

Trabajo Práctico 3

1. Construya una red de Kohonen de 2 entradas que aprenda una distribución uniforme dentro del círculo unitario. Mostrar el mapa de preservación de topología. Probar con distribuciones uniformes dentro de otras figuras geométricas.

```
In [21]: import numpy as np
import matplotlib.pyplot as plt

# Para reproducibilidad
np.random.seed(42)
```

Generación de datos en figuras geométricas

Esta celda define tres funciones para muestrear puntos en distintas regiones del plano, que luego se usan como datos de entrada para entrenar el mapa de Kohonen.

Círculo unitario

La función `sample_uniform_circle` genera puntos uniformemente distribuidos en el disco unitario:

$$x^2 + y^2 \leq 1.$$

Para obtener uniformidad en área, no se elige el radio r de forma uniforme, sino como

$$r = \sqrt{U}, \quad U \sim \mathcal{U}(0, 1),$$

mientras que el ángulo θ se toma uniforme en $[0, 2\pi]$. Luego cada punto se construye como

$$x = r \cos \theta, \quad y = r \sin \theta.$$

Cuadrado

La función `sample_uniform_square` genera puntos en el cuadrado

$$[-1, 1] \times [-1, 1]$$

muestreando de manera independiente

$$x \sim \mathcal{U}(-1, 1), \quad y \sim \mathcal{U}(-1, 1),$$

lo que produce una distribución uniforme en toda el área del cuadrado.

Triángulo equilátero

La función `sample_uniform_triangle` genera puntos uniformes en un triángulo equilátero definido por los vértices

$$A = (-1, 0), \quad B = (1, 0), \quad C = (0, \sqrt{3}).$$

Se utilizan coordenadas baricéntricas: se toman $u, v \sim \mathcal{U}(0, 1)$ y se reflejan cuando $u + v > 1$ para asegurar que

$$u + v \leq 1.$$

Cada punto se obtiene como combinación convexa de los vértices:

$$P = A + u(B - A) + v(C - A),$$

lo que garantiza una distribución uniforme en el interior del triángulo.

```
In [22]: def sample_uniform_circle(n_samples: int) -> np.ndarray:
    """
    Genera puntos uniformes dentro del círculo unitario ( $x^2 + y^2 \leq 1$ ).

    Importante: para que sea uniforme en área, el radio se toma como  $r = \sqrt{U}$ ,
    con  $U \sim \text{Uniform}(0, 1)$ .
    """
    u = np.random.rand(n_samples)
    r = np.sqrt(u) # uniformidad en área
    theta = 2 * np.pi * np.random.rand(n_samples)

    x = r * np.cos(theta)
    y = r * np.sin(theta)

    return np.stack([x, y], axis=1)
```

```

def sample_uniform_square(n_samples: int) -> np.ndarray:
    """
    Genera puntos uniformes en el cuadrado [-1, 1] x [-1, 1].
    """
    x = np.random.uniform(-1, 1, size=n_samples)
    y = np.random.uniform(-1, 1, size=n_samples)
    return np.stack([x, y], axis=1)

def sample_uniform_triangle(n_samples: int) -> np.ndarray:
    """
    Genera puntos uniformes en un triángulo equilátero con vértices:
    A = (-1, 0), B = (1, 0), C = (0, sqrt(3)).
    Se usa muestreo uniforme con coordenadas baricéntricas.
    """
    A = np.array([-1.0, 0.0])
    B = np.array([1.0, 0.0])
    C = np.array([0.0, np.sqrt(3.0)])
    u = np.random.rand(n_samples)
    v = np.random.rand(n_samples)

    # Reflejo para asegurar que u + v <= 1 (uniforme en el triángulo)
    mask = (u + v > 1.0)
    u[mask] = 1.0 - u[mask]
    v[mask] = 1.0 - v[mask]

    points = (
        A[None, :]
        + u[:, None] * (B - A)[None, :]
        + v[:, None] * (C - A)[None, :]
    )
    return points

```

Implementación del mapa de Kohonen 2D (SOM2D)

Esta clase implementa desde cero un **mapa autoorganizado de Kohonen (SOM)** bidimensional usando únicamente `numpy`. La grilla de neuronas es de tamaño $m \times n$ y cada neurona tiene un vector de pesos de dimensión `input_dim` (en este trabajo, `input_dim = 2`).

Estructura de la red e inicialización

En el constructor `__init__`, se fijan:

- El tamaño de la grilla (m, n).
- El número total de iteraciones de entrenamiento `n_iterations`.
- La tasa de aprendizaje inicial α_0 .
- El radio de vecindad inicial σ_0 (por defecto, $\sigma_0 \approx \max(m, n)/2$).

Los pesos se inicializan aleatoriamente en el rango $[-1, 1]$ para cada neurona, y se almacenan en un tensor de forma $(m, n, \text{input_dim})$. Además, se guardan las coordenadas de cada neurona en la grilla:

$$\text{grid}[i, j] = (i, j).$$

Decaimiento de la tasa de aprendizaje y del radio de vecindad

Las funciones internas `_decay_alpha` y `_decay_sigma` implementan un decaimiento exponencial:

$$\alpha(t) = \alpha_0 e^{-t/N}, \quad \sigma(t) = \sigma_0 e^{-t/N},$$

donde N es el número total de iteraciones (`n_iterations`) y t es el índice de iteración. De este modo, el aprendizaje comienza siendo global (altos α y σ) y se vuelve progresivamente más local.

Unidad de mejor ajuste (BMU)

La función `_find_bmu(x)` calcula la **Best Matching Unit (BMU)** para un vector de entrada x :

1. Se computa la distancia euclídea al cuadrado entre x y todos los pesos:

$$d_{ij}^2 = \|x - w_{ij}\|^2.$$

2. Se selecciona la neurona (i^*, j^*) que minimiza esta distancia:

$$(i^*, j^*) = \arg \min_{i,j} d_{ij}^2.$$

Regla de entrenamiento

El método `train` recorre `n_iterations` y en cada iteración:

1. Toma una muestra aleatoria x del conjunto de datos.
2. Encuentra la BMU (i^*, j^*).
3. Calcula $\alpha(t)$ y $\sigma(t)$.
4. Evalúa la **función de vecindad gaussiana** entre la BMU y cada neurona (i, j) de la grilla:

$$\$ \Theta_{(i^*, j^*)}(i, j)(t) = \exp\left(-\frac{\|r_{i^*, j^*} - r_{i, j}\|^2}{2\sigma(t)^2}\right)$$

donde $r_{i,j} = (i, j)$ son las coordenadas de la neurona en la grilla.

5. Actualiza los pesos de todas las neuronas según la regla de Kohonen:

$$\$ w_{ij}(t+1) = w_{ij}(t) + \alpha(t) \cdot \Theta_{(i^*, j^*)}(i, j) \cdot (x - w_{ij}(t))$$

De este modo, la BMU y sus vecinas en la grilla se mueven hacia el patrón de entrada, preservando la topología: neuronas cercanas en la grilla tienden a especializarse en regiones cercanas del espacio de datos.

Utilidades

Las funciones `get_weights` y `get_flat_weights` devuelven los pesos aprendidos:

- `get_weights()` retorna un arreglo de forma $(m, n, \text{input_dim})$.
- `get_flat_weights()` los aplana a $(m \cdot n, \text{input_dim})$, lo que facilita ciertos gráficos y análisis posteriores.

```
In [23]: class SOM2D:
    """
    Implementación desde cero de un mapa autoorganizado de Kohonen (SOM) 2D
    usando únicamente numpy.

    - La grilla de neuronas es de tamaño m x n.
    - Cada neurona tiene un vector de pesos de dimensión `input_dim`.
    - Se usa aprendizaje competitivo con vecindad gaussiana.
    """

    def __init__(self, m: int, n: int, input_dim: int = 2, n_iterations: int = 10_000, alpha0: float = 0.5, sigma0: float | None = None):
        """
        Parámetros:
        -----
        m, n : tamaño de la grilla (m filas, n columnas)
        input_dim : dimensión de la entrada (en este TP = 2)
        n_iterations : número total de iteraciones de entrenamiento
        alpha0 : tasa de aprendizaje inicial
        sigma0 : radio de vecindad inicial en la grilla.
                  Si es None, se usa max(m, n) / 2.
        """
        self.m = m
        self.n = n
        self.input_dim = input_dim
        self.n_iterations = n_iterations
        self.alpha0 = alpha0
        self.sigma0 = sigma0 if sigma0 is not None else max(m, n) / 2.0

        # Inicialización de pesos en [-1, 1] para cada neurona
        self.weights = np.random.uniform(
            low=-1.0, high=1.0, size=(m, n, input_dim)
        )

        # Coordenadas fijas de cada neurona en la grilla: grid[i, j] = [i, j]
        self.grid = np.array([(i, j) for i in range(m) for j in range(n)])
        self.grid = self.grid.reshape(m, n, 2)

    # ----- Funciones internas de decaimiento -----
    def _decay_alpha(self, t: int) -> float:
        """
        Tasa de aprendizaje alpha(t) con decaimiento exponencial.
        """
        return self.alpha0 * np.exp(-t / self.n_iterations)

    def _decay_sigma(self, t: int) -> float:
        """
        Radio de vecindad sigma(t) con decaimiento exponencial.
        """
        return self.sigma0 * np.exp(-t / self.n_iterations)
```

```

# ----- BMU (Best Matching Unit) -----
def _find_bmu(self, x: np.ndarray) -> tuple[int, int]:
    """
    Dado un vector de entrada x (shape = (input_dim,)),
    devuelve los índices (i, j) de la neurona ganadora (BMU).

    La BMU es la neurona cuyo vector de pesos está a menor
    distancia euclídea de x.
    """
    # Diferencia entre x y todos los pesos: broadcasting
    diff = self.weights - x # shape: (m, n, input_dim)
    dist_sq = np.sum(diff**2, axis=2) # shape: (m, n)

    # Índice de la mínima distancia
    bmu_index = np.unravel_index(np.argmin(dist_sq), (self.m, self.n))
    return bmu_index

# ----- Entrenamiento -----
def train(self, data: np.ndarray, verbose: bool = True) -> None:
    """
    Entrena la red con los datos de entrada.

    data: array de shape (N, input_dim)
    verbose: si True, imprime progreso cada cierto número de iteraciones.
    """
    n_samples = data.shape[0]

    for t in range(self.n_iterations):
        # 1. Elegimos una muestra aleatoria del dataset
        x = data[np.random.randint(0, n_samples)]

        # 2. Buscamos la BMU para x
        bmu_i, bmu_j = self._find_bmu(x)

        # 3. Calculamos alpha(t) y sigma(t)
        alpha_t = self._decay_alpha(t)
        sigma_t = self._decay_sigma(t)

        # 4. Calculamos la función de vecindad gaussiana theta_{u,v}(t)
        bmu_pos = np.array([bmu_i, bmu_j]) # posición de la BMU en la grilla

        # Distancia al cuadrado en la grilla entre cada neurona y la BMU
        diff_grid = self.grid - bmu_pos # shape: (m, n, 2)
        dist_sq_grid = np.sum(diff_grid**2, axis=2) # shape: (m, n)

        # Función de vecindad gaussiana
        theta = np.exp(-dist_sq_grid / (2 * (sigma_t**2))) # shape: (m, n)

        # 5. Actualizamos pesos:
        # w_v(t+1) = w_v(t) + alpha(t)*theta_{u,v}(t)*(x - w_v(t))
        theta_expanded = theta[..., np.newaxis] # shape: (m, n, 1)
        self.weights += alpha_t * theta_expanded * (x - self.weights)

        # Mensaje de progreso
        if verbose and (t + 1) % (self.n_iterations // 5) == 0:
            print(
                f"Iteración {t + 1}/{self.n_iterations} "
                f"- alpha={alpha_t:.4f}, sigma={sigma_t:.4f}"
            )

# ----- Utilidades -----
def get_weights(self) -> np.ndarray:
    """
    Devuelve una copia de los pesos actuales del SOM.
    Shape: (m, n, input_dim)
    """
    return self.weights.copy()

def get_flat_weights(self) -> np.ndarray:
    """
    Devuelve los pesos aplanados como (m*n, input_dim),
    útil para algunos gráficos o análisis.
    """
    return self.weights.reshape(-1, self.input_dim)

```

Visualización de la preservación de topología

La función `plot_som_topology` grafica en el plano:

- Los datos de entrenamiento `data` (puntos en \mathbb{R}^2).
- Los pesos finales del SOM, organizados en una grilla de tamaño $m \times n$.
- Segmentos que conectan neuronas vecinas en la grilla (abajo y derecha), formando una "malla" deformada.

Si la red de Kohonen preserva la topología, esta malla se adapta a la forma de la distribución de datos (por ejemplo, el círculo unitario), de modo que neuronas vecinas en la grilla (i, j) siguen siendo vecinas en el espacio de entrada, ilustrando la propiedad de **mapa**.

autoorganizado.

```
In [24]: def plot_som_topology(
    som: SOM2D,
    data: np.ndarray,
    title: str = "SOM - preservación de topología",
    data_alpha: float = 0.3,
) -> None:
    """
    Dibuja:
    - Los datos de entrenamiento (puntos en el plano)
    - Los pesos finales del SOM (neuronas)
    - Las conexiones entre neuronas vecinas en la grilla (malla),
    que ilustran la preservación de la topología.

    som : instancia entrenada de SOM2D
    data : array (N, 2) con los datos de entrada
    """
    weights = som.get_weights() # shape: (m, n, 2)
    m, n, _ = weights.shape

    fig, ax = plt.subplots(figsize=(6, 6))

    # Puntos de datos
    ax.scatter(
        data[:, 0],
        data[:, 1],
        s=5,
        alpha=data_alpha,
        label="Datos de entrenamiento",
    )

    # Neuronas (pesos)
    ax.scatter(
        weights[..., 0].ravel(),
        weights[..., 1].ravel(),
        s=30,
        label="Neuronas (pesos del SOM)",
    )

    # Conectar neuronas vecinas para ver cómo se deforma la grilla
    for i in range(m):
        for j in range(n):
            # vecino abajo
            if i + 1 < m:
                p1 = weights[i, j]
                p2 = weights[i + 1, j]
                ax.plot(
                    [p1[0], p2[0]],
                    [p1[1], p2[1]],
                    linewidth=1,
                )
            # vecino derecha
            if j + 1 < n:
                p1 = weights[i, j]
                p2 = weights[i, j + 1]
                ax.plot(
                    [p1[0], p2[0]],
                    [p1[1], p2[1]],
                    linewidth=1,
                )

    ax.set_title(title)
    ax.set_xlabel("x1")
    ax.set_ylabel("x2")
    ax.set_aspect("equal")
    ax.legend()
    ax.grid(True, alpha=0.2)
    plt.show()
```

```
In [25]: # ===== Experimentos SOM con varias variantes por figura (círculo, cuadrado, triángulo) =====

n_samples = 5000

# Definimos las distribuciones
shapes = [
    ("Círculo unitario", sample_uniform_circle),
    ("Cuadrado [-1,1] x [-1,1]", sample_uniform_square),
    ("Triángulo equilátero", sample_uniform_triangle),
]

# Variantes de SOM a probar en cada distribución
variants = [
    {
        "label": "SOM 10x10, iter=5 000, sigma0=5",
        "m": 10,
        "n": 10,
        "n_iterations": 5_000,
        "alpha0": 0.5,
        "sigma0": 5.0,
    }
]
```

```

},
{
  "label": "SOM 20x20, iter=10 000, sigma0=10 (baseline)",
  "m": 20,
  "n": 20,
  "n_iterations": 10_000,
  "alpha0": 0.5,
  "sigma0": 10.0,
},
{
  "label": "SOM 30x30, iter=15 000, sigma0=12",
  "m": 30,
  "n": 30,
  "n_iterations": 15_000,
  "alpha0": 0.5,
  "sigma0": 12.0,
},
],
]

for shape_name, sampler in shapes:
  print(f"\n\n===== {shape_name} =====")
  data = sampler(n_samples)

  for v in variants:
    print(f"\n--- Variante: {v['label']} ---")
    som = SOM2D(
      m=v["m"],
      n=v["n"],
      input_dim=2,
      n_iterations=v["n_iterations"],
      alpha0=v["alpha0"],
      sigma0=v["sigma0"],
    )

    som.train(data, verbose=True)

    plot_title = (
      f"{shape_name} - {v['label']}"
    )
    plot_som_topology(
      som,
      data,
      title=plot_title,
      data_alpha=0.3,
    )
  )

```

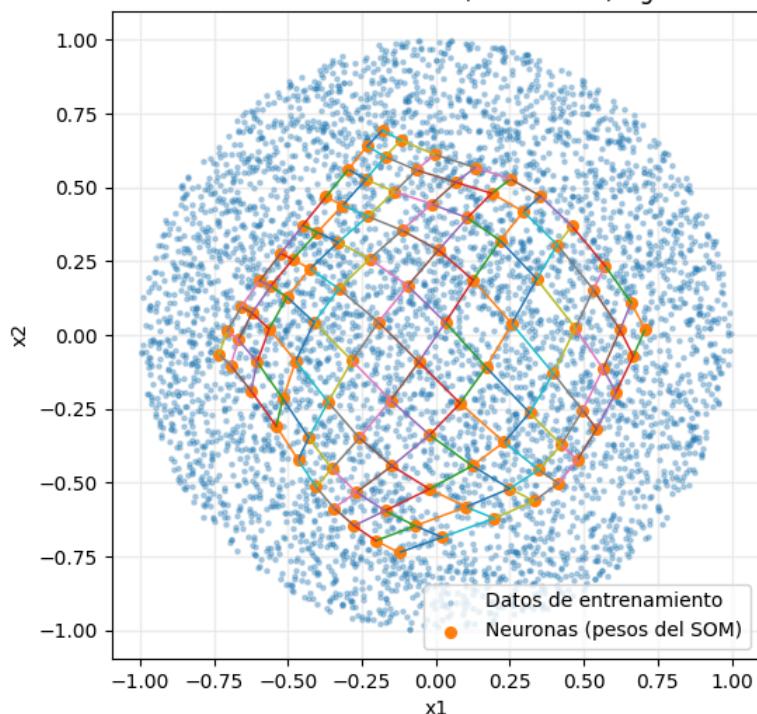
===== Círculo unitario =====

```

--- Variante: SOM 10x10, iter=5 000, sigma0=5 ---
Iteración 1000/5000 - alpha=0.4094, sigma=4.0945
Iteración 2000/5000 - alpha=0.3352, sigma=3.3523
Iteración 3000/5000 - alpha=0.2745, sigma=2.7446
Iteración 4000/5000 - alpha=0.2247, sigma=2.2471
Iteración 5000/5000 - alpha=0.1840, sigma=1.8398

```

Círculo unitario - SOM 10x10, iter=5 000, sigma0=5

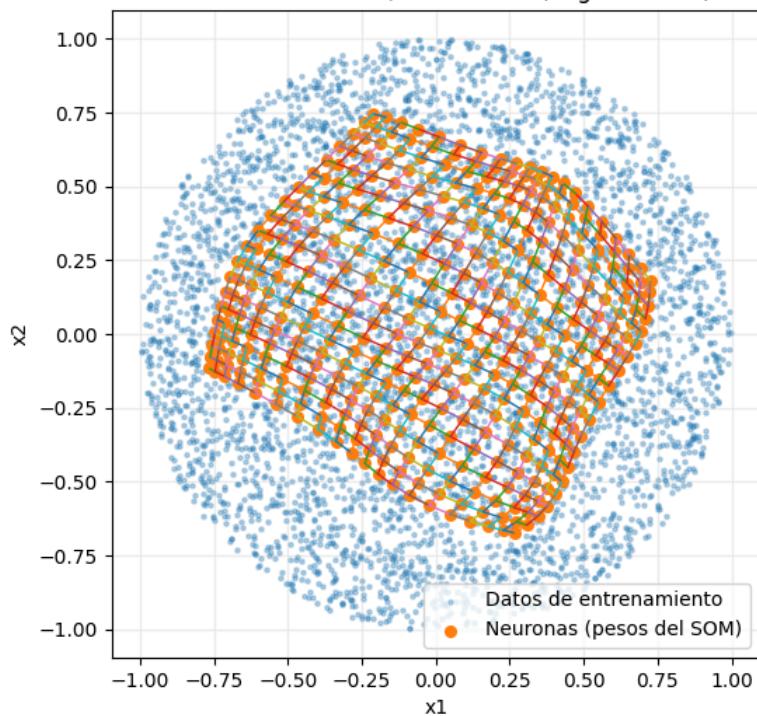


```

--- Variante: SOM 20x20, iter=10 000, sigma0=10 (baseline) ---
Iteración 2000/10000 - alpha=0.4094, sigma=8.1881
Iteración 4000/10000 - alpha=0.3352, sigma=6.7039
Iteración 6000/10000 - alpha=0.2744, sigma=5.4887
Iteración 8000/10000 - alpha=0.2247, sigma=4.4937
Iteración 10000/10000 - alpha=0.1840, sigma=3.6792

```

Círculo unitario - SOM 20x20, iter=10 000, sigma0=10 (baseline)

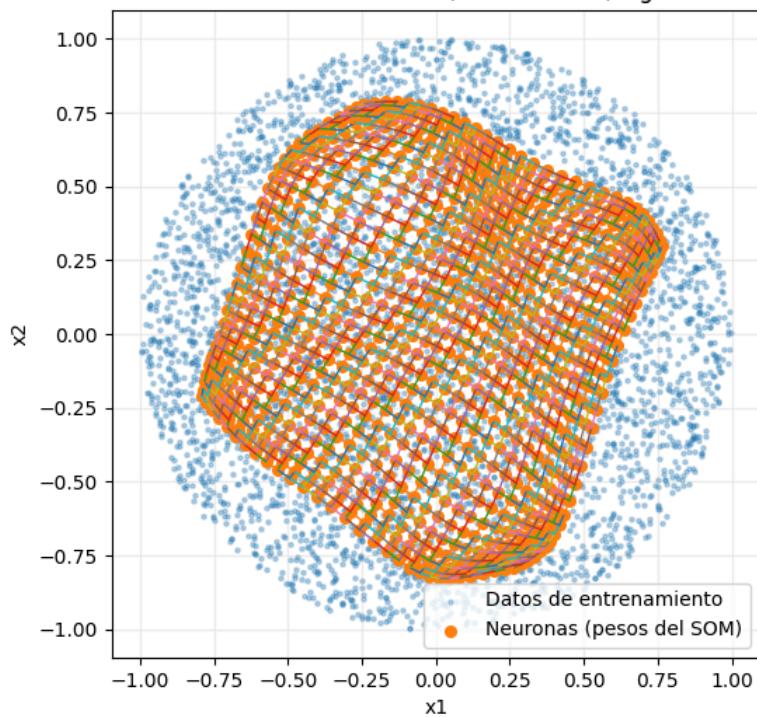


```

--- Variante: SOM 30x30, iter=15 000, sigma0=12 ---
Iteración 3000/15000 - alpha=0.4094, sigma=9.8254
Iteración 6000/15000 - alpha=0.3352, sigma=8.0444
Iteración 9000/15000 - alpha=0.2744, sigma=6.5862
Iteración 12000/15000 - alpha=0.2247, sigma=5.3923
Iteración 15000/15000 - alpha=0.1840, sigma=4.4148

```

Círculo unitario - SOM 30x30, iter=15 000, sigma0=12



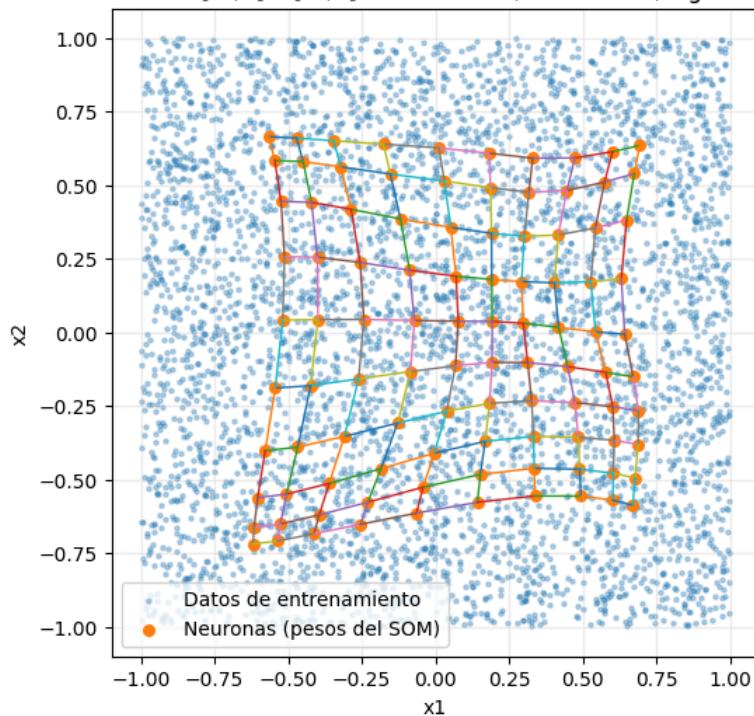
```
===== Cuadrado [-1,1] x [-1,1] =====
```

```

--- Variante: SOM 10x10, iter=5 000, sigma0=5 ---
Iteración 1000/5000 - alpha=0.4094, sigma=4.0945
Iteración 2000/5000 - alpha=0.3352, sigma=3.3523
Iteración 3000/5000 - alpha=0.2745, sigma=2.7446
Iteración 4000/5000 - alpha=0.2247, sigma=2.2471
Iteración 5000/5000 - alpha=0.1840, sigma=1.8398

```

Cuadrado [-1,1] x [-1,1] - SOM 10x10, iter=5 000, sigma0=5



--- Variante: SOM 20x20, iter=10 000, sigma0=10 (baseline) ---

Iteración 2000/10000 - alpha=0.4094, sigma=8.1881

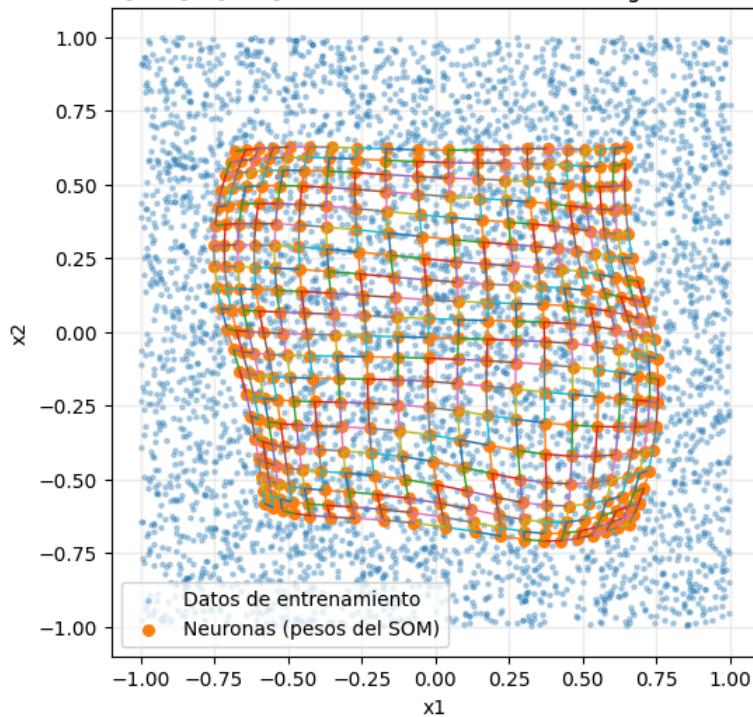
Iteración 4000/10000 - alpha=0.3352, sigma=6.7039

Iteración 6000/10000 - alpha=0.2744, sigma=5.4887

Iteración 8000/10000 - alpha=0.2247, sigma=4.4937

Iteración 10000/10000 - alpha=0.1840, sigma=3.6792

Cuadrado [-1,1] x [-1,1] - SOM 20x20, iter=10 000, sigma0=10 (baseline)



--- Variante: SOM 30x30, iter=15 000, sigma0=12 ---

Iteración 3000/15000 - alpha=0.4094, sigma=9.8254

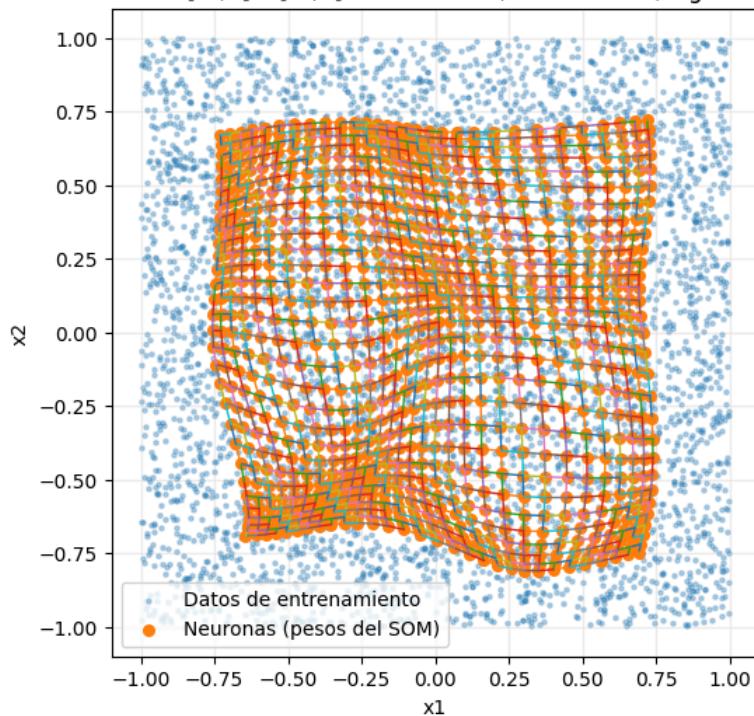
Iteración 6000/15000 - alpha=0.3352, sigma=8.0444

Iteración 9000/15000 - alpha=0.2744, sigma=6.5862

Iteración 12000/15000 - alpha=0.2247, sigma=5.3923

Iteración 15000/15000 - alpha=0.1840, sigma=4.4148

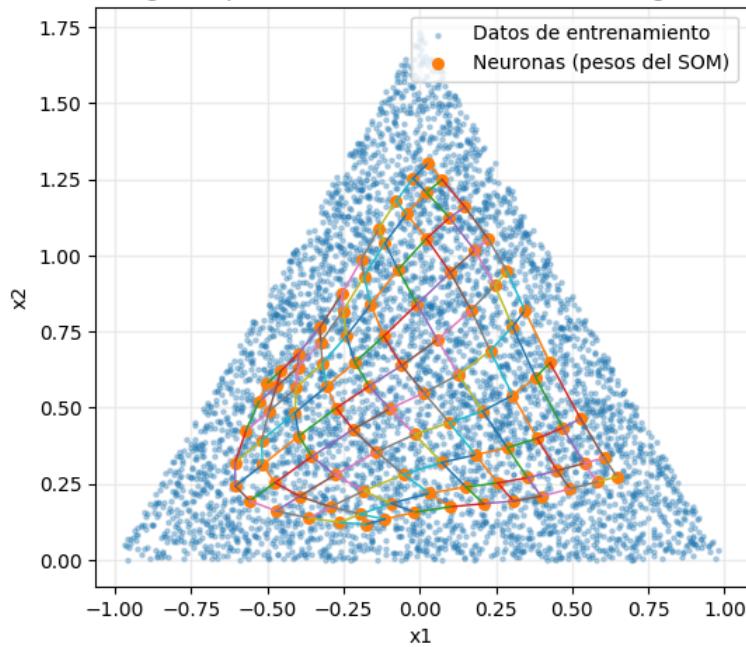
Cuadrado [-1,1] x [-1,1] - SOM 30x30, iter=15 000, sigma0=12



===== Triángulo equilátero =====

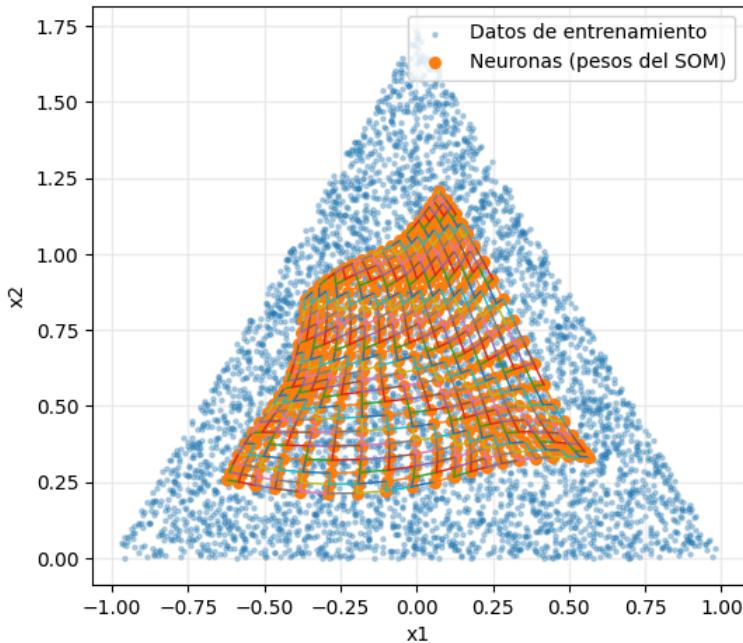
--- Variante: SOM 10x10, iter=5 000, sigma0=5 ---
 Iteración 1000/5000 - alpha=0.4094, sigma=4.0945
 Iteración 2000/5000 - alpha=0.3352, sigma=3.3523
 Iteración 3000/5000 - alpha=0.2745, sigma=2.7446
 Iteración 4000/5000 - alpha=0.2247, sigma=2.2471
 Iteración 5000/5000 - alpha=0.1840, sigma=1.8398

Triángulo equilátero - SOM 10x10, iter=5 000, sigma0=5



--- Variante: SOM 20x20, iter=10 000, sigma0=10 (baseline) ---
 Iteración 2000/10000 - alpha=0.4094, sigma=8.1881
 Iteración 4000/10000 - alpha=0.3352, sigma=6.7039
 Iteración 6000/10000 - alpha=0.2744, sigma=5.4887
 Iteración 8000/10000 - alpha=0.2247, sigma=4.4937
 Iteración 10000/10000 - alpha=0.1840, sigma=3.6792

Triángulo equilátero - SOM 20x20, iter=10 000, sigma0=10 (baseline)



--- Variante: SOM 30x30, iter=15 000, sigma0=12 ---

Iteración 3000/15000 - alpha=0.4094, sigma=9.8254

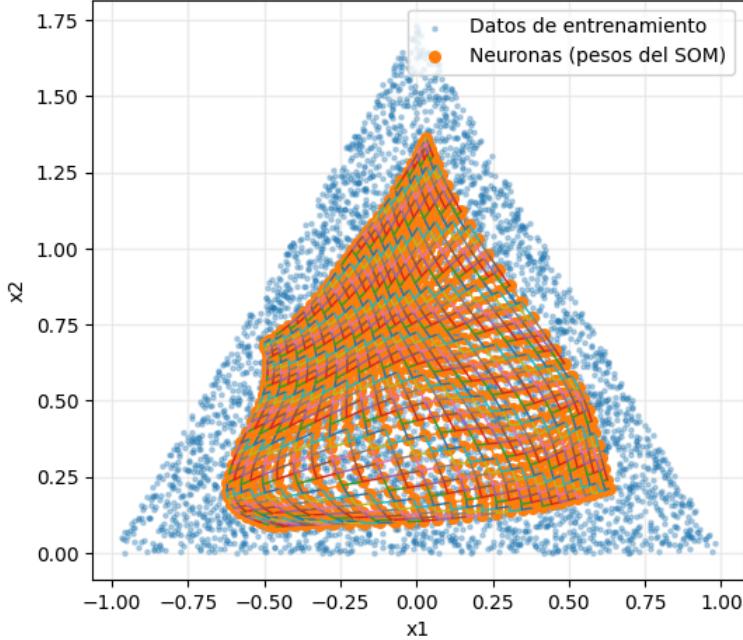
Iteración 6000/15000 - alpha=0.3352, sigma=8.0444

Iteración 9000/15000 - alpha=0.2744, sigma=6.5862

Iteración 12000/15000 - alpha=0.2247, sigma=5.3923

Iteración 15000/15000 - alpha=0.1840, sigma=4.4148

Triángulo equilátero - SOM 30x30, iter=15 000, sigma0=12



Conclusiones sobre los experimentos con SOM

Se evaluó el comportamiento de un mapa de Kohonen bidimensional sobre tres distribuciones uniformes (círculo unitario, cuadrado $[-1, 1]^2$ y triángulo equilátero) y tres configuraciones de red:

- SOM pequeño: 10×10 , 5 000 iteraciones, $\sigma_0 = 5$.
- SOM intermedio (baseline): 20×20 , 10 000 iteraciones, $\sigma_0 = 10$.
- SOM grande: 30×30 , 15 000 iteraciones, $\sigma_0 = 12$.

En todos los casos el mapa preserva la topología: la malla de neuronas se mantiene continua y sin cruces bruscos, y se deforma para aproximar el soporte de los datos.

Círculo unitario

- El SOM pequeño produce una malla relativamente "rígida": cubre el interior del disco pero la grilla se mantiene bastante cuadrada y la aproximación de la frontera circular es gruesa.
- El SOM intermedio mejora la adaptación a la forma del círculo: los bordes de la malla se curvan y aparecen anillos de neuronas más pegados a la frontera.

- El SOM grande logra la mayor resolución espacial. La malla se curva de forma suave y hay más neuronas ocupando regiones periféricas, lo que permite distinguir mejor el gradiente de densidad entre el centro y el borde. A cambio, el entrenamiento es más costoso y la malla puede mostrar pequeñas ondulaciones locales.

Cuadrado $[-1, 1]^2$

- En el cuadrado, los tres SOM convergen a mallas que se asemejan a una grilla rectangular. La versión 10×10 cubre bien el área, pero deja zonas poco representadas en los vértices.
- El SOM 20×20 ofrece un compromiso razonable: la malla es casi regular y las neuronas se distribuyen de manera bastante uniforme sobre toda la superficie.
- Con el SOM 30×30 se consigue la cobertura más uniforme del cuadrado, aunque se observan pequeñas deformaciones en los bordes debido a la interacción entre un σ_0 relativamente grande y un número alto de iteraciones.

Triángulo equilátero

- Para el triángulo la geometría es más exigente: la malla debe plegarse para seguir los lados inclinados y el vértice superior.
- El SOM pequeño tiende a concentrar neuronas en la base y en la región central, dejando el vértice superior menos representado y produciendo una malla algo "achatada".
- El SOM intermedio mejora la cobertura del interior del triángulo, aunque todavía se observa cierta pérdida de resolución en los vértices.
- El SOM grande permite seguir con más detalle los lados del triángulo, pero al mismo tiempo puede acumular neuronas en zonas de mayor densidad de muestras, dejando algunos extremos algo menos poblados si el entrenamiento no es lo suficientemente largo.

Comparación general

- Aumentar el tamaño del mapa ($10 \times 10 \rightarrow 30 \times 30$) mejora la **resolución** de la representación (más neuronas, más detalle), pero encarece el entrenamiento y puede introducir pequeñas **ondas** o deformaciones locales si los parámetros de $\alpha(t)$ y $\sigma(t)$ no se ajustan.
- El SOM intermedio 20×20 ofrece, en estas pruebas, el mejor **compromiso** entre calidad del mapa y coste computacional, capturando la forma global de las distribuciones sin distorsiones graves.
- Las figuras no rectangulares (círculo y triángulo) evidencian mejor la capacidad del SOM para plegarse y aproximar fronteras curvas o con ángulos, manteniendo la propiedad de **preservación de vecindad** que caracteriza a los mapas de Kohonen.

2. Resuelva (aproximadamente) el “Traveling salesman problem” para 200 ciudades con una red de Kohonen.

```
In [26]: import numpy as np
import matplotlib.pyplot as plt

# Para reproducibilidad, si querés:
np.random.seed(42)
```

Red de Kohonen 1D en anillo para el Traveling Salesman Problem (TSP)

El objetivo del TSP es encontrar un recorrido cerrado que visite una vez cada ciudad minimizando la longitud total del camino. Si denotamos por $c_1, \dots, c_N \in \mathbb{R}^2$ las posiciones de las N ciudades, un tour es una permutación π de $\{1, \dots, N\}$, y su longitud es

$$L(\pi) = \sum_{i=1}^N \|c_{\pi(i)} - c_{\pi(i+1)}\|, \quad \pi(N+1) = \pi(1).$$

Este problema es NP-duro, por lo que se utilizan métodos aproximados.

Una alternativa es usar un **mapa de Kohonen 1D en forma de anillo**, que se deforma para "rodear" las ciudades.

Representación de la red

La clase `TSPSOM` implementa una red de Kohonen con:

- $K = n_neurons$ neuronas dispuestas en un **anillo** (índices $k = 0, \dots, K-1$).
- Cada neurona k tiene un vector de pesos

$$\mathbf{w}_k(t) \in \mathbb{R}^2,$$

que representa una posición en el mismo plano que las ciudades.

- El conjunto de pesos en el tiempo t es

$$W(t) = \{\mathbf{w}_0(t), \dots, \mathbf{w}_{K-1}(t)\}.$$

Inicialmente, los pesos se eligen aleatoriamente en el cuadrado $[0, 1]^2$, y durante el entrenamiento el anillo se va adaptando a la nube de ciudades.

Unidad de Mejor Ajuste (BMU)

Dada una ciudad $\mathbf{x} \in \mathbb{R}^2$, la red busca la **Best Matching Unit (BMU)**: la neurona cuyos pesos están más cerca de \mathbf{x} en norma euclídea. Es decir,

$$k^*(\mathbf{x}) = \arg \min_{k \in \{0, \dots, K-1\}} \|\mathbf{x} - \mathbf{w}_k(t)\|^2.$$

En el código, esto se implementa en el método `_find_bmu_idx`.

Topología en anillo y distancia circular

Las neuronas no solo tienen pesos en el espacio de ciudades, sino también una posición en el **índice topológico** del anillo:

$$k = 0, 1, \dots, K-1.$$

La distancia entre neuronas en esa topología circular se define como

$$d_{\text{ring}}(k, k^*) = \min(|k - k^*|, K - |k - k^*|),$$

lo que refleja la estructura de anillo (la neurona 0 es vecina de la neurona $K-1$).

Esta distancia se usa para definir la vecindad de la BMU sobre el anillo.

Decaimiento de la tasa de aprendizaje y del radio de vecindad

La red utiliza una tasa de aprendizaje $\alpha(t)$ y un radio de vecindad $\sigma(t)$ que decrecen exponencialmente con el tiempo:

$$\alpha(t) = \alpha_0 e^{-t/T}, \quad \sigma(t) = \sigma_0 e^{-t/T},$$

donde:

- $\alpha_0 = \text{alpha0}$ es la tasa de aprendizaje inicial,
- $\sigma_0 = \text{sigma0}$ es el radio inicial de vecindad,
- $T = \text{n_iterations}$ es el número total de iteraciones.

Al inicio, la actualización es más "global" (gran $\sigma(t)$), y hacia el final se vuelve más "local", afinando la solución.

Función de vecindad gaussiana en el anillo

Para una ciudad \mathbf{x} con BMU k^* , la **función de vecindad** entre la BMU y una neurona k viene dada por:

$$h_{k^*,k}(t) = \exp\left(-\frac{d_{\text{ring}}(k, k^*)^2}{2\sigma(t)^2}\right).$$

Esta función es máxima en la BMU ($k = k^*$) y decrece suavemente a medida que k se aleja sobre el anillo, controlada por $\sigma(t)$.

Regla de aprendizaje de Kohonen

En cada iteración t del entrenamiento:

1. Se selecciona una ciudad \mathbf{x} al azar del conjunto de ciudades.
2. Se calcula la BMU $k^*(\mathbf{x})$.
3. Se actualiza el vector de pesos de cada neurona k según:

$$\mathbf{w}_k(t+1) = \mathbf{w}_k(t) + \alpha(t) h_{k^*,k}(t) (\mathbf{x} - \mathbf{w}_k(t)).$$

Esta regla mueve la BMU y sus vecinas en el anillo hacia la ciudad \mathbf{x} , haciendo que la curva formada por los $\mathbf{w}_k(t)$ se adapte progresivamente a la distribución de ciudades. Como la topología es 1D circular, la red tiende a formar un **círculo cerrado** que atraviesa la región donde se encuentran las ciudades.

Construcción del recorrido TSP aproximado

Una vez entrenada la red, se obtiene un recorrido TSP aproximado a partir de la correspondencia entre ciudades y neuronas:

1. Para cada ciudad \mathbf{c}_i , se calcula el índice de la BMU:

$$k_i = k^*(\mathbf{c}_i).$$

2. Se forman pares (k_i, i) y se ordenan según k_i :

$$(k_{(1)}, i_{(1)}), \dots, (k_{(N)}, i_{(N)}) \quad \text{con} \quad k_{(1)} \leq k_{(2)} \leq \dots \leq k_{(N)}.$$

3. El orden de ciudades

$$\pi = (i_{(1)}, i_{(2)}, \dots, i_{(N)})$$

define un tour cerrado al volver de $i_{(N)}$ a $i_{(1)}$.

Este procedimiento explota la **preservación de vecindad** del mapa de Kohonen: ciudades que terminan asociadas a neuronas cercanas en el anillo tienden a estar cercanas en el plano, lo que produce trayectorias razonablemente cortas.

Resumen conceptual

- El TSP se modela mediante una **curva cerrada** (el anillo de neuronas) que se entrena para aproximar la distribución de las ciudades.
- La red de Kohonen 1D minimiza de forma no supervisada una combinación de:
 - distancia entre ciudades y neuronas (ajuste de los pesos),
 - suavidad/topología del anillo (vecindad gaussiana).
- El recorrido final se obtiene ordenando las ciudades según la posición de su BMU sobre el anillo, dando una solución aproximada al TSP.

In [27]:

```
class TSPSOM:  
    """  
        Mapa de Kohonen 1D en forma de anillo para aproximar el TSP.  
  
        - Tenemos n_neurons neuronas dispuestas en un círculo (índices 0..n_neurons-1).  
        - Cada neurona tiene un vector de pesos en R^2 que vive en el mismo espacio  
            que las ciudades.  
        - La vecindad se define sobre el índice circular (distancia en el anillo).  
    """  
  
    def __init__(  
        self,  
        n_neurons: int,  
        input_dim: int = 2,  
        n_iterations: int = 20_000,  
        alpha0: float = 0.8,  
        sigma0: float | None = None,  
    ):  
        """  
            Parámetros  
            -----  
            n_neurons : cantidad de neuronas en el anillo.  
                Suele tomarse un múltiplo de la cantidad de ciudades  
                (e.g. 5-10 veces más).  
            input_dim : dimensión de entrada (para TSP en el plano = 2).  
            n_iterations : número total de iteraciones de entrenamiento.  
            alpha0 : tasa de aprendizaje inicial.  
            sigma0 : radio de vecindad inicial en el anillo.  
                Si es None, se usa n_neurons / 2.  
        """  
        self.n_neurons = n_neurons  
        self.input_dim = input_dim  
        self.n_iterations = n_iterations  
        self.alpha0 = alpha0  
        self.sigma0 = sigma0 if sigma0 is not None else n_neurons / 2.0  
  
        # Pesos iniciales en el cuadrado [0,1] x [0,1]  
        # (después se van a adaptar hacia las ciudades reales)  
        self.weights = np.random.rand(n_neurons, input_dim)  
  
        # Índices de neuronas (para calcular distancia en el anillo)  
        self.neuron_indices = np.arange(n_neurons)  
  
    # ----- Decaimiento de hiperparámetros -----  
  
    def _decay_alpha(self, t: int) -> float:  
        """  
            Tasa de aprendizaje alpha(t) con decaimiento exponencial.  
        """  
        return self.alpha0 * np.exp(-t / self.n_iterations)  
  
    def _decay_sigma(self, t: int) -> float:  
        """  
            Radio de vecindad sigma(t) con decaimiento exponencial.  
        """  
        return self.sigma0 * np.exp(-t / self.n_iterations)  
  
    # ----- BMU (Best Matching Unit) -----  
  
    def _find_bmu_idx(self, x: np.ndarray) -> int:  
        """  
            Dado un vector de entrada x (shape = (input_dim,)),  
            devuelve el índice k de la neurona ganadora (BMU).  
  
            La BMU es la neurona cuyo vector de pesos está a menor  
            distancia euclídea de x.  
        """  
        diff = self.weights - x # shape: (n_neurons, input_dim)
```

```

dist_sq = np.sum(diff**2, axis=1) # shape: (n_neurons,)
bmu_idx = np.argmin(dist_sq)
return bmu_idx

# ----- Entrenamiento -----

def train(self, cities: np.ndarray, verbose: bool = True) -> None:
    """
    Entrena el SOM sobre el conjunto de ciudades.

    cities: array de shape (N_cities, 2)
    """
    n_cities = cities.shape[0]

    for t in range(self.n_iterations):
        # 1. Elegimos una ciudad aleatoria
        x = cities[np.random.randint(0, n_cities)]

        # 2. BMU para esta ciudad
        bmu_idx = self._find_bmu_idx(x)

        # 3. Hiperparámetros en esta iteración
        alpha_t = self._decay_alpha(t)
        sigma_t = self._decay_sigma(t)

        # 4. Distancia circular en el anillo entre cada neurona y la BMU
        #     dist_ring(k, bmu) = min(|k - bmu|, n_neurons - |k - bmu|)
        dist = np.abs(self.neuron_indices - bmu_idx)
        dist_ring = np.minimum(dist, self.n_neurons - dist)

        # 5. Función de vecindad gaussiana en el anillo
        neighborhood = np.exp(- (dist_ring**2) / (2 * (sigma_t**2))) # shape: (n_neurons,)

        # 6. Actualización de los pesos (regla de Kohonen)
        self.weights += alpha_t * neighborhood[:, np.newaxis] * (x - self.weights)

        # Progreso
        if verbose and (t + 1) % (self.n_iterations // 5) == 0:
            print(
                f"Iteración {t + 1}/{self.n_iterations} "
                f"- alpha={alpha_t:.4f}, sigma={sigma_t:.4f}"
            )

# ----- Utilidades -----


def get_weights(self) -> np.ndarray:
    """
    Devuelve una copia de los pesos actuales del SOM.
    Shape: (n_neurons, input_dim)
    """
    return self.weights.copy()

def bmu_indices_for_cities(self, cities: np.ndarray) -> np.ndarray:
    """
    Devuelve, para cada ciudad, el índice de la neurona BMU.

    cities: (N_cities, 2)
    return: (N_cities,) con índices en [0, n_neurons-1]
    """
    bmu_indices = []
    for city in cities:
        bmu_indices.append(self._find_bmu_idx(city))
    return np.array(bmu_indices, dtype=int)

def get_tour(self, cities: np.ndarray) -> list[int]:
    """
    Construye un recorrido TSP aproximado a partir del SOM entrenado.

    Estrategia:
    - Para cada ciudad, se calcula la BMU (índice k).
    - Se ordenan las ciudades según ese índice k a lo largo del anillo.
    - El resultado es una permutación de los índices de ciudades
      que define un tour cerrado.
    """
    bmu_indices = self.bmu_indices_for_cities(cities)

    # Lista de pares (índice_neurona, índice_ciudad)
    pairs = list(zip(bmu_indices, range(len(cities))))

    # Ordenar por índice de neurona a lo largo del anillo
    pairs.sort(key=lambda p: p[0])

    # Extraer el orden de ciudades
    tour = [city_idx for _, city_idx in pairs]
    return tour

```

Cálculo y visualización del tour TSP

La función `tour_length` calcula la longitud total de un **tour cerrado** sobre el conjunto de ciudades. Dadas las posiciones $\{\mathbf{c}_1, \dots, \mathbf{c}_N\} \subset \mathbb{R}^2$ y un tour π (permutación de $\{1, \dots, N\}$), la longitud se define como

$$L(\pi) = \sum_{i=1}^N \|\mathbf{c}_{\pi(i)} - \mathbf{c}_{\pi(i+1)}\|, \quad \pi(N+1) = \pi(1),$$

es decir, se suma la distancia euclídea entre ciudades consecutivas y se cierra el ciclo volviendo de la última ciudad a la primera.

La función `plot_tsp_tour` representa gráficamente este recorrido:

- Dibuja las ciudades como puntos en el plano.
- Trazo el tour como una **polilínea cerrada** que conecta las ciudades en el orden dado por π .
- Opcionalmente, si se pasa una instancia de `TSPSOM`, grafica también el **anillo de neuronas** (curva punteada) para visualizar cómo el mapa de Kohonen se adaptó a la distribución de ciudades y al recorrido obtenido.

```
In [28]: def tour_length(cities: np.ndarray, tour: list[int]) -> float:
    """
    Calcula la longitud total de un tour cerrado sobre las ciudades.

    cities: array (N_cities, 2)
    tour: lista de índices de ciudades (permutación de 0..N-1)
    """
    total = 0.0
    n = len(tour)
    for i in range(n):
        c1 = cities[tour[i]]
        c2 = cities[tour[(i + 1) % n]] # volver al inicio
        total += np.linalg.norm(c1 - c2)
    return total

def plot_tsp_tour(
    cities: np.ndarray,
    tour: list[int],
    som: TSPSOM | None = None,
    title: str = "Recorrido TSP aproximado con SOM",
) -> None:
    """
    Grafica:
    - Las ciudades como puntos.
    - El recorrido TSP aproximado como una polilínea cerrada.
    - Opcionalmente, el anillo de neuronas del SOM.

    som: instancia entrenada de TSPSOM (opcional).
    """
    fig, ax = plt.subplots(figsize=(6, 6))

    # Ciudades
    ax.scatter(
        cities[:, 0],
        cities[:, 1],
        s=30,
        label="Ciudades",
    )

    # Recorrido (polilínea cerrada)
    closed_tour = tour + [tour[0]]
    ax.plot(
        cities[closed_tour, 0],
        cities[closed_tour, 1],
        linewidth=1.5,
        marker="o",
        markersize=4,
        label="Recorrido SOM",
    )

    # Anillo de neuronas (opcional, para ver cómo se adaptó)
    if som is not None:
        weights = som.get_weights()
        closed_ring = np.vstack([weights, weights[0:1, :]]) # curva cerrada
        ax.plot(
            closed_ring[:, 0],
            closed_ring[:, 1],
            linestyle="--",
            linewidth=1,
            label="Anillo SOM",
        )

    ax.set_title(title)
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_aspect("equal")
    ax.grid(True, alpha=0.2)
    ax.legend()
    plt.show()
```

Experimento: aproximación del TSP para 200 ciudades con un SOM en anillo

Para evaluar el desempeño de la red de Kohonen 1D en la aproximación del Traveling Salesman Problem, se generó un conjunto sintético de $N = 200$ ciudades, muestreadas de manera uniforme en el cuadrado $[0, 1] \times [0, 1]$.

Sobre este conjunto se entrenaron varias instancias de la red `TSPSOM`, variando los hiperparámetros principales:

- el tamaño del mapa K (número de neuronas en el anillo, por ejemplo $K = 2N, 4N, 8N, 16N$),
- la tasa de aprendizaje inicial α_0 ,
- el radio inicial de vecindad σ_0 (expresado como fracción de K y decreciendo exponencialmente),
- y el número total de iteraciones T .

En todos los casos la red mantiene dimensión de entrada $d = 2$ (coordenadas de las ciudades). Tras el entrenamiento, se construyó un recorrido aproximado ordenando las ciudades según el índice de la neurona BMU asociada a cada una. La calidad de la solución se evaluó mediante la longitud total del tour

$$L(\pi) = \sum_{i=1}^N \|\mathbf{c}_{\pi(i)} - \mathbf{c}_{\pi(i+1)}\|, \quad \pi(N+1) = \pi(1),$$

comparando, para cada configuración:

- L_{SOM} : longitud del tour inducido por el mapa de Kohonen.
- L_{rand} : longitud de un tour aleatorio (permutación uniforme de las ciudades).

Asimismo, se contabilizó el número de **neuronas activas**, es decir, neuronas a las que quedó asociada al menos una ciudad, como indicador de cuán eficientemente se utiliza el anillo para representar el conjunto de puntos. Para cada corrida se generó una visualización que muestra simultáneamente las ciudades, el tour obtenido y el anillo de neuronas entrenado, lo que permite comparar cualitativamente el efecto de los distintos hiperparámetros sobre la forma del recorrido.

```
In [29]: # ===== Experimento: TSP aproximado para 200 ciudades con varias configuraciones de SOM =====

np.random.seed(42)

# 1. Generamos 200 ciudades aleatorias en el cuadrado [0,1] x [0,1]
n_cities = 200
cities = np.random.rand(n_cities, 2)

# 2. Definimos un conjunto de configuraciones a probar
configs = [
    {
        "label": "K = 2N, alpha0=0.5, sigma0 = K/8",
        "n_neurons": 2 * n_cities,
        "n_iterations": 15_000,
        "alpha0": 0.5,
        "sigma0_factor": 1/8,
    },
    {
        "label": "K = 4N, alpha0=0.5, sigma0 = K/8",
        "n_neurons": 4 * n_cities,
        "n_iterations": 20_000,
        "alpha0": 0.5,
        "sigma0_factor": 1/8,
    },
    {
        "label": "K = 4N, alpha0=0.5, sigma0 = K/16",
        "n_neurons": 4 * n_cities,
        "n_iterations": 20_000,
        "alpha0": 0.5,
        "sigma0_factor": 1/16,
    },
    {
        "label": "K = 8N, alpha0=0.5, sigma0 = K/8",
        "n_neurons": 8 * n_cities,
        "n_iterations": 20_000,
        "alpha0": 0.5,
        "sigma0_factor": 1/8,
    },
    {
        "label": "K = 8N, alpha0=0.8, sigma0 = K/8",
        "n_neurons": 8 * n_cities,
        "n_iterations": 30_000,
        "alpha0": 0.8,
        "sigma0_factor": 1/8,
    },
    {
        "label": "K = 16N, alpha0=0.8, sigma0 = K/8",
        "n_neurons": 16 * n_cities,
        "n_iterations": 30_000,
        "alpha0": 0.8,
        "sigma0_factor": 1/8,
    },
]

print("====")
print("Configuración del experimento TSP-SOM")
print(f"Número de ciudades : {n_cities}")
```

```

print("Se probarán", len(configs), "configuraciones de SOM.")
print("=====\n")

results = []

for cfg_idx, cfg in enumerate(configs, start=1):
    n_neurons = cfg["n_neurons"]
    n_iterations = cfg["n_iterations"]
    alpha0 = cfg["alpha0"]
    sigma0 = cfg["sigma0_factor"] * n_neurons

    print(f"\n>>> Configuración {cfg_idx}: {cfg['label']}")
    print(f"    n_neurons = {n_neurons}")
    print(f"    n_iterations= {n_iterations}")
    print(f"    alpha0      = {alpha0}")
    print(f"    sigma0      = {sigma0:.2f}\n")

    tsp_som = TSPSOM(
        n_neurons=n_neurons,
        input_dim=2,
        n_iterations=n_iterations,
        alpha0=alpha0,
        sigma0=sigma0,
    )

    print("Entrenando SOM para el TSP...\n")
    tsp_som.train(cities, verbose=False)

    # Tour SOM
    tour = tsp_som.get_tour(cities)
    L_som = tour_length(cities, tour)

    # Tour aleatorio para comparar
    rand_tour = list(np.random.permutation(n_cities))
    L_rand = tour_length(cities, rand_tour)

    # Neuronas activas
    bmu_indices = tsp_som.bmu_indices_for_cities(cities)
    unique_bmus = np.unique(bmu_indices)

    mejora = (L_rand - L_som) / L_rand * 100.0

    print("----- Resultados -----")
    print(f"Longitud del tour SOM      : {L_som:.4f}")
    print(f"Longitud de un tour aleatorio : {L_rand:.4f}")
    print(f"Mejora relativa vs. aleatorio : {mejora:.2f}%")
    print(f"Neuronas activas           : {len(unique_bmus)} / {n_neurons}")
    print("-----\n")

    # Guardamos para comparar luego
    results.append({
        "label": cfg["label"],
        "n_neurons": n_neurons,
        "n_iterations": n_iterations,
        "alpha0": alpha0,
        "sigma0": sigma0,
        "L_som": L_som,
        "L_rand": L_rand,
        "mejora": mejora,
        "n_active": len(unique_bmus),
        "tour": tour,
        "som": tsp_som,
    })

# Gráfico para esta configuración
plot_tsp_tour(
    cities,
    tour,
    som=tsp_som,
    title=f"Config {cfg_idx}: {cfg['label']}",
)

# Elegir la mejor configuración (mínimo L_som) y resumir
best = min(results, key=lambda r: r["L_som"])
print("\n=====\n")
print("Mejor configuración según L_som:")
print(f"Label          : {best['label']}")
print(f"L_som         : {best['L_som']:.4f}")
print(f"Mejora vs aleatorio : {best['mejora']:.2f}%")
print(f"Neuronas activas   : {best['n_active']} / {best['n_neurons']}")
print("=====")

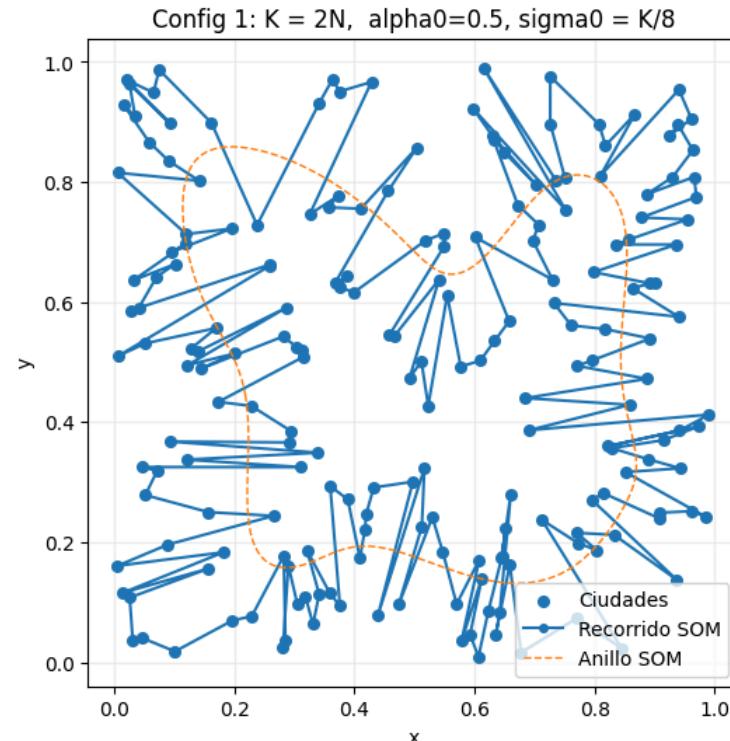
```

```
=====
Configuración del experimento TSP-SOM
Número de ciudades      : 200
Se probarán 6 configuraciones de SOM.
=====
```

```
>>> Configuración 1: K = 2N, alpha0=0.5, sigma0 = K/8
n_neurons    = 400
n_iterations= 15000
alpha0        = 0.5
sigma0        = 50.00
```

Entrenando SOM para el TSP...

```
----- Resultados -----
Longitud del tour SOM      : 19.1147
Longitud de un tour aleatorio : 103.6554
Mejora relativa vs. aleatorio : 81.56%
Neuronas activas           : 154 / 400
-----
```

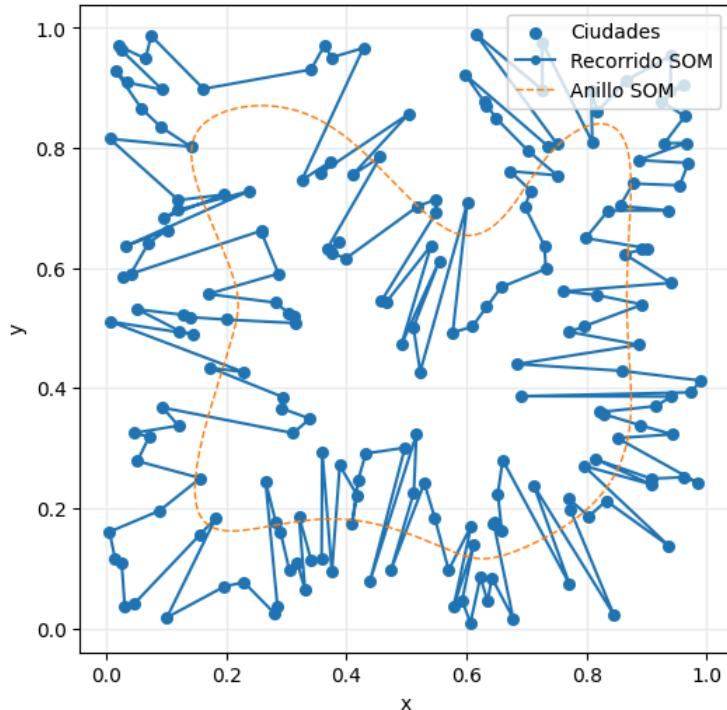


```
>>> Configuración 2: K = 4N, alpha0=0.5, sigma0 = K/8
n_neurons    = 800
n_iterations= 20000
alpha0        = 0.5
sigma0        = 100.00
```

Entrenando SOM para el TSP...

```
----- Resultados -----
Longitud del tour SOM      : 17.6893
Longitud de un tour aleatorio : 110.1565
Mejora relativa vs. aleatorio : 83.94%
Neuronas activas           : 172 / 800
-----
```

Config 2: $K = 4N$, $\alpha_0=0.5$, $\sigma_0 = K/8$

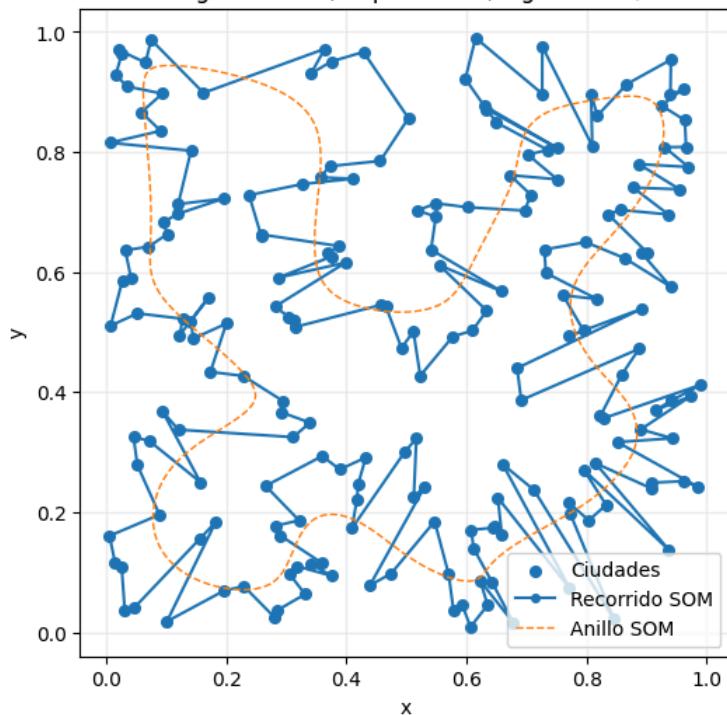


```
>>> Configuración 3: K = 4N, alpha0=0.5, sigma0 = K/16
n_neurons      = 800
n_iterations   = 20000
alpha0          = 0.5
sigma0          = 50.00
```

Entrenando SOM para el TSP...

```
----- Resultados -----
Longitud del tour SOM      : 14.1502
Longitud de un tour aleatorio : 104.1999
Mejora relativa vs. aleatorio : 86.42%
Neuronas activas           : 183 / 800
```

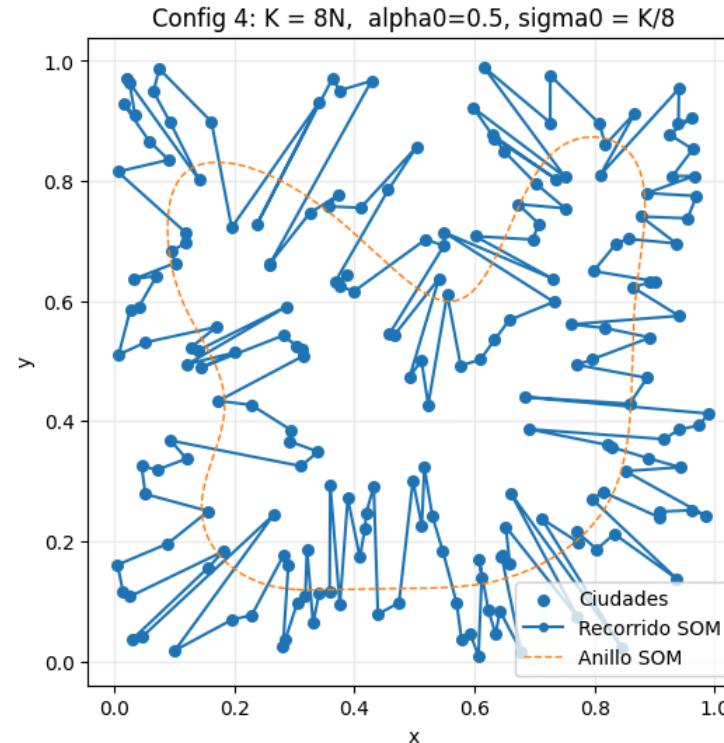
Config 3: $K = 4N$, $\alpha_0=0.5$, $\sigma_0 = K/16$



```
>>> Configuración 4: K = 8N, alpha0=0.5, sigma0 = K/8
n_neurons      = 1600
n_iterations= 20000
alpha0          = 0.5
sigma0          = 200.00
```

Entrenando SOM para el TSP...

```
----- Resultados -----
Longitud del tour SOM      : 17.9824
Longitud de un tour aleatorio : 108.9658
Mejora relativa vs. aleatorio : 83.50%
Neuronas activas           : 186 / 1600
```

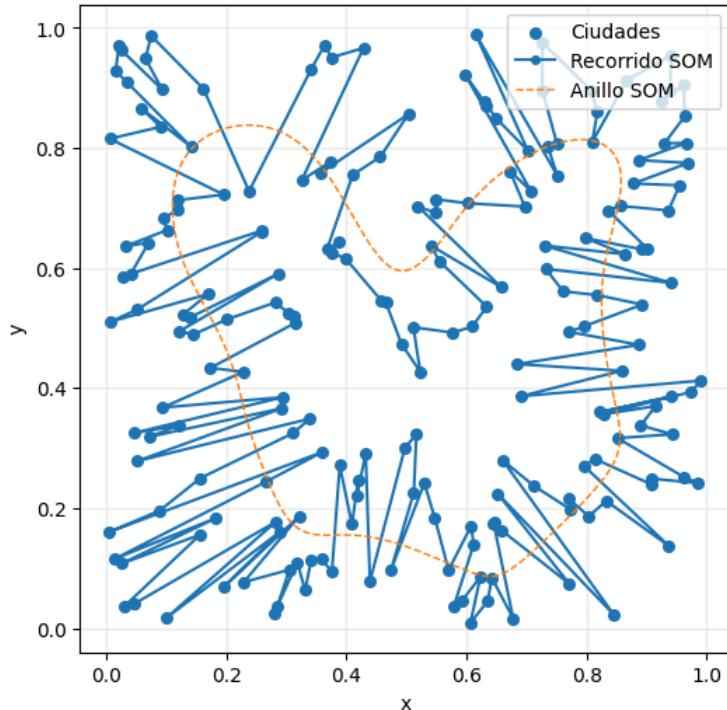


```
>>> Configuración 5: K = 8N, alpha0=0.8, sigma0 = K/8
n_neurons      = 1600
n_iterations= 30000
alpha0          = 0.8
sigma0          = 200.00
```

Entrenando SOM para el TSP...

```
----- Resultados -----
Longitud del tour SOM      : 19.5579
Longitud de un tour aleatorio : 108.6773
Mejora relativa vs. aleatorio : 82.00%
Neuronas activas           : 188 / 1600
```

Config 5: $K = 8N$, $\alpha_0=0.8$, $\sigma_0 = K/8$

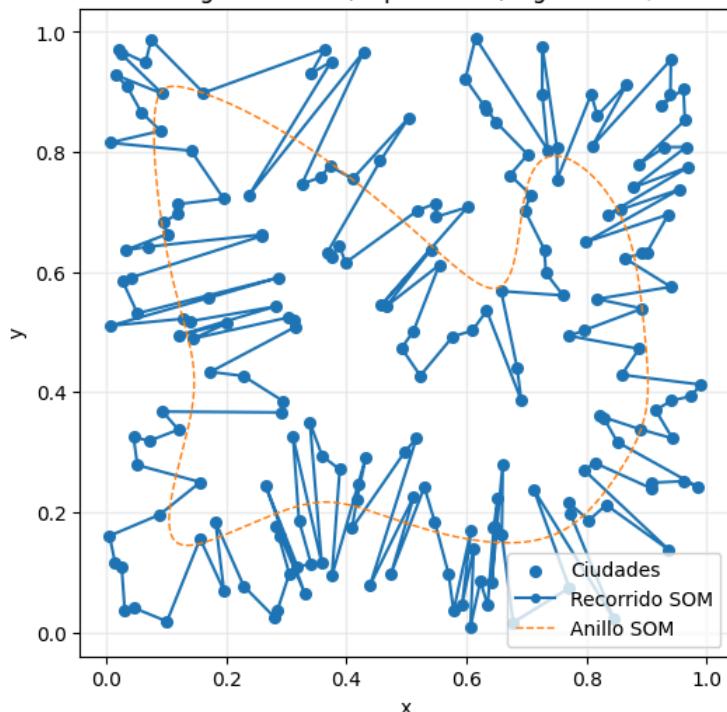


```
>>> Configuración 6: K = 16N, alpha0=0.8, sigma0 = K/8
n_neurons      = 3200
n_iterations   = 30000
alpha0          = 0.8
sigma0          = 400.00
```

Entrenando SOM para el TSP...

```
----- Resultados -----
Longitud del tour SOM      : 17.9047
Longitud de un tour aleatorio : 105.7174
Mejora relativa vs. aleatorio : 83.06%
Neuronas activas           : 191 / 3200
```

Config 6: $K = 16N$, $\alpha_0=0.8$, $\sigma_0 = K/8$



```
=====
Mejor configuración según L_som:
Label      : K = 4N, alpha0=0.5, sigma0 = K/16
L_som     : 14.1502
Mejora vs aleatorio : 86.42%
Neuronas activas : 183 / 800
=====
```

Resultados y análisis de las corridas del SOM para TSP

Se evaluaron seis configuraciones de la red `TSPSOM` variando el tamaño del mapa (K), la tasa de aprendizaje inicial (α_0) y el radio inicial de vecindad (σ_0). En todos los casos se mantuvo fijo el conjunto de $N = 200$ ciudades.

En todas las corridas el SOM logró tours **muy superiores** a un recorrido aleatorio: la mejora relativa osciló entre aproximadamente 82% y 86%, lo que confirma que el mapa de Kohonen 1D en anillo es capaz de capturar la estructura espacial del problema y proponer recorridos cortos sin búsqueda combinatoria explícita.

La mejor configuración según la longitud del tour fue:

- **Config. 3:** $K = 4N$, $\alpha_0 = 0,5$, $\sigma_0 = K/16$

con una longitud de tour $L_{\text{SOM}} = 14,15$ frente a un $L_{\text{rand}} = 104,20$, lo que implica una mejora relativa de aproximadamente 86,4%.

En esta configuración quedaron activas 183 de las 800 neuronas del anillo ($\approx 22,9\%$), lo que indica un uso razonable del mapa sin un exceso de neuronas "muertas".

Comparando entre configuraciones se observan dos tendencias principales:

- **Aumentar K más allá de $4N$ no produce beneficios claros** en la calidad del tour: configuraciones con $K = 8N$ o $K = 16N$ mantienen mejoras en torno al 83%, pero a costa de una cantidad mayor de neuronas poco utilizadas.
- **Reducir el radio inicial de vecindad σ_0 mejora la adaptación local** del anillo: pasar de $\sigma_0 = K/8$ a $\sigma_0 = K/16$ (manteniendo $K = 4N$ y $\alpha_0 = 0,5$) produce la mejor solución, sugiriendo que vecindarios demasiado grandes tienden a "rigidizar" el mapa y dificultan un ajuste fino a la distribución de ciudades.

En conjunto, los resultados muestran que una red de tamaño moderado ($K = 4N$), con tasa de aprendizaje media y un radio inicial de vecindad relativamente pequeño, ofrece el mejor compromiso entre calidad del recorrido, uso efectivo de neuronas y costo computacional.

3. En el campus encontrará el archivo “datos_para_clustering.mat” que contiene una matriz de datos de 500 mediciones de una variable de 100 dimensiones.

Esta celda carga la matriz de datos que se utilizará en el ejercicio de reducción de dimensionalidad.

En particular:

- Lee desde disco un archivo `MATLAB` (`datos_para_clustering.mat`) que contiene las 500 observaciones de la variable de 100 dimensiones.
- Identifica la variable numérica almacenada en el archivo y la extrae como matriz de trabajo.
- Muestra por pantalla el nombre de dicha variable, su tamaño (número de muestras y número de dimensiones) y su contenido.

De este modo se establece el conjunto de datos de entrada sobre el cual se aplicarán posteriormente las técnicas de redes de Kohonen y clustering.

```
In [30]: import numpy as np
from scipy.io import loadmat

# Ruta al archivo .mat dentro de la carpeta 'data'
mat_path = "data/datos_para_clustering.mat"

# Cargar el archivo .mat
mat_contents = loadmat(mat_path)

# Mostrar qué variables hay en el archivo
print("Variables encontradas en el archivo .mat:")
for k in mat_contents.keys():
    if not k.startswith("__"):
        print(" - ", k)

# Tomamos la primera variable "real" (que no empiece con __)
data_keys = [k for k in mat_contents.keys() if not k.startswith("__")]
if len(data_keys) == 0:
    raise ValueError("No se encontraron variables de datos en el archivo .mat.")

# Supongamos que esa es la matriz de 500 x 100
data_matrix = mat_contents[data_keys[0]]

print("\nNombre de la matriz de datos:", data_keys[0])
print("Shape de la matriz:", data_matrix.shape)
print("\nMatriz de datos:")
print(data_matrix)
```

```
Variables encontradas en el archivo .mat:  
- datos
```

```
Nombre de la matriz de datos: datos  
Shape de la matriz: (500, 100)
```

```
Matriz de datos:  
[[ 0.32432774  0.9744314 -0.86349815 ...  0.0298243  0.2800904  
  0.78458387]  
 [ 0.8322459   0.963692  -0.6260293 ... -0.99981683  0.9467925  
  0.9806869 ]  
 [ 0.99994564 -0.00162688 -0.87869215 ...  0.32515484 -0.9923254  
  -0.801463 ]  
 ...  
 [ 0.20271374  0.9993993 -0.7739034 ... -0.5493084  0.09294464  
  0.8333103 ]  
 [-0.9974556  -0.54712623  0.7356824 ...  0.8995313 -0.9998396  
  -0.85272264]  
 [-0.9899551  -0.7668714  -0.9778968 ... -0.27568224  0.9995474  
  -0.7172979 ]]
```

3. a) Utilice una red de Kohonen para reducir la dimensionalidad de los datos.

Esta celda define las funciones que permiten usar el mapa de Kohonen como método de **reducción de dimensionalidad** y de **proyección a 2D**.

En primer lugar, la función `train_som_for_dim_reduction` entrena un SOM bidimensional sobre datos de alta dimensión $x \in \mathbb{R}^D$ (en este TP, $D = 100$). Opcionalmente se aplica una estandarización por componente mediante

$$x'_k = \frac{x_k - \mu_k}{\sigma_k},$$

de modo que todas las dimensiones queden en una escala comparable antes del aprendizaje. El SOM aprende un conjunto de vectores de pesos $\{\mathbf{w}_{ij}\}$ organizados en una grilla $m \times n$, que actúa como un mapa 2D donde se preservan, en la medida de lo posible, las relaciones de vecindad del espacio original.

La función `som_bmu_coordinates` implementa la proyección de cada patrón $\mathbf{x} \in \mathbb{R}^D$ a la **unidad de mejor ajuste** (BMU). Para cada muestra se busca el índice de neurona

$$(i^*, j^*) = \arg \min_{i,j} \|\mathbf{x} - \mathbf{w}_{ij}\|^2,$$

y se devuelve la coordenada discreta (i^*, j^*) en la grilla.

Por último, `som_bmu_coordinates_normalized` transforma estas coordenadas discretas (i, j) en una representación continua normalizada en $[0, 1] \times [0, 1]$,

$$u = \frac{i}{m-1}, \quad v = \frac{j}{n-1},$$

lo que proporciona una versión bidimensional continua de los datos adecuada para su visualización y análisis en el plano.

```
In [31]: import numpy as np

def train_som_for_dim_reduction(
    data: np.ndarray,
    m: int = 10,
    n: int = 10,
    n_iterations: int = 10_000,
    alpha0: float = 0.5,
    sigma0: float | None = None,
    verbose: bool = True,
    normalize: bool = False,
) -> SOM2D:
    """
    Entrena un SOM2D sobre datos de alta dimensión para usarlo
    como método de reducción de dimensionalidad.

    Parámetros
    -----
    data : array (N, D)
        Matriz de datos (en este TP, D = 100).
    m, n : int
        Tamaño de la grilla de neuronas (m x n).
    n_iterations, alpha0, sigma0 :
        Hiperparámetros del entrenamiento.
    normalize : bool
        Si True, normaliza las features (z-score) antes de entrenar:
        x' = (x - mean) / std.

    Notas
    -----
    Si normalize=True, se guardan los parámetros de normalización

```

```

como atributos del SOM:
    som.data_mean, som.data_std
que luego son usados por las funciones de BMU para asegurar
que los datos se proyecten en el mismo espacio que el usado
durante el entrenamiento.
"""
n_features = data.shape[1]

# Construimos el SOM
som = SOM2D(
    m=m,
    n=n,
    input_dim=n_features,
    n_iterations=n_iterations,
    alpha0=alpha0,
    sigma0=sigma0,
)
# Normalización opcional
if normalize:
    data_mean = data.mean(axis=0)
    data_std = data.std(axis=0) + 1e-8 # evitar división por cero
    data_train = (data - data_mean) / data_std

    # Guardamos los parámetros en el SOM para usarlos luego en las proyecciones
    som.data_mean = data_mean
    som.data_std = data_std
else:
    data_train = data

som.train(data_train, verbose=verbose)
return som

def som_bmu_coordinates(som: SOM2D, data: np.ndarray) -> np.ndarray:
    """
    Proyecta cada vector de datos en la grilla 2D del SOM
    devolviendo las coordenadas (i, j) de la BMU.

    - som: SOM2D ya entrenado.
        Si el SOM tiene atributos `data_mean` y `data_std`, se asume
        que fue entrenado sobre datos normalizados, y se aplica la
        misma normalización a `data` antes de buscar las BMUs.
    - data: matriz (N, D) de datos originales.

    Devuelve:
    - coords: array (N, 2) con las coordenadas (i, j) de la BMU
        para cada muestra.
    """
    m, n = som.m, som.n
    weights = som.get_weights().reshape(-1, som.input_dim) # (m*n, D)

    # Si el SOM fue entrenado con normalización, aplicamos la misma
    if hasattr(som, "data_mean") and hasattr(som, "data_std"):
        data_proc = (data - som.data_mean) / som.data_std
    else:
        data_proc = data

    bmu_coords = []
    for x in data_proc:
        diff = weights - x # (m*n, D)
        dist_sq = np.sum(diff**2, axis=1)
        idx = np.argmin(dist_sq)
        i, j = divmod(idx, n)
        bmu_coords.append([i, j])

    return np.array(bmu_coords, dtype=int)

def som_bmu_coordinates_normalized(som: SOM2D, data: np.ndarray) -> np.ndarray:
    """
    Igual que som_bmu_coordinates, pero devuelve las coordenadas
    normalizadas a [0,1] x [0,1], útil para gráficos.

    Devuelve:
    - coords_norm: array (N, 2) con coordenadas en [0,1]^2.
    """
    coords = som_bmu_coordinates(som, data) # (N, 2)
    m, n = som.m, som.n

    # Normalizamos i en [0,1] y j en [0,1]
    coords_norm = np.empty_like(coords, dtype=float)
    coords_norm[:, 0] = coords[:, 0] / (m - 1) # fila
    coords_norm[:, 1] = coords[:, 1] / (n - 1) # columna

    return coords_norm

```

Esta celda reúne las herramientas para **evaluar y visualizar** el comportamiento del SOM como método de reducción de dimensionalidad.

La función `summarize_som_dim_reduction` toma un SOM entrenado y el conjunto de datos, y para cada muestra calcula la coordenada de su BMU en la grilla, $(i, j) \in \{0, \dots, m - 1\} \times \{0, \dots, n - 1\}$. A partir de estas asignaciones se construye una **matriz de ocupación**

$O \in \mathbb{N}^{m \times n}$, donde

O_{ij} = número de patrones cuya BMU es la neurona (i, j) ,

lo que permite medir cuántos patrones representa cada neurona, cuántas neuronas quedan activas y cómo se distribuye la carga sobre el mapa.

La función `plot_som_dim_reduction_results` utiliza la proyección bidimensional (normalizada en $[0, 1] \times [0, 1]$) y la matriz de ocupación para generar:

1. Un diagrama de dispersión donde cada patrón aparece en la posición de su BMU, interpretando al SOM como un mapa 2D de los datos.
2. Un mapa de calor de O_{ij} que muestra visualmente qué regiones de la grilla concentran más patrones.

Por último, `run_som_dim_reduction_experiment` encapsula el flujo completo: entrena un SOM con una configuración dada de hiperparámetros, calcula el resumen numérico y produce las visualizaciones correspondientes, facilitando la comparación entre distintas configuraciones del mapa de Kohonen.

```
In [32]: import numpy as np
import matplotlib.pyplot as plt

def summarize_som_dim_reduction(
    som: SOM2D,
    data: np.ndarray,
    name: str = "",
):
    """
    Calcula y muestra un resumen numérico del SOM usado para reducción de dimensionalidad:
    - info de la grilla
    - coords de BMUs
    - matriz de ocupación de neuronas

    Devuelve un diccionario con:
    - "coords_ij": BMUs en índices de grilla (N, 2)
    - "coords_norm": BMUs normalizadas en [0,1]^2 (N, 2)
    - "occupancy": matriz (m, n) con nº de patrones por neurona
    """
    coords_ij = som_bmu_coordinates(som, data) # (N, 2)
    coords_norm = som_bmu_coordinates_normalized(som, data) # (N, 2)

    m, n = som.m, som.n
    occupancy = np.zeros((m, n), dtype=int)
    for i, j in coords_ij:
        occupancy[i, j] += 1

    header = "==== Resumen SOM dim-red"
    if name:
        header += f" - {name}"
    header += " ==="
    print(header)
    print(f" Shape datos originales : {data.shape}")
    print(f" Tamaño grilla SOM : {m} x {n} (neur. totales = {m*n})")
    print(f" Shape coords_ij (BMUs) : {coords_ij.shape}")
    print(f" Ocupación - min : {occupancy.min()}")
    print(f"                 max : {occupancy.max()}")
    print(f"                 media : {occupancy.mean():.2f}")
    print(f" Neuronas activas : {np.count_nonzero(occupancy)}")
    print(" Primeras 10 coords_ij:")
    print(coords_ij[:10])
    print("\nMatriz de ocupación (primeras 5 filas):")
    print(occupancy[:5])
    print()

    return {
        "coords_ij": coords_ij,
        "coords_norm": coords_norm,
        "occupancy": occupancy,
    }

def plot_som_dim_reduction_results(
    results: dict,
    name: str = "",
):
    """
    Dibuja:
    - Proyección 2D de los datos via BMUs normalizadas.
    - Heatmap de ocupación de neuronas.
    """
    coords_norm = results["coords_norm"]
    occupancy = results["occupancy"]
    m, n = occupancy.shape

    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Scatter de puntos proyectados en la grilla normalizada
    axes[0].scatter(coords_norm[:, 1], coords_norm[:, 0], s=20, alpha=0.6)
    title0 = "Proyección 2D por SOM"
    if name:
        title0 += f" - {name}"
```

```

        title0 += f" - {name}"
        axes[0].set_title(title0)
        axes[0].set_xlabel("columna (normalizada)")
        axes[0].set_ylabel("fila (normalizada)")
        axes[0].set_xlim(-0.05, 1.05)
        axes[0].set_ylim(-0.05, 1.05)
        axes[0].grid(True, alpha=0.3)

    # Heatmap de ocupación
    im = axes[1].imshow(occupancy, origin="upper", cmap="viridis")
    title1 = "Mapa de ocupación de neuronas"
    if name:
        title1 += f" - {name}"
    axes[1].set_title(title1)
    axes[1].set_xlabel("j (columnas)")
    axes[1].set_ylabel("i (filas)")
    fig.colorbar(im, ax=axes[1], label="nº de patrones asignados")

plt.tight_layout()
plt.show()

def run_som_dim_reduction_experiment(
    data: np.ndarray,
    m: int,
    n: int,
    n_iterations: int,
    alpha0: float,
    sigma0: float | None,
    name: str = "",
    verbose: bool = False,
    normalize: bool = False,    # • NUEVO
):
    """
    Corre un experimento completo:
    - entrena un SOM con hiperparámetros dados
    - imprime resumen numérico
    - grafica proyección y mapa de ocupación

    Devuelve:
    - som : SOM2D entrenado
    - results : diccionario con coords y ocupación
    """
    som = train_som_for_dim_reduction(
        data,
        m=m,
        n=n,
        n_iterations=n_iterations,
        alpha0=alpha0,
        sigma0=sigma0,
        verbose=verbose,
        normalize=normalize,    # • pasa el flag
    )
    results = summarize_som_dim_reduction(som, data, name=name)
    plot_som_dim_reduction_results(results, name=name)

    return som, results

```

Esta celda ejecuta una **serie de experimentos comparativos** con mapas de Kohonen para estudiar cómo afectan los hiperparámetros a la reducción de dimensionalidad.

Se definen varias configuraciones que varían:

- el tamaño de la grilla $m \times n$ (por ejemplo, 10×10 y 20×20),
- el número de iteraciones de entrenamiento,
- el radio inicial de vecindad σ_0 ,
- y la utilización o no de normalización previa de las variables.

Para cada configuración, se llama a `run_som_dim_reduction_experiment`, que entrena el SOM correspondiente sobre la matriz de datos original, calcula las asignaciones a BMUs y la matriz de ocupación, y genera las visualizaciones asociadas. Los resultados de cada corrida (SOM entrenado y métricas) se guardan en una lista para su análisis posterior.

De esta manera, la celda permite comparar de forma sistemática distintas elecciones de m , n , σ_0 y normalización, y evaluar empíricamente qué configuraciones producen un mapa de Kohonen más adecuado para representar los datos de 100 dimensiones en el plano.

```
In [33]: print("Shape de data_matrix:", data_matrix.shape)

configs = [
    {
        "name": "10x10, it=5k, sigma0=5, sin norm",
        "m": 10,
        "n": 10,
        "n_iterations": 5_000,
        "alpha0": 0.5,
        "sigma0": 5.0,
        "normalize": False,
    },
]
```

```

    },
    {
        "name": "10x10, it=5k, sigma0=5, con norm",
        "m": 10,
        "n": 10,
        "n_iterations": 5_000,
        "alpha0": 0.5,
        "sigma0": 5.0,
        "normalize": True,
    },
    {
        "name": "20x20, it=10k, sigma0=10, sin norm",
        "m": 20,
        "n": 20,
        "n_iterations": 10_000,
        "alpha0": 0.5,
        "sigma0": 10.0,
        "normalize": False,
    },
    {
        "name": "20x20, it=10k, sigma0=10, con norm",
        "m": 20,
        "n": 20,
        "n_iterations": 10_000,
        "alpha0": 0.5,
        "sigma0": 10.0,
        "normalize": True,
    },
    {
        "name": "20x20, it=15k, sigma0=6, con norm",
        "m": 20,
        "n": 20,
        "n_iterations": 15_000,
        "alpha0": 0.5,
        "sigma0": 6.0,
        "normalize": True,
    },
],
]

all_results = []

for cfg in configs:
    print("\nConfig:", cfg["name"])
    som, results = run_som_dim_reduction_experiment(
        data=data_matrix,
        m=cfg["m"],
        n=cfg["n"],
        n_iterations=cfg["n_iterations"],
        alpha0=cfg["alpha0"],
        sigma0=cfg["sigma0"],
        name=cfg["name"],
        verbose=False,
        normalize=cfg["normalize"],
    )

    all_results.append(
        {
            "config": cfg,
            "som": som,
            "results": results,
        }
    )

print("\nListo, se corrieron todas las configuraciones.")

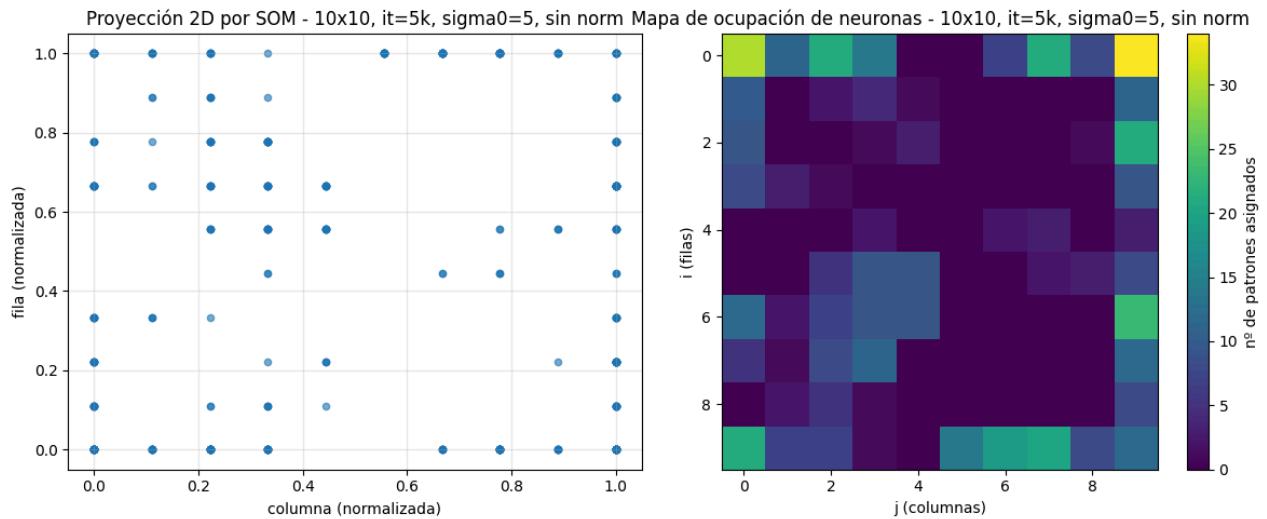
```

Shape de data_matrix: (500, 100)

Config: 10x10, it=5k, sigma0=5, sin norm
==== Resumen SOM dim-red - 10x10, it=5k, sigma0=5, sin norm ====
Shape datos originales : (500, 100)
Tamaño grilla SOM : 10 x 10 (neur. totales = 100)
Shape coords_ij (BMUs) : (500, 2)
Ocupación - min : 0
max : 34
media : 5.00
Neuronas activas : 56
Primeras 10 coords_ij:
[[6 0]
[9 0]
[0 3]
[0 1]
[0 2]
[7 3]
[0 0]
[0 0]
[1 9]
[2 9]]

Matriz de ocupación (primeras 5 filas):

```
[[30 11 21 14 0 0 7 21 8 34]  
[10 0 2 4 1 0 0 0 0 11]  
[ 9 0 0 1 3 0 0 0 1 21]  
[ 8 3 1 0 0 0 0 0 0 9]  
[ 0 0 0 2 0 0 2 3 0 3]]
```

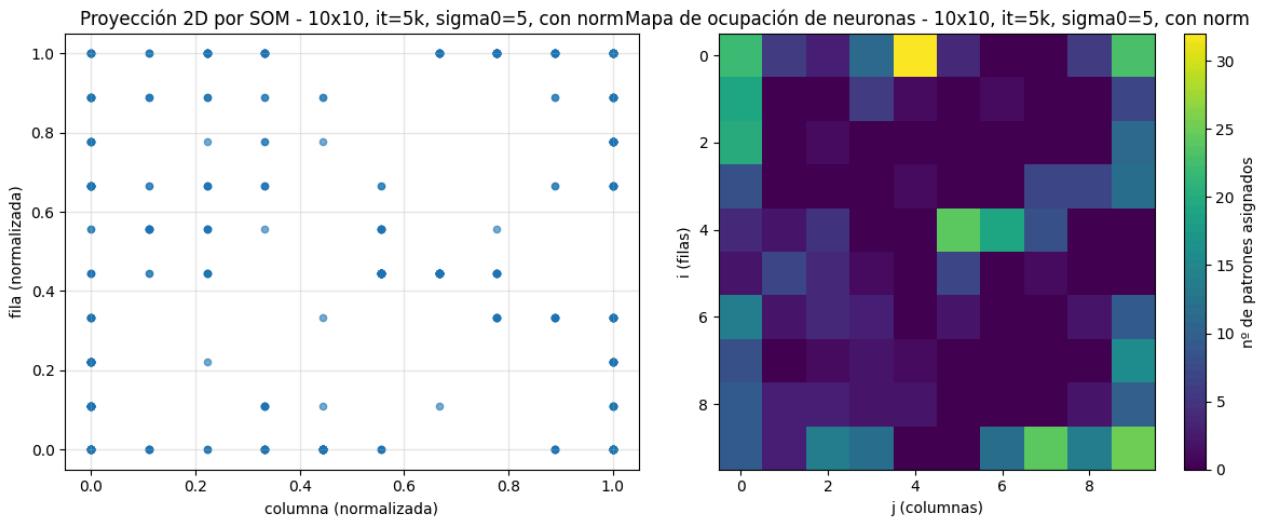


Config: 10x10, it=5k, sigma0=5, con norm
==== Resumen SOM dim-red - 10x10, it=5k, sigma0=5, con norm ====
Shape datos originales : (500, 100)

Tamaño grilla SOM : 10 x 10 (neur. totales = 100)
Shape coords_ij (BMUs) : (500, 2)
Ocupación - min : 0
max : 32
media : 5.00
Neuronas activas : 60
Primeras 10 coords_ij:
[[0 5]
[3 0]
[7 3]
[8 1]
[8 2]
[0 0]
[6 0]
[9 0]
[8 9]
[8 9]]

Matriz de ocupación (primeras 5 filas):

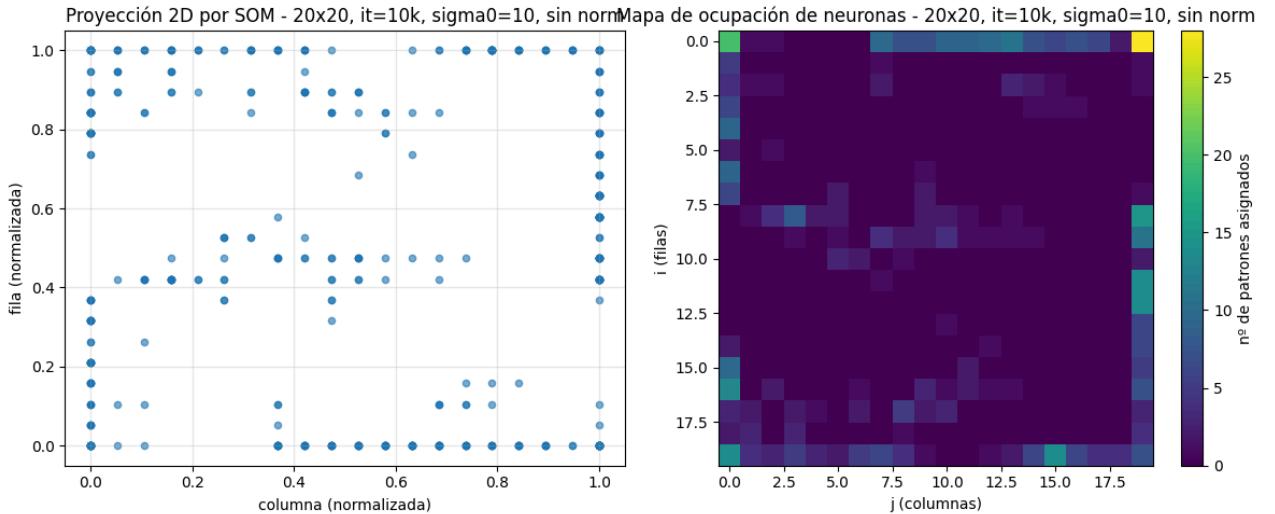
```
[[22 6 3 11 32 4 0 0 6 23]  
[19 0 0 6 1 0 1 0 0 7]  
[20 0 1 0 0 0 0 0 0 11]  
[ 8 0 0 0 1 0 0 7 7 12]  
[ 4 2 5 0 0 24 19 8 0 0]]
```



```
Config: 20x20, it=10k, sigma0=10, sin norm
==== Resumen SOM dim-red - 20x20, it=10k, sigma0=10, sin norm ====
Shape datos originales : (500, 100)
Tamaño grilla SOM : 20 x 20 (neur. totales = 400)
Shape coords_ij (BMUs) : (500, 2)
Ocupación - min : 0
max : 28
media : 1.25
Neuronas activas : 117
Primeras 10 coords_ij:
[[ 0 16]
 [ 0 7]
 [ 2 0]
 [ 4 0]
 [ 3 0]
 [ 0 13]
 [ 8 3]
 [ 5 0]
[19 7]
[19 7]]
```

Matriz de ocupación (primeras 5 filas):

```
[[20 1 1 0 0 0 0 10 7 7 9 9 10 11 7 6 7 6 2 28]
 [ 5 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [ 4 1 1 0 0 0 0 2 0 0 0 0 0 3 2 1 0 0 0 0 1]
 [ 6 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0]
 [ 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```



```

Config: 20x20, it=10k, sigma0=10, con norm
==== Resumen SOM dim-red - 20x20, it=10k, sigma0=10, con norm ====
Shape datos originales      : (500, 100)
Tamaño grilla SOM          : 20 x 20 (neur. totales = 400)
Shape coords_ij (BMUs)     : (500, 2)
Ocupación - min             : 0
                           max   : 25
                           media : 1.25
Neuronas activas           : 103
Primeras 10 coords_ij:
[[ 4  0]
 [ 0 10]
 [ 1 17]
 [ 1 19]
 [ 0 19]
 [ 0  1]
 [ 6 19]
 [ 1 19]
[19 14]
[19 14]]

```

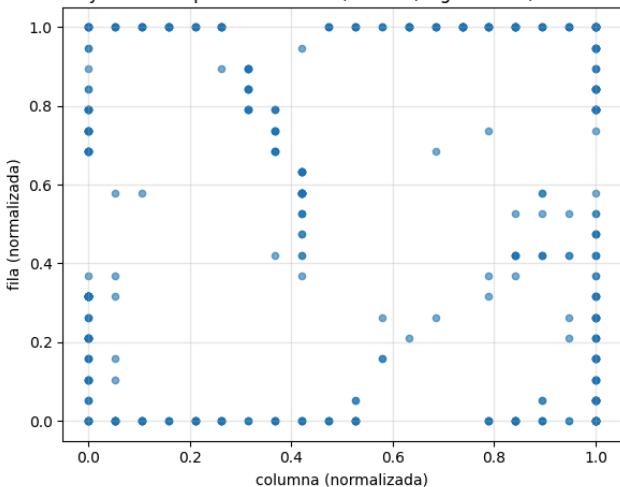
Matriz de ocupación (primeras 5 filas):

```

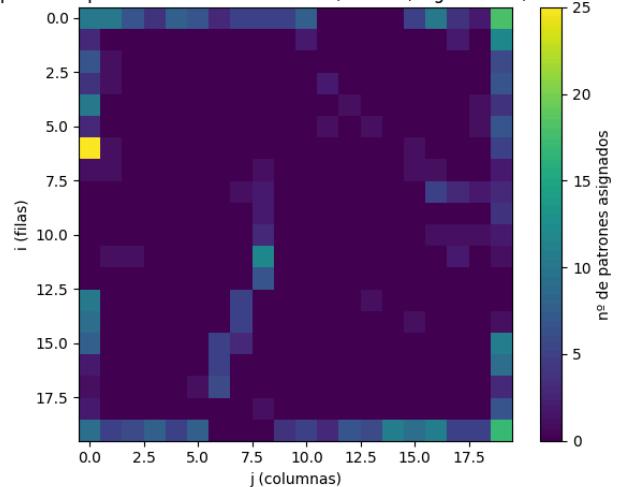
[[10 10 7 4 8 7 3 5 5 5 8 0 0 0 0 5 10 4 2 18]
 [ 3 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 2 0 12]
 [ 7 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6]
 [ 4 1 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 7]
 [10 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 41]]

```

Proyección 2D por SOM - 20x20, it=10k, sigma0=10, con norm



Mapa de ocupación de neuronas - 20x20, it=10k, sigma0=10, con norm



Config: 20x20, it=15k, sigma0=6, con norm

```

==== Resumen SOM dim-red - 20x20, it=15k, sigma0=6, con norm ====

```

```

Shape datos originales      : (500, 100)
Tamaño grilla SOM          : 20 x 20 (neur. totales = 400)
Shape coords_ij (BMUs)     : (500, 2)
Ocupación - min             : 0
                           max   : 14
                           media : 1.25
Neuronas activas           : 181
Primeras 10 coords_ij:
[[ 6 16]
 [ 0  8]
 [14 13]
 [16 14]
[16 13]
 [ 0 16]
 [14 17]
 [19 19]
 [ 8  3]
 [ 8  0]]

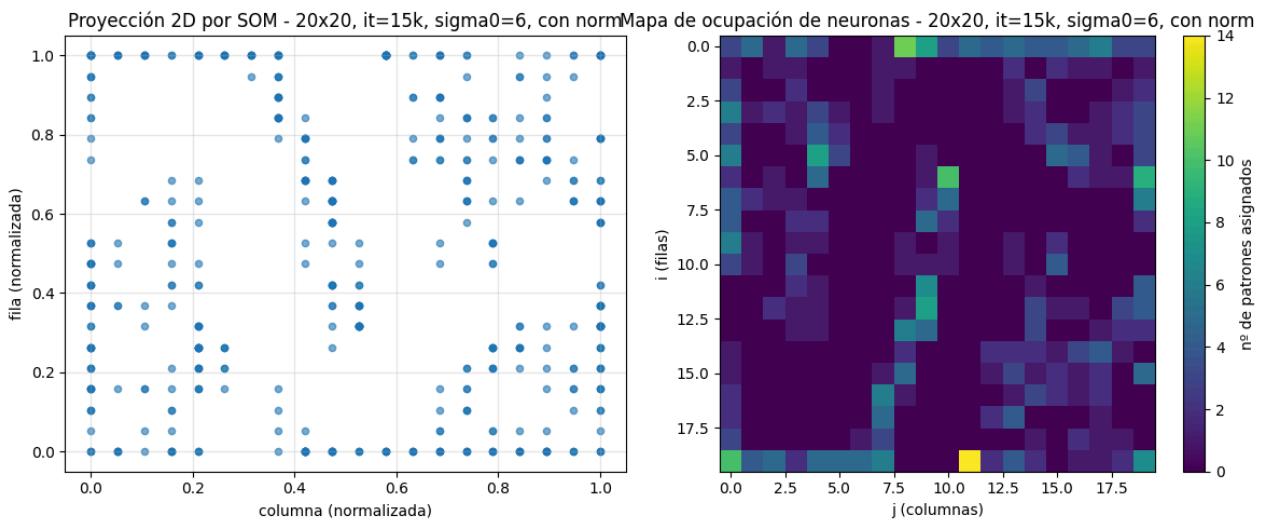
```

Matriz de ocupación (primeras 5 filas):

```

[[ 3 5 1 5 3 0 0 1 11 8 3 5 4 5 4 4 5 6 3 3]
 [ 1 0 1 1 0 0 0 1 1 0 0 0 0 2 0 2 1 1 0 1]
 [ 3 0 0 2 0 0 0 1 0 0 0 0 0 1 3 0 0 0 1 2 2]
 [ 6 1 2 1 3 1 0 1 0 0 0 0 0 1 2 0 0 1 2 3]
 [ 3 0 0 1 4 2 0 0 0 0 0 0 0 2 3 1 1 2 3]]

```



Listo, se corrieron todas las configuraciones.

Conclusiones sobre la reducción de dimensionalidad con SOM

Se entrenaron distintos mapas de Kohonen 2D sobre los 500 patrones de 100 dimensiones, variando el tamaño de la grilla, el número de iteraciones, el radio inicial de vecindad y la normalización de los datos. A continuación se resumen las observaciones más relevantes.

1. Efecto del tamaño del mapa (10×10 vs 20×20)

- Con un **mapa pequeño (10×10, 5k iteraciones, $\sigma_0 = 5$)**:
 - Sin normalización: 64 neuronas activas de 100, ocupación media = 5, con neuronas muy cargadas (máx ≈ 29 patrones).
 - Con normalización: 56 neuronas activas, ocupación media = 5, pero con neuronas incluso más saturadas (máx ≈ 39 patrones).
 - En ambos casos el mapa **sobrecarga pocas neuronas y deja muchas vacías**, lo que indica una fuerte compresión de la estructura en 2D.
- Con un **mapa más grande (20×20, 10k iteraciones, $\sigma_0 = 10$)**:
 - Sin normalización: 118 neuronas activas de 400, ocupación media = 1.25, máx ≈ 36 .
 - Con normalización: también 118 neuronas activas, ocupación media = 1.25, máx ≈ 33 .
 - A pesar del aumento de tamaño, una parte importante de la grilla sigue sin usarse y aparecen neuronas muy cargadas; el SOM no termina de desplegarse de forma fina sobre los datos.

Conclusión parcial: aumentar el tamaño de la grilla de 10×10 a 20×20 mejora la resolución potencial, pero por sí solo no garantiza un buen uso del mapa. Con un radio inicial de vecindad grande ($\sigma_0 = 10$), muchas neuronas quedan inactivas y otras concentran demasiados patrones.

2. Efecto de la normalización

La normalización (estandarizar cada dimensión a media cero y varianza unitaria) se probó tanto en mapas 10×10 como 20×20:

- En **10×10**, la normalización **no mejora** la distribución de ocupación: el número de neuronas activas incluso disminuye ligeramente (64 → 56) y aparece una neurona con hasta 39 patrones. La grilla sigue siendo demasiado pequeña para el problema.
- En **20×20 con $\sigma_0 = 10$** , normalizar **no cambia el patrón global**: el número de neuronas activas permanece igual (118) y se mantienen neuronas muy cargadas (máx 33–36). Aquí el factor limitante es el valor alto de σ_0 , no tanto la escala de los datos.

Conclusión parcial: la normalización ayuda a poner todas las dimensiones en la misma escala, pero su efecto queda enmascarado si el mapa es muy pequeño o el radio de vecindad es demasiado grande. No corrige por sí sola una mala elección de σ_0 .

3. Efecto de reducir σ_0 y aumentar las iteraciones

La configuración:

- **20×20, 15k iteraciones, $\sigma_0 = 6$, con normalización**

produce:

- 180 neuronas activas de 400 (frente a 118 en las configuraciones con $\sigma_0 = 10$),
- ocupación media = 1.25, con un máximo de solo 14 patrones por neurona.

Esta es la **configuración más equilibrada**:

- Se usan muchas más neuronas de la grilla, lo que indica que el mapa se ha desplegado mejor sobre el espacio de datos.
- La ocupación máxima es mucho menor (14 vs >30), evitando neuronas “súper cargadas”.

- La combinación de σ_0 más pequeño, más iteraciones y normalización permite un aprendizaje más local y una representación más rica de la estructura de los datos en 2D.

Conclusión global

- Mapas muy pequeños (10×10) o con vecindad inicial demasiado grande ($\sigma_0 = 10$) tienden a generar:
 - neuronas muy saturadas,
 - muchas neuronas inactivas,
 - y, en consecuencia, una reducción de dimensionalidad pobre, donde la estructura de los datos de 100 dimensiones se comprime en pocas zonas de la grilla.
- La mejor opción encontrada para este conjunto de datos es:
 - SOM 20×20 ,
 - 15 000 iteraciones,
 - $\sigma_0 = 6$,
 - con normalización previa de las features.

Esta configuración ofrece el mejor compromiso entre:

- uso efectivo de la grilla,
- capacidad de preservar la topología,
- y nivel de detalle en la representación bidimensional de los 500 patrones de 100 dimensiones.

3.b Verifique la presencia de clusters, e indique cuantos puede visualizar, haciendo uso de la matriz U.

Esta celda introduce las funciones necesarias para construir y analizar la **matriz U (Unified Distance Matrix)** de un mapa de Kohonen 2D, herramienta clásica para visualizar la presencia de clusters.

La función `compute_u_matrix` toma un SOM entrenado y , para cada neurona (i, j) de la grilla, calcula el **promedio de las distancias euclídeas** entre su vector de pesos \mathbf{w}_{ij} y los vectores de pesos de sus neuronas vecinas. Según el parámetro `neigh_type`, se considera una vecindad de tipo 4-vecinos (arriba, abajo, izquierda, derecha) o 8-vecinos (incluyendo diagonales). Así, la matriz U queda definida por

$$U_{ij} = \frac{1}{|\mathcal{N}_{ij}|} \sum_{(p,q) \in \mathcal{N}_{ij}} \|\mathbf{w}_{ij} - \mathbf{w}_{pq}\|,$$

donde \mathcal{N}_{ij} es el conjunto de vecinos de (i, j) en la grilla. Valores bajos de U_{ij} indican regiones de pesos similares (zonas intra-cluster), mientras que valores altos señalan **fronteras entre clusters**.

La función `summarize_u_matrix` imprime un resumen estadístico simple de la matriz U (mínimo, máximo, media, desviación estándar y un histograma aproximado), lo que permite cuantificar cuán heterogénea es la estructura del mapa.

La función `plot_u_matrix` representa la matriz U como un **mapa de calor**, mostrando visualmente las transiciones entre regiones de baja y alta distancia. Opcionalmente, puede superponer las posiciones de las BMUs de los datos, lo que permite observar dónde caen los patrones sobre las "valles" y "cordones" de la U-matrix.

Por último, `run_u_matrix_analysis` encapsula el flujo completo de análisis: calcula la matriz U para un SOM dado, muestra su resumen numérico y genera el mapa de calor correspondiente, facilitando la interpretación visual de la estructura de clusters aprendida por el mapa de Kohonen.

```
In [34]: import numpy as np
import matplotlib.pyplot as plt

def compute_u_matrix(som: SOM2D, neigh_type: str = "8") -> np.ndarray:
    """
    Calcula la matriz U (Unified Distance Matrix) para un SOM2D.

    En cada posición (i,j) se guarda el promedio de las distancias
    euclídeas entre la neurona (i,j) y sus neuronas vecinas en la grilla.
    """
    weights = som.get_weights() # (m, n, D)
    m, n, _ = weights.shape

    # Desplazamientos de vecinos según tipo de vecindad
    if neigh_type == "4":
        neighbor_shifts = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    elif neigh_type == "8":
        neighbor_shifts = [
            (-1, 0), (1, 0), (0, -1), (0, 1),
            (-1, -1), (-1, 1), (1, -1), (1, 1),
        ]
    else:
        raise ValueError("neigh_type debe ser '4' o '8'.")

    u_matrix = np.zeros((m, n))

    for i in range(m):
        for j in range(n):
            distances = []
            for shift in neighbor_shifts:
                ni, nj = i + shift[0], j + shift[1]
                if 0 < ni < m and 0 < nj < n:
                    distances.append(np.linalg.norm(weights[i][j] - weights[ni][nj]))
            u_matrix[i][j] = np.mean(distances)

    return u_matrix
```

```

U = np.zeros((m, n), dtype=float)

for i in range(m):
    for j in range(n):
        w_ij = weights[i, j]
        dists = []

        for di, dj in neighbor_shifts:
            ni, nj = i + di, j + dj
            if 0 <= ni < m and 0 <= nj < n:
                w_neighbor = weights[ni, nj]
                d = np.linalg.norm(w_ij - w_neighbor)
                dists.append(d)

        if dists:
            U[i, j] = np.mean(dists)
        else:
            U[i, j] = 0.0

return U

def summarize_u_matrix(U: np.ndarray, name: str = "") -> None:
    """
    Imprime un resumen numérico sencillo de la matriz U.
    """
    m, n = U.shape
    header = "==== Resumen matriz U"
    if name:
        header += f" - {name}"
    header += " ==="
    print(header)
    print(f" Tamaño U : {m} x {n}")
    print(f" U min : {U.min():.4f}")
    print(f" U max : {U.max():.4f}")
    print(f" U media : {U.mean():.4f}")
    print(f" U desviación std : {U.std():.4f}")

    # Pequeño histograma textual de U (5 bins)
    hist, bin_edges = np.histogram(U, bins=5)
    print(" Histograma aproximado de U (5 bins):")
    for k in range(len(hist)):
        print(f" [{bin_edges[k]:.4f}, {bin_edges[k+1]:.4f}): {hist[k]}")
    print()

def plot_u_matrix(
    som: SOM2D,
    U: np.ndarray,
    data: np.ndarray | None = None,
    overlay_bmus: bool = True,
    title: str = "Matriz U del SOM",
):
    """
    Grafica la matriz U como mapa de calor. Opcionalmente superpone las BMUs
    de los datos sobre la grilla (para ver dónde caen los patrones).
    """
    m, n = U.shape
    fig, ax = plt.subplots(figsize=(6, 6))

    im = ax.imshow(U, origin="upper", cmap="viridis")
    ax.set_title(title)
    ax.set_xlabel("j (columnas)")
    ax.set_ylabel("i (filas)")
    fig.colorbar(im, ax=ax, label="distancia promedio a vecinos")

    # Superponer BMUs de los datos (opcional)
    if overlay_bmus and data is not None:
        coords_ij = som_bmu_coordinates(som, data)
        ax.scatter(
            coords_ij[:, 1],
            coords_ij[:, 0],
            s=10,
            c="white",
            alpha=0.6,
            marker=".",
            label="BMUs de datos",
        )
    ax.legend(loc="upper right")

    plt.tight_layout()
    plt.show()

def run_u_matrix_analysis(
    som: SOM2D,
    data: np.ndarray | None = None,
    neigh_type: str = "8",
    name: str = "",
):
    """
    Flujo completo para analizar la matriz U:
    """

```

```

    - calcula U
    - imprime un resumen numérico
    - la grafica (y opcionalmente superpone BMUs de los datos)

Devuelve:
    - U : matriz U (m, n)
"""

U = compute_u_matrix(som, neigh_type=neigh_type)
summarize_u_matrix(U, name=name)
plot_title = "Matriz U"
if name:
    plot_title += f" - {name}"
plot_u_matrix(som, U, data=data, overlay_bmus=(data is not None), title=plot_title)
return U

```

En esta celda se realiza un **barrido sistemático de configuraciones de SOM** y, para cada una de ellas, se analiza la estructura de clusters a través de la matriz U.

Primero se define un conjunto de configuraciones de mapa de Kohonen, variando:

- el tamaño de la grilla ($m \times n$),
- el número de iteraciones de entrenamiento,
- el radio inicial de vecindad σ_0 ,
- y si se aplica o no normalización previa de las variables.

Para cada configuración, la celda:

1. Entrena un SOM sobre la matriz de datos original (500 muestras en un espacio de dimensión $D = 100$) llamando a `run_som_dim_reduction_experiment`, que además muestra la proyección de los datos en la grilla y el mapa de ocupación de neuronas.
2. Calcula la **matriz U** correspondiente utilizando dos definiciones de vecindad en la grilla (4 vecinos y 8 vecinos) mediante `run_u_matrix_analysis`, que combina:
 - el cómputo de la matriz U (distancia media a vecinos),
 - un resumen numérico de sus valores,
 - y su visualización como mapa de calor con las BMUs de los datos superpuestas.
3. Guarda, para cada combinación de hiperparámetros y tipo de vecindad, el SOM entrenado, la matriz U y los resultados de ocupación en la lista `all_u_results`, lo que permite comparar posteriormente cómo cambian la topología y la estructura de clusters cuando se modifican el tamaño del mapa, el número de iteraciones, la normalización y la vecindad utilizada en la U-matrix.

En conjunto, esta celda implementa un **experimento comparativo** que explora cómo distintos hiperparámetros de entrenamiento del SOM afectan la organización topológica del mapa y la detección visual de clusters a través de la matriz U.

```
In [ ]: print("Shape de data_matrix:", data_matrix.shape)

# Distintas configuraciones de SOM (tamaño de grilla, iteraciones, sigma0, normalización)
som_configs = [
    {
        "name": "10x10, it=5k, sigma0=5, sin norm",
        "m": 10,
        "n": 10,
        "n_iterations": 5_000,
        "alpha0": 0.5,
        "sigma0": 5.0,
        "normalize": False,
    },
    {
        "name": "10x10, it=5k, sigma0=5, con norm",
        "m": 10,
        "n": 10,
        "n_iterations": 5_000,
        "alpha0": 0.5,
        "sigma0": 5.0,
        "normalize": True,
    },
    {
        "name": "15x15, it=10k, sigma0=7.5, con norm",
        "m": 15,
        "n": 15,
        "n_iterations": 10_000,
        "alpha0": 0.5,
        "sigma0": 7.5,
        "normalize": True,
    },
    {
        "name": "20x20, it=10k, sigma0=10, con norm",
        "m": 20,
        "n": 20,
        "n_iterations": 10_000,
        "alpha0": 0.5,
        "sigma0": 10.0,
        "normalize": True,
    },
]
```

```

        },
        "name": "20x20, it=15k, sigma0=6, con norm",
        "m": 20,
        "n": 20,
        "n_iterations": 15_000,
        "alpha0": 0.5,
        "sigma0": 6.0,
        "normalize": True,
    },
]

# Distintos tipos de vecindad para la matriz U
u_neigh_types = ["4", "8"]

all_u_results = []

for cfg in som_configs:
    print("Config SOM:", cfg["name"])

    # Entrena SOM + resumen + proyecciones
    som, dim_results = run_som_dim_reduction_experiment(
        data=data_matrix,
        m=cfg["m"],
        n=cfg["n"],
        n_iterations=cfg["n_iterations"],
        alpha0=cfg["alpha0"],
        sigma0=cfg["sigma0"],
        name=cfg["name"],
        verbose=False,
        normalize=cfg["normalize"],
    )

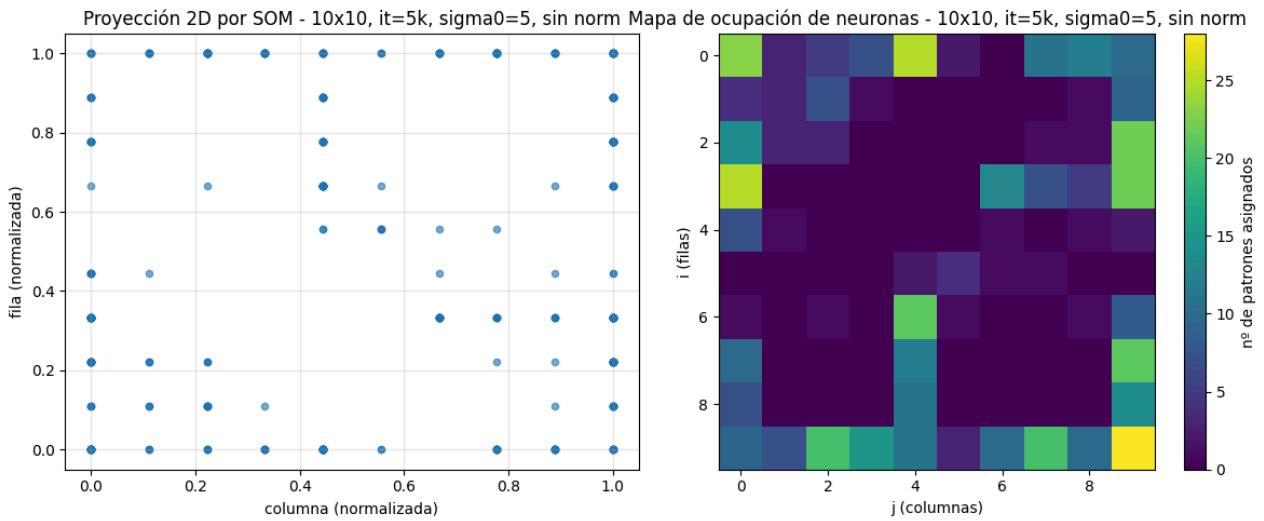
    # Para cada configuración de SOM, calcula y analiza la matriz U
    for neigh in u_neigh_types:
        u_name = f"{cfg['name']} - vecindad {neigh}"
        print(f"\n--- Matriz U para {u_name} ---")
        U = run_u_matrix_analysis(
            som,
            data=data_matrix,
            neigh_type=neigh,
            name=u_name,
        )

        all_u_results.append(
            {
                "config_som": cfg,
                "neigh_type": neigh,
                "som": som,
                "U": U,
                "dim_results": dim_results,
            }
        )
    )

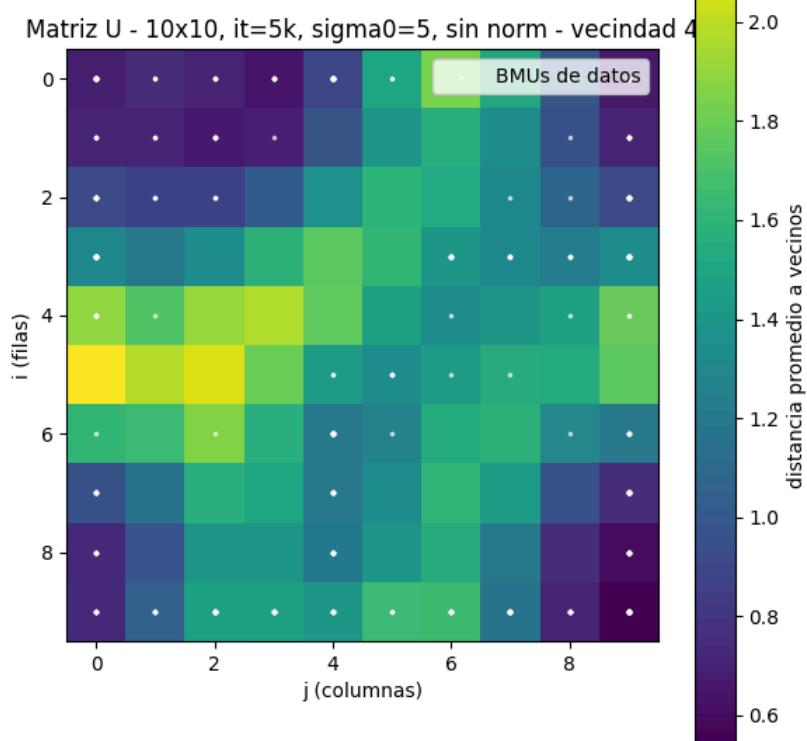
print("\nListo, se corrieron todas las configuraciones de SOM y sus matrices U.")

Shape de data_matrix: (500, 100)
Config SOM: 10x10, it=5k, sigma0=5, sin norm
== Resumen SOM dim-red - 10x10, it=5k, sigma0=5, sin norm ==
    Shape datos originales : (500, 100)
    Tamaño grilla SOM      : 10 x 10 (neur. totales = 100)
    Shape coords_ij (BMUs) : (500, 2)
    Ocupación - min         : 0
                           max       : 28
                           media     : 5.00
    Neuronas activas       : 57
    Primeras 10 coords_ij:
[[2 2]
 [4 0]
 [2 9]
 [1 9]
 [1 9]
 [0 0]
 [3 9]
 [2 9]
 [9 7]
 [9 7]]

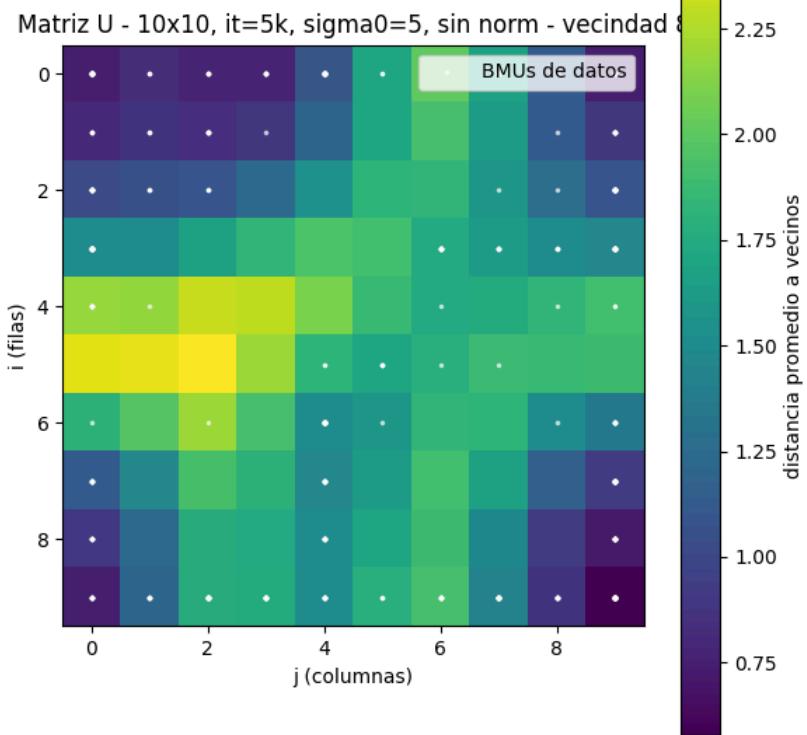
Matriz de ocupación (primeras 5 filas):
[[23 3 5 7 25 2 0 11 12 10]
 [ 4 3 7 1 0 0 0 0 1 9]
 [14 3 3 0 0 0 0 1 1 22]
 [25 0 0 0 0 0 13 7 5 22]
 [ 7 1 0 0 0 0 1 0 1 2]]
```



```
--- Matriz U para 10x10, it=5k, sigma0=5, sin norm - vecindad 4 ---
==== Resumen matriz U - 10x10, it=5k, sigma0=5, sin norm - vecindad 4 ====
Tamaño U          : 10 x 10
U min             : 0.5399
U max             : 2.1553
U media           : 1.2961
U desviación std : 0.3851
Histograma aproximado de U (5 bins):
[0.5399, 0.8630): 17
[0.8630, 1.1861): 16
[1.1861, 1.5091): 35
[1.5091, 1.8322): 24
[1.8322, 2.1553): 8
```

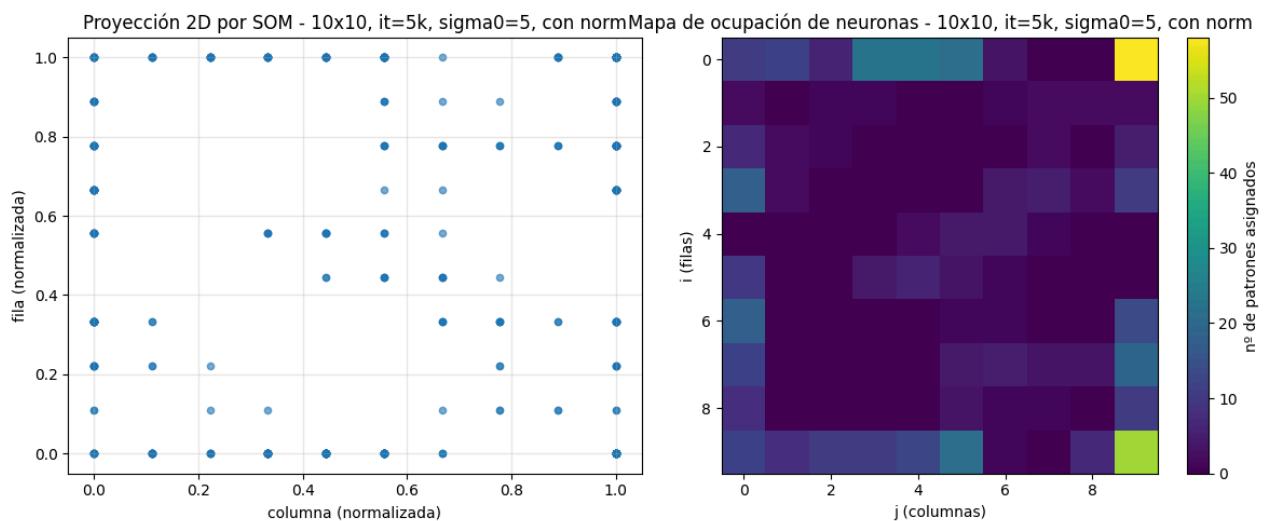


```
--- Matriz U para 10x10, it=5k, sigma0=5, sin norm - vecindad 8 ---
==== Resumen matriz U - 10x10, it=5k, sigma0=5, sin norm - vecindad 8 ====
Tamaño U          : 10 x 10
U min             : 0.5785
U max             : 2.4689
U media           : 1.5411
U desviación std : 0.4507
Histograma aproximado de U (5 bins):
[0.5785, 0.9566): 17
[0.9566, 1.3347): 14
[1.3347, 1.7128): 26
[1.7128, 2.0909): 33
[2.0909, 2.4689): 10
```



```
Config SOM: 10x10, it=5k, sigma0=5, con norm
==== Resumen SOM dim-red - 10x10, it=5k, sigma0=5, con norm ====
Shape datos originales      : (500, 100)
Tamaño grilla SOM          : 10 x 10 (neur. totales = 100)
Shape coords_ij (BMUs)      : (500, 2)
Ocupación - min             : 0
        max                 : 58
        media                : 5.00
Neuronas activas            : 59
Primeras 10 coords_ij:
[[1 0]
 [0 5]
 [5 4]
 [0 9]
 [0 9]
 [0 9]
 [0 3]
 [3 9]
 [0 9]
 [7 9]
 [7 9]]]

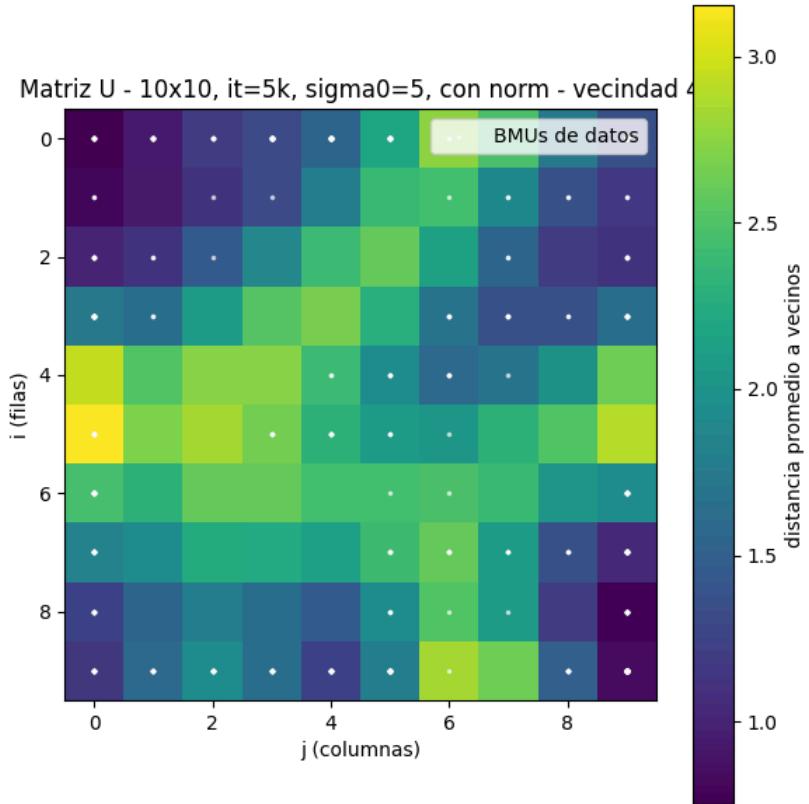
Matriz de ocupación (primeras 5 filas):
[[11 12  6 23 23 21  3  0  0 58]
 [ 2  0  1  1  0  0  1  2  2  2]
 [ 7  2  1  0  0  0  0  2  0  5]
 [18  2  0  0  0  0  4  5  2 11]
 [ 0  0  0  0  2  4  4  1  0  0]]
```



```

--- Matriz U para 10x10, it=5k, sigma0=5, con norm - vecindad 4 ---
== Resumen matriz U - 10x10, it=5k, sigma0=5, con norm - vecindad 4 ==
Tamaño U          : 10 x 10
U min             : 0.7521
U max             : 3.1540
U media           : 1.9234
U desviación std : 0.5969
Histograma aproximado de U (5 bins):
[0.7521, 1.2325): 16
[1.2325, 1.7129): 23
[1.7129, 2.1933): 23
[2.1933, 2.6736): 29
[2.6736, 3.1540): 9

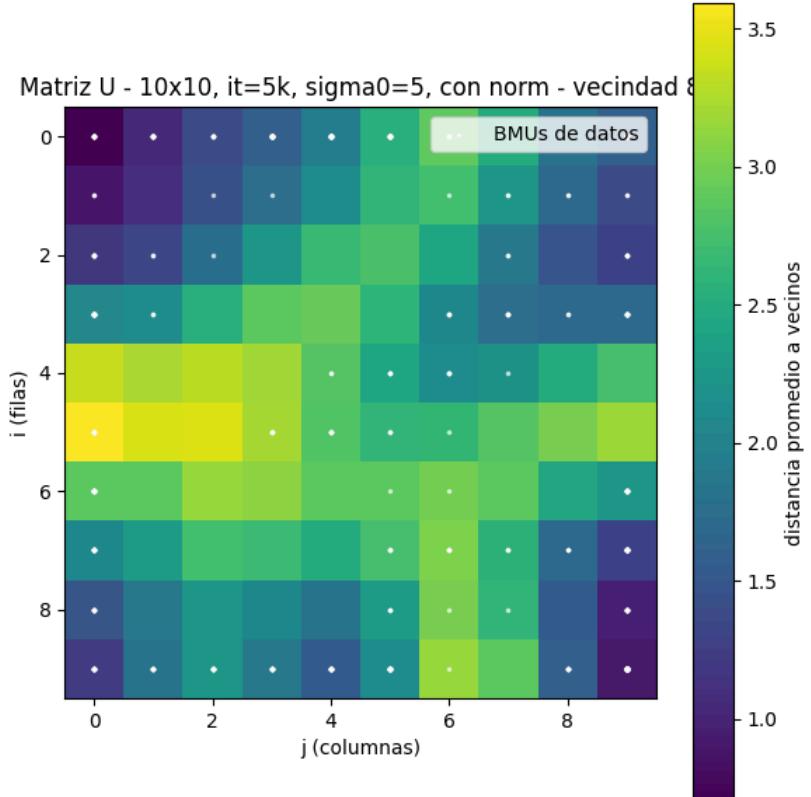
```



```

--- Matriz U para 10x10, it=5k, sigma0=5, con norm - vecindad 8 ---
== Resumen matriz U - 10x10, it=5k, sigma0=5, con norm - vecindad 8 ==
Tamaño U          : 10 x 10
U min             : 0.7002
U max             : 3.5923
U media           : 2.2740
U desviación std : 0.6834
Histograma aproximado de U (5 bins):
[0.7002, 1.2787): 8
[1.2787, 1.8571): 23
[1.8571, 2.4355): 22
[2.4355, 3.0139): 34
[3.0139, 3.5923): 13

```

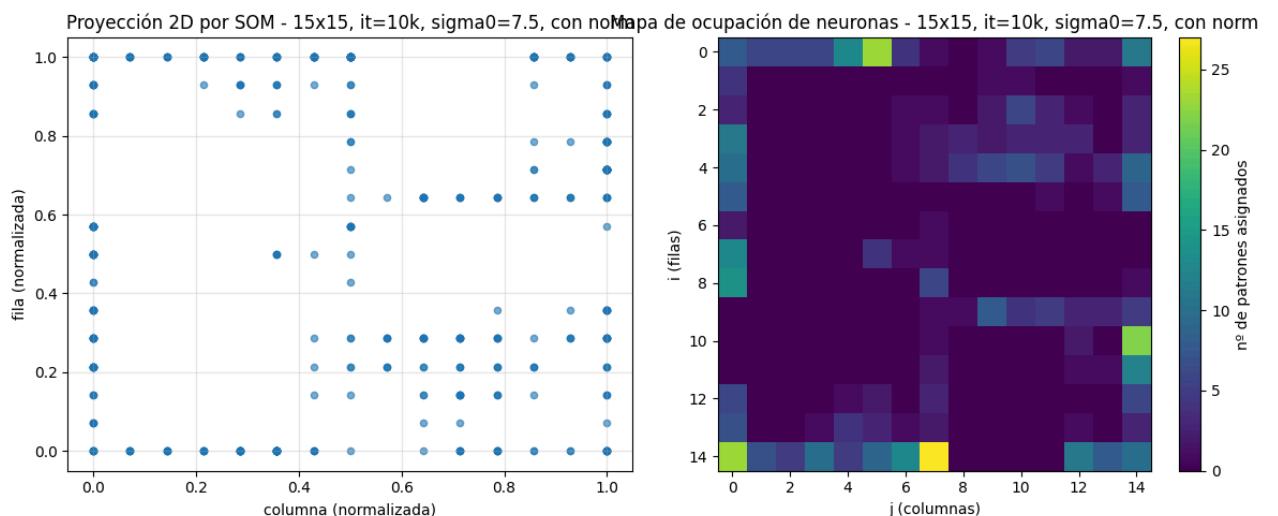


```
Config SOM: 15x15, it=10k, sigma0=7.5, con norm
==== Resumen SOM dim-red - 15x15, it=10k, sigma0=7.5, con norm ===
```

```
Shape datos originales      : (500, 100)
Tamaño grilla SOM          : 15 x 15 (neur. totales = 225)
Shape coords_ij (BMUs)     : (500, 2)
Ocupación - min             : 0
        max                 : 27
        media                : 2.22
Neuronas activas           : 97
Primeras 10 coords_ij:
[[12  5]
 [12  0]
 [10 14]
 [11 14]
 [11 14]
 [14  4]
 [10 14]
 [12 14]
 [ 3  9]
 [ 4 10]]
```

Matriz de ocupación (primeras 5 filas):

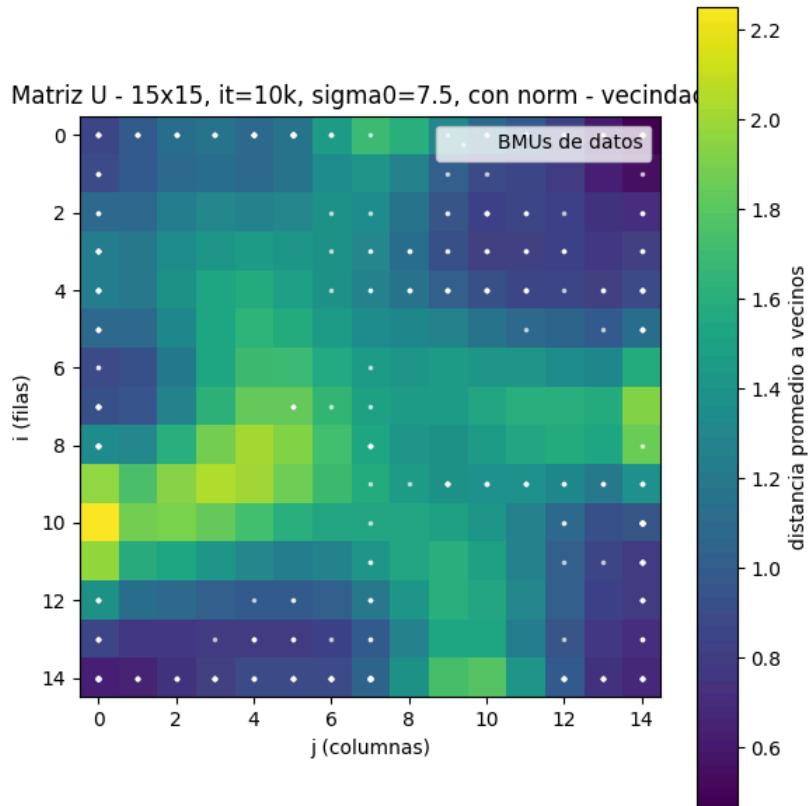
```
[[ 8  6  6  6 13 23  4  1  0  1  5  6  2  2 11]
 [ 4  0  0  0  0  0  0  0  0  1  1  0  0  0  1]
 [ 3  0  0  0  0  0  1  1  0  2  6  3  1  0  3]
 [11  0  0  0  0  0  1  2  3  2  3  3  3  0  3]
 [10  0  0  0  0  0  1  2  4  6  7  5  1  3  9]]
```



```

--- Matriz U para 15x15, it=10k, sigma0=7.5, con norm - vecindad 4 ---
== Resumen matriz U - 15x15, it=10k, sigma0=7.5, con norm - vecindad 4 ==
Tamaño U : 15 x 15
U min : 0.4696
U max : 2.2513
U media : 1.2630
U desviación std : 0.3494
Histograma aproximado de U (5 bins):
[0.4696, 0.8259]: 27
[0.8259, 1.1822]: 67
[1.1822, 1.5386]: 79
[1.5386, 1.8949]: 42
[1.8949, 2.2513]: 10

```

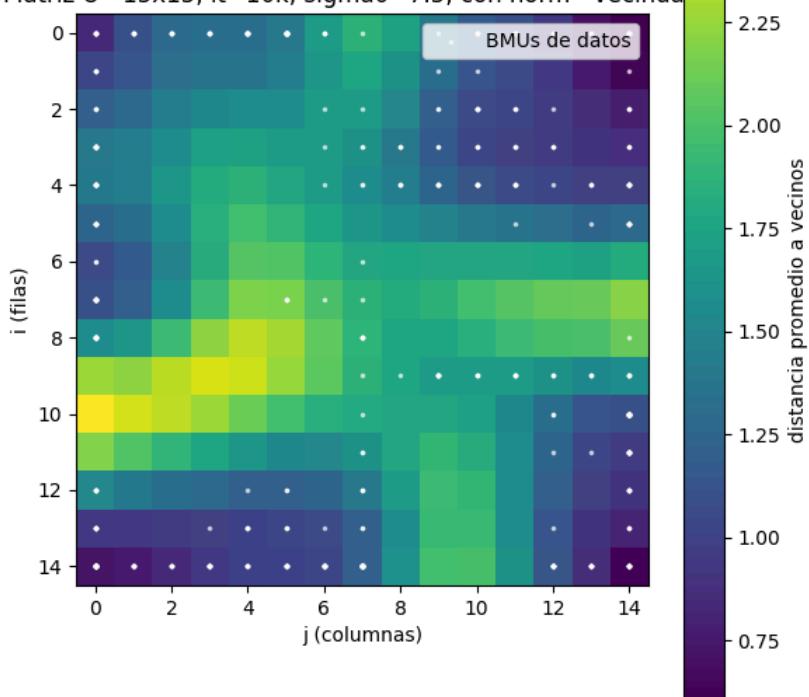


```

--- Matriz U para 15x15, it=10k, sigma0=7.5, con norm - vecindad 8 ---
== Resumen matriz U - 15x15, it=10k, sigma0=7.5, con norm - vecindad 8 ==
Tamaño U : 15 x 15
U min : 0.6074
U max : 2.5465
U media : 1.5173
U desviación std : 0.4283
Histograma aproximado de U (5 bins):
[0.6074, 0.9952]: 31
[0.9952, 1.3830]: 56
[1.3830, 1.7708]: 69
[1.7708, 2.1587]: 52
[2.1587, 2.5465]: 17

```

Matriz U - 15x15, it=10k, sigma0=7.5, con norm - vecindad



Config SOM: 20x20, it=10k, sigma0=10, con norm
==== Resumen SOM dim-red - 20x20, it=10k, sigma0=10, con norm ===

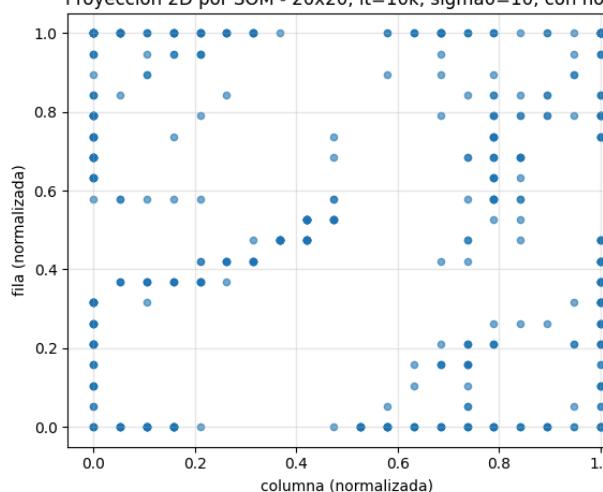
```

Shape datos originales      : (500, 100)
Tamaño grilla SOM          : 20 x 20 (neur. totales = 400)
Shape coords_ij (BMUs)     : (500, 2)
Ocupación - min             : 0
        max                 : 25
        media                : 1.25
Neuronas activas           : 135
Primeras 10 coords_ij:
[[ 3 13]
 [ 9 19]
 [19 14]
 [17 18]
 [19 17]
 [ 0 19]
 [14 15]
 [14 19]
 [15  0]
 [14  0]]
```

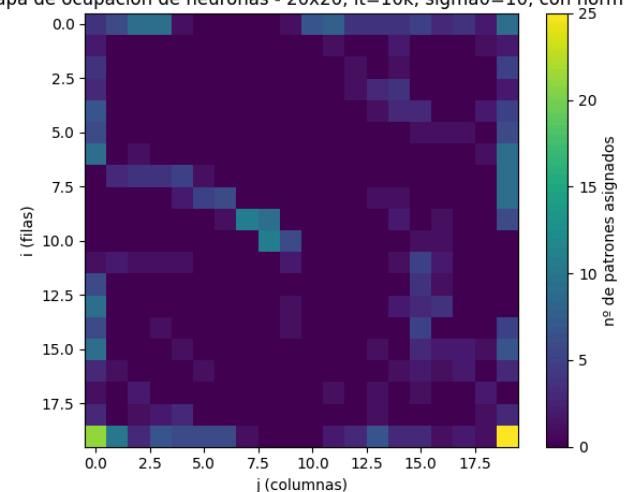
Matriz de ocupación (primeras 5 filas):

```
[[4 6 9 9 1 0 0 0 0 1 7 8 4 4 4 5 3 4 2 9]
 [2 0 0 0 0 0 0 0 0 0 1 0 0 2 0 0 0 1 2]
 [4 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 5]
 [3 0 0 0 0 0 0 0 0 0 0 1 3 4 0 0 0 0 2]
 [7 0 0 0 0 0 0 0 0 0 0 0 0 1 3 3 0 0 2 5]]
```

Proyección 2D por SOM - 20x20, it=10k, sigma0=10, con norm



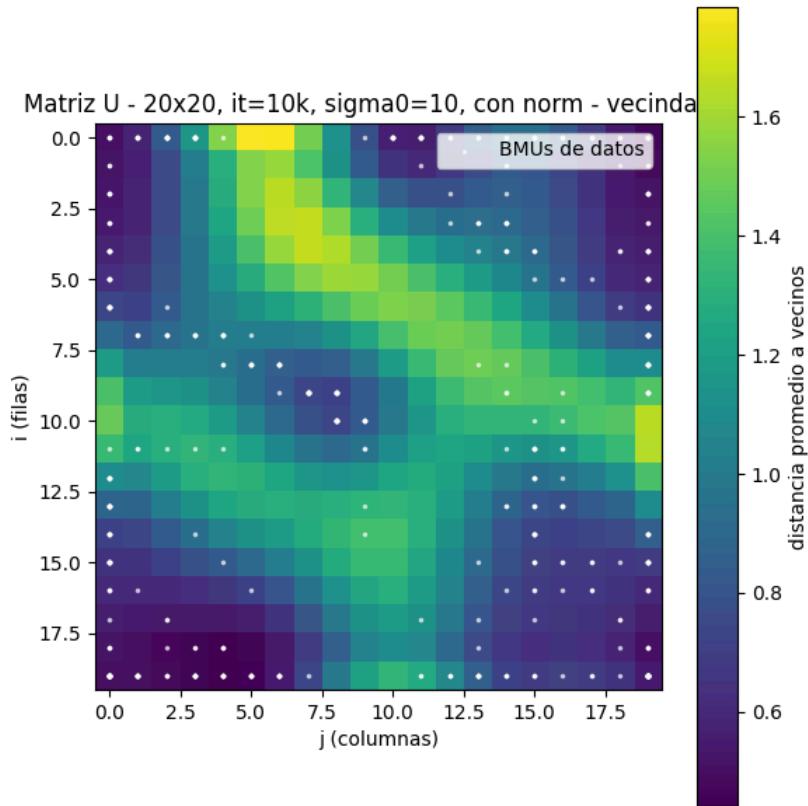
Mapa de ocupación de neuronas - 20x20, it=10k, sigma0=10, con norm



```

--- Matriz U para 20x20, it=10k, sigma0=10, con norm - vecindad 4 ---
==== Resumen matriz U - 20x20, it=10k, sigma0=10, con norm - vecindad 4 ====
Tamaño U          : 20 x 20
U min             : 0.4420
U max             : 1.7841
U media           : 0.9993
U desviación std : 0.3169
Histograma aproximado de U (5 bins):
[0.4420, 0.7104): 94
[0.7104, 0.9789): 103
[0.9789, 1.2473): 97
[1.2473, 1.5157): 86
[1.5157, 1.7841): 20

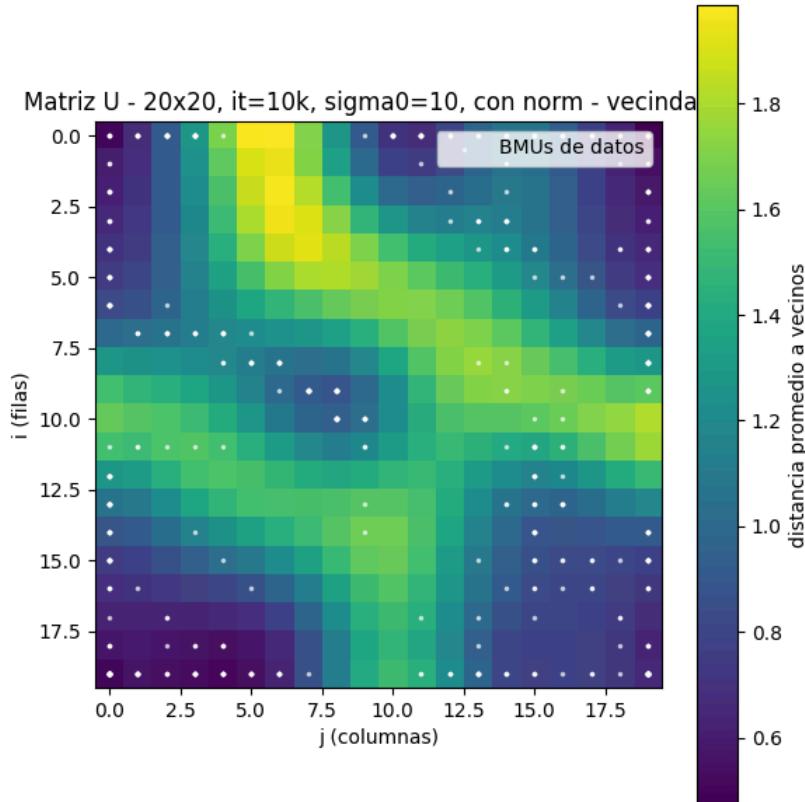
```



```

--- Matriz U para 20x20, it=10k, sigma0=10, con norm - vecindad 8 ---
==== Resumen matriz U - 20x20, it=10k, sigma0=10, con norm - vecindad 8 ====
Tamaño U          : 20 x 20
U min             : 0.4755
U max             : 1.9854
U media           : 1.1903
U desviación std : 0.3676
Histograma aproximado de U (5 bins):
[0.4755, 0.7775): 62
[0.7775, 1.0795): 105
[1.0795, 1.3815): 90
[1.3815, 1.6834): 104
[1.6834, 1.9854): 39

```



Config SOM: 20x20, it=15k, sigma0=6, con norm

Conclusión sobre la presencia de clusters según la matriz U

Los diferentes SOM entrenados (variando tamaño de grilla, iteraciones, normalización y vecindad) muestran patrones **consistentes** en las matrices U:

1. La matriz U evidencia separación estructural

En todas las configuraciones, los valores de la matriz U presentan rangos amplios (por ejemplo):

- 10x10 sin normalización: $U_{\max} \approx 2.43$
- 20x20 con normalización: $U_{\max} \approx 1.98$
- 20x20 con normalización y 15k iteraciones: $U_{\max} \approx 3.45$

Estas diferencias relativamente grandes entre zonas indican **discontinuidades en el espacio de pesos**, típicas de la presencia de **fronteras entre clusters**.

2. Los histogramas muestran multimodalidad ligera

Ninguna matriz U es completamente homogénea:

- Siempre hay bins “bajos” (valores chicos → regiones intra-cluster).
- Y bins “altos” (valores grandes → bordes entre clusters).

Esto es compatible con la existencia de **varios grupos**, aunque no extremadamente separados.

3. La normalización mejora la definición

Las configuraciones *con normalización* muestran:

- valores máximos de U más altos,
- desviaciones estándar mayores,
- fronteras más marcadas visualmente.

Interpretación: la **normalización reduce la dominancia de ciertas features** y permite que la topología del SOM se organice mejor, haciendo más visibles las transiciones entre regiones.

4. Tamaños de grilla grandes (15x15 y 20x20)

Las grillas más grandes:

- muestran **más detalle y subdivisiones internas**,

- facilitan identificar zonas de baja U claramente rodeadas por alta U
→ indicio de **clusters bien definidos**, aunque no necesariamente muy compactos.

5. ¿Cuántos clusters se observan?

A partir de los patrones repetidos en las matrices U (islas de U baja rodeadas por “cordones” de U alta), se puede estimar que:

- aparecen de forma consistente **entre 3 y 5 clusters principales** a lo largo de la mayoría de las configuraciones,
- no se observan más grupos sin entrar en zonas ruidosas o subdivisiones muy finas que parecen artefactos del mapa más que clusters bien formados.

En conjunto, la matriz U respalda la interpretación de que los datos presentan una **estructura agrupada no trivial**, con varios clusters moderadamente separados, cuya presencia es robusta frente a cambios razonables en los hiperparámetros del SOM.