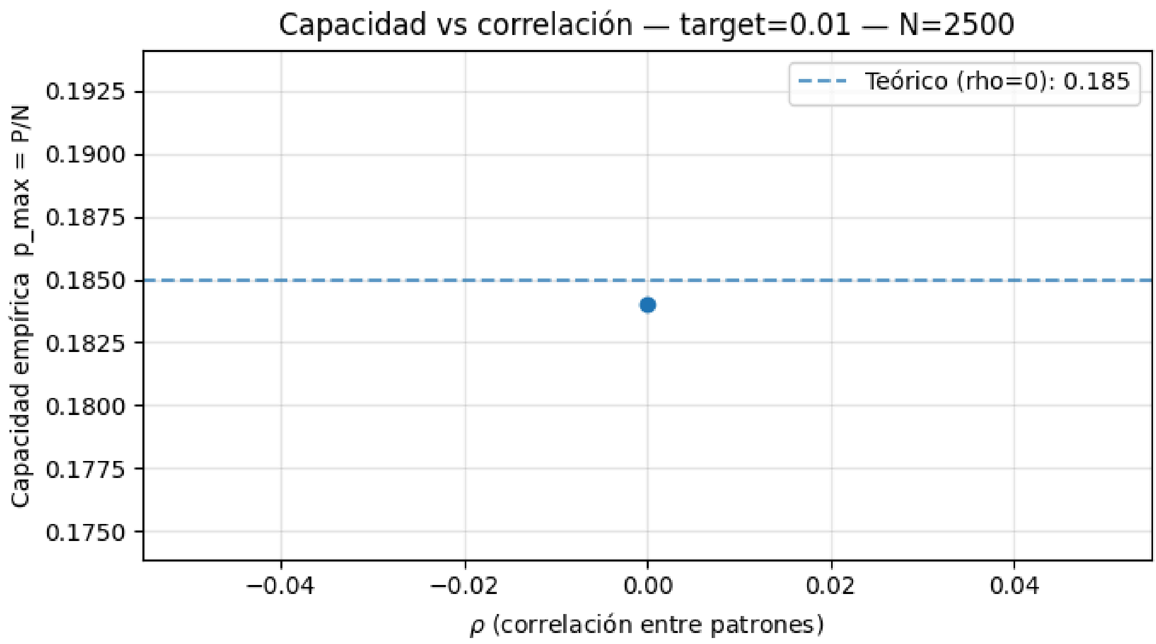
In [272…

```
# Fábrica de proveedores de patrones correlacionados (usa tu generar_patr
def make_correlated_provider(rho, seed_base=0):
    def _prov(N, P, trial):
        # generar_patrones_correlacionados debe devolver (P,N) en {-1,+1}
        return generar_patrones_correlacionados(N=N, P=P, rho=rho, seed=s
    return _prov

def capacidad_vs_rho(N, rhos, P_values, target=0.01, trials=3, norm='N',
    """
    Para un N fijo y un umbral de error 'target', barre rhos y devuelve p
    alcanzado empíricamente con patrones de correlación 'rho'.
    """
    p_emp_list = []
    for rho in rhos:
        prov = make_correlated_provider(rho, seed_base=seed_base)
        resultados = experimento_simple(
            N, P_values,
            trials=trials,
            targets=(target,),     # pedimos solo ese umbral
```

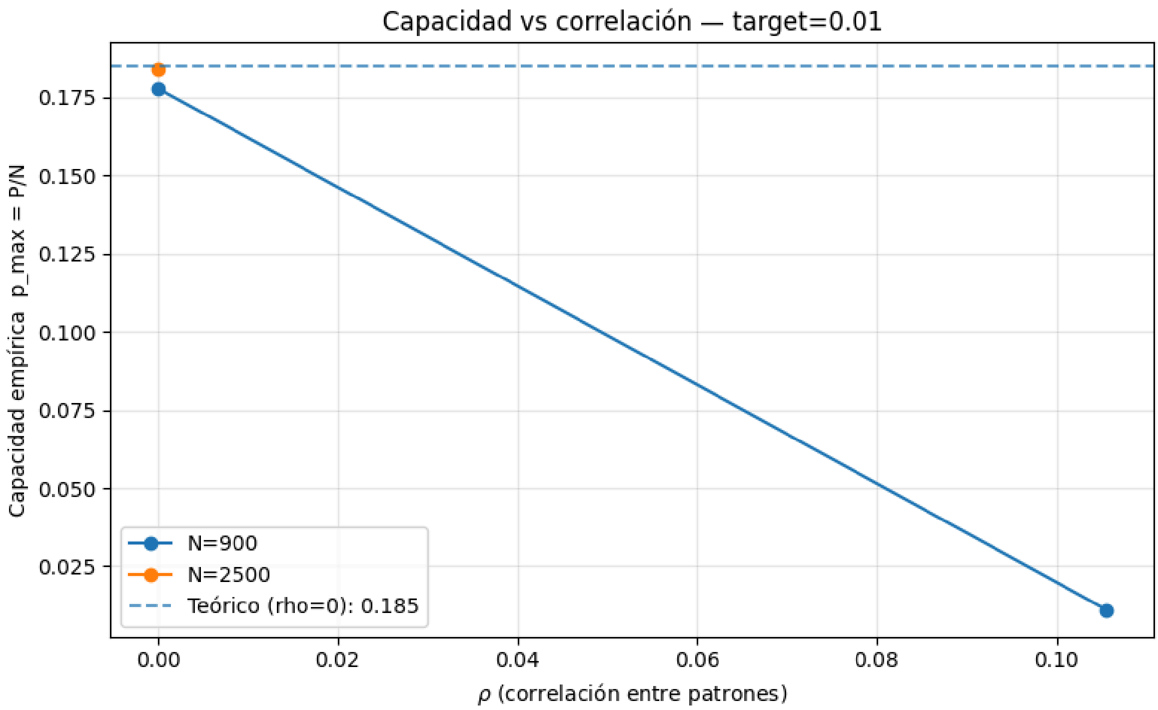
```
            norm=norm,
            patrones_source=prov,
            seed_base=seed_base
        )
        # resultados es lista de tuplas [(target, p_emp, err)]
        _, p_emp, _ = resultados[0]
        p_emp_list.append(p_emp)   # puede ser None si no se alcanzó
    return p_emp_list

def plot_capacidad_vs_rho(rhos, p_emp_list, target, p_teo_ref=None, title
    """
    Grafica p_max (empírico) vs rho. Opcional: línea horizontal con p_teo
    """
    rhos = np.asarray(rhos, dtype=float)
    y = np.array([np.nan if v is None else v for v in p_emp_list], dtype=

    plt.figure(figsize=(7,4))
    plt.plot(rhos, y, marker='o')
    if p_teo_ref is not None:
        plt.axhline(p_teo_ref, linestyle='--', alpha=0.7, label=f"Teórico
        plt.legend()
    plt.xlabel(r"$\rho$ (correlación entre patrones)")
    plt.ylabel("Capacidad empírica  p_max = P/N")
    ttl = f"Capacidad vs correlación — target={target:g}"
    if title_extra:
        ttl += f" — {title_extra}"
    plt.title(ttl)
    plt.grid(alpha=0.3)
    plt.tight_layout()
    plt.show()

def capacidad_multiN_vs_rho(Ns, rhos, P_range_dict, target=0.01, trials=3
    """
    Corre capacidad_vs_rho para varios N y devuelve dict: N -> lista p_em
    """
    out = {}
    for N in Ns:
        P_vals = P_range_dict[N]
        p_emp = capacidad_vs_rho(N, rhos, P_vals, target=target, trials=t
        out[N] = p_emp
    return out

def plot_multiN_capacidad_vs_rho(rhos, resultados_dict, target, p_teo_ref
    """
    Grafica en una figura p_max vs rho para múltiples N (una curva por N)
    """
    plt.figure(figsize=(8,5))
    for N, p_list in resultados_dict.items():
        y = np.array([np.nan if v is None else v for v in p_list], dtype=
        plt.plot(rhos, y, marker='o', label=f"N={N}")
    if p_teo_ref is not None:
        plt.axhline(p_teo_ref, linestyle='--', alpha=0.7, label=f"Teórico
    plt.xlabel(r"$\rho$ (correlación entre patrones)")
    plt.ylabel("Capacidad empírica  p_max = P/N")
    plt.title(f"Capacidad vs correlación — target={target:g}")
    plt.grid(alpha=0.3)
    plt.legend()
    plt.tight_layout()
    plt.show()
```

In [ ]:
```
# Umbral de error a evaluar (cambiá por 0.05 o 0.1 si querés)
target = 0.01

# Referencia teórica para patrones independientes (rho=0)
THEORETICAL = {0.001: 0.105, 0.0036: 0.138, 0.01: 0.185, 0.05: 0.37, 0.1:
p_teo_ref = THEORETICAL.get(target, None)

# Rango de P por N
P_range = {
    300: list(range(10, 500, 10)),
    600: list(range(10, 800, 10)),
    900: list(range(10, 1000, 10)),
}

# Conjunto de Ns y barrido de correlaciones
Ns = [300, 600, 900]
rhos = np.linspace(0.0, 0.95, 10)  # 0 → independientes ; 1 → idénticos (

# --- Un N por figura ---
for N in Ns:
    p_emp_list = capacidad_vs_rho(
        N=N,
        rhos=rhos,
        P_values=P_range[N],
        target=target,
        trials=3,
        norm='N',
        seed_base=1234
    )
    # Gráfico para este N
    plot_capacidad_vs_rho(rhos, p_emp_list, target=target, p_teo_ref=p_te

# --- Todas las N en una sola figura ---
res_multi = capacidad_multiN_vs_rho(Ns, rhos, P_range_dict=P_range, targe
plot_multiN_capacidad_vs_rho(rhos, res_multi, target=target, p_teo_ref=p_
```





Capacidad vs correlación — target=0.01 — N=2500
Teórico (rho=0): 0.185
Capacidad empírica p_max = P/N
ρ (correlación entre patrones)
0.1925
0.1900
0.1875
0.1850
0.1825
0.1800
0.1775
0.1750
−0.04
−0.02
0.00
0.02
0.04



Capacidad vs correlación — target=0.01
N=900
N=2500
Teórico (rho=0): 0.185
Capacidad empírica p_max = P/N
ρ (correlación entre patrones)
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.00
0.02
0.04
0.06
0.08
0.10