Ejercicio 1

Red de Kohonen

Implementación

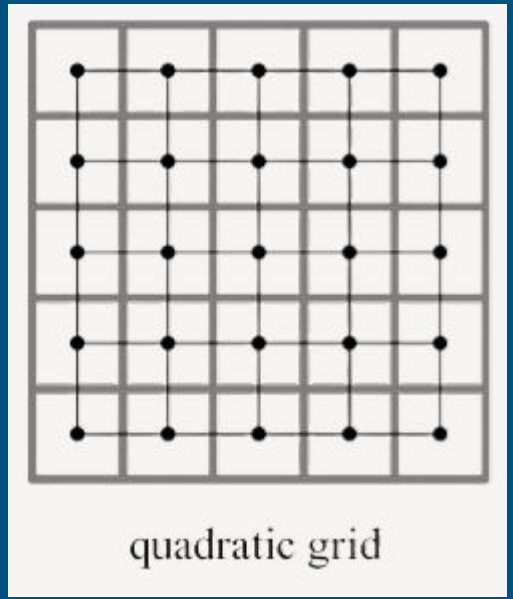
UML Class Diagram

Kohonen

- k: int
- grid: [k, k, n]
- winners: [k, k, float]
- initial_radius: float
- radius: float
- initial_eta: float
- eta: float
- decay_factor: float
- threshold: float
- iteration: int
- max_iterations: int
- print_neighborhood: bool
- print_iteration_results: bool
- print_final_results: bool

- init(...config_data)
- initialize_weights_random()
- initialize_weights_input()
- tuple[int, int] find_winner_neuron(np.ndarray input_data)
- update_weights(np.ndarray input_data, tuple winner_indexes)
- bool has_converged()
- print_iteration_results()
- print_final_results()
- plot_heatmap_u_matrix()
- plot_heatmap_final_entries(bool delete_smalls)
- plot_heatmap_variable(int variable_index, str variable_name)
- create_neighborhood_gif()
- start()

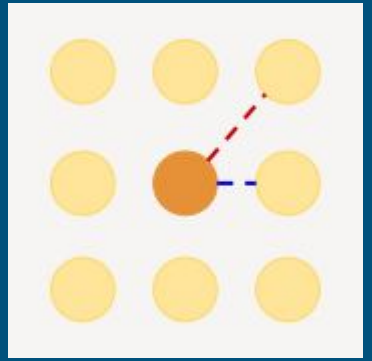
Grid



Similitud

$$W_{\hat{k}} = \arg\min_{1 \le j \le N} \{d(X^p - W_j)\}$$

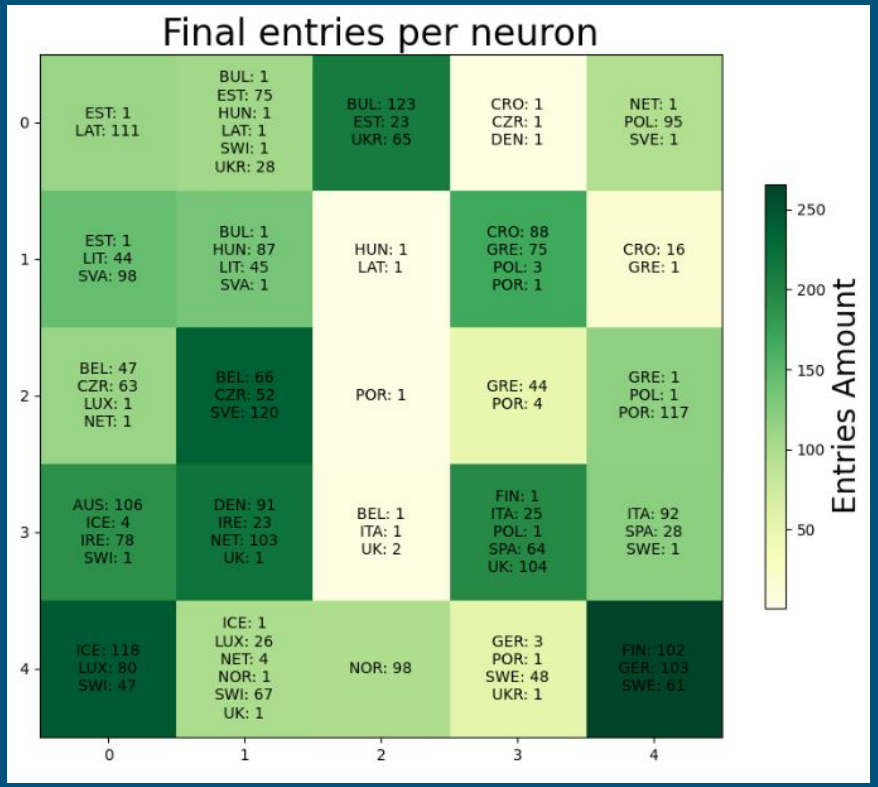
Neighborhood



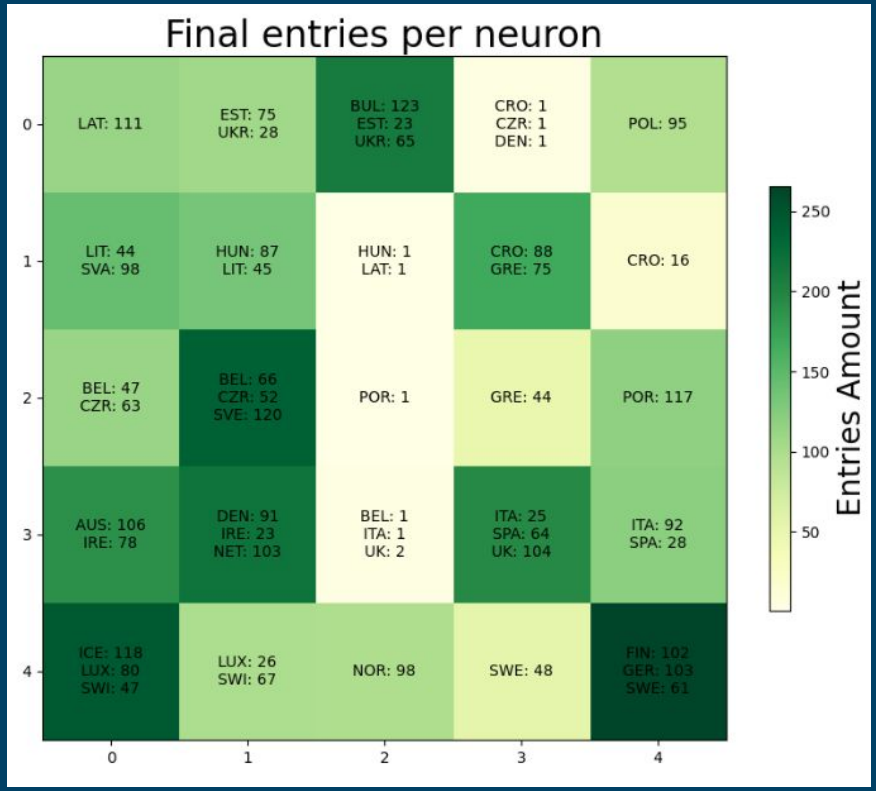
Euclidean Distance

4-vecinos → R = 1
8-vecinos → R = sqrt(2)

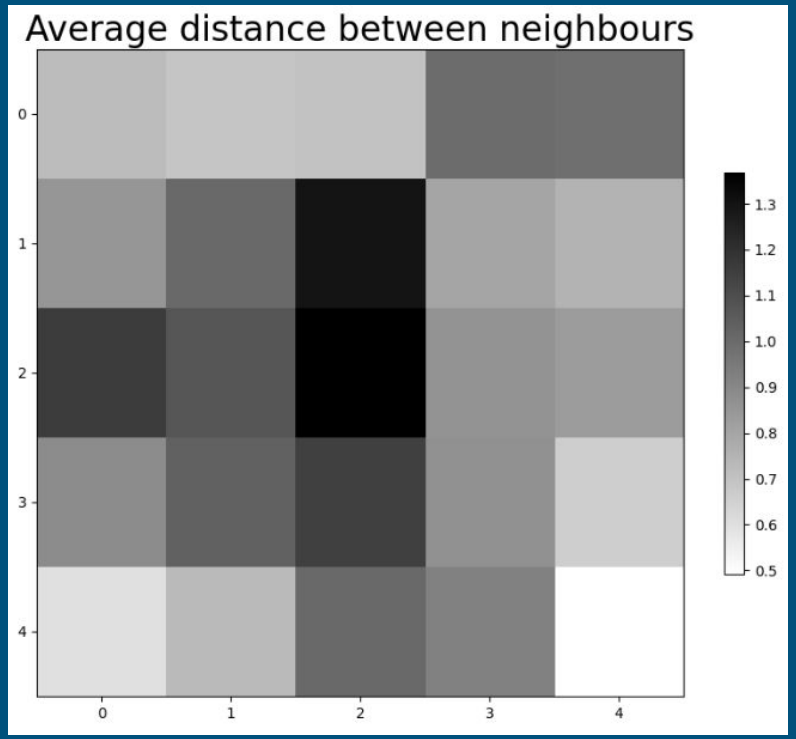
Heatmap Final



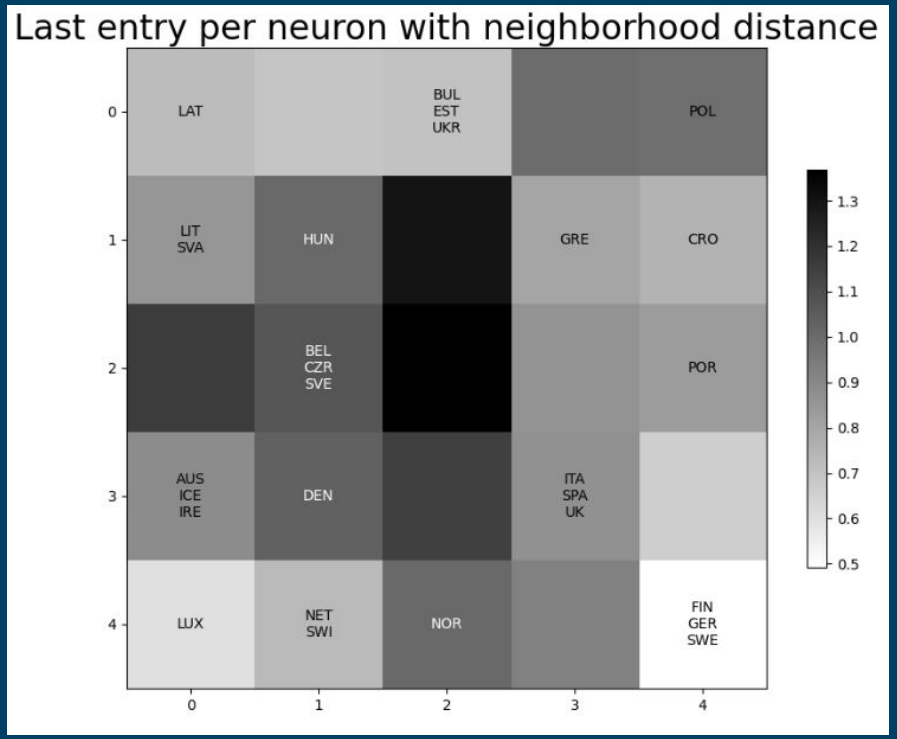
Sin “Pequeños”



U-Matrix

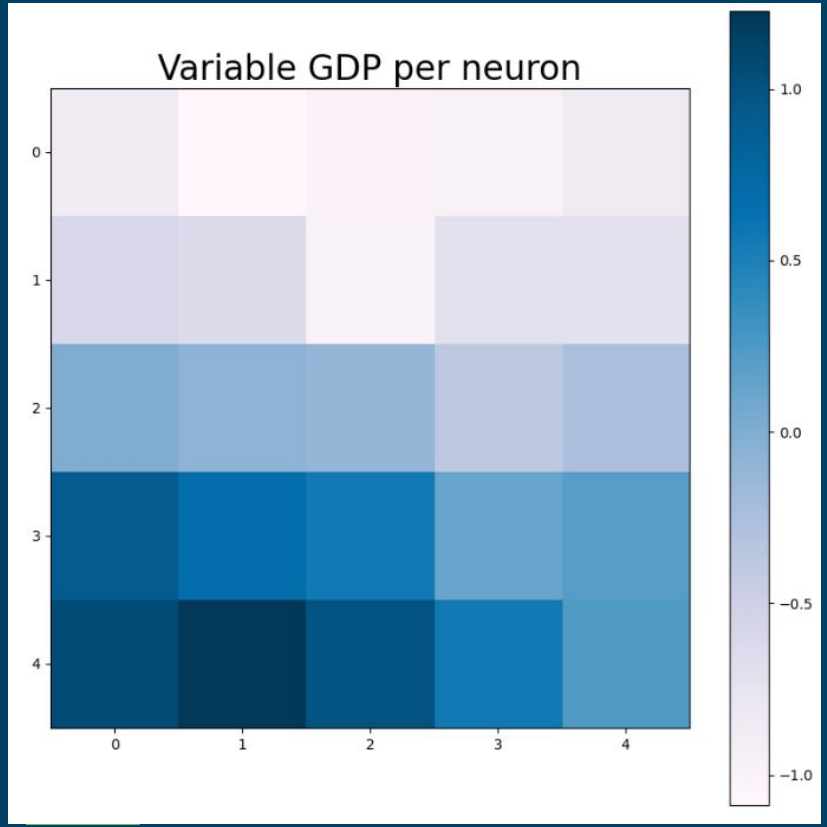


Last entry

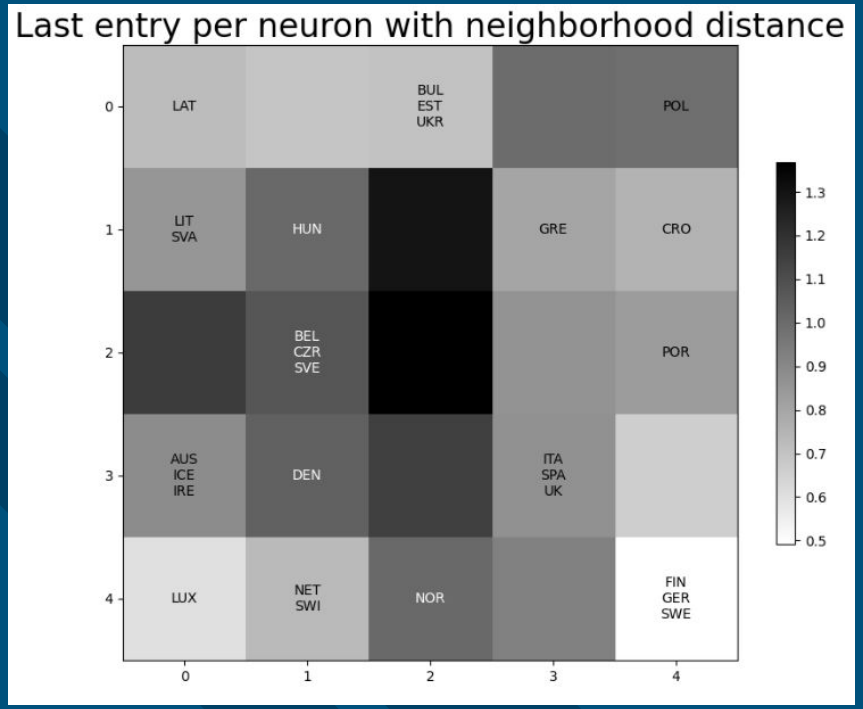


¿Y cómo se ven las variables?

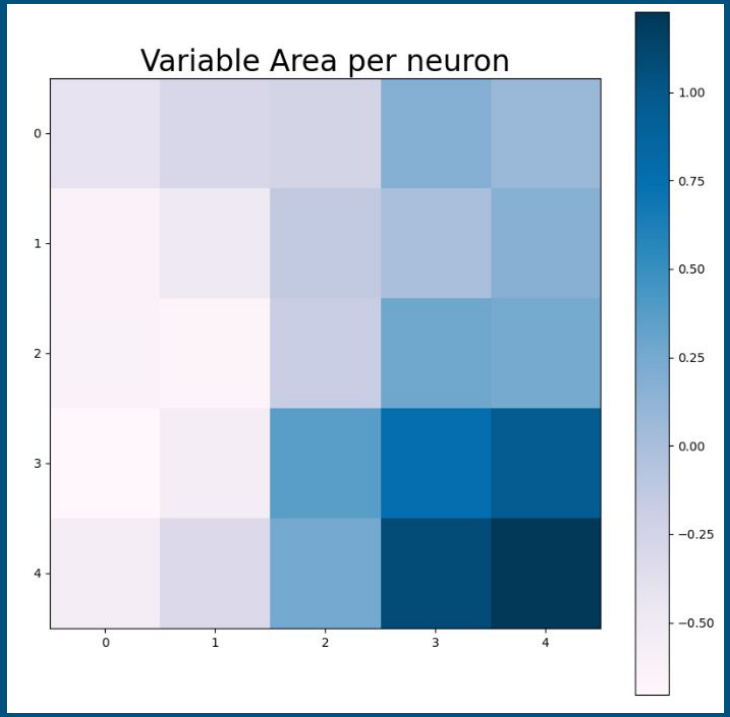
Ejemplo: Plot del GDP por neurona



Entradas finales vs algunas variables (I)

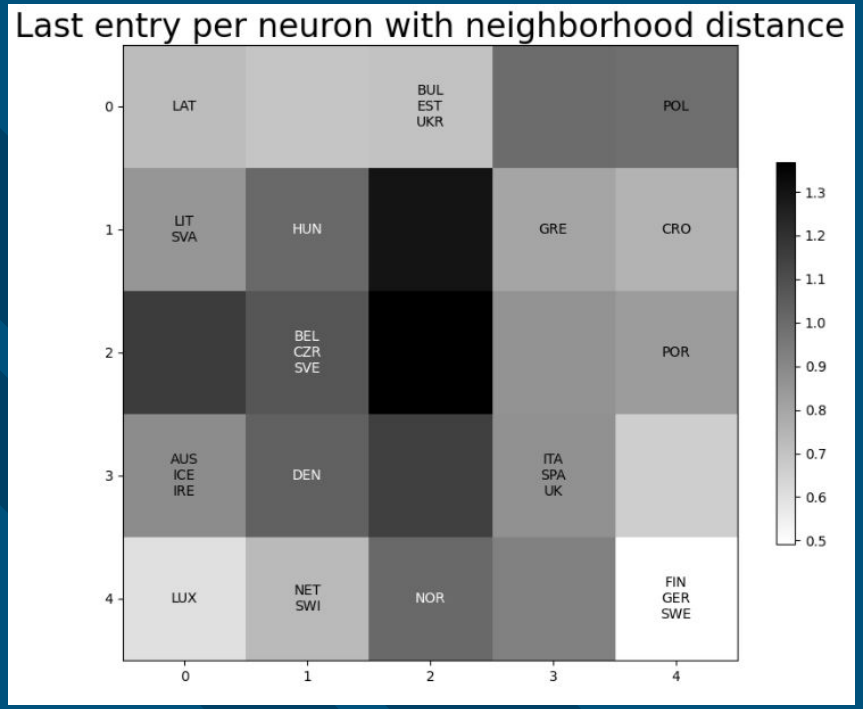


Entradas Finales

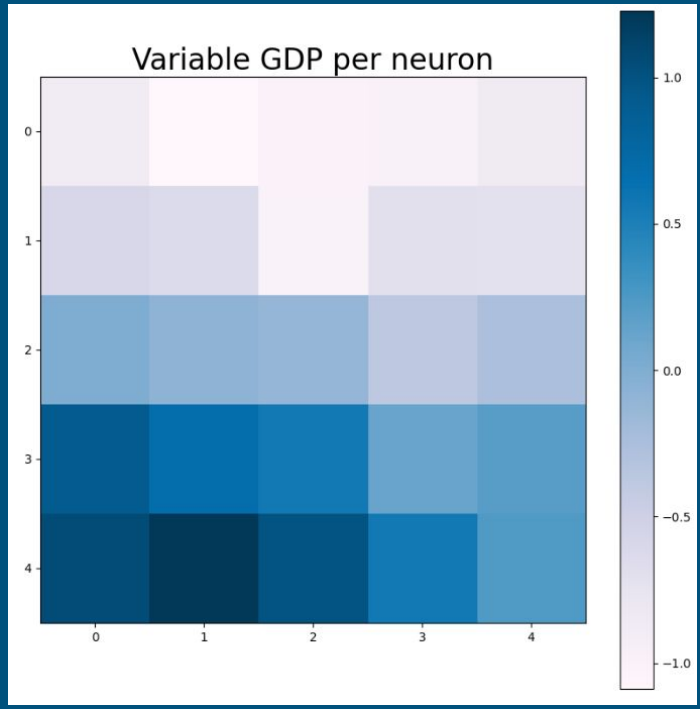


Variable Area

Entradas finales vs algunas variables (II)

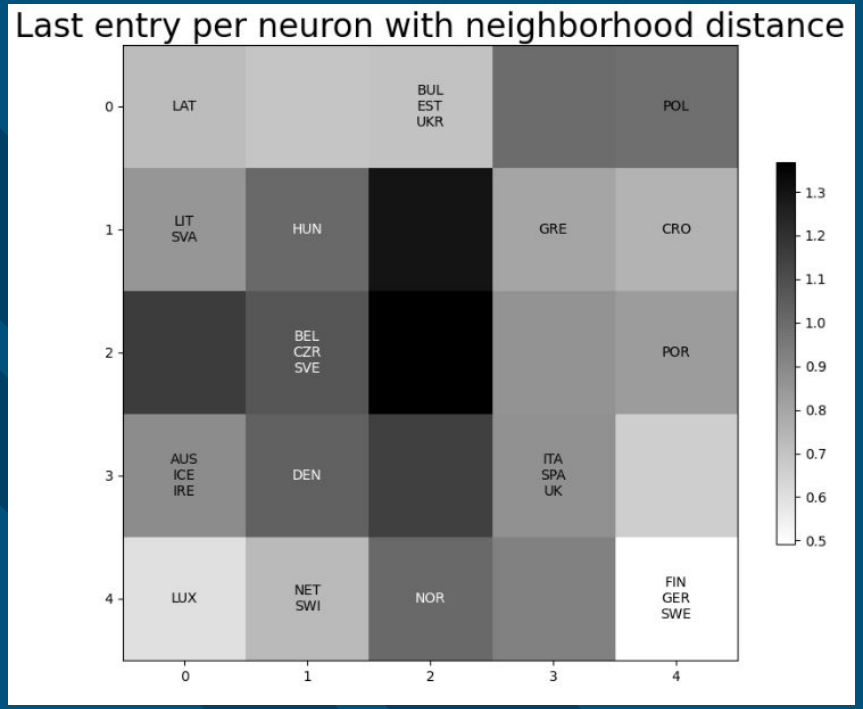


Entradas Finales

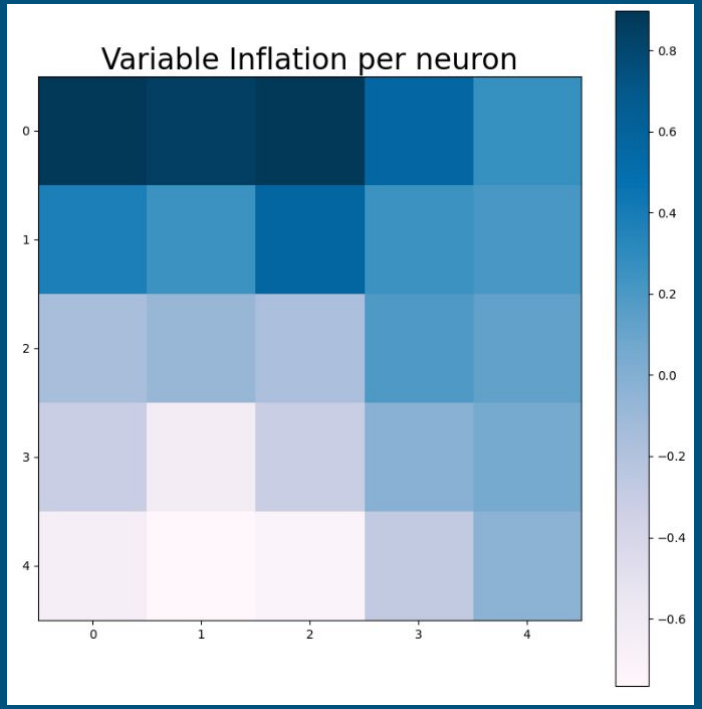


Variable GDP

Entradas finales vs algunas variables (III)



Entradas Finales



Variable Inflation

Conclusiones (I)

- El análisis de las **distancias euclídeas** entre los **vectores de pesos de las neuronas y sus vecinos** ofrece información sobre la organización y coherencia de los grupos en el mapa.

- **Si las distancias euclídeas entre una neurona y sus vecinos son pequeñas**, indica que los vectores de pesos son similares entre sí. Esto sugiere que las neuronas **se encuentran en un mismo grupo** o cluster coherente.

- Por el contrario, **si las distancias euclídeas entre una neurona y sus vecinos son grandes**, indica que los vectores de pesos son diferentes entre sí. Esto sugiere una **separación o disociación entre las neuronas**, lo que puede indicar una **mayor diversidad** o dispersión en los grupos del mapa.

Conclusiones (II)

- **En diferentes iteraciones, los mapas de clasificación pueden ser distintos.**
- Es posible que en una iteración España, Italia y Reino Unido estén cerca debido a similitudes en términos de características o relaciones en los datos, mientras que en otra iteración pueden estar separados debido a nuevas similitudes o diferencias que han sido capturadas por el modelo durante el entrenamiento.

Implementación

Constantes

initial_radius	3.5
initial_eta	0.9
max_iterations	3000
threshold	10^{-5}
decay_factor	0.96

➔ Imprimimos solo los resultados finales

Variables

Inicialización de pesos
k

¿Cómo actualizamos el radio y η?

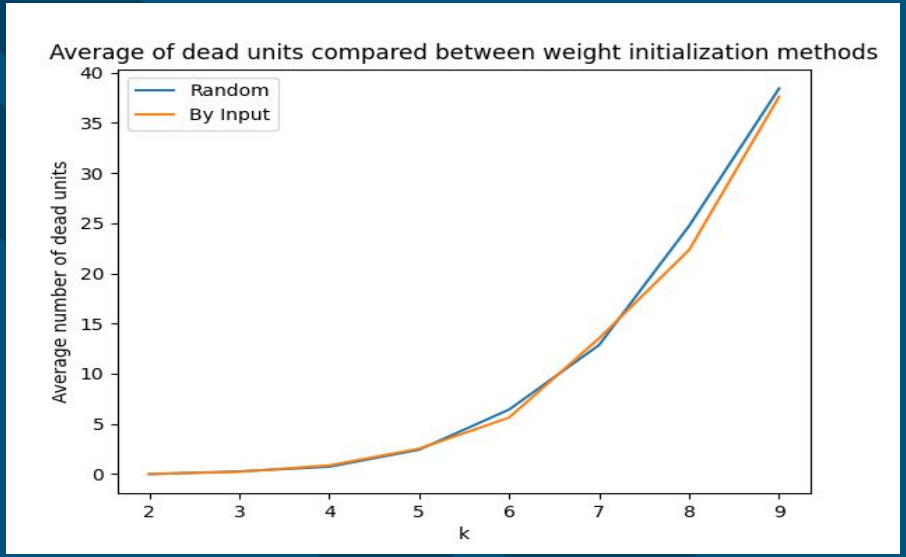
➔ $radius_k = max(1, radius_0 * decayFactor^k)$

➔ $\eta_k = max(0.001, \eta_0 * decayFactor^k)$

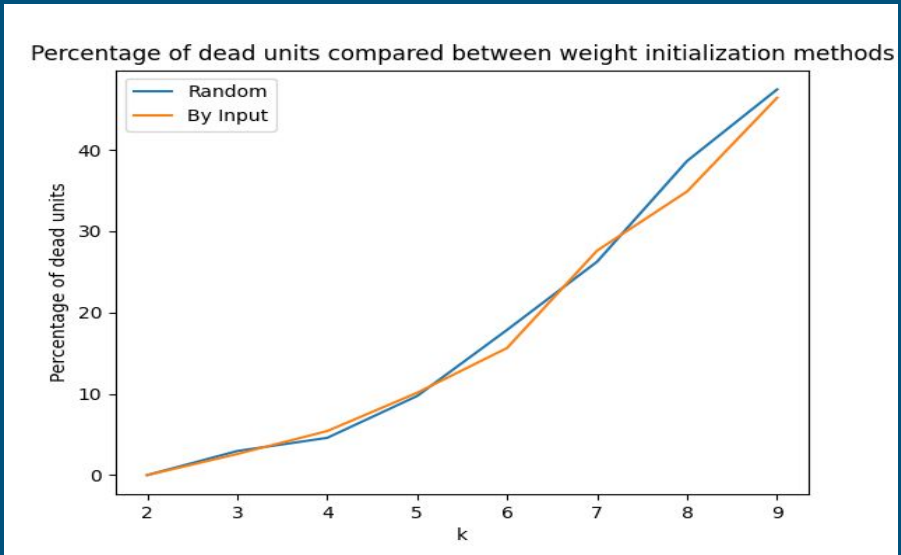
Inicializar los pesos aleatorios vs con valores de entrada (I)

Cantidad de unidades muertas - 30 iteraciones

Promedio



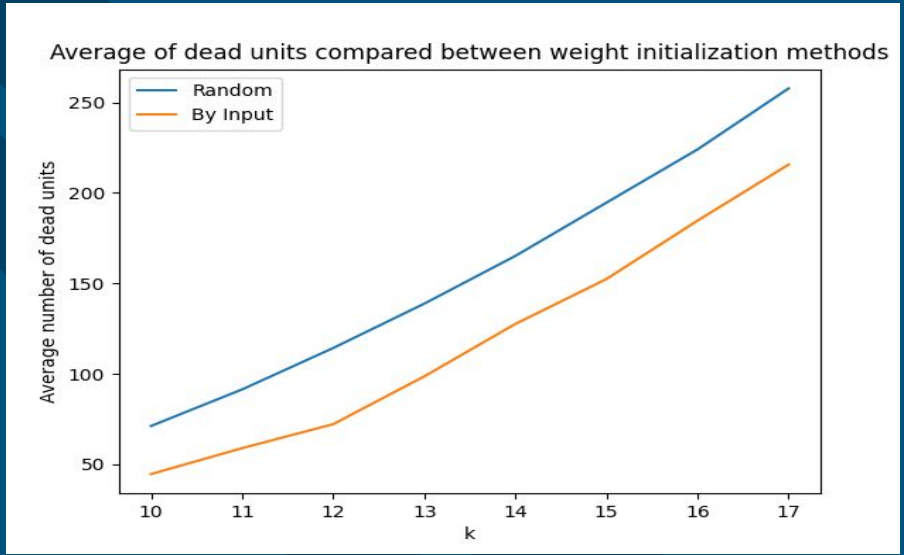
Porcentaje



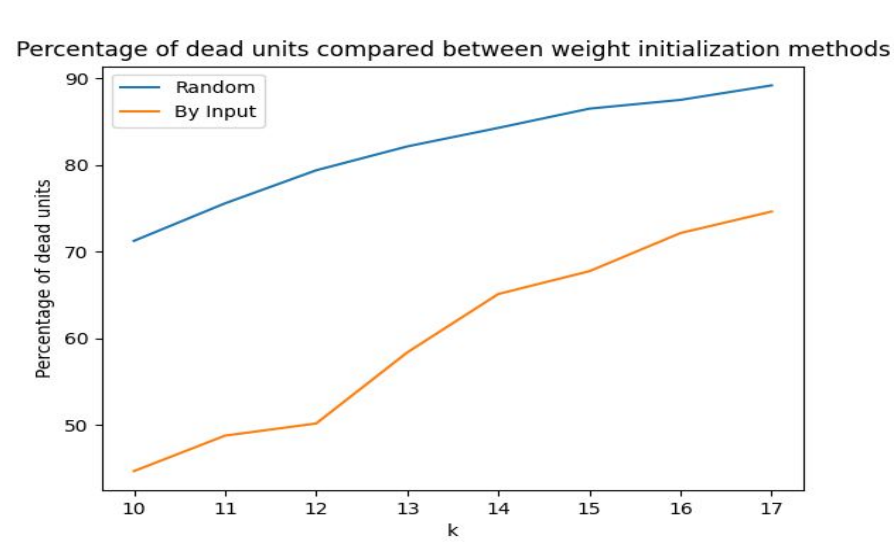
Inicializar los pesos aleatorios vs con valores de entrada (II)

Cantidad de unidades muertas - 30 iteraciones, k ≥ 10

Promedio



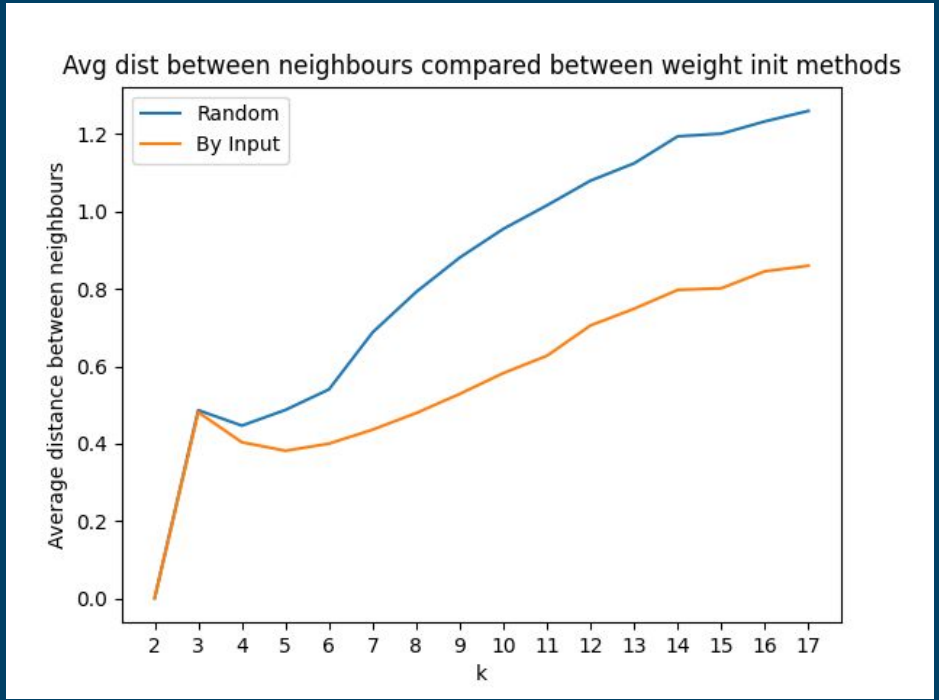
Porcentaje



¡No tiene sentido estudiarlo!

Inicializar los pesos aleatorios vs con valores de entrada (III)

- ➔ 30 iteraciones
- ➔ Graficamos la distancia promedio entre vecinos



Conclusiones (III)

- **Aumentar** el *k*, **aumenta** la cantidad (y proporción) de *neuronas no ganadoras*.

- Al tener un número reducido de inputs, ganarán algunas y convergerán a los valores de dichas entradas, por lo que tenderán a ganar siempre, haciendo que algunas neuronas nunca ganen por más que se hagan *muchas* iteraciones.
- Estas neuronas no nos aportan nada, ya que no generan clasificaciones.
- Podemos analizar la eficiencia y estabilidad del mapa mirando los gráficos de cantidad de neuronas "muertas" en función de los parámetros del algoritmo.

Analicemos otros parámetros (I)

Constantes

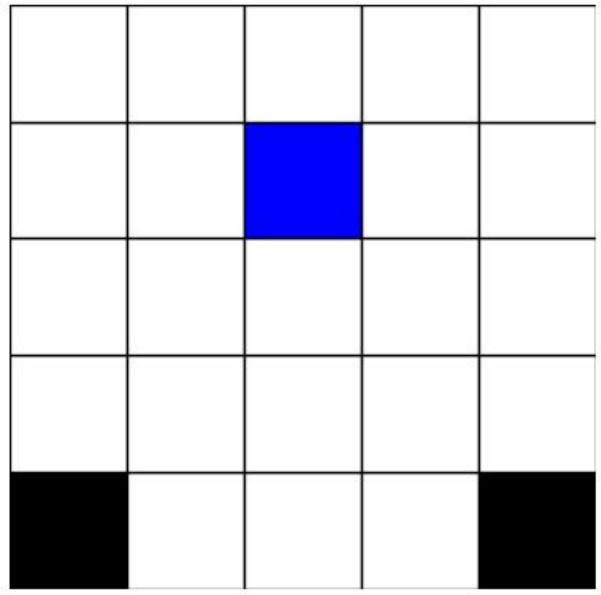
max_iterations	3000
threshold	10^{-4}
Inicialización de pesos	By Input

Variables

decay_factor
initial_radius
initial_eta
k

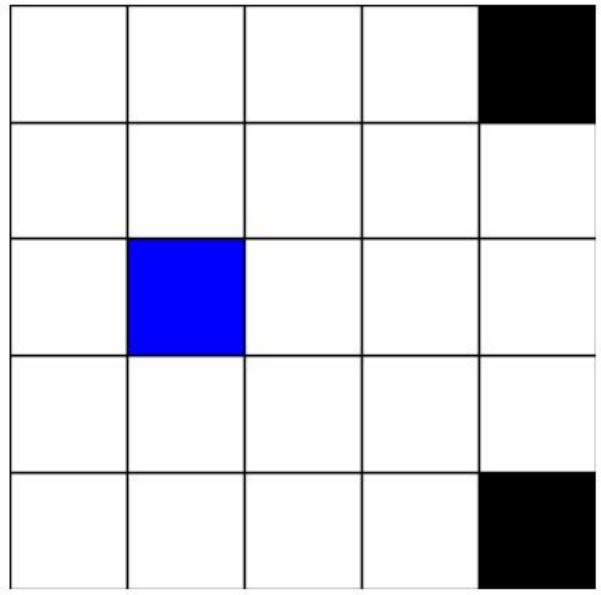
Analicemos otros parámetros (II)

Neighborhood and radius evolution con decay factor 0.96



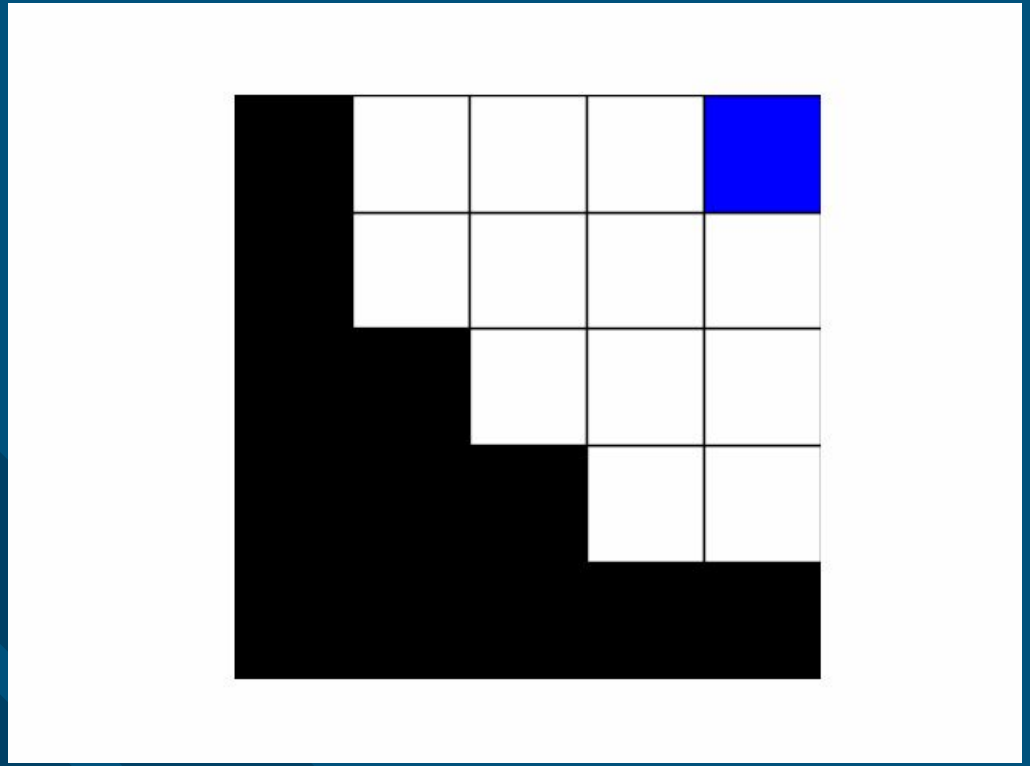
Analicemos otros parámetros (II)

Neighborhood and radius evolution con decay factor 0.975



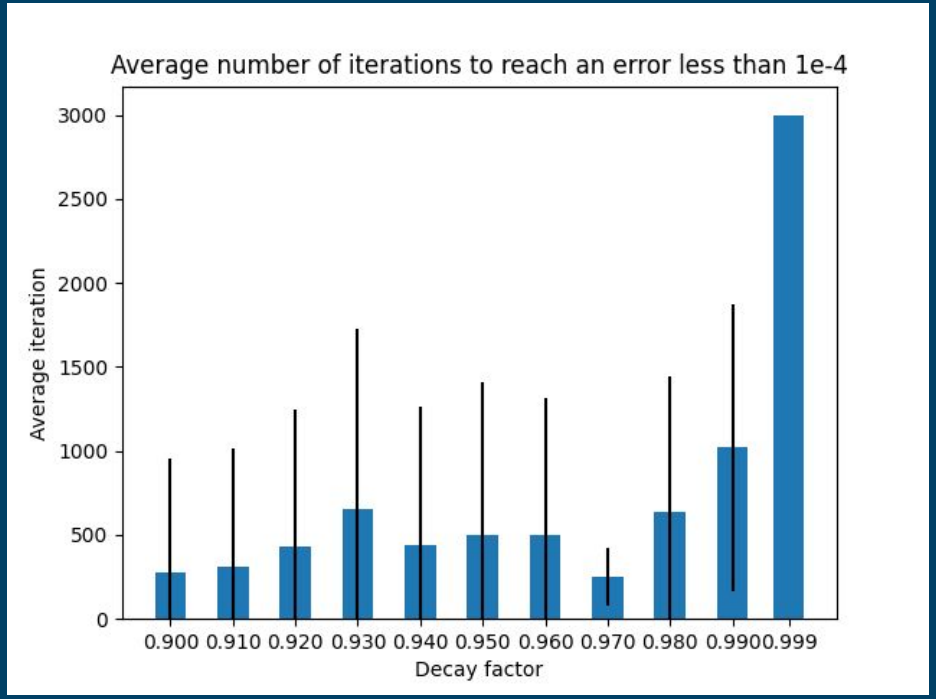
Analicemos otros parámetros (II)

Neighborhood and radius evolution con decay factor 0.98



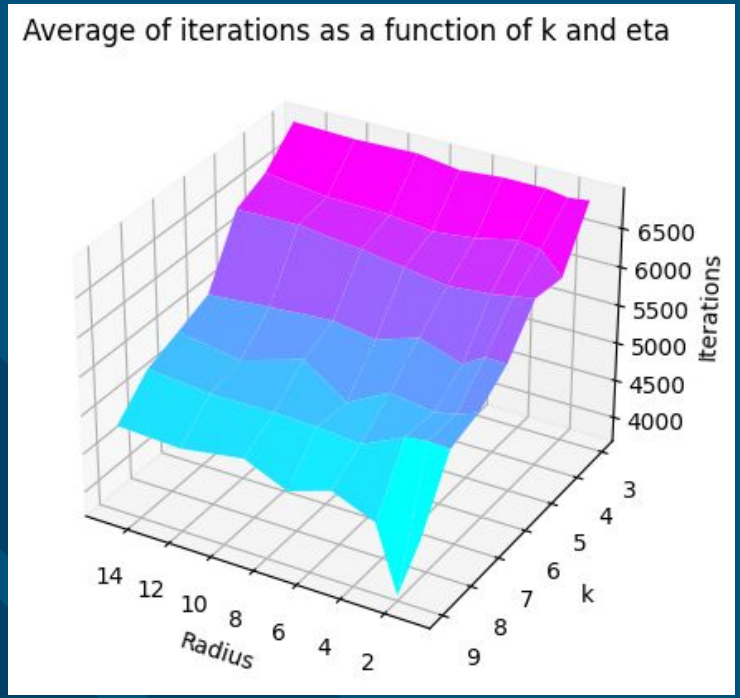
Analicemos otros parámetros (III)

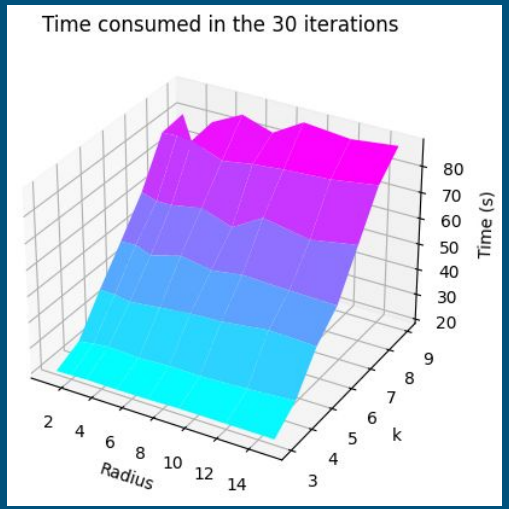
- Radio inicial de 3.5
- Taza de aprendizaje inicial de 0.9
- Grilla de tamaño k = 5
- 30 iteraciones



Analicemos otros parámetros (III)

Average number of iterations to reach an error less than 10^{-4}

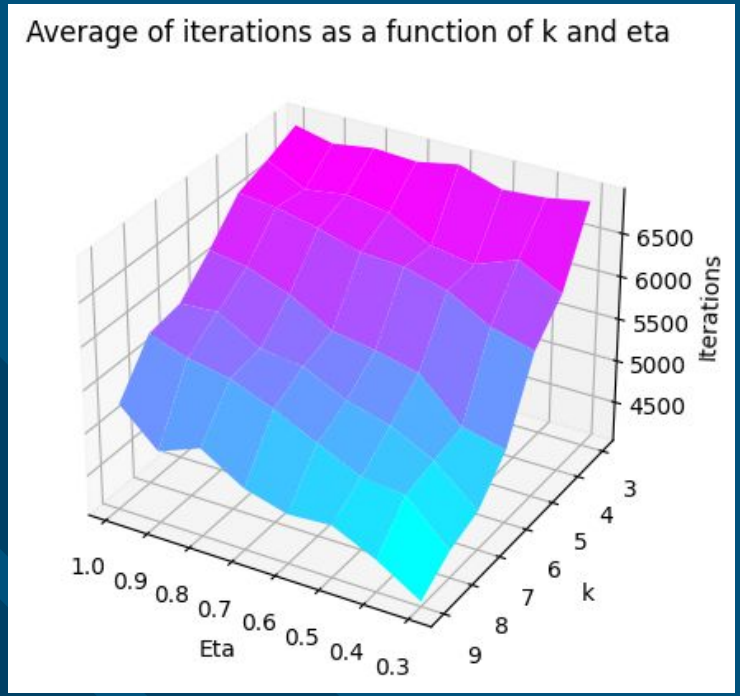


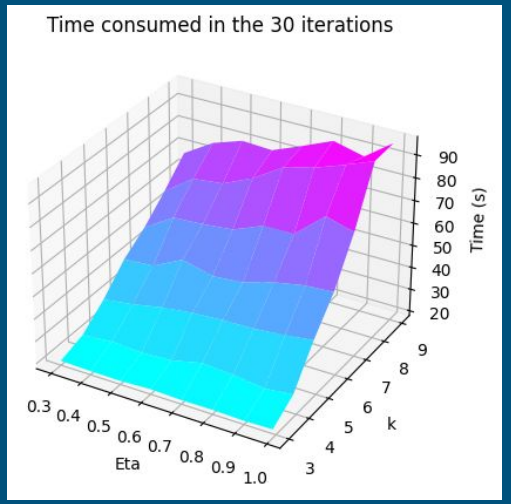


30 Iteraciones
Eta inicial de 0.9
Decay factor = 0.999

Analicemos otros parámetros (III)

Average number of iterations to reach an error less than 10^{-4}





30 Iteraciones
Radio inicial de 3.5
Decay factor = 0.999

Conclusiones (IV)

- El **initial_radius** e **initial_eta** son parámetros que también afectan a la convergencia. Igualmente, dependiendo del **decay_factor**, suelen tener más importancia en las primeras iteraciones del programa. Valores “altos” pero que luego vayan decayendo, suelen dar los mejores resultados.

- Un valor mayor de "**initial_radius**" implica un vecindario más amplio, lo que permite una exploración inicial más extensa del espacio de características.

- Un valor alto de "**initial_eta**" implica un ajuste más rápido de los pesos, lo que puede conducir a una convergencia más rápida pero también puede provocar una oscilación excesiva en el proceso de entrenamiento. Un valor bajo de "**initial_eta**" permite un ajuste más lento y suave de los pesos.

- **Requieren ajustes y experimentación para obtener los mejores resultados.**

Conclusiones (V)

¿Para qué sirve el algoritmo de Kohonen?

- El **algoritmo de Kohonen** es una técnica útil para la clasificación y organización de datos, especialmente en problemas de reducción de dimensionalidad y visualización.

Ejercicio 2

Modelo de Oja

Implementación

- Perceptrón Lineal con una capa de salida
- Δw con fórmula de Oja

$$\nabla w = \eta(O x_i - O_5 w_i)$$

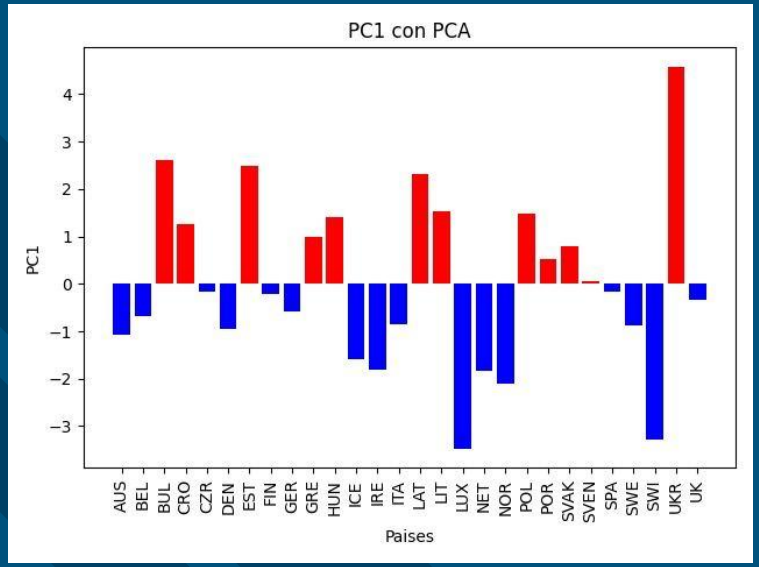
- Datos estandarizados
- Pesos iniciales aleatorios uniformes entre 0.5 y -0.5
- Para cálculo de PCA se utilizó la librería sklearn

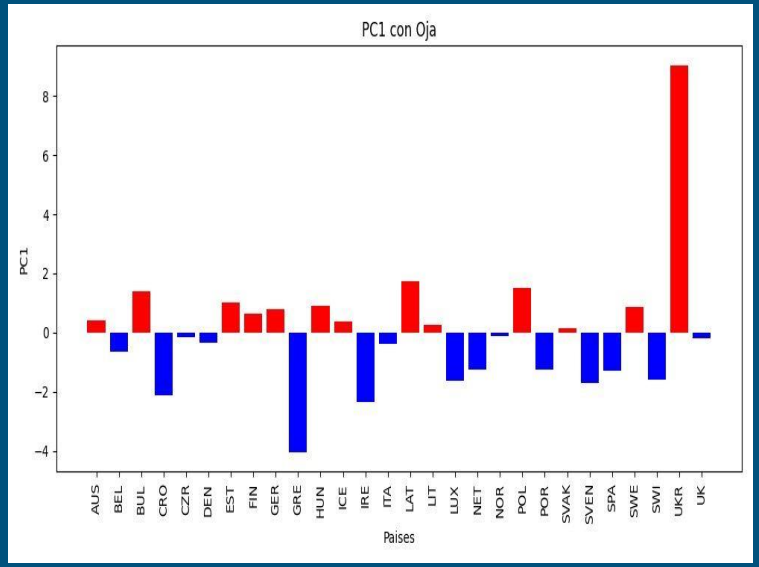
PCA vs. Oja

Objetivos

- ➔ Evaluaremos la capacidad del Modelo de Oja para aproximar la PC1
- ➔ Analizaremos la aproximación para distintos valores de η

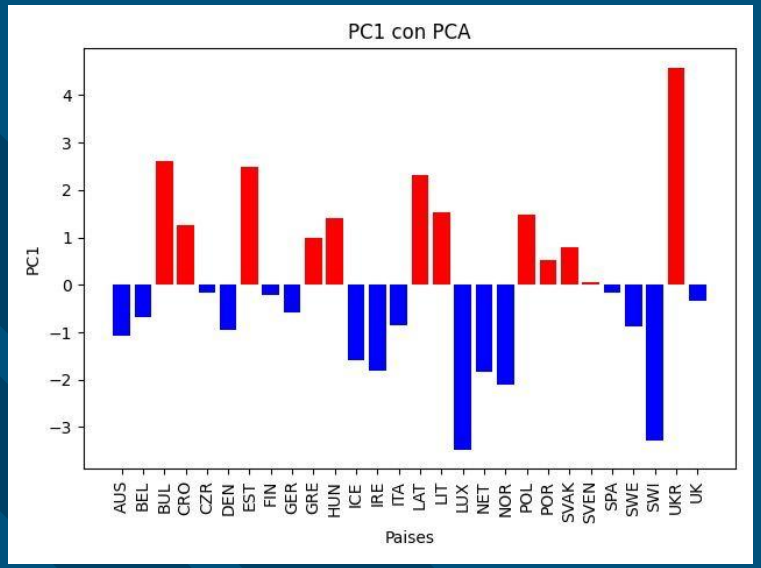
Analizamos Resultados para distintos η
Epochs: 10000 ~ η: 10^{-1}

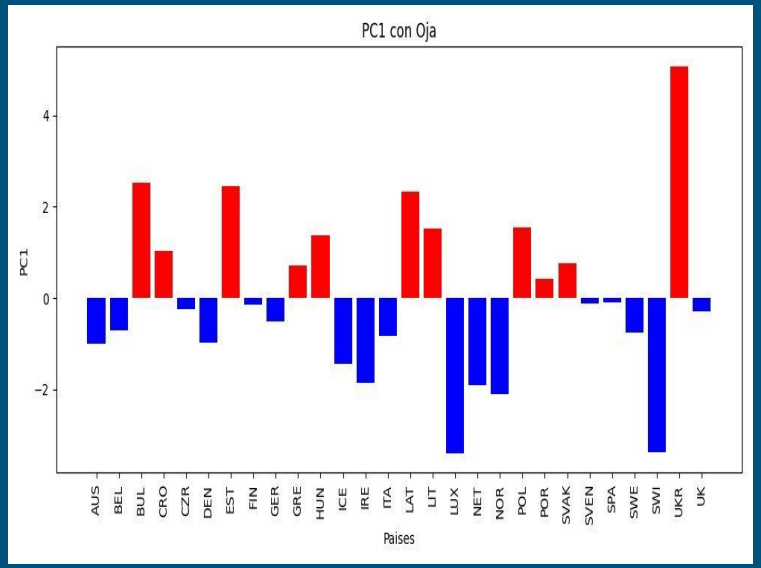




	Área	GDP	Inflation	Life.expect	Military	Pop.growth	Unemployment
PCA	0.1248739	-0.50050586	0.40651815	-0.48287333	0.18811162	-0.47570355	0.27165582
Oja	0.97357514	-0.43844389	0.59036815	-1.04145423	-0.52176422	0.01039984	-0.96686829
Error Promedio							0.58395677

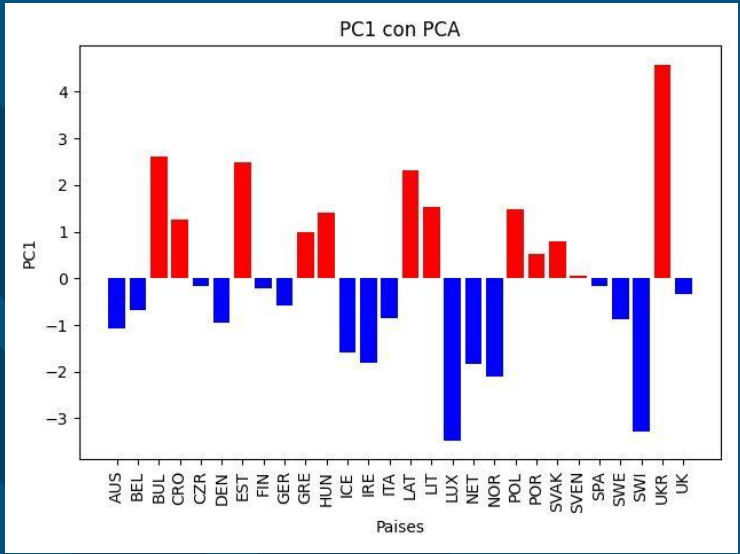
Analizamos Resultados para distintos η
Epochs: 10000 ~ η: 10^{-2}

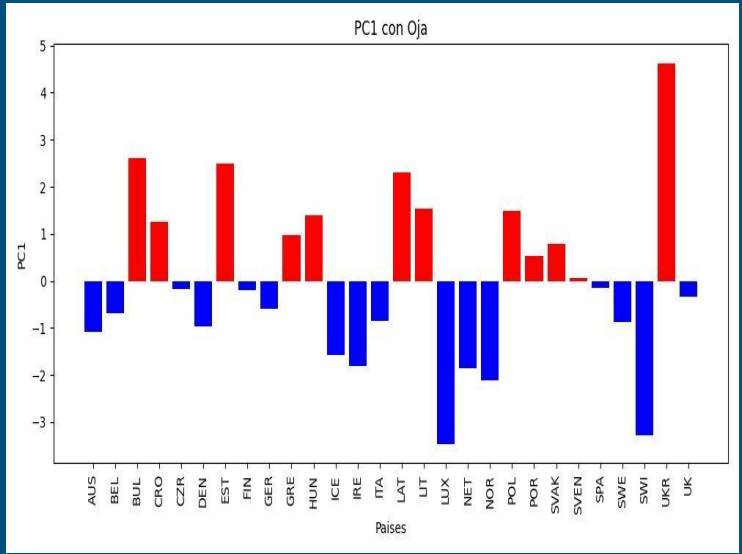




	Área	GDP	Inflation	Life.expect	Military	Pop.growth	Unemployment
PCA	0.1248739	-0.50050586	0.40651815	-0.48287333	0.18811162	-0.47570355	0.27165582
Oja	0.20275538	-0.48993905	0.47946847	-0.50095825	0.13074739	-0.45684623	0.232006
Error Promedio	0.04219355 ~ 10^{-2}						

Analizamos Resultados para distintos η
Epochs: 10000 - η: 10^{-3} (Recomendado en clase)





	Área	GDP	Inflation	Life.expect	Military	Pop.growth	Unemployment
PCA	0.1248739	-0.50050586	0.40651815	-0.48287333	0.18811162	-0.47570355	0.27165582
Oja	0.1320901	-0.49983558	0.41358595	-0.48437194	0.18215693	-0.47415024	0.26813229
Error Promedio	0.00392634 ~ 10^{-3}						

Conclusiones: Tasa de Aprendizaje

- Tomar una tasa de aprendizaje adecuada en Modelo de Oja nos garantiza que el algoritmo converja a la Componente Principal 1.
- Por el contrario, tomar una **tasa de aprendizaje alta** puede provocar que el Modelo de Oja no converja a la PC1.
- Incluso, observamos un ejemplo en donde las cargas obtenidas indicaron correlaciones inversas a las esperadas.

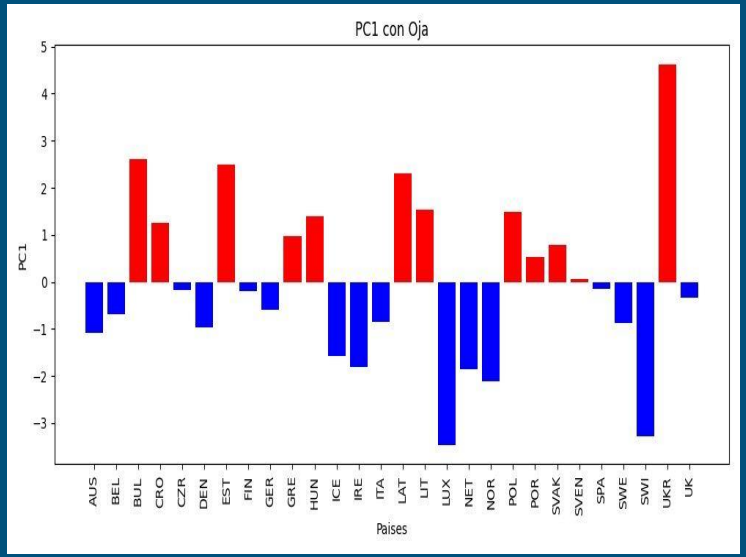
Conclusiones: ¿Por qué Oja?

- Para el cálculo de PCA necesitamos operar con la matriz de covarianza, y para calcular está matriz necesitamos acceder a todos los datos a la vez.
- Este proceso puede resultar imposible de implementar para datasets muy grandes.
- El Modelo de Oja, al estar basado en un perceptrón, puede ser entrenado de un dato a la vez.
- Por ende, **utilizaremos el modelo de Oja si queremos obtener una buena aproximación de la PC1 y el dataset que tenemos que utilizar en muy grande para ser cargado completamente en memoria.**

Conclusiones: Resultados de PC1

	Area	GDP	Inflation	Life.expect	Military	Pop.growth	Unemployment
PCA	0.1248739	-0.50050586	0.40651815	-0.48287333	0.18811162	-0.47570355	0.27165582
Oja	0.1320901	-0.49983558	0.41358595	-0.48437194	0.18215693	-0.47415024	0.26813229

- La **inflación** y el **desempleo** tiene una **correlación positiva fuerte** en el índice.
- El **GDP**, la **expectativa de vida** y el **crecimiento poblacional** tienen una **correlación negativa fuerte** en el índice.
- Dicho esto, observamos que **Ucrania** lidera el índice por poseer un desempleo y inflacion muy altos.
- Suiza y Luxemburgo lideran el lado negativo del índice por poseer un gran GDP, expectativa de vida y crecimiento.

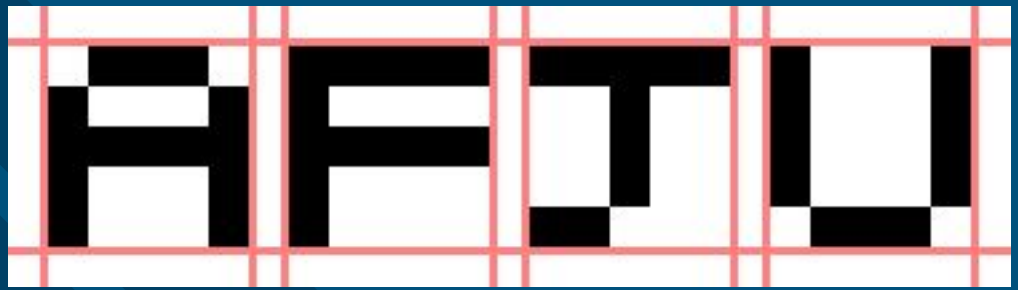


Ejercicio 3

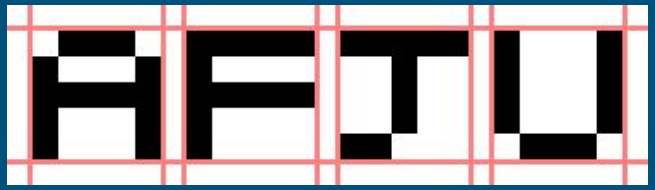
Modelo de Hopfield

Experimentos iniciales

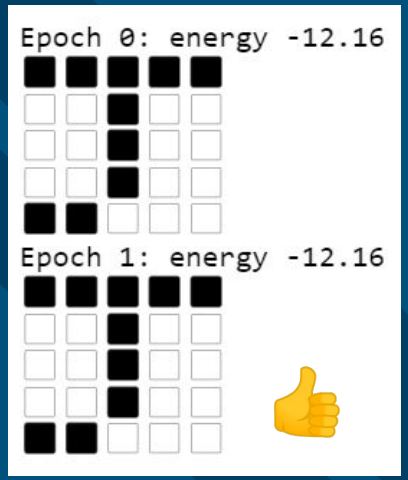
- Creamos una red de 5x5 y le enseñamos a reconocer cuatro letras; ‘AFJU’.
- Elegimos estas letras porque sus patrones son considerablemente distintos.
- Probamos darle para reconocer letras que conozca y que no, y luego probamos la letra ‘J’ con cantidades incrementales de ruido.
- Calculamos la función de energía del sistema en cada iteración.
- El ruido utilizado es “salt & pepper”.



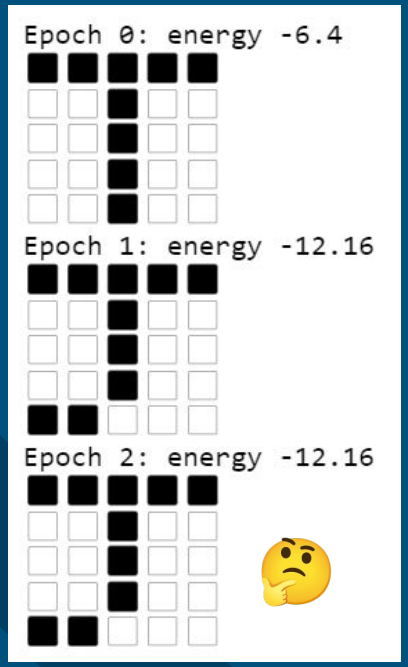
Pruebas básicas - letras



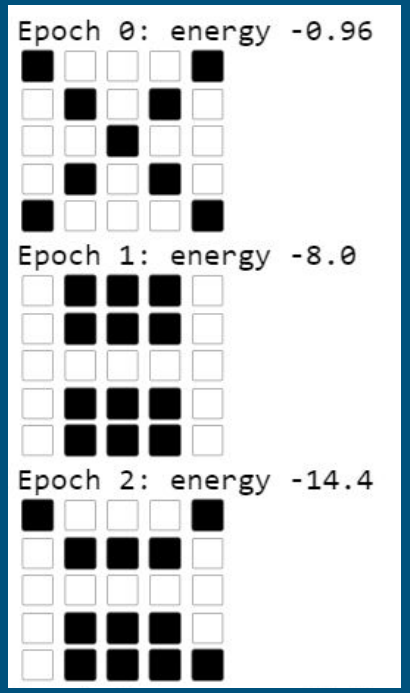
¿Si le doy la letra J?

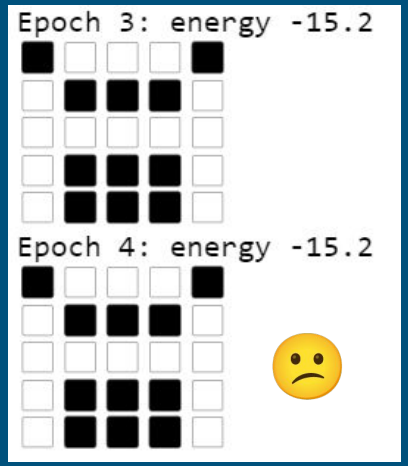


¿Y si le doy la T?



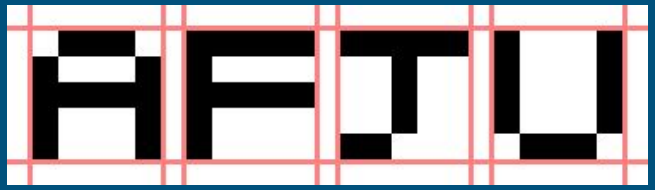
¿Y si le doy la X?



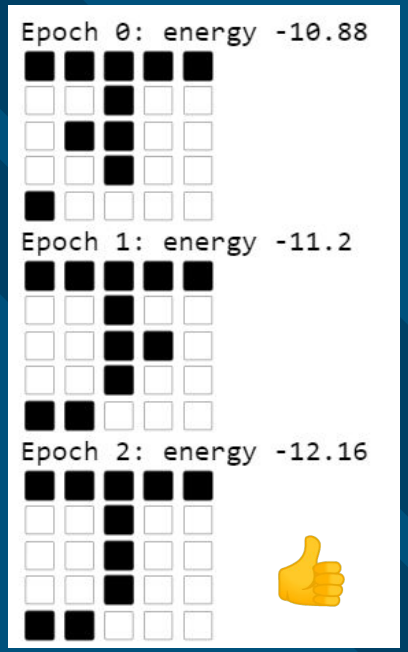


¡Estado espúreo!

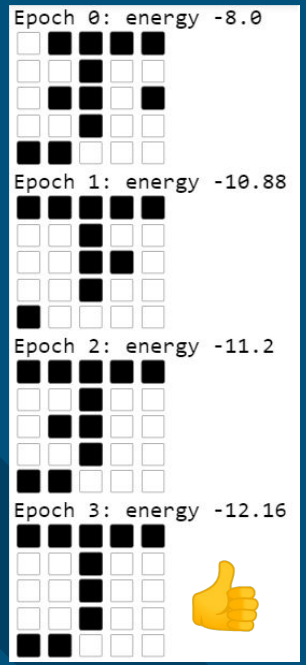
Pruebas básicas - ‘J’ con ruido



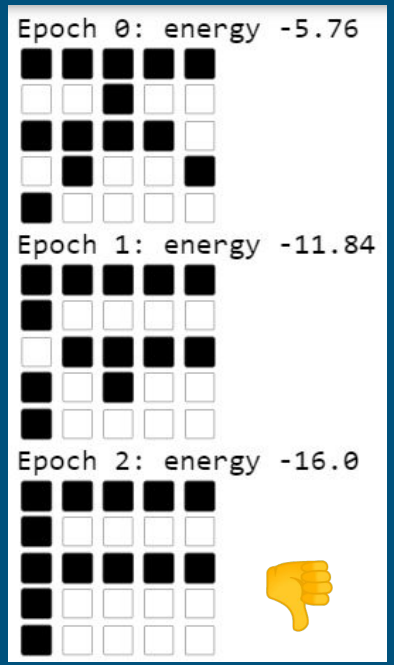
10% de ruido



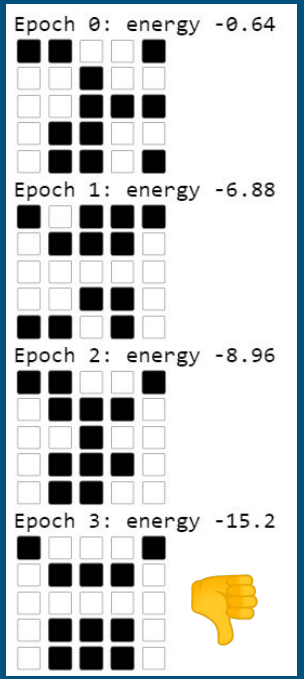
20% de ruido



40% de ruido

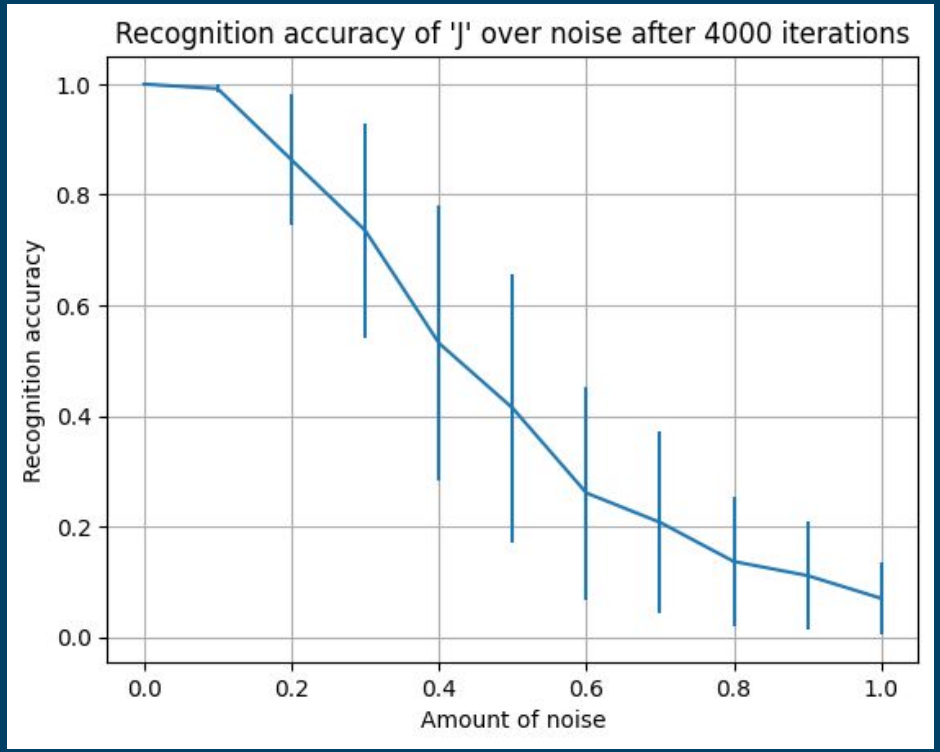


60% de ruido



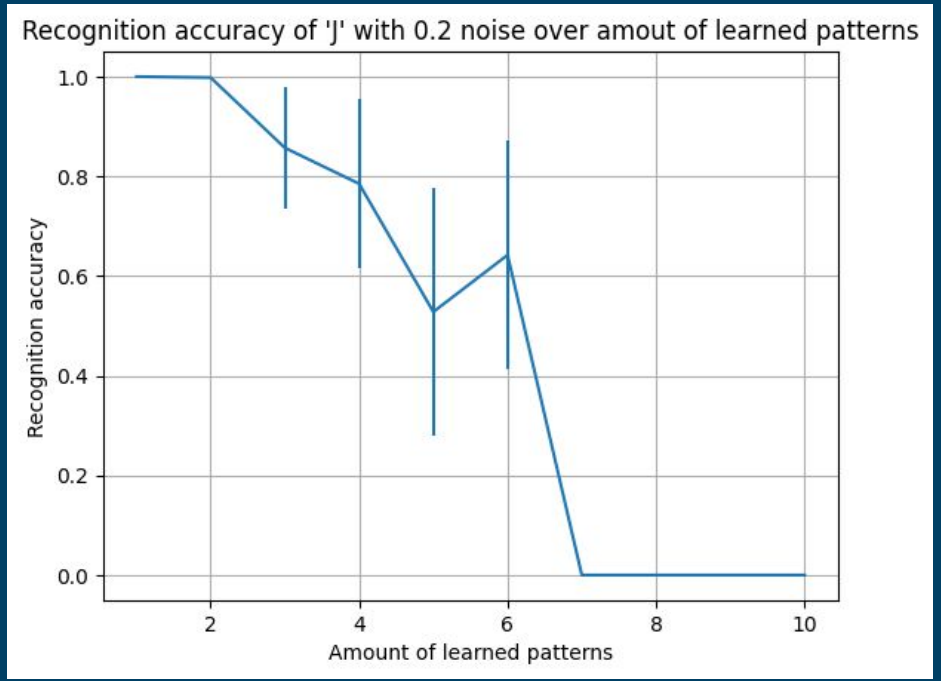
¿Qué tan resistente es frente al ruido?

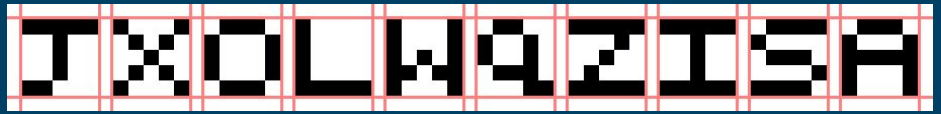
- ➔ Incrementamos el ruido de a pasos de 0.1
- ➔ 4000 iteraciones cada uno
- ➔ Graficamos el promedio de aciertos



¿Y si queremos reconocer más letras?

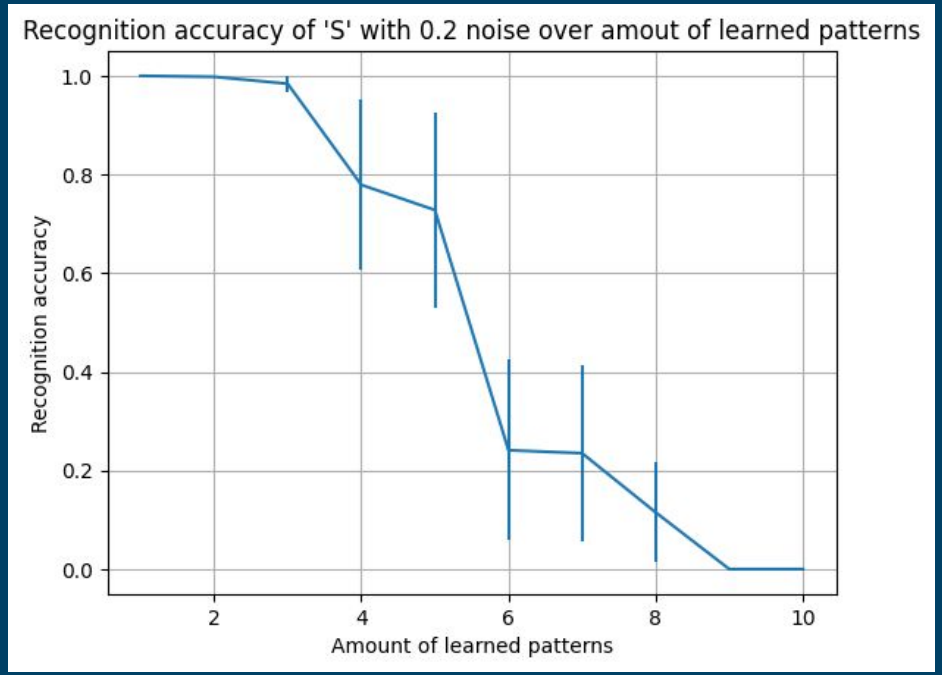
- ➔ Probamos enseñar de 1 a 10 letras, siempre probando la ‘J’
- ➔ Ruido constante de 0.2
- ➔ 1000 iteraciones
- ➔ graficamos el promedio de aciertos

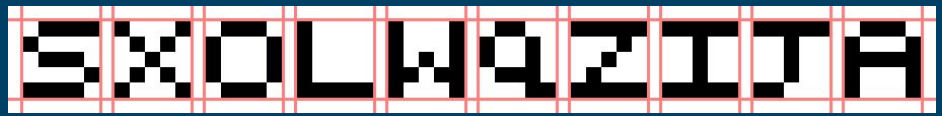




¿Y si queremos reconocer más letras?

- ➔ Volvemos a probar con la letra ‘S’, los demás parámetros se mantienen
- ➔ La historia se repite
- ➔ La regla del 15% nos impide retener más de 4 letras

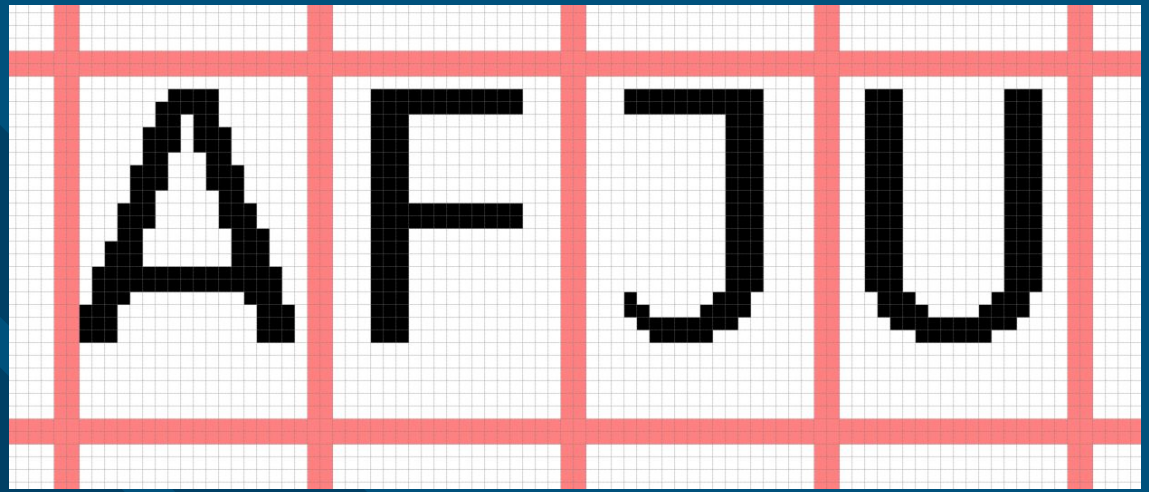




HIPÓTESIS: Más Neuronas = Más Mejor

¡Volvamos a probar pero con letras más grandes!

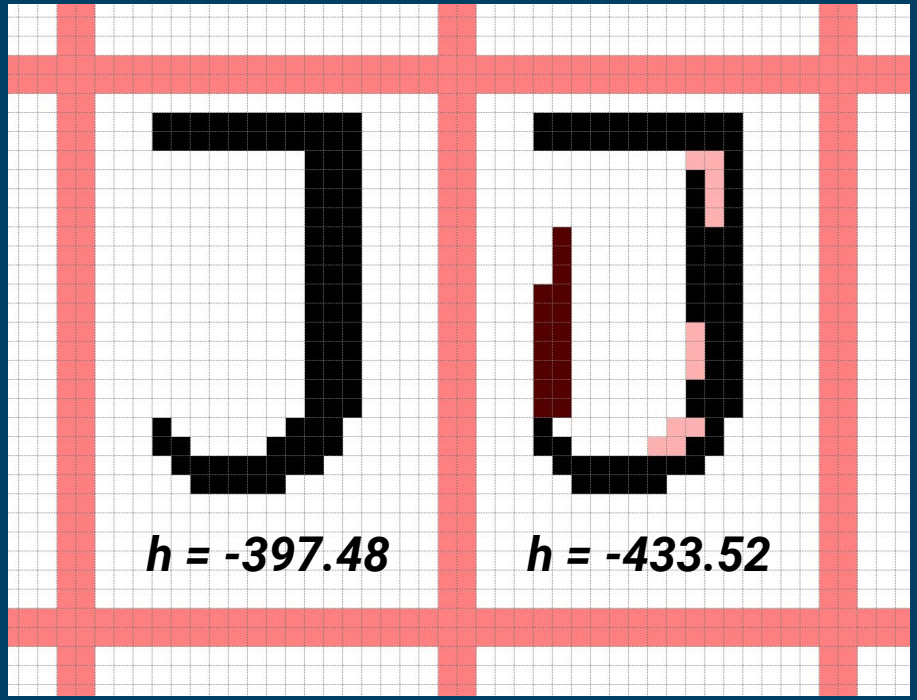
18 x 27 = 486 ⇒ Debería poder guardar 72 patrones



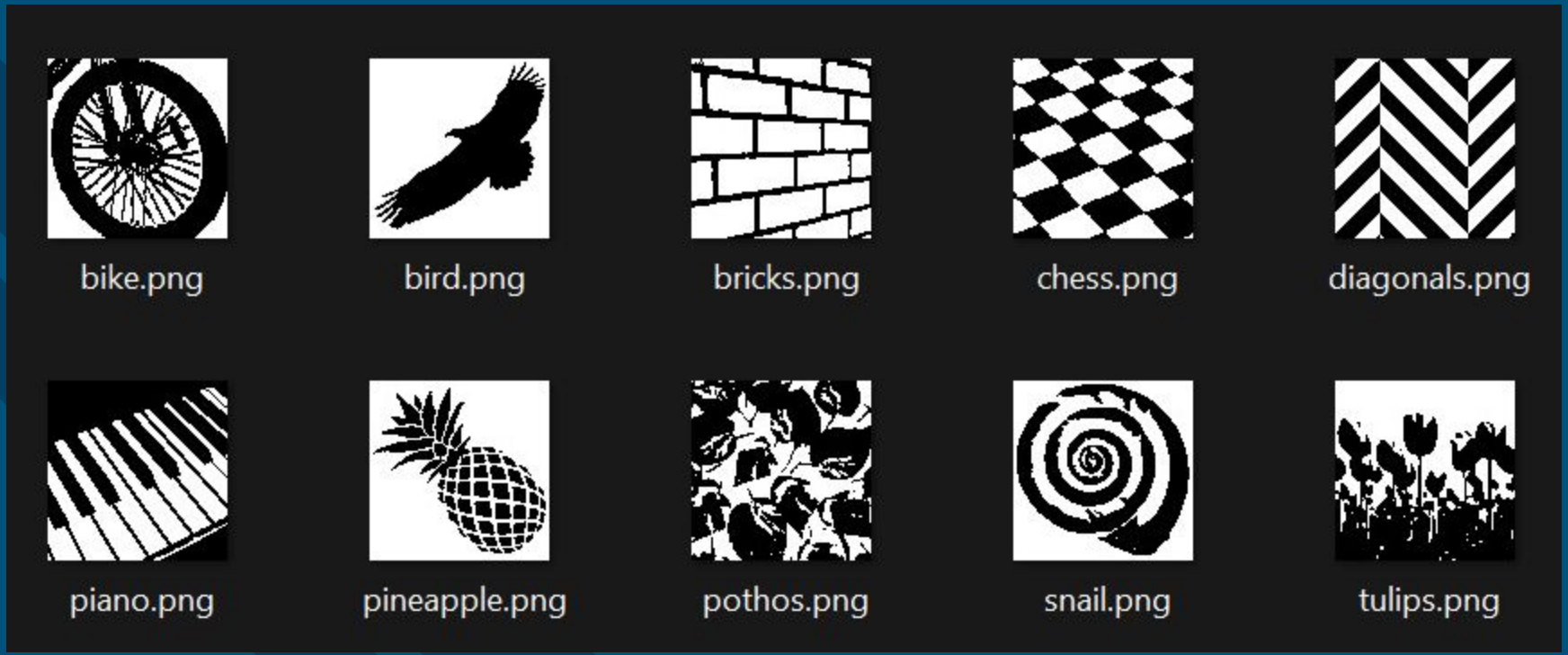
RESULTADOS:

Nop 💀

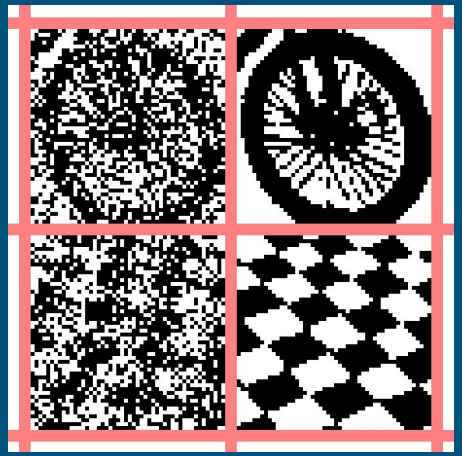
- ¡Con solo 4 letras la red es incapaz de reconocer un patrón aprendido!
- La función de energía baja en cada paso, no es un problema de implementación
- Resaltamos en rojo los píxeles que están mal

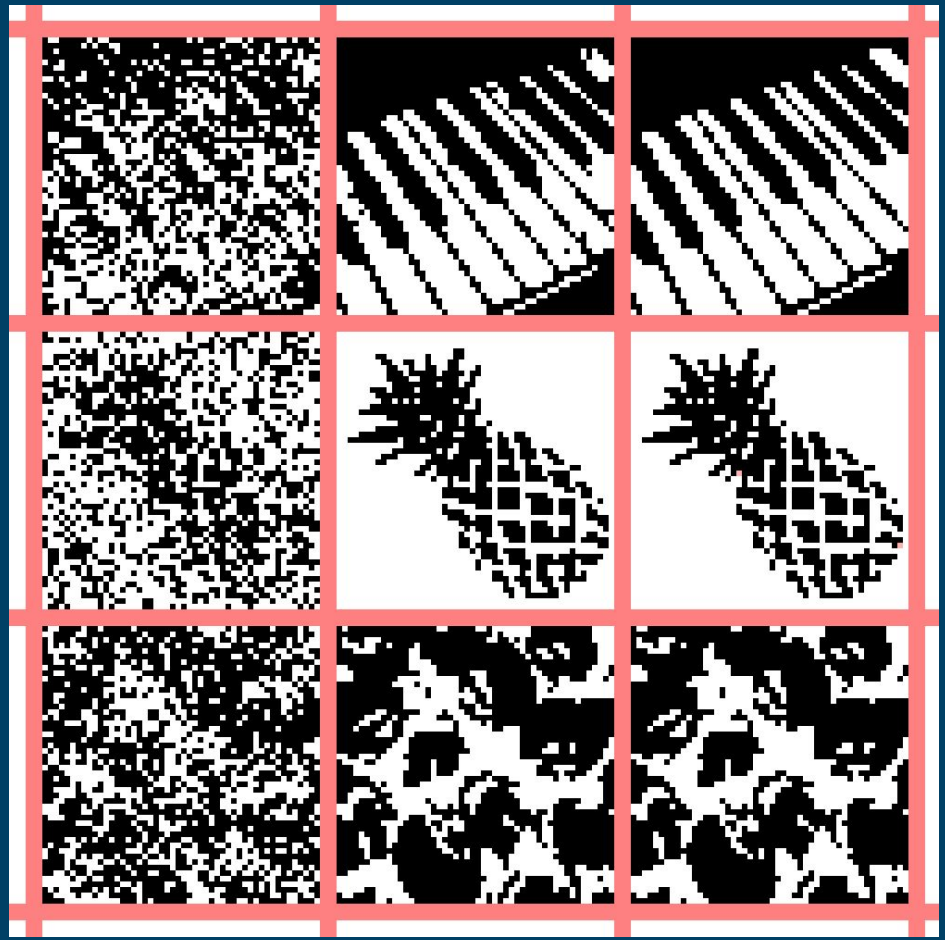


La prueba de fuego...



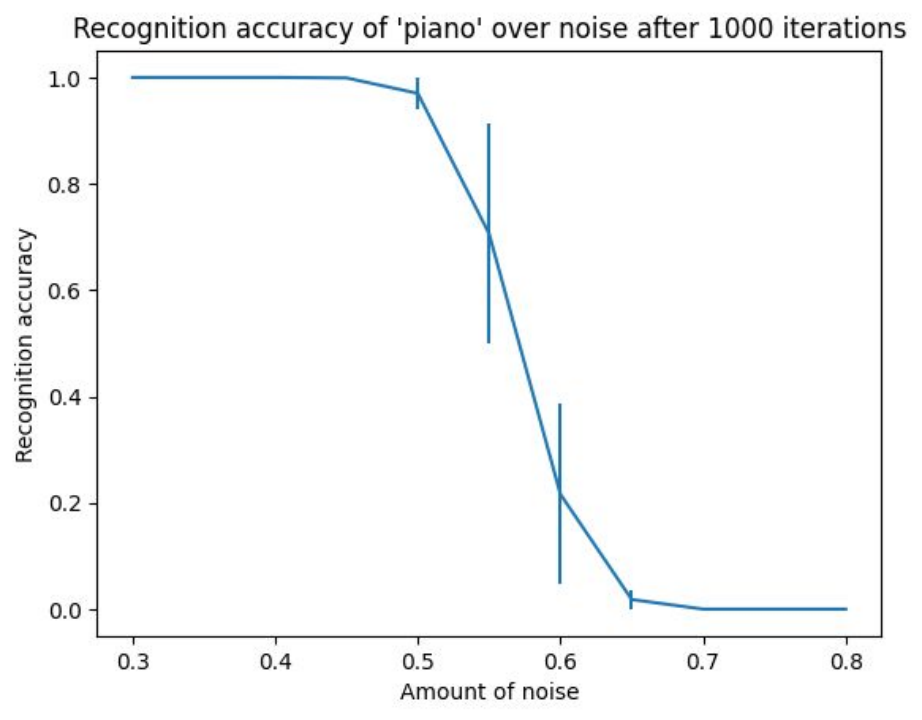
!Es efectivo con 40% de ruido!





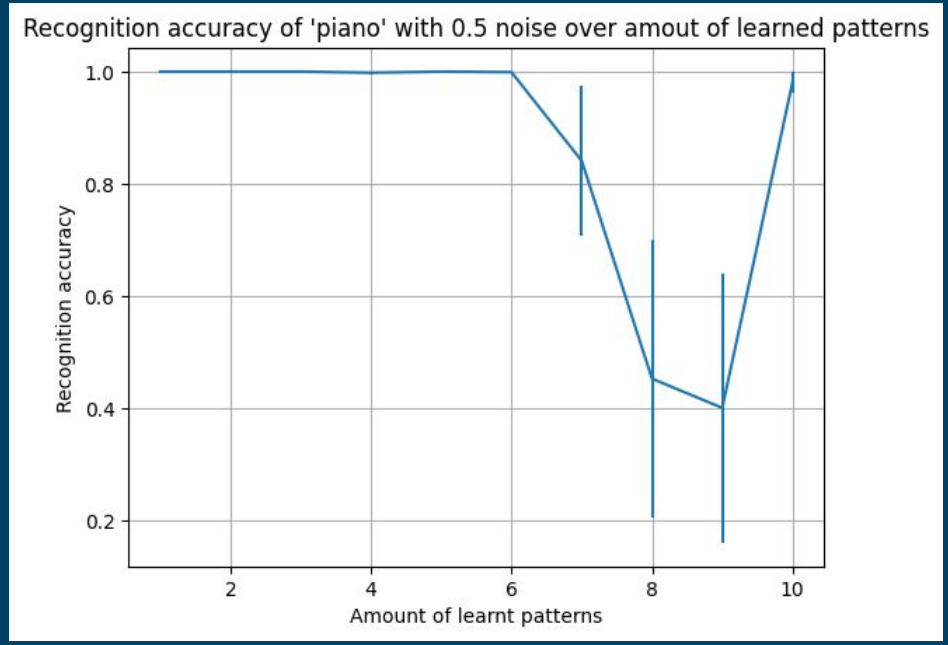
Pero... ¿Cuán resistente al ruido?

- Con ruidos entre [0.3, 0.8]
- 1000 iteraciones cada uno
- Graficamos el promedio de aciertos
- ¡0.5 sigue dando buenos resultados en este patrón!

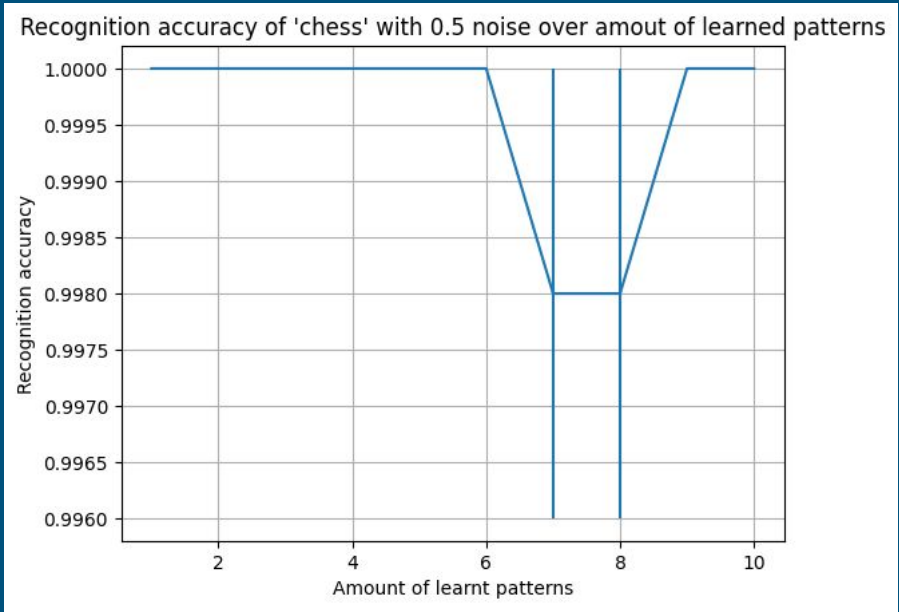


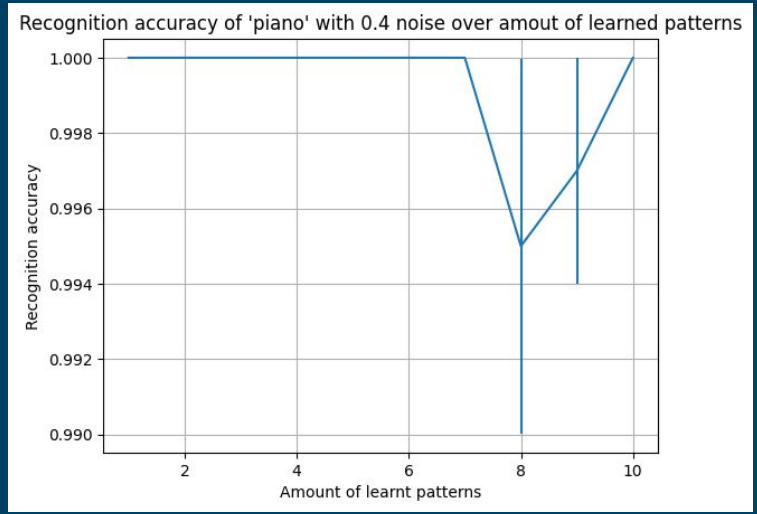
El ruido trae incertidumbre

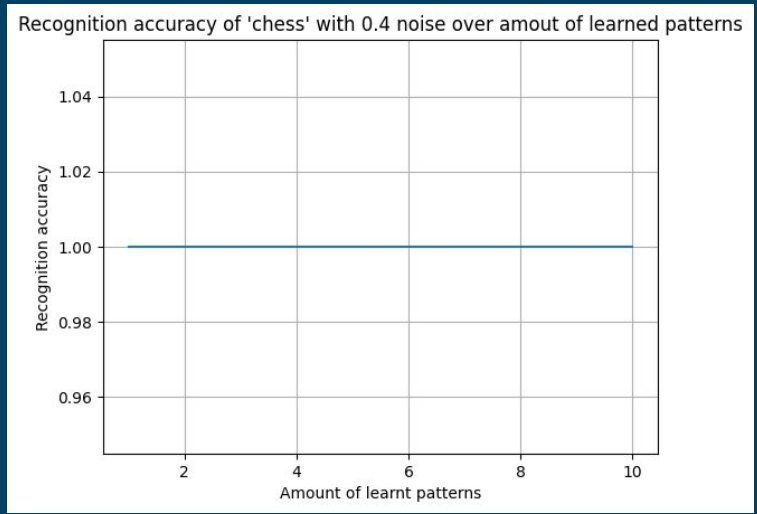
- Ruido constante de 0.5
- Vamos incrementando la cantidad de patrones aprendidos
- 1000 iteraciones cada uno
- Graficamos el promedio de aciertos



Son casos particulares.







Conclusiones

- Una red pequeña tiene reducida capacidad de retención de patrones.
- Pueden ocurrir estados espúreos, que no son un patrón aprendido pero son un mínimo en la función de energía, y por ende converge ahí.
- Sin importar el tamaño de la red, los patrones *deben* ser más o menos ortogonales. No fue capaz de reconocer las letras agrandadas, pero funciona impecable con las imágenes.
- Con un set de patrones bien distinguibles, puede tolerar grandes cantidades de ruido.

Conclusiones

“Otro tipo de memoria”

- Todas las estructuras de datos que vimos hasta ahora en la carrera guardan los datos de alguna forma explícita.
- ¡Las redes de Hopfield tienen una capacidad de retención de patrones, sin guardar dichos patrones!
- *“Memorias Asociativas”*

¿Preguntas?