APROPIACIÓN DE CONOCIMIENTO Y PUESTA EN MARCHA DE RED NEURONAL ARTIFICIAL MORFOLÓGICA DE DENDRITAS MEDIANTE ALGORITMO DE EVOLUCIÓN DIFERENCIAL

Omar Jordán Jordán
Código:1038239, omar.jordan@correounivalle.edu.co

Universidad del Valle, Facultad de ingeniería, Escuela de ingeniería Eléctrica y Electrónica, Santiago de Cali, Colombia 2018

I. INTRODUCCIÓN

En el presente informe se muestra la estructura de una red neuronal morfológica de dendritas (DMNN), sus capacidades en comparación a otras redes neuronales artificiales (ANN), y un ejemplo programado hecho con la finalidad de entender cómo puede esta red ser sintonizada mediante el algoritmo estocástico de evolución diferencial (DE); los resultados fueron satisfactorios y esperados, en concordancia con el artículo de Fernando Arce, Erik Zamora, Humberto Sossa y Ricardo Barron [1], en el cual se basa este trabajo.

II. MARCO TEÓRICO

Red Neuronal Perceptrón:

En la estructura básica clásica de la neurona artificial, sean “n” entradas, la salida “y” es el resultado de una función de activación “f(x)” que puede ser o no derivable, lineal, continua, etc; entre las más comunes se encuentran: identidad (lineal), lineal a tramos, escalón, escalón binario, sigmoidal, sigmoidal binaria, gaussiana y rampa. Esta función recibe la sumatoria de el valor bias “b” (que evita matemáticamente que la solución pase necesariamente por el origen) con las entradas “x” multiplicadas por un peso sináptico “w” asociado a cada una de ellas. La solución es un hiperplano en el hiperespacio de entradas (combinación lineal de parámetros) [6].

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

Red Neuronal Morfológica:

Una concepción diferente de neurona artificial nace de los avances en la teoría del álgebra de imagen [3], la cual usa a su vez operaciones matemáticas morfológicas [4] de donde se resaltan las operaciones de Erosión y Dilatación pues son usadas por la neurona artificial morfológica. En ésta cambia las operaciones de multiplicación de los pesos sinápticos con las entradas en la neurona clásica por operaciones de suma y la sumatoria de dichas multiplicaciones es cambiada por operadores máximo o mínimo, con lo cual queda claro que la ecuación de esta neurona artificial no es derivable ni lineal incluso desde antes de la función de activación [5].

$$y = f\left(b + a_0 . \bigwedge_{i=1}^{n} a_i . (x_i + w_i)\right)$$

Luego parámetros extra que valen ±1 son añadidos, el “ai” multiplica cada sumatoria dentro de la función mínimo o máximo, habilitando o deshabilitando dicha entrada; el “a0” multiplica a toda la función mínimo o máximo con lo cual deshabilita o habilita la salida (ya que suele usarse la función de activación hard-limiter o escalón con este tipo de neuronas).

Las propiedades son diferentes a la clásica ANN, por ejemplo, un perceptrón monocapa basado en MNN puede ser entrenado con un número finito de iteraciones, evitando el problema de convergencia; además separa las clases a clasificar en forma de superficie de decisión correspondiente a un impulso infinito (como una curva de potencia en 2D) en lugar de una separación lineal como lo haría un clásico perceptrón.

Red Neuronal Morfológica con Dendritas:

Para explicar el funcionamiento de una neurona morfológica con dendritas se requiere primero de “n” neuronas en la capa inmediatamente anterior que proporcionen las entradas a la red “x”, a diferencia del perceptrón clásico donde todas las entradas llegan directamente al cuerpo de la neurona siguiente. Aquí dicho cuerpo posee una cantidad dada de dendritas o terminales de entrada “K”, estos enviarán su información al cuerpo de la neurona para ser operada con la función máximo o mínimo y luego pasar por la función de activación “f(x)” para generar la salida [2].

$$y = f\left(\bigwedge_{k=1}^{K}\left(p_k . \bigwedge_{i=1}^{n} l_{ik} . (x_i + w_{ik})\right)\right)$$

Cada neurona “n” de la capa anterior divide su salida “x” (axón) en varias ramas, donde cada una puede adherirse a la dendrita “k” de la neurona siguiente; a su vez cada rama del axón tiene su peso sináptico “w” con la dendrita y un parámetro “l” ±1 que denota excitación o inhibición. La salida “x” que viaja por el axón es sumada a dicho peso “w” y esta suma se multiplica por -1 si el parámetro “l” es inhibitorio. Todos estos axones que llegan a la dendrita son operados con la función mínimo o máximo y finalmente multiplicado por el parámetro “p” ±1 de excitación o inhibición de la dendrita. Este es el procesamiento de la dendrita, Luego ¿cuantas dendritas debe tener una neurona? ahí es donde radica el reto de entrenar la red generando la cantidad adecuada de dendritas para solucionar el problema, además de estimar todos los parámetros.

La interpretación geométrica es que cada dendrita crea un hipercubo en el hiperespacio [7], de aquí que un algoritmo de aprendizaje se basa en crear un hipercubo que englobe todos los patrones e irlo dividiendo iterativamente confirmando que cada hipercubo encierre solo una clase de los patrones, haciendo además combinación de hipercubos cercanos para optimizar. Finalmente, una vez todos los patrones están encapsulados, se genera la red neuronal asociando a cada dendrita uno de estos hipercubos [8]; en otros algoritmos este método se usa exclusivamente como un inicializador de la DMNN a causa de su menor capacidad de generalización y optimización en cuanto al número de dendritas [1].

Evolución Diferencial:

El principio de funcionamiento de los algoritmos evolutivos, consiste en empaquetar los parámetros del problema a resolver como una cadena, que representa las características de una posible solución. Dicha cadena corresponde a un individuo y emula la idea del genoma biológico [9]. Por lo tanto, ante un número de individuos, seleccionados a través de diversos métodos, se produce una nueva generación de potenciales soluciones; entre más iteraciones (generaciones) transcurran, se iran alcanzando soluciones más óptimas. Las funciones aleatorias de selección, cruce y mutación juegan un rol importante y hacen que este tipo de algoritmos estén en el grupo de los llamados estocásticos.

La evolución diferencial usa la idea de perturbar con diferencias vectoriales la creación de una nueva generación [10], estos vectores diferenciales de perturbación “v” son creados a partir de (en la variante más básica) la selección de 3 individuos diferentes al azar “r1, r2 y r3” y su combinación matemática vectorial controlada por un parámetro “h” (tasa de mutación). La nueva generación se crea entonces en dos pasos; primero se compara componente a componente un valor aleatorio con un parámetro “c” (tasa de cruce) para escoger componentes de “x” o “v” y obtener los llamados vectores de prueba “u” (cruzados de los anteriores); luego éstos son

comparados en desempeño (mediante la función de costo) con los individuos de la población actual “x” (de tamaño “m”) y son ordenados de tal manera que representan la nueva generación de individuos.

```
Repetir i hasta m:
        xi = rand
repetir:
        repetir i hasta m:
                r1 = rand % m
                r2 = rand % m
                r3 = rand % m
                vi = x(r1) + h . ( x(r2) – x(r3) )
                repetir k hasta tamaño(vi):
                        si rand < c
                                ui(k) = vi(k)
                        sino
                                ui(k) = xi(k)
        repetir i hasta m:
                si error(ui) < error(xi)
                        xi = ui
        condición parada
```

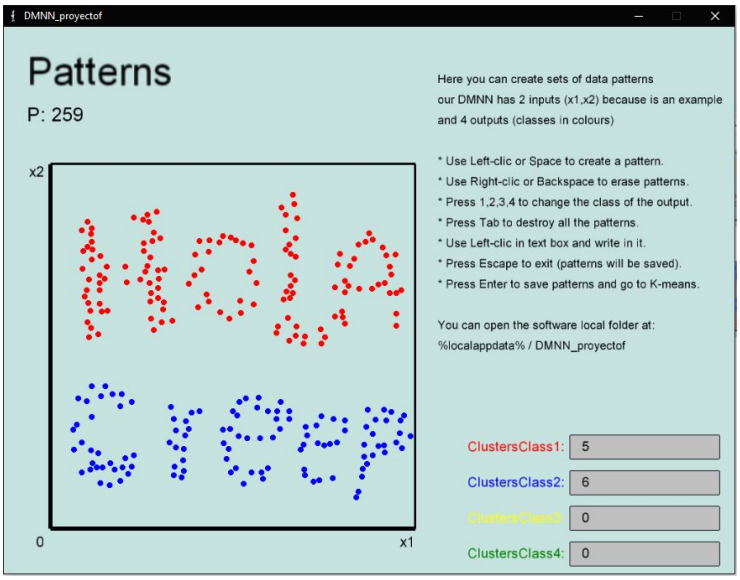
Este algoritmo ha dado resultados superiores a los clásicos algoritmos genéticos, y es ampliamente usado en el campo de la inteligencia computacional; por ejemplo, este algoritmo ha sido implementado para la sintonización de parámetros de una DMNN [1].

III. EXPERIMENTACIÓN

La idea fue clasificar patrones de 4 clases representadas por colores (rojo, azul, amarillo, verde), estos patrones se encuentran esparcidos por el espacio bidimensional x1, x2 (dos entradas), esto con el fin de que el problema sea fácilmente graficado y visualizado.

Primero que todo se eligió como software de desarrollo Game Maker Studio, por ser un entorno y lenguaje (similar a C++ pero de alto nivel) con el cual ya se había trabajado antes en el curso y en la vida, brinda la velocidad de ejecución requerida y una flexibilidad propia de un motor de videojuegos (2D).

Se creó una interfaz de usuario donde pueden ingresarse manualmente los patrones, clickeando directamente sobre la gráfica, eligiendo previamente la clase a instanciar; también en esta GUI se elige la cantidad de clusters por clase para ser sintonizados con el algoritmo K-means de la otra GUI.

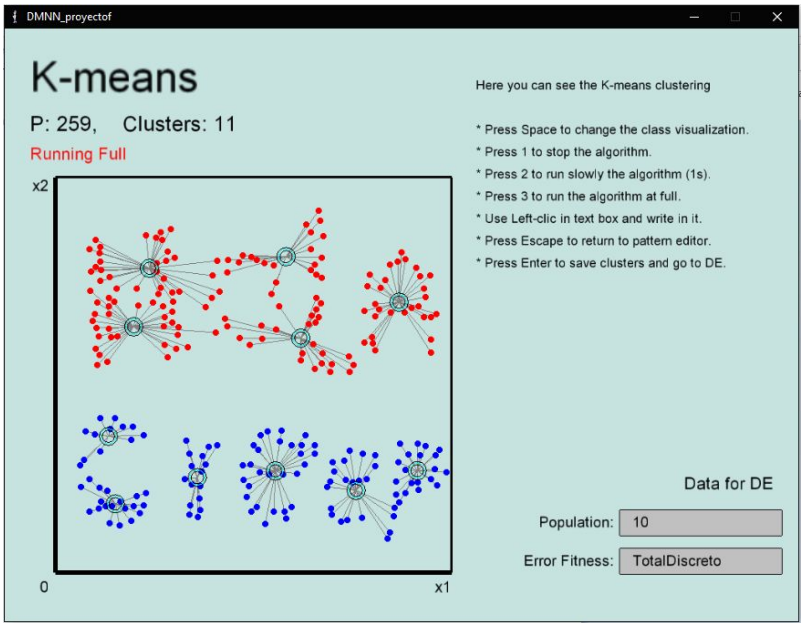


Cada dendrita representa una hiper-caja en el hiper-espacio, es por ello que la solución a un problema de clasificación es la organización de estas hiper-cajas (cajas para nuestro caso), en el estado del arte [1] se plantea inicializarlas en lugares convenientes en lugar de lanzarlas al azar como se hace tradicionalmente con los pesos sinápticos de redes como el multi layer perceptron (MLP); esto ofrece ventajas de inicio para el entrenamiento de la DMNN.

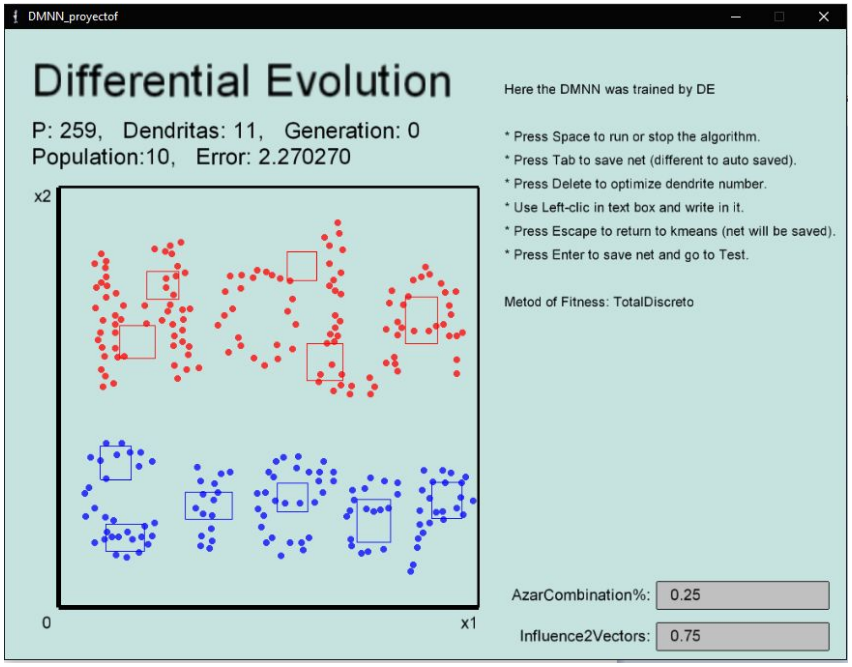
Se utiliza el algoritmo K-means con una leve modificación en su propia inicialización, el algoritmo es el siguiente:

```
repetir c hasta numero_clusters:
        cluster(c) = patrón(rand)
repetir:
        repetir p hasta numero_patrones
                patrón(p).pertenece = cluster_mas_cercano(p)
        repetir c hasta numero_clusters:
                cluster(c) = promedio_patrones_pertenecen(c)
        condición parada
```

La GUI del K-means muestra la ubicación de los clusters y a que patrones están asociados, el algoritmo puede correr lentamente (cada 1s) para ver su funcionamiento o a lo que da el software; por otra parte el usuario puede elegir la cantidad de individuos que tendrá el algoritmo genético de la otra GUI así como uno de los tres algoritmos de error (fitness) que usará el método estocástico.



Luego en el apartado de evolución diferencial se inicializa la primera generación tomando las posiciones de los clusters como centros de las dendritas, las cuales adquieren una dimensión azarosa, esto se repite para todos los individuos y por comodidad de observación sólo un individuo será mostrado en la gráfica (todas las cajas de las cuatro posibles clases pertenecen a un individuo), este siempre es el que posee el mejor error, mismo que es mostrado en su respectivo label.



El algoritmo de evolución diferencial funciona tal como se explica en el apartado del marco teórico. En esta GUI están presentes los dos parámetros que se involucran, los cuales pueden ser modificados en plena marcha.

La función de error en general inicia mostrando a la red DMNN todo el set de patrones, ejecutando su proceso y comparando la salida deseada con la obtenida. El proceso de la DMNN es el siguiente:

$$y_k = \overset{1}{\underset{h=0}{V}} \left(\overset{n}{\underset{i=1}{V}} \left((-1)^h . \left(x_i + w_{ik}^h \right) \right) \right)$$

$$y = \overset{K}{\underset{k=1}{\Lambda}} \left(y_k \right)$$

Sea “y” la salida de una neurona y “yk” la salida de su “k-ésima” dendrita respectivamente. La variable “h” como superíndice del peso “w” no es una operación de potencia, es simplemente un “w” tipo 1 y otro tipo 0 donde cada uno representa un extremo de la hiper-caja, entonces la operación “menor” más externo siempre seleccionará entre dos valores que representan dichos extremos.

Para el presente caso se tienen 4 neuronas, cada una representa una clase, entonces para una entrada dada se selecciona al final la neurona con el valor mayor como la clase ganadora, hay una opcion de poner valores menores a cero como clase indefinida, esto lleva a problemas de mala generalización y discontinuidades.

Al final este resultado se usa para hallar el error, se proponen 3 métodos donde se comparte que todos obtienen un aporte de error dado por cada patrón y luego se divide esta neta entre el número de patrones (promedio):

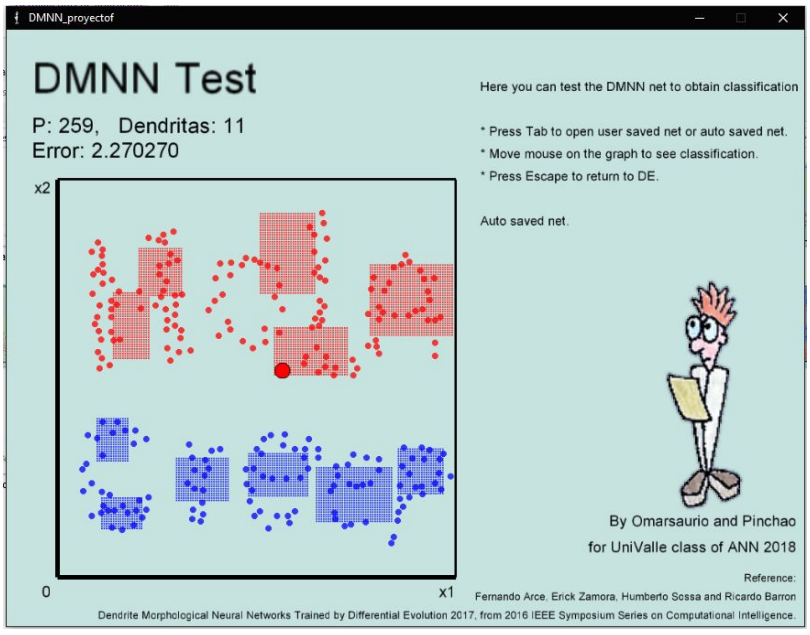
- SalidaTodoNada:
  si la clase deseada es diferente de la esperada entonces se suma uno a la neta del error.
- TotalContinuo:
  recorre todas las neuronas, si la neurona actual es la asociada a la clase deseada y su valor está por debajo de 0 entonces agrega a la neta del error el valor absoluto de esta salida; si no es la asociada a la clase deseada y su valor es mayor que cero entonces sumará este valor a la neta del error.
- TotalDiscreto:
  igual al anterior pero agrega uno al error en lugar del valor de la salida.

En esta GUI los algoritmos corren hasta alcanzar error 0 o hasta que el usuario da orden de detener. Otro comando disponible es el de optimización de dendritas, esto lo que hace es eliminar dendritas (hiper-cajas) bajo dos condiciones:

- Vacías:
  si no encierran a ningún patrón de su respectiva clase.
- Cruzadas:
  evalúa todas las dendritas en orden ascendente respecto a su área (para 2D o volumen para 3D) eliminando aquellas que al simular ser eliminadas no aumentan el error, esto es un algoritmo de fuerza bruta y utiliza el primer tipo de función de error “SalidaTodoNada” por lo que puede haber cambios en el error si se ha entrenado con otra función.

Esta función de optimización puede disminuir dramáticamente el número de dendritas, pero puede crear daños irreversibles, así que hay una opción de que el usuario pueda guardar una red cuando el lo desee.

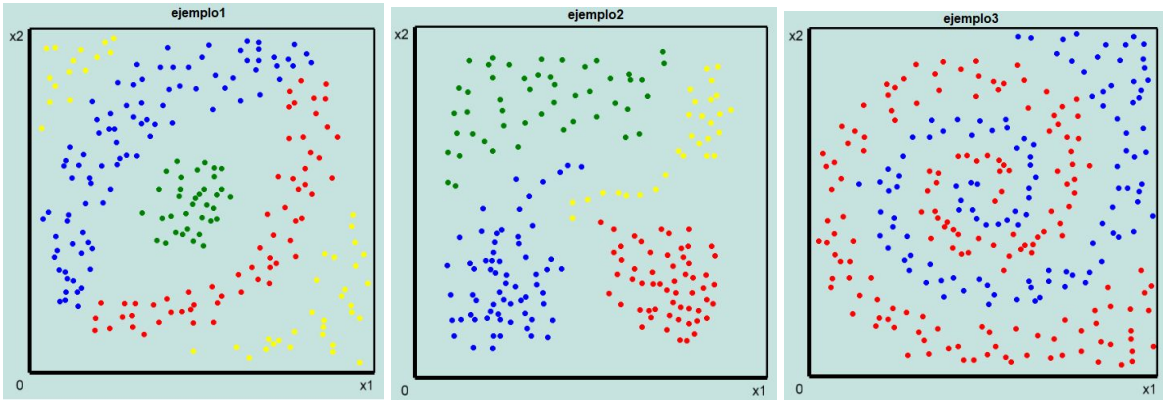
El programa finaliza con la siguiente GUI:



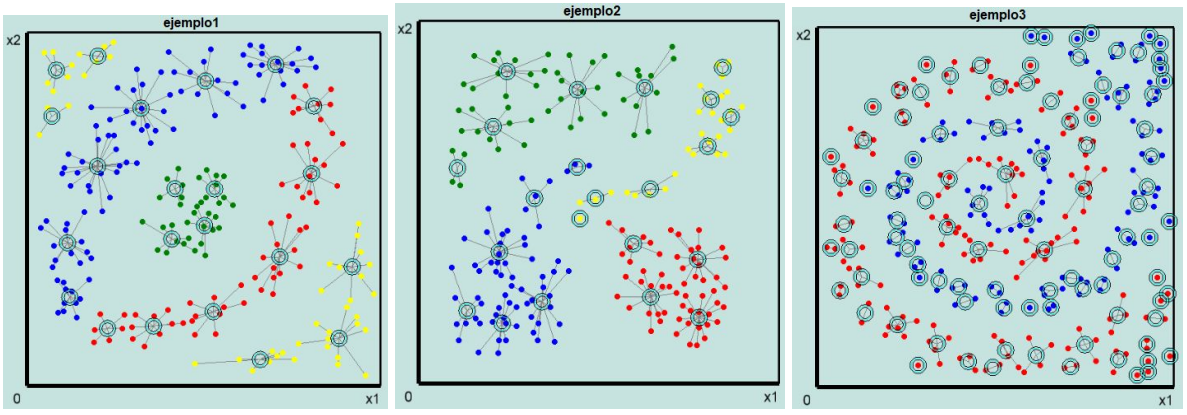
Aquí se puede observar que zonas de la gráfica pertenecen a cada clase, puede escogerse entre mostrar la red neuronal guardada por el usuario o la auto-guardada.

IV. RESULTADOS

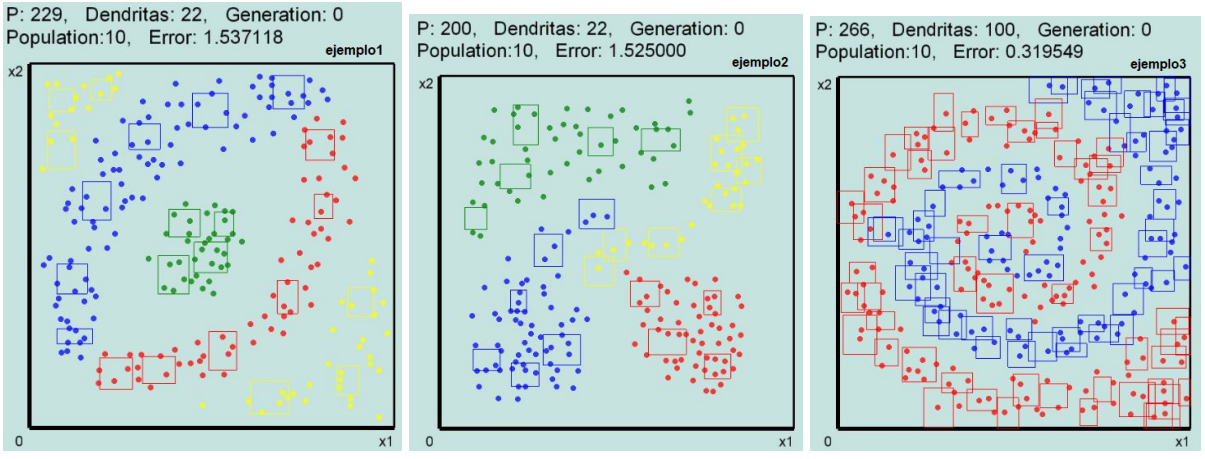
Se probaron 3 ejemplos de patrones ingresados a mano:



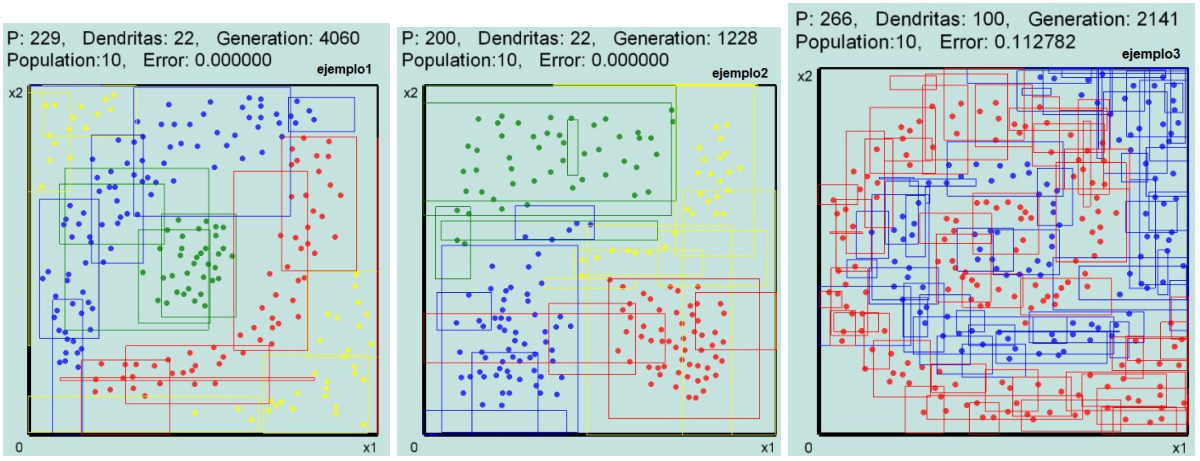
Los números de clusters se escogieron viendo a ojo más o menos cuantos se necesitan, en un hiper-espacio hacer esto no es muy viable así que una opción además del continuo testeo es elegir valores grandes de dendritas y al final eliminar las sobrantes.



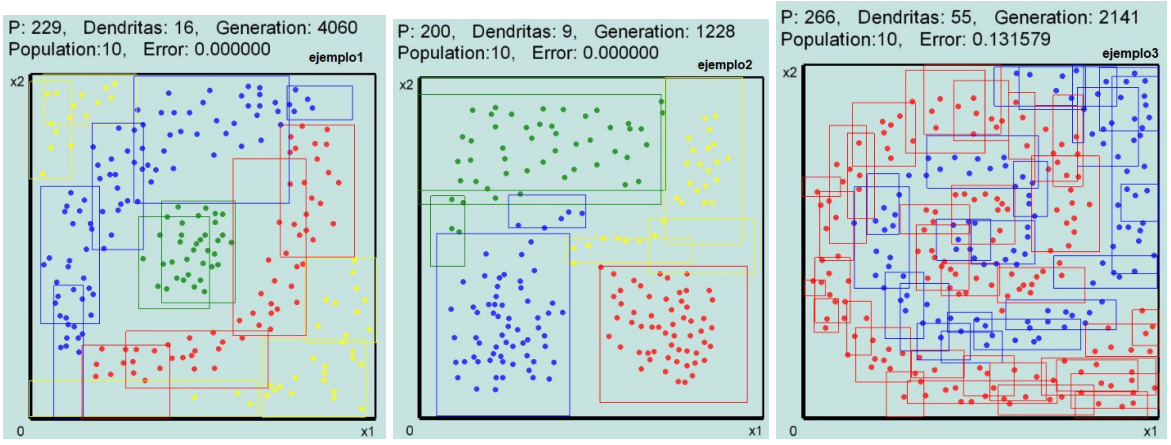
Se inicializan las redes neuronales para todos los individuos, aunque se muestra solo al mejor.



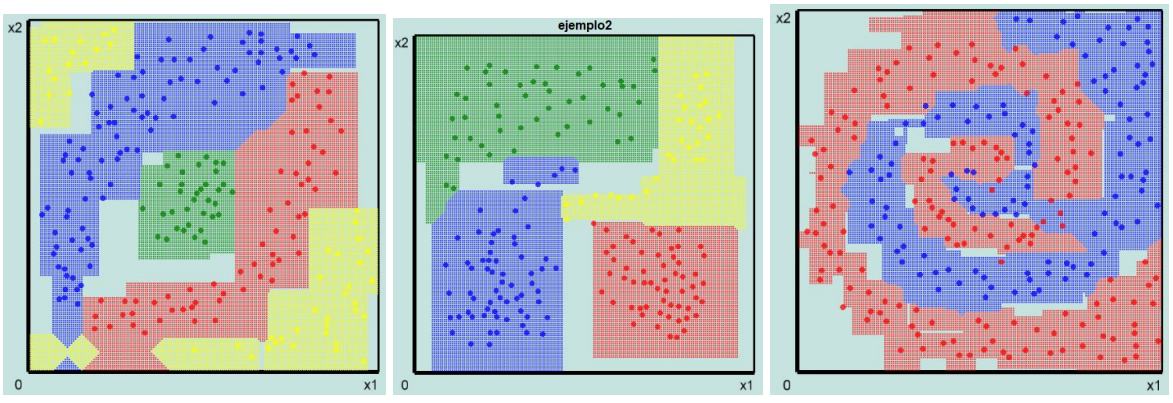
Y comienza el entrenamiento, usando algoritmo de error “TotalDiscreto”, 10 individuos por generación, azar combinacional de 0.25 e influencia de dos vectores de 0.75; como se observa los dos primeros casos obtuvieron error 0, es viable alcanzarlo a causa de la forma geométrica compleja que pueden mostrar todas las cajas de las dendritas, sin necesariamente perder capacidad de generalización.



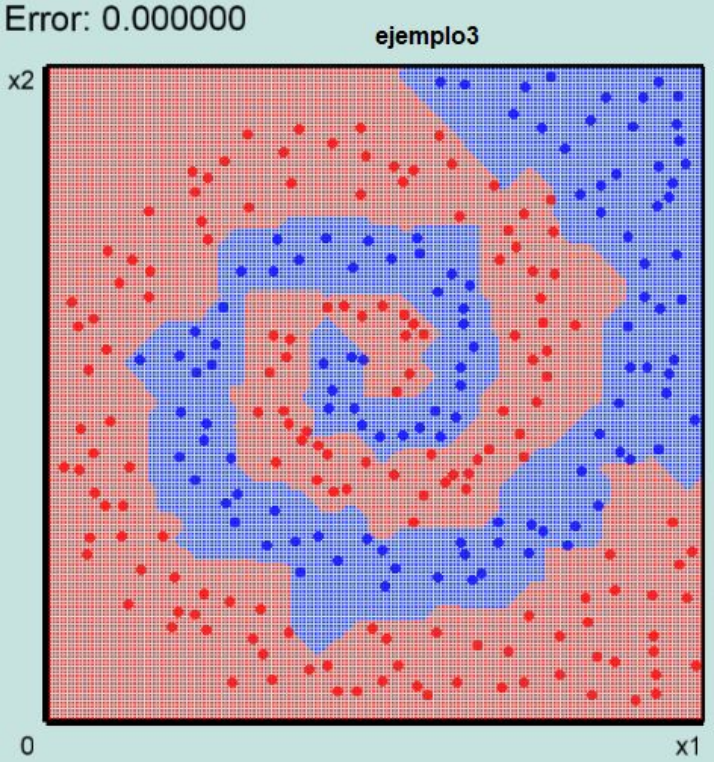
Luego del entreno se ejecuta el código de optimización.



Como puesta a prueba vemos a continuación todas las posibilidades dentro del gráfico, se aclara que aquí se usaron valores menores a 0 como clase indefinida; de allí que el ejemplo 3 posea esos bordes tan toscos y zonas indefinidas que afectan la continuidad de la solución.



Como caso final, se re-entreno el ejemplo 3 sin poner valores negativos a la salida de la red como indeterminado; esto claramente deja ver una mayor capacidad de generalización y continuidad en los límites de las figuras.



V. CONCLUSIONES

Los resultados fueron muy satisfactorios puesto que para problemas de mayor complejidad a lo habitual como lo es la espiral doble del ejemplo 3, lograron separarse las zonas de decisión; aún así la generalización de las soluciones es notoriamente un problema latente. Lo experimentado puede llevarse a cabo para “n” entradas y promete crear zonas de decisión complejas sin necesidad de añadir capas ocultas de DMNN. La función de error es susceptible de ser mejorada, esta es crucial para el algoritmo genético, así como el hallar un criterio para elegir clusters.

VI. REFERENCIAS

[1] Fernando Arce, Erick Zamora, Humberto Sossa and Ricardo Barron, “Dendrite Morphological Neural Networks Trained by Differential Evolution”, 2017, from 2016 IEEE Symposium Series on Computational Intelligence.

[2] Gerhard X. Ritter, Laurentiu Iancu and Gonzalo Urcid, “Morphological Perceptrons With Dendritic Structure”, 2003 Fuzzy Systems The 12th IEEE International Conference On.

[3] Joseph N. Wilson and Gerhard X. Ritter, “Handbook of Computer Vision Algorithms in Image Algebra”, 2000 by CRC Press, ISBN: 9780849300752.

[4] Vincent Morard, Etienne Decenciere and Peter Dokladal, “Mathematical Morphology and Its Applications to Image and Signal Processing”, 2011 CMMCentre de Morphologie Mathematique, ISBN: 978-3-642-21569-8.

[5] Gerhard X. Ritter and Peter Sussner, “An Introduction to Morphological Neural Networks”, 1996 Proceedings – International Conference on Pattern Recognition 4,547657, pp. 709-717.

[6] Eduardo F. Caicedo Bravo y Jesus A. Lopez Sotelo, “Una Aproximación Práctica a las Redes Neuronales Artificiales”, 2013 Universidad del Valle, ISBN: 978-958-670-767-1.

[7] Gerhard X. Ritter and Gonzalo Urcid, “Lattice Algebra Approach to Single Neuron Computation”, 2003 IEEE Transactions On Neural Networks, vol 14 no. 2.

[8] Humberto Sossa and Elizabeth Guevara, “Efficient Training for Dendrite Morphological Neural Networks”, 2014 Neurocomputing 131, pp. 132-142.

[9] Carlos A. Coello Coello, David A. Van Veldhuizen, Gary B. Lamont, “Evolutionary Algorithms for Solving Multi-Objective Problems”, 2002 ISBN: 978-0-306-46762-2.

[10] Kenneth V. Price, Rainer M. Storn and Jouni A. Lampinen, “Differential Evolution a Practical Approach to Global Optimization”, 2005 Springer, Natural Computing Series, ISBN-10 3-540-20950-6.