Universidad de Buenos Aires
Facultad de
Ciencias Exactas y Naturales
Departamento de Computación

Redes Neuronales

**Primer Cuatrimestre de 2014**

# Trabajo práctico

# Modelo de HopField

**Fecha de entrega: 26 de Junio**

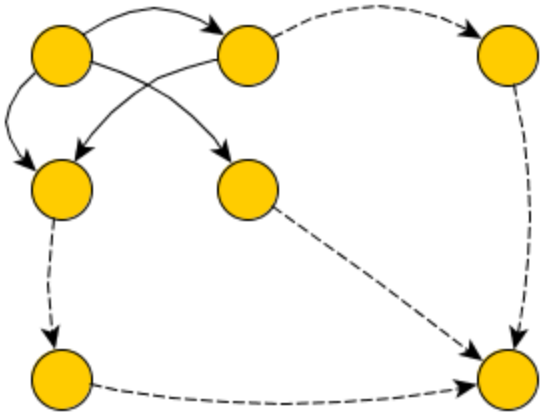


| Integrante | LU | Correo electrónico |
|---|---|---|
| Cammi, Martín | 676/02 | martincammi@gmail.com |
| De Sousa Bispo, Mariano | 389/08 | marian_sabianaa@hotmail.com |

Ejercicio 1: **Patrones Simples**

## Arquitectura de la red

La red se conforma de 20 neuronas interconectadas todas entre sí.



## Introducción

A continuación presentamos los resultados a la implementación del algoritmo de Hopfield de memorias asociativas, el algoritmo fue ejecutado con entrenamiento sincrónico en primera instancia y luego con entrenamiento asincrónico.

Algunas definiciones:

Los **valores de ortogonalidad** mostrados corresponden al cálculo de la ortogonalidad para todo par de vectores.

El **% de Ortogonalidad** corresponde a cuan ortogonales son los vectores en su conjunto con respecto a un conjunto de vectores completamente ortogonales. Así los vectores [[1,0,0],[0,1,0],[0,0,1]] tienen un % de ortogonalidad de 100% (son todos linealmente independientes) y los vectores [[1,0,0],[1,0,0],[1,0,0]] tienen un 0% de ortogonalidad (son todos linealmente dependientes)

La **Distancia Hamming learning-result** es un vector que corresponde a la diferencia de hamming entre las memorias aprendidas y el resultado de una misma cantidad de vectores de activación. En la posición i de este vector estará la distancia de hamming de la iésima memoria con respecto al i ésimo vector de activación.

## Implementación

1) Se programó el algoritmo de Hopfield con parametrización de entrenamiento sincrónico y asincrónico.

## Conjuntos de entrenamiento Asincrónico

2) Proponemos a continuación varios sets de memorias con diferentes porcentajes de ortogonalidad para verificar que al ser lo suficientemente ortogonales la activación converja a los mismos patrones.

Patrón 1: Se hizo aprender a la red con las siguientes memorias:

m1 = [[ 1. -1.  1.  1. -1. -1.  1.  1.  1. -1.  1.  1. -1.  1.  1.  1.  1.  1.  -1.  1.]]
m2 = [[ 1.  1. -1.  1.  1.  1.  1.  1. -1.  1.  1. -1.  1. -1.  1.  1.  1.  1.  1.  1.]]
m3 = [[ 1. -1.  1. -1. -1.  1.  1. -1. -1.  1.  1.  1. -1. -1. -1.  1. -1.  1.  1. -1.]]

**Valores de Ortogonalidad:** [0, 0, 0]
**% de Ortogonalidad:** 100.0
**Resultado:** Al invocar la activación con los mismos patrones se obtuvieron las mismas memorias almacenadas.

Patrón 2: Se hizo aprender a la red con las siguientes memorias:

m1 = [[-1. -1.  1.  1.  1.  1. -1. -1. -1. -1.  1. -1.  1. -1. -1. -1. -1. -1. 1.  -1.]]
m2 = [[-1. -1.  1.  1.  1. -1.  1.  1.  1.  1.  1. -1. -1. -1. -1. -1.  1. -1. -1. -1.]]
m3 = [[ 1. -1.  1.  1. -1.  1.  1. -1. -1. -1. -1.  1.  1.  1.  1. -1.  1.  1. -1. -1.]]

**Valores de Ortogonalidad:** [4.0, 0.0, -4.0]
**% de Ortogonalidad**: 86.66
**Resultado:** Nuevamente la activación con los mismos patrones que en este caso tienen una ortogonalidad de 86.66% se obtuvieron las memorias almacenadas.

Patrón 3: Se realizó otra prueba con memorias con un porcentaje de ortogonalidad aún menor:

m1 = [[ 1.  1.  1.  1.  1. -1.  1.  1.  1. -1.  1.  1. -1.  1.  1.  1.  1.  1.  -1.  1.]]
m2 = [[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. -1.  1.  1. -1.  1.  1.  1. ]]
m3 = [ 1.  1.  1. -1.  1.  1.  1.  1.  1.  1.  1.  1. -1. -1. -1. -1.  1.  1.  1.  -1. ]]

**Valores de Ortogonalidad:** [8.0, 4.0, 8.0]
**% de Ortogonalidad:** 66.66
**Resultado:** Aún con un nivel de ortogonalidad más bajo todavía se siguen obteniendo las memorias originales en la activación.

3) Parece verificarse entonces qué patrones con una suficiente ortogonalidad convergen a ellos mismos para menor porcentaje de ortogonalidad (menor a 60%) las memorias dejan de converger y se obtienen estados espurios para ellas.

4) Verificaremos la atracción de las memorias con patrones levemente alterados a los de las memorias almacenadas.

Utilizando el Patrón 1 de memorias de máxima ortogonalidad citado anteriormente para el aprendizaje, activamos Hopfield con los siguientes vectores donde en **negrita** se aprecian las posiciones que se han alterado de las memorias iniciales

[[ 1. -1. 1. 1. -1. **1**. 1. 1. 1. -1. 1. 1. -1. 1. 1. 1. 1. 1. -1. 1.]]
[[ 1. 1. -1. 1. 1. 1. 1. 1. -1. **-1**. 1. -1. 1. -1. 1. 1. 1. 1. 1. 1.]]
[[ 1. -1. 1. -1. -1. 1. 1. -1. -1. 1. 1. 1. -1. **1**. -1. 1. -1. 1. 1. -1.]]

Distancia Hamming learning-activation: [1, 1, 1]
**Resultado:** Las memorias devueltas siguen siendo las mismas que las aprendidas originalmente.

Cambiando algunos bits más:

[[ 1. -1. 1. 1. -1. **1**. 1. 1. 1. **1**. 1. 1. -1. 1. 1. 1. 1. 1. -1. 1.]]
[[ 1. 1. **1**. 1. 1. 1. 1. 1. -1. **-1**. 1. -1. 1. -1. 1. 1. 1. 1. 1. 1.]]
[[ 1. -1. 1. -1. -1. **1**. 1. -1. -1. 1. 1. 1. -1. **1**. -1. 1. -1. 1. 1. -1.]]

Distancia Hamming learning-activation: [2, 2, 2]
**Resultado:** Las memorias devueltas siguen siendo las mismas que las aprendidas originalmente.

Haciendo una nueva prueba con 4 bits modificados

[[ 1. -1. 1. 1. -1. **1**. 1. 1. 1. **1**. 1. 1. **1**. 1. 1. 1. 1. 1. -1. **-1**.]]
[[ 1. 1. -1. 1. 1. -**1**. **-1**. 1. -1. **-1**. 1. -1. 1. **1**. 1. 1. 1. 1. 1. 1.]]
[[ 1. **1**. 1. **1**. -1. 1. 1. -1. -1. 1. -**1**. 1. -1. **1**. -1. 1. -1. 1. 1. -1.]]

**Distancia Hamming learning-activation**: [4, 4, 4]

**Resultado:** Las memorias devueltas siguen siendo las mismas que las aprendidas originalmente, probando con una alteración de 5 bits ya comienzan a aparecer estados espurios.

5) Búsqueda de estados espurios en forma analítica.

a) A continuación proponemos combinaciones lineales de los vectores ortogonales del patrón 1 para generar estados espurios analíticamente.

a.1) Suma de las memorias y aplicación de signo. Sumamos todas las memorias componente a componente y luego tomamos signo de las mismas, siendo que son tres memorias los resultados serán o bien positivos o bien negativos, cada negativo lo representaremos con -1 y cada positivo con 1.

[[ 1. -1. 1. 1. -1. -1. 1. 1. 1. -1. 1. 1. -1. 1. 1. 1. 1. 1. -1. 1.]]
[[ 1. 1. -1. 1. 1. 1. 1. 1. -1. 1. 1. -1. 1. -1. 1. 1. 1. 1. 1. 1.]]
[[ 1. -1. 1. -1. -1. 1. 1. -1. -1. 1. 1. 1. -1. -1. -1. 1. -1. 1. 1. -1.]]
Suma: [3, -1, 1, 1, -1, 1, 3, 1, -1, 1, 3, 1, -1, -1, 1, 3, 1, 3, 1, 1]
Signo: **[1, -1, 1, 1, -1, 1, 1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, 1, 1, 1]**

Realizamos las siguientes pruebas con el estados espurio llamado e0 y vectores que difieren en pocos bits.

**e0=** [[1, -1, 1, 1, -1, 1, 1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, 1, 1, 1]]
[[1, **1**, 1, 1, -1, 1, 1, 1, -1, 1, 1, 1, -1, -1, **-1**, 1, 1, 1, 1, 1]]
[[1, -1, 1, **-1**, -1, 1, 1, 1, -1, 1, 1, 1, -1, **1**, 1, 1, 1, 1, 1, 1]]

**Resultado:** Los resultados devueltas coinciden todas con el estado espurio con lo cual el estado espurio es un estado atractor.
**Distancia Hamming learning-result**: [5, 5, 5], lo que significa que el estado espurio dista cinco unidades de cada una de las memorias iniciales.

a.2) Suma y resta de memorias. Sumaremos y restaremos las memorias del Patrón 1 para obtener más combinaciones lineales que serán estados espurios:

e1 = signo( m1 - m2 + m3)
e2 = signo( m1 + m2 - m3)
e3 = signo(- m1 + m1 - m3)

e1 =
m1 = [[ 1. -1. 1. 1. -1. -1. 1. 1. 1. -1. 1. 1. -1. 1. 1. 1. 1. 1. -1. 1.]]
m2 = [[ 1. 1. -1. 1. 1. 1. 1. 1. -1. 1. 1. -1. 1. -1. 1. 1. 1. 1. 1. 1.]]
m3 = [[ 1. -1. 1. -1. -1. 1. 1. -1. -1. 1. 1. 1. -1. -1. -1. 1. -1. 1. 1. -1.]]

suma e1 = [[ 1, -3, 3, -1, -3, -1, 1, -1, 1, -1, 1, 3, -3, 1, -1, 1, -1, 1, -1, -1]]
e1 = [[ 1, -1, 1, -1, -1, -1, 1, -1, 1, -1, 1, 1, -1, 1, -1, 1, -1, 1, -1, -1]]

suma e2 = [[ 1, 1, -1, 3, 1, -1, 1, 3, 1, -1, 1, -1, 1, 1, 3, -1, 3, -1, -1, 3]]
e2 = [[ 1, 1, -1, 1, 1, -1, 1, 1, 1, -1, 1, -1, 1, 1, 1, -1, 1, -1, -1, 1]]

suma e3 = [[ -1, 3, -3, 1, 3, 1, -1, 1, -1, 1, -1, -3, 3, -1, 1, -1, 1, -1, 1, 1]]
e3 = [[ -1, 1, -1, 1, 1, 1, -1, 1, -1, 1, -1, -1, 1, -1, 1, -1, 1, -1, 1, 1]]

**Resultado:** Al activar Hopfield con los estados espurios e1, e2 y e3, Los resultados devueltos coinciden con cada uno de ellos, y habiendo activado con vectores con diferencias en algunos de bits se ha observado que devuelven los mismos estados espurios con lo cual son atractores.
**Distancia Hamming learning-result**: para e1 -> [5, 15, 5], e2 -> [5, 5, 15], e3 -> [15, 5, 15]

b) Búsqueda de estados espurios en forma empírica.

Habiendo generado cerca de 3000 vectores aleatorios uniformemente para la activación para la red de Hopfield de 20 neuronas, que admite sólamente 3 patrones, se detectaron 11 estados espurios.

Sobre un total de 14 estados, el porcentaje de estados espurios corresponde al 78% y la cantidad de patrones que admite la red un 22%.

Los siguientes son los estados espurios encontrados:

[[ 1, 1, -1, -1, 1, 1, 1, -1, -1, 1, 1, -1, 1, -1, -1, 1, -1, 1, 1, -1]],
e2 = [[ 1, 1, -1, 1, 1, -1, 1, 1, 1, -1, 1, -1, 1, 1, 1, 1, 1, 1, -1, 1]],
[[ 1, -1, 1, 1, -1, 1, 1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, 1, 1, 1]],
e1 = [[ 1, -1, 1, -1, -1, -1, 1, -1, 1, -1, 1, 1, -1, 1, -1, 1, -1, 1, -1, -1]],
[[-1, 1, -1, 1, 1, -1, -1, 1, 1, -1, -1, -1, 1, 1, 1, -1, 1, -1, -1, 1]],
[[-1, -1, 1, -1, -1, -1, -1, -1, 1, -1, -1, 1, -1, 1, -1, -1, -1, -1, -1, -1]],
[[-1, 1, -1, -1, 1, 1, -1, -1, -1, 1, -1, -1, 1, -1, -1, -1, -1, -1, 1, -1]],
[[-1, -1, 1, -1, -1, 1, -1, -1, -1, 1, -1, 1, -1, -1, -1, -1, -1, 1, -1]],
[[-1, -1, 1, 1, -1, -1, -1, 1, 1, -1, -1, 1, -1, 1, 1, -1, 1, -1, -1, 1]],
[[-1, 1, -1, -1, 1, -1, -1, -1, 1, -1, -1, -1, 1, 1, -1, -1, -1, -1, -1, -1]],
e3 = [[-1, 1, -1, 1, 1, 1, -1, 1, -1, 1, -1, -1, 1, -1, 1, -1, 1, -1, 1, 1]]

c) Se realizaron más mediciones modificando la cantidad de neuronas y memorias sobre un total de 1000 vectores de activación con patrones de memorias lo más ortogonales posibles en la mayoría de los casos.

| Cantidad de Neuronas (N) | %Ortogonalidad de memorias | Cantidad de Memorias | Estados espurios | %Efectividad en promedio de las memorias | Memorias / Neuronas | %Captado por memorias |
|---|---|---|---|---|---|---|
| 20 | 100% | 3 | 11 | 13.1% | 0.15 | 39.3% |
| 30 | 97% | 3 | 11 | 13.7% | 0.15 | 41.3% |
| 40 | 100% | 3 | 11 | 14.43% | 0.15 | 43.3% |
| 50 | 98% | 3 | 11 | 14.1% | 0.15 | 42.5% |
| 50 | 97% | 4 | 39 | 10% | 0.10 | 40% |
| 60 | 97% | 4 | 45 | 9.4% | 0.06 | 37.6% |
| 50 | 95% | 5 | 104 | 7% | 0.10 | 34.7% |
| 70 | 97% | 5 | 156 | 6.6% | 0.07 | 33% |
| 50 | 92% | 6 | 52 | 6% | 0.12 | 36.2% |
| 60 | 91% | 6 | 95 | 5% | 0.10 | 30.1% |
| 60 | 90% | 7 | 82 | 4.2% | 0.11 | 30% |
| 70 | 90% | 8 | 69 | 5% | 0.11 | **46.2%** |
| 70 | 92% | 8 | 96 | 3.5% | 0.11 | 26% |
| 80 | 91% | 9 | 69 | 1.5% | 0.11 | 14.3% |

La tabla anterior muestra las mediciones para al algoritmo de Hopfield con diferentes **cantidades de neuronas**, y variando también la **cantidad de memorias**, también figura la **ortogonalidad** de las mismas. Sobre un total de 1000 activaciones se detectaron el porcentaje de vectores de activación **atraídos por memorias** este porcentaje dividido por la cantidad de memorias identifica la **efectividad en promedio de las memorias.** También figuran la cantidad de **estados espurios** y el factor de memoria/Neuronas.

Se comenzó con una cantidad de 3 neuronas realizando corridas de activación para intentar obtener los mejores porcentajes de atracción. Se detectaron 11 estados espurios, y el porcentaje de atracción de las memorias fue cercano al 40%.

Se fueron aumentando la cantidad de neuronas viendo que se mantuviera o aumentara el porcentaje de atracción de las memorias y en cuanto esto no se cumpliese se fueron aumentando la cantidad de neuronas para lograr identificar los parámetros que maximicen este porcentaje lo más posible.

A partir del pasaje a 5 memorias se detectó un salto significativo en la cantidad de estados espurios y el porcentaje de atracción disminuyó un 5% con respecto al porcentaje de atracción de 4 memorias.

## Conjuntos de entrenamiento Sincrónico

2.a) Realizaremos a continuación similares sets pero esta vez activando el flag sincrónico.

Con los patrones 1, 2 y 3 utilizados en el entrenamiento asincrónico con porcentajes de ortogonalidad similares (100%, 86% y 66%), en todos ellos se obtuvieron los mismos patrones al activarse con ellos.

3) Similar a la activación asincrónica se verifica que patrones con una suficiente ortogonalidad convergen a ellos mismos.

4) Verificaremos la atracción de las memorias con patrones levemente alterados a los de las memorias almacenadas.

Utilizando el Patrón 1 de memorias alterado una cantidad de 1 a 7 bits: hemos obtenido resultados similares siendo las memorias devueltas las mismas que las aprendidas.

Aunque para alteraciones de más de 7 bits, se notan algunas diferencias pero no sustanciales. Ingresando las 3 memorias ortogonales del patrón 1 para el aprendizaje y realizando activaciones con otros tres vectores aleatorios se obtuvieron los siguientes resultados:

**Asincrónico**
Hamming distances learning-result: [10, 10, 20]
Hamming distances learning-activation: [11, 11, 13]

**Sincrónico**

Hamming distances learning-result: [12, 11, 20]
Hamming distances learning-activation: [11, 11, 13]

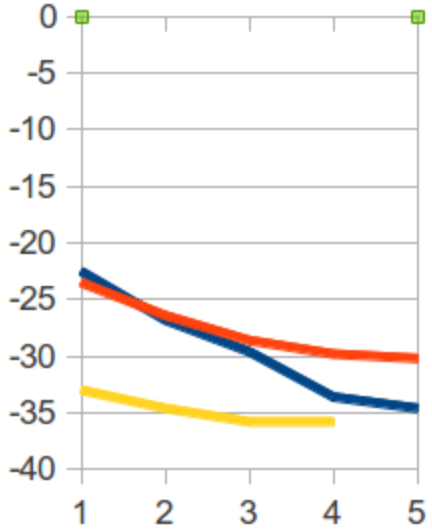
**Asincrónico**
Hamming distances learning-result: [5, 5, 10]
Hamming distances learning-activation: [9, 8, 12]

**Sincrónico**
Hamming distances learning-result: [5, 5, 15]
Hamming distances learning-activation: [9, 8, 12]

Las distancias de hamming de los vectores de activación superan el 50% de alteración en sus bits con respecto a las memorias originales, es lógico que los vectores resultantes tengan también una gran desviación y no se identifiquen con ninguna de las memorias.

El algoritmo en modo sincrónico parece devolver resultados con no más de un 10% peor en algunos casos con respecto a que devuelven vectores más alejados de las memorias que los del modo asincrónico, pero en ambos casos cuando una devuelve estados espurios la otra también.

Los valores de energía se mantuvieron constantes en algunos casos y en otros variaron decreciendo rápidamente en cada iteración de activación para ambos modos y en alguna situación tomaron sólo uno o dos valores, pero luego rápidamente convergieron a un valor de energía.




Inicialmente el algoritmo tenía una condición de parada por igualdad del vector resultante en comparación la iteración anterior, pero nos dimos cuenta que esto no alcanzaba.

Sucedió que varias veces los valores resultantes alternaban entre dos pares de valores, haciendo imposible cumplir la condición de parada ya que en cada iteración era diferente a la anterior (aunque se volvían a repetir) es por ello que se estableció una condición de parada por energía cuando esta se repetía en dos iteraciones.

En esta oportunidad ocurrió que la energía generada no era exactamente constante sino que difería en milésimas o inclusive en menos con lo que se agregó un umbral para establecer una cercanía razonable entre dos valores de energía y así poder finalizar la iteración de activación.

# Ejercicio 2: OCR

## Introducción

Inicialmente comenzamos a plantear el ejercicio con las 26 letras del abecedario, aunque posteriormente se mencionó en clase que la poca ortogonalidad de las mismas hace que sus patrones no puedan ser recuperados, es por ello que luego decidimos intentar aplicar la técnica a los símbolos decimales. Tampoco se tuvo el éxito esperado y muchas de los resultados se correspondieron con estados espurios. A continuación presentamos los datos y análisis.

Efectuamos otro análisis con un algoritmo naive de remapeo de las letras de 5x5 del primer trabajo práctico a NxN. Obtuvimos resultados muy interesantes que pasaremos a contar al final.

## Arquitectura de la red

**Símbolos decimales**

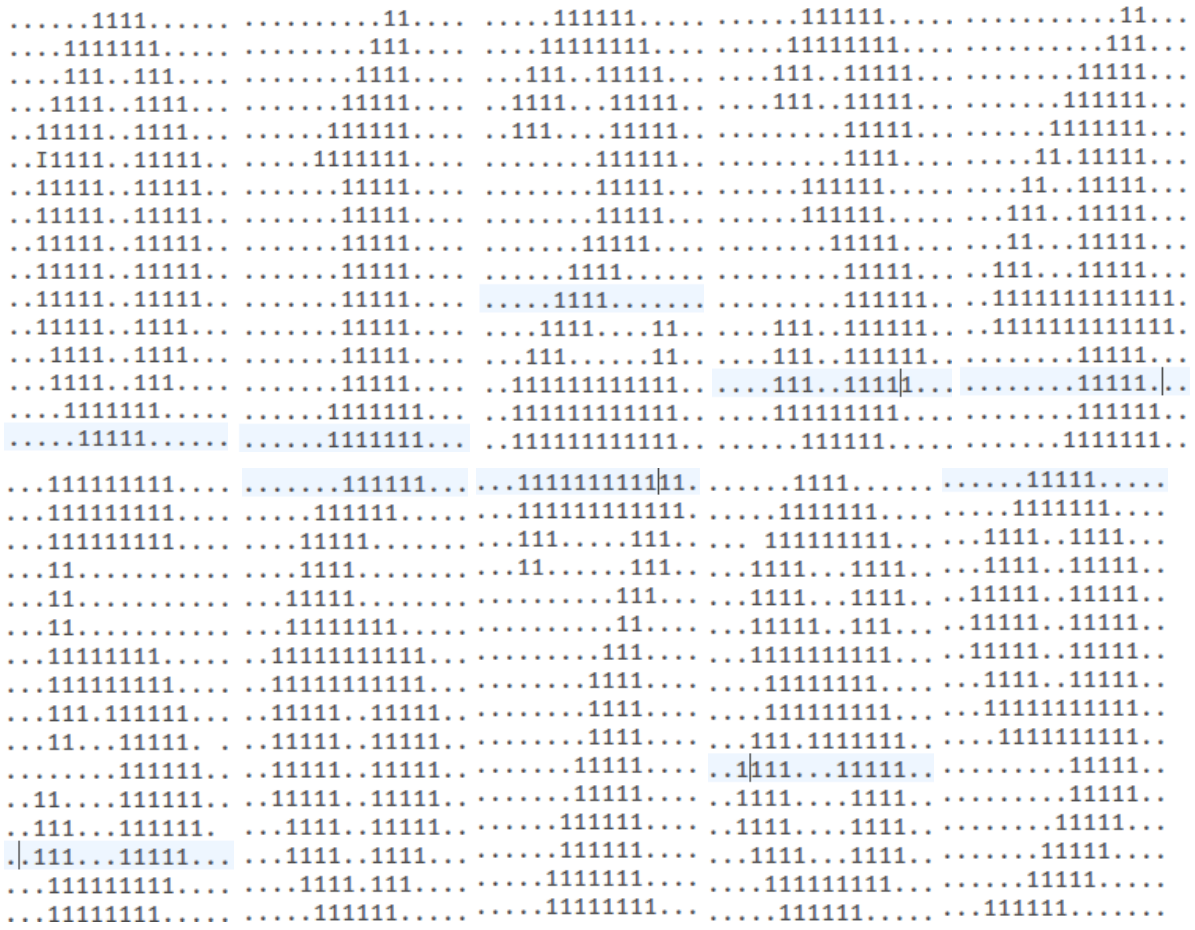
Se utilizó una red de Hopfield de 256 neuronas para almacenar las memorias iniciales de símbolos decimales y estos se representaron con vectores de 256 posiciones (visualizables matricialmente en dimensión de 16x16).

**Letras del abecedario**

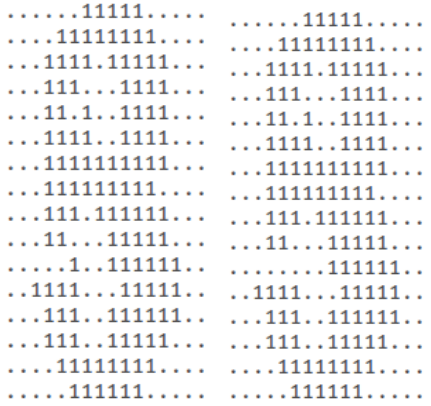
Se utilizaron redes de Hopfield de 5 neuronas.

## Implementación de símbolos decimales

1) El set propuesto a la red consta de los 10 dígitos decimales transformados a un vector de 256 posiciones de valores bipolares. A continuación se muestran las cifras decimales presentadas a la red. (los puntos '.' serán reemplazados por -1 en la activación)



2) Se analizó el set anterior detectándose que poseen una ortogonalidad del 59% entre ellos pero tras entrenarse la red con estas cifras y luego activarse con ellas mismas no se obtuvieron las mismas memorias almacenadas. Por el contrario, se obtuvieron estados espurios. Lo curioso es que para las 10 cifras sólo se devolvieron 2 estados espurios y son los siguientes:



Ambos estados difieren en solamente un bit en la fila 11 columna 6. Aparentemente las memorias no son lo suficientemente ortogonales como para permitir que atraigan a sus mismos vectores. Esto da paso a la generación de estados espurios que atraen estos vectores, lo interesante es que el estados espurio se parece sorprendentemente a varios de las cifras, en una especie de combinación entre ellas.

Es posible que las características intrínsecas de cada cifra se vean condensadas en este estado espurio y que por esa razón se convierta este en atractor de todas ellas.

3) Siendo que las memorias no convergen, no tiene demasiado sentido hacer pruebas sobre la convergencia a ellas. De todos para validar la hipótesis anterior probamos con pocas cifras como memorias y es ese caso si converge. Para un aprendizaje con las cifras 0 1 y 2 las memorias convergen (ya que poseen una ortogonalidad del 74%), pero de ahí en más si el aprendizaje agrega una memoria más comienzan a aparecer los estados espurios. Por ejemplo aprendiendo con las cifras 0, 1, 2 y 3 (Ortogonalidad del 62%) la memoria 0 converge, pero las 1, 2 y 3 convergen a los siguientes estados espurios:

```
[.....111111....0]   [.....111111....0]   [.....111111....0]
[....11111111...0]   [....11111111...0]   [....11111111...0]
[...111..11111..0]   [...111..11111..0]   [...111..11111..0]
[...111..11111..0]   [...111...1111..0]   [...111..11111..0]
[........11111..0]   [..111....1111..0]   [........11111..0]
[........1111...0]   [........111111.0]   [........1111...0]
[.......11111...0]   [........11111..0]   [.....1111111...0]
[.......11111...0]   [........11111..0]   [.....1111111...0]
[.......11111...0]   [.......11111...0]   [.......11111...0]
[.......11111...0]   [......111111...0]   [........11111..0]
[.......11111...0]   [.....1111111...0]   [........111111.0]
[....11.1111111.0]   [....1111.11111.0]   [...111..111111.0]
[...111..111111.0]   [...111...11111.0]   [...111..111111.0]
[...111.111111..0]   [...1111111111..0]   [...111..11111..0]
[...1111111111..0]   [...1111111111..0]   [...111111111...0]
[.....11111111..0]   [.....11111111..0]   [.....111111....0]
```

Estos estados se parecen bastante a las cifras originales 1, 2 y 3
En general cualquier combinación de cifras que tengan un porcentaje de ortogonalidad menor al 70% comienza a generar mayor cantidad de estados espurios que hits en memorias y esto ocurre cuando la cantidad de memorias (cifras) supera las 4 para cualquier permutación de ellas.

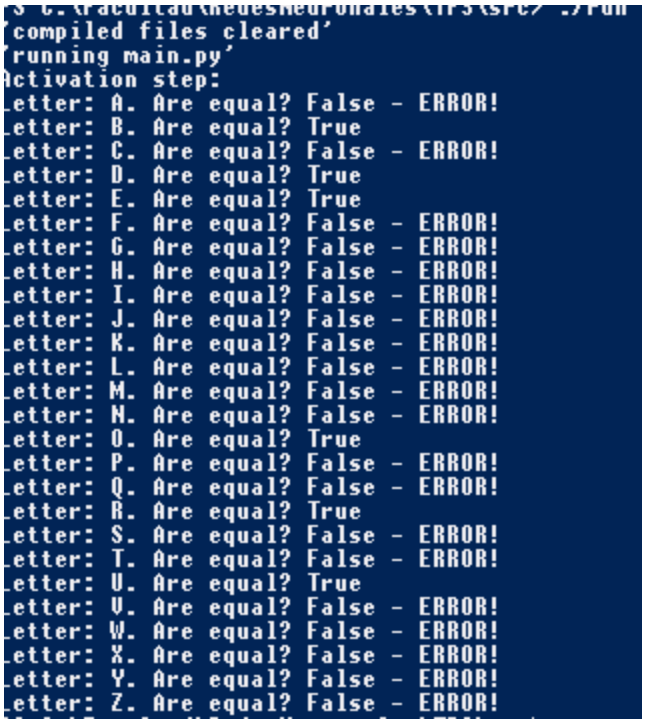
Como conclusión, Hopfield funciona bien si las memorias originales son lo más ortogonales posibles. Si el dominio del problema no conlleva intrínsecamente esta propiedad, será difícil para el algoritmo poder retener los patrones y producirá mayor cantidad de estados espurios.

## Implementación de letras del abecedario

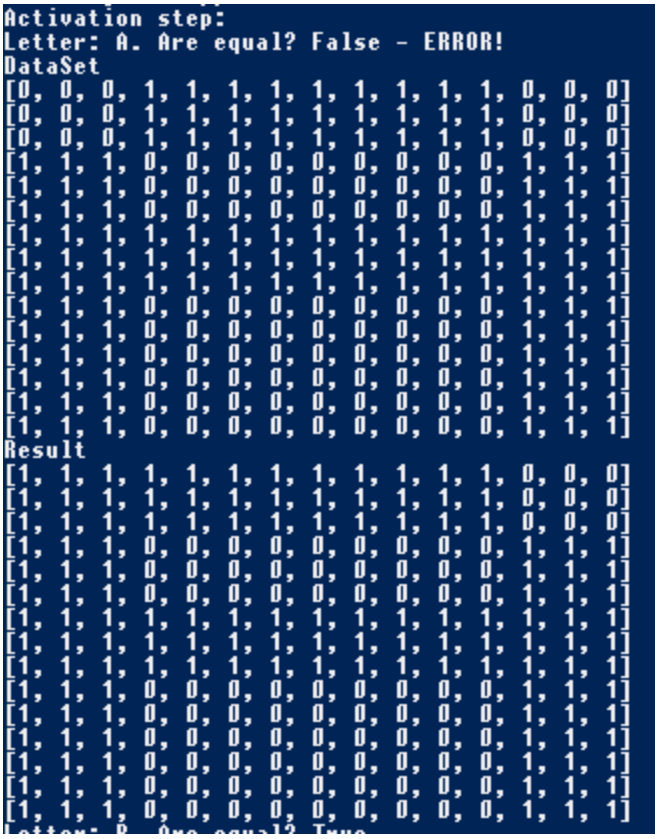
Para poder de alguna forma aprovechar la implementación de las letras del primer trabajo práctico, las cuales estaban representadas en matrices de 5x5 se implementó un algoritmo de remapeo que escala esas representaciones de la dimensión original de 5x5 a cualquier dimensión de NxN con N múltiplo de 5.

## Remapeo de 5x5 a NxN

El algoritmo consiste de expandir cada bit a una matriz de $KxK$ tal que $5K = N$. Tal como sucedió con los números, no logró aprender las letras pero sí surgieron notas interesantes. Al probar la activación de las memorias reconoció algunas pocas (6), pero aún modificando $K$ (de 3 a 50), siempre reconoció las mismas (B,D,E,O,R,V).




Decidimos imprimir las letras para ver qué sucedía y los resultados nos asombraron. El algoritmo parece reconocer cada matriz de $KxK$ y la trata como un sólo valor. Es decir o todos valen -1 o todos valen 1. Pareciera que se da cuenta que en realidad la información relevante sólo es la original de $5x5$ y todo el resto es decoro de tamaño. En el gráfico que se ve a continuación se modificaron los -1 por 0 simplemente para que sea posible ver a simple vista la letra que representa, el cómputo del algoritmo se hizo utilizando la representación bipolar.



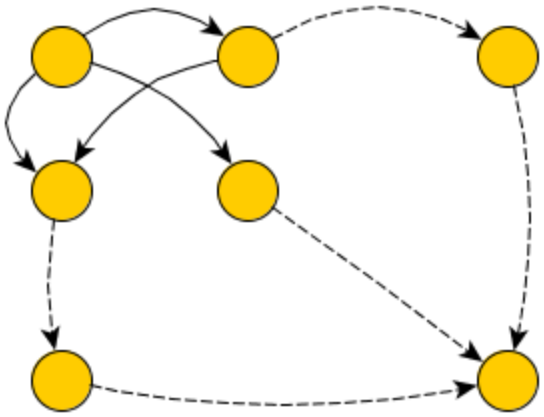

En la esquina superior izquierda puede verse una submatriz de 3x3 que difiere que el resultado de la activación del vector original de aprendizaje.Eesta submatriz de 3x3 fue la expansión del bit [0,0] de la representación de 5x5. Parece ser que el algoritmo reconoce esa información replicada en la pequeña submatriz de 3x3.

Dado que se comporta como una red neuronal de 25 neuronas, resulta trivial ver que es imposible que aprenda las 26 letras del abecedario, sin importar la cantidad de neuronas con las que se lo termine ejecutando al algoritmo.

# Ejercicio 3: **Hopfield estocástico**

## Arquitectura de la red

La red se conforma de 100 neuronas interconectadas todas entre sí.



### Introducción

A continuación se detallan los resultados obtenidos utilizando el algoritmo de Hopfield estocástico.

La forma de validar la condición de parada del algoritmo varió dependiendo de si se estaba testeando la activación a partir de una memoria (con 0 o algún porcentaje de ruido) o, si se estaba testeando la activación con vectores al azar.

En el primer caso la condición de parada fue que el vector de activación posea una distancia de Hamming inferior o igual al 10% (o sea a lo sumo difieran en 10 neuronas) en comparación con el vector en el paso anterior. En el segundo caso, se observó cada una cierta cantidad de iteraciones el valor del vector $s$ , exigiendo que lo observado se encuentre en promedio a menos de una cierta distancia de Hamming de al menos de una de las memorias

Se entrenó la red con diez vectores elegidos al azar de dimensión 100, generando una carga del 10% sobre la red.

## Implementación

La implementación del algoritmo fue realizada en Python teniendo parametrizada en la creación de instancia la temperatura con la cual trabajar.

El archivo “HopfieldStochastic.py” posee la implementación asociada.

# Conjuntos de entrenamiento

**Activación de memorias y memorias con ruido**

A continuación se detalla en una grilla la verificación de las memorias como atractores. Consta de 9 columnas: Las primeras 3 corresponden a temperatura, porcentaje de ruido y porcentaje de éxito (es decir haber recuperado la memoria que se esperaba debía recuperar).

El ruido que se utilizó fue invertir el valor de la neurona. Las segundas 3 columnas sirveN para validar la correctitud del agregado de ruido, para eso tenemos la distancia de Hamming máxima, mínima y promedio entre el vector con ruido y la memoria que el algoritmo debiera reconocer. Por último, se muestra la mejora de distancia de Hamming entre la memoria con ruido y la esperada; y la memoria obtenida después de la activación con la esperada. Se muestra la mejora máxima, mínima y promedio.

Dado que es una gran cantidad de información se procedió a mostrar únicamente lo considera más relevante.

Tomamos a 0.0001 como temperatura cero para realizar las pruebas ya que obtuvimos los resultados esperados y pudimos dejar una implementación más prolija (ya que utilizar el cero exacto produciría un error de división por cero).

| Temperatura | % Ruido | % Éxito | Hamming Max | Hamming Min | Hamming AVG | Mejora Hamming Max | Mejora Hamming Min | Mejora Hamming AVG |
|---|---|---|---|---|---|---|---|---|
| 0.0001 | 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.05 | 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 100 | 3 | 0 | 1 | 3 | 0 | 1 |
| | 21 | 70 | 28 | 17 | 22 | 27 | 11 | 20 |
| | 41 | 20 | 48 | 37 | 41 | 42 | 2 | 17 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 51 | 0 | 58 | 42 | 51 | 25 | 3 | 10 |
| 0.1 | 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.15 | 0 | 40 | 0 | 0 | 0 | 3 | 0 | 1 |
| | 1 | 40 | 5 | 0 | 1 | 4 | 0 | 1 |
| | 21 | 50 | 26 | 15 | 19 | 21 | 4 | 15 |
| | 31 | 0 | 39 | 24 | 29 | 28 | 12 | 21 |
| | 51 | 0 | 57 | 48 | 51 | 41 | 1 | 8 |
| 0.2 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 11 | 10 | 16 | 8 | 11 | 11 | 4 | 8 |
| | 31 | 0 | 36 | 24 | 30 | 28 | 1 | 15 |
| 0.25 | 1 | 10 | 2 | 0 | 0 | 8 | 0 | 4 |
| | 11 | 0 | 13 | 8 | 10 | 21 | 1 | 6 |
| | 61 | 0 | 69 | 55 | 61 | 29 | 2 | 11 |
| 0.3 | 0 | 10 | 0 | 0 | 0 | 10 | 0 | 4 |
| | 11 | 0 | 19 | 8 | 11 | 25 | 0 | 8 |
| | 61 | 0 | 65 | 53 | 60 | 34 | 1 | 12 |
| 0.35 | 0 | 0 | 0 | 0 | 0 | 53 | 4 | 20 |
| | 11 | 0 | 17 | 8 | 12 | 61 | 6 | 30 |
| | 61 | 0 | 67 | 57 | 61 | 19 | 0 | 10 |
| 0.4 | 0 | 0 | 0 | 0 | 0 | 64 | 5 | 42 |
| | 61 | 0 | 60 | 54 | 61 | 41 | 2 | 16 |

Efectuando la activación con la memoria exacta obtuvimos excelentes resultados a temperaturas bajas. A medida que se aumentaba, disminuyó la cantidad de memorias correctamente encontradas. La distancia de Hamming obtenida comenzó a aumentar, implicando que la memoria resultante difirió mucho de la esperada.

En general a mayor ruido, menor probabilidad de encontrar la memoria exacta. A menor temperatura, más probabilidades de encontrarlo hubo: por ejemplo, temperatura 0.15 y un 21% de ruido, se encontraron la mitad de las memorias.

Resulta muy interesante notar que el algoritmo mejora en todos los casos la distancia de hamming a las memorias. Aún teniendo temperaturas altas como 0.4 y mucho ruido (61%), se mejoraron un promedio de 16 neuronas por vector.

Suena razonable utilizar una temperatura media (alrededor de 0.5) en primera instancia para reducir la cantidad de estados espurios, a la vez que acercamos el vector a una memoria. Luego, se podría reducirla para que algoritmo converja a la memoria deseada. Puede suceder que si las memorias son poco ortogonales entre sí, aplicar esta técnica nos haga que el resultado converja a la memoria "similar" en vez de la cual con la que se creó.

**Activación de vectores aleatorios**

A continuación se muestra la información asociada a la activación de 5000 vectores aleatorios (todos distintos) dependiendo de la temperatura dada. Para la condición de parada del algoritmo se requirió que: cada 8 iteraciones se tomara una muestra (es decir, visualizar el vector S). A partir de las 10 muestras, requerir que el vector actual coincida o esté como máximo a 10 de distancia de Hamming con el 65% de las muestras tomadas.

| Temperatura | Memorias encontradas | Estados espurios encontrados |
|---|---|---|
| 0.01 | 1065 | 3935 |
| 0.06 | 1191 | 3809 |
| 0.11 | 895 | 4105 |
| 0.16 | 433 | 4567 |
| 0.21 | 61 | 4939 |
| 0.26 | 4 | 4996 |
| 0.31 | 0 | 5000 |
| 0.36 | 0 | 5000 |
| 0.41 | 0 | 5000 |

Es interesante ver como con 0.06 tuvimos mejores resultados en con 0.01, aumentar la temperatura permitió en este caso mejorar el resultado que teníamos con el que es casi determinístico. El aumento de temperatura tiende a perder resultados exactos y aumentar la cantidad de resultados espurios.

Considerando que es posible que los parámetros elegidos puedan ser demasiado laxos, hemos decidido aumentarlos a: cada 15 iteraciones se tomara una muestra (es decir, visualizar el vector S) y a partir de la muestra 20 requerir que el vector actual coincida o esté como máximo a 5 de distancia de Hamming con el 90% de las muestras tomadas. Hemos tomado 0.3 como temperatura para efectuar esta prueba.

El programa no logró terminar, el simple hecho de cambiar de 8 a 15 las iteraciones antes de tomar la muestra ya provocó que el algoritmo no finalice.

Dado que hemos comprobado que es muy difícil que el algoritmo converja efectivamente a una memoria, se generó un nuevo test diferenciando los inputs que se encuentran (una vez activado) a una distancia de Hamming menor o igual a 25 neuronas de alguna de las memorias.

| Temperatura | Memorias encontradas | Estados espurios con dist Hamming< 25 | Estados espurios encontrados |
|---|---|---|---|
| 0.2 | 12 | 3449 | 1551 |
| 0.25 | 5 | 3219 | 1781 |
| 0.30 | 0 | 2793 | 2207 |

Debemos notar dos cosas de estos resultados. La primera es que hacer la condición de parada más laxa hace que se encuentren muchas menos memorias de manera exacta. Sin embargo, es muy interesante pensar que mucho más de la mitad de los patrones se encuentra a menos de un 25% de ser una memoria

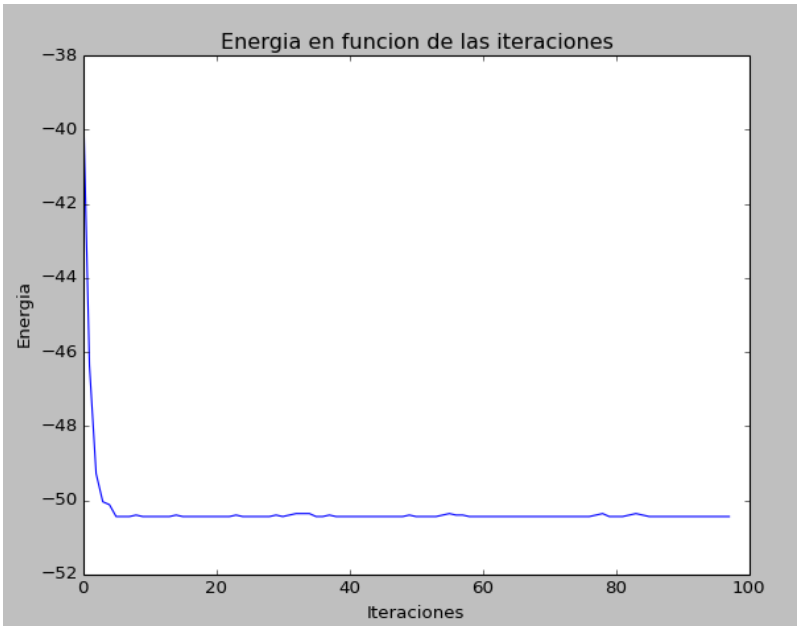
**Combinaciones Lineales**

Se efectuaron pruebas con combinaciones lineales de las entradas haciendo variar la temperatura. La combinación lineal se efectuó con 3, 5 y 7 memorias. La condición de parada fue: “entre dos iteraciones, la distancia de Hamming debe ser menor a 5”.

A continuación se muestran los resultados variando la temperatura.

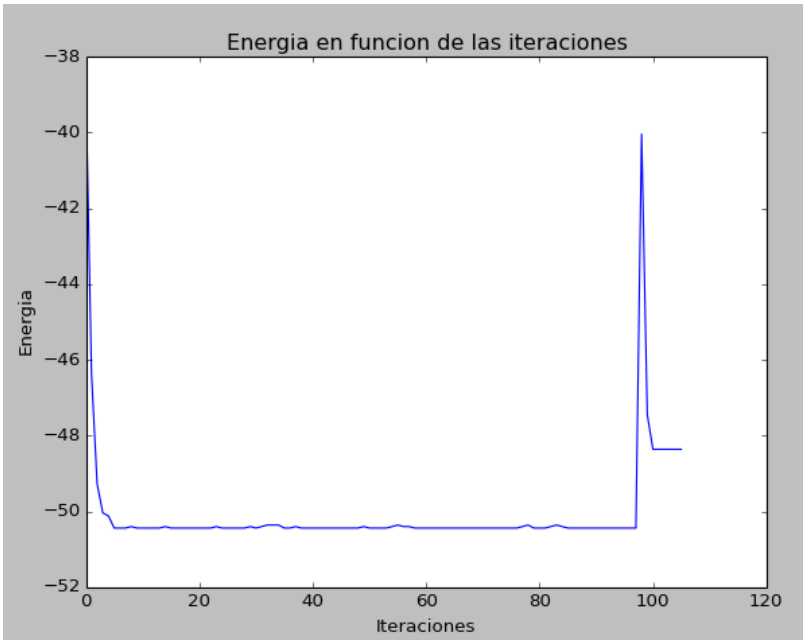
**Temperatura: 0.01**

- **Cantidad de memorias en la combinación lineal: 3**

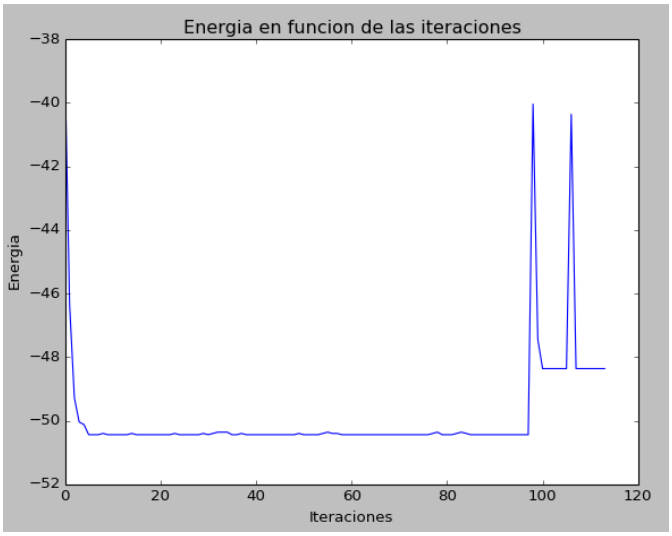



En pocas iteraciones el algoritmo logra disminuir la energía y sentarse alrededor de una memoria.

- **Cantidad de memorias en la combinación lineal: 5**



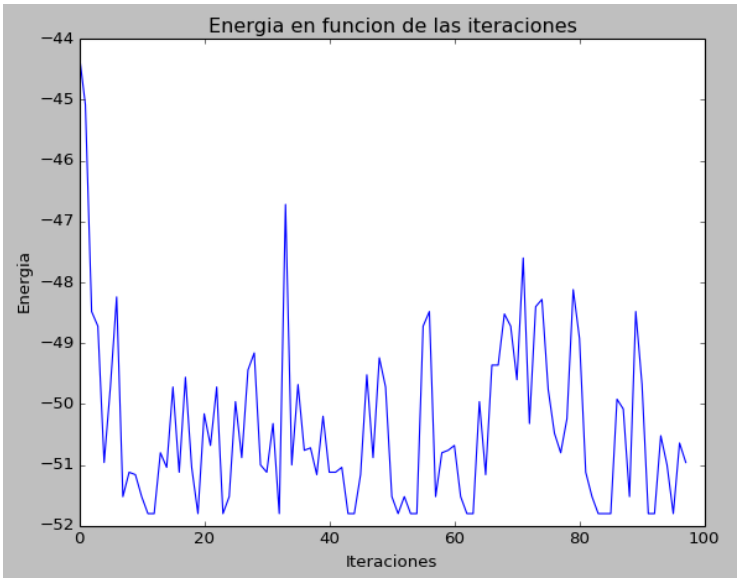
- **Cantidad de memorias en la combinación lineal: 7**



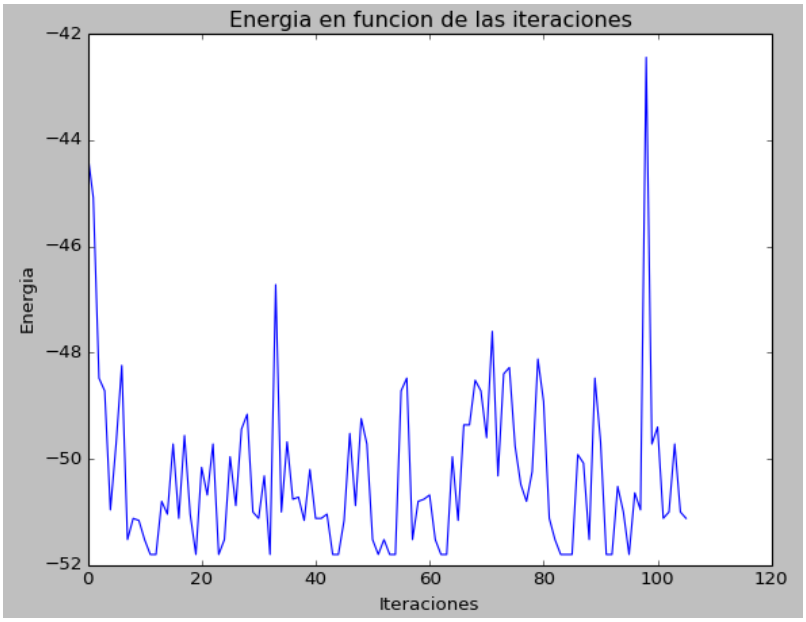

Cuando la combinación lineal se produce con más vectores y la temperatura es baja el algoritmo puede ir a un estado espurio y converger ahí. Vemos con combinaciones de 5 y 7 vectores pareciera converger a la misma solución que el de 3 hasta que en una iteración cercana a la 100 disminuye la energía y pierde el mínimo ya alcanzado

**Temperatura: 0.15**

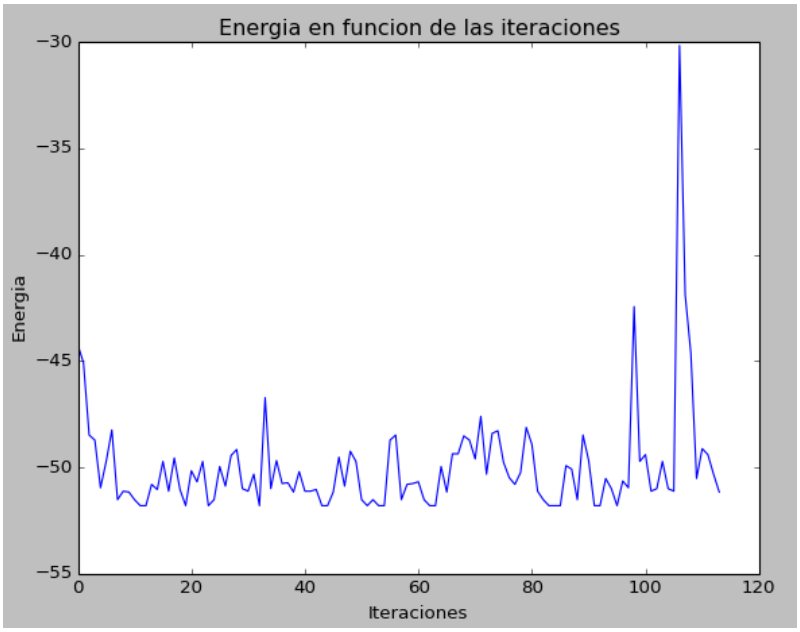
- **Cantidad de memorias en la combinación lineal: 3**



- **Cantidad de memorias en la combinación lineal: 5**




- **Cantidad de memorias en la combinación lineal: 7**

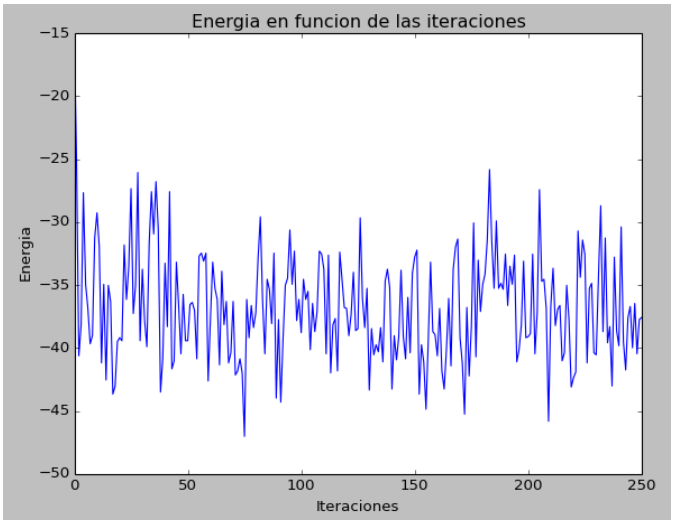



Al aumentar la temperatura se ve que fluctúa mucho más la energía. Además la cantidad de iteraciones aumenta conforme la cantidad de memorias en la combinación lineal. Es importante notar, que el aumentar la temperatura resta importancia a la cantidad de combinaciones ya que los tres gráficos muestran convergencia a energía mínima.

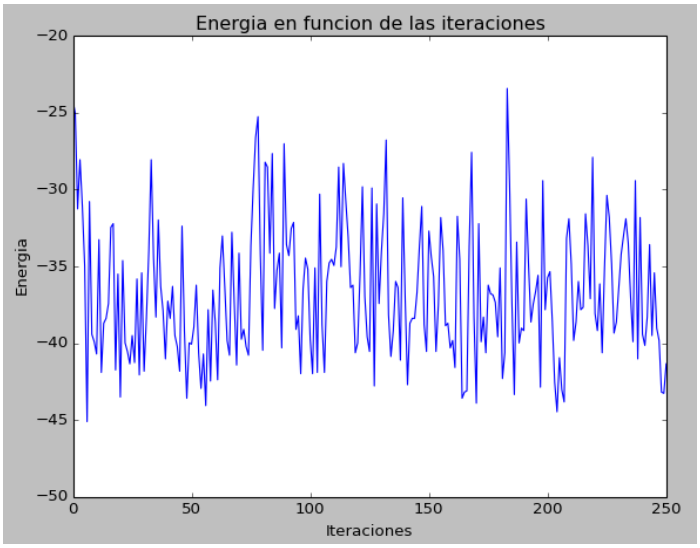
**Temperatura: 0.35**

**NOTA:** Se tuvo que limitar la cantidad de iteraciones a 250 para mejor claridad de los gráficos ya que el algoritmo no lograba minimizar la distancia de Hamming (hicimos pruebas y efectuó al menos 750 iteraciones sin lograrlo).

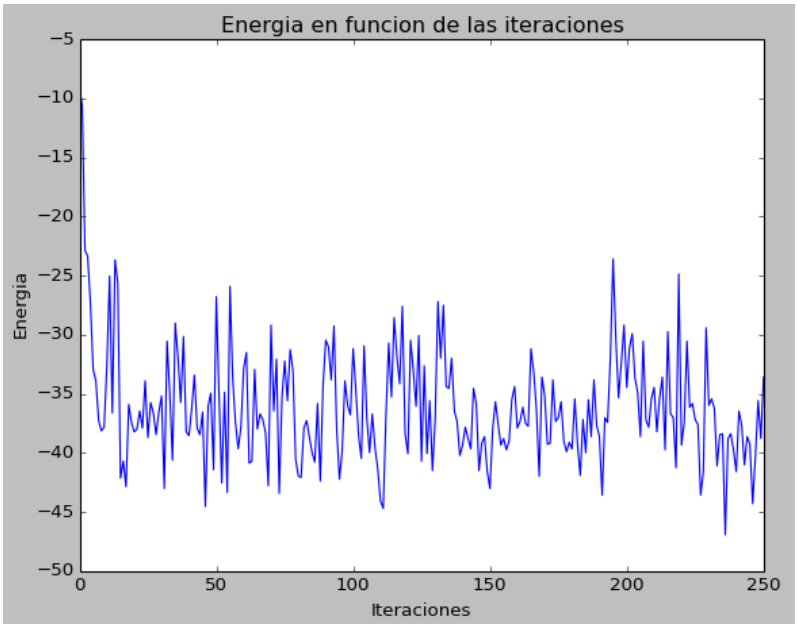
- **Cantidad de memorias en la combinación lineal: 3**




- **Cantidad de memorias en la combinación lineal: 5**



- **Cantidad de memorias en la combinación lineal: 7**




Los últimos tres gráficos confirman que el aumento de la temperatura agudiza la fluctuación en la energía. Podemos ver que aumentar la temperatura aumenta la cantidad de iteraciones para converger a una solución, y si es muy alta el algoritmo inclusive podría nunca terminar al nunca cumplir la condición de parada exigida.

Por otro lado, agrega más peso a nuestra hipótesis de que quizás es una buena idea modificar la temperatura de manera decreciente para poder salir de los estados espurios.