FACI: Un Framework para Inteligencia Computacional Académica

Jorgelina E. Barnij[1]

Universidad Tecnológica Nacional, FRSF, Santa Fe, Argentina

Resumen. Uno de los principales desafíos en la utilización de redes neuronales es el entrenamiento de las mismas. En la mayoría de los casos, se recurre para tal fin, a técnicas de descenso por el gradiente, como el algoritmo de Retropropagación del error. Sin embargo, éstas a veces pueden ser muy costosas en tiempo de cómputo, utilización de memoria, etc. En este trabajo se presenta un framework para el diseño de redes neuronales que pueden ser entrenadas a través de distintos algoritmos de entrenamiento.El framework se describe para el caso de una red neuronal del tipo Perceptrón Multicapa que permite la detección de números naturales de un dígito. Se muestra el entrenamiento mediante la utilización del algoritmo de Retropropagación del error y algoritmos Genéticos y se compara mediante el tiempo promedio de entrenamiento y la exactitud de la red entrenada. En base a los resultados, se verá que el Algoritmo Genético es una alternativa atractiva respecto al clásico algoritmo de Retropropagación del error.

Palabras clave: Algoritmo Genético, Algoritmo de Retropropagación del Error, Redes Neuronales Multicapa, Aprendizaje Supervisado, Clasificación.

1 Introducción

La mayor parte de los métodos que se utilizan en el entrenamiento de redes neuronales está basada en métodos de entrenamiento que se apoyan en las técnicas de descenso del gradiente, como por ejemplo, el algoritmo de Retropropagación del error (BP - del inglés Back Propagation) desarrollada por Werbos [1] y popularizada por Rumelhart y McClelland [2]. Sin embargo, Curry y Morgan [3] señalan que las técnicas de gradientes no proveen la mejor forma ni la más rápida para entrenar redes neuronales.

Habitualmente no se utilizan otros métodos para entrenar redes neuronales debido a que se cuenta con pocas herramientas adecuadas para ello, o bien las que hay disponibles son cerradas, es decir que no se puede acceder a sus códigos fuentes y modificarlos en pos de las necesidades particulares. Esto se hace necesario principalmente debido a que el método de entrenamiento más eficiente depende en gran medida del problema que se esté queriendo resolver mediante la utilización de la red neuronal.

Una buena alternativa a los métodos de gradiente descendiente para el entrenamiento de las redes neuronales que se ha propuesto, son los algoritmos

genéticos [4]. Esto es posible ya que si se conoce la topología de la red, la misma puede representarse como un string binario que pueda ser fácilmente utilizado por los operadores de cruza y mutación de un algoritmo genético (AG) [5]. Definiendo una función de fitness adecuada, cuando el algoritmo genético converge, se obtienen los parámetros de una red debidamente entrenada, codificados en la estructura de un cromosoma.

En este trabajo, se presenta un Framework denominado FACI que puede ser utilizado para el desarrollo de redes neuronales y el uso de algoritmos genéticos, de tal forma que se puede realizar el entrenamiento de una red neuronal mediante distintos métodos de entrenamiento. Se muestra como ejemplificación de su uso, una red neuronal que identifica números naturales de un dígito, la cual se la entrenó a través de BP y AG comparándose los resultados obtenidos.

Como trabajos similares, se puede citar a JCortex [1] y Joone [2] , que son frameworks que permiten, entre otras cosas, realizar el entrenamiento de una red neuronal utilizando los modelos de redes neuronales más conocidos o definiendo nuevos. El framework presentado en este trabajo, se diferencia de los mencionados anteriormente en que si bien permite crear y entrenar una red neuronal, también permite abordar soluciones de problemas de búsqueda y optimización con algoritmos genéticos independientemente de las redes neuronales. Asimismo, permite también trabajar con diferentes topologías de redes neuronales, y entrenarlas con distintas estrategias, como se mencionó previamente.

2 Descripción general del modelo

Específicamente este framework se desarrolló para ser usado como herramienta didáctica por los alumnos de quinto año de la carrera Ingeniería en Sistemas de Información, más específicamente para aquellos que se encuentran cursando la materia Inteligencia Computacional de la UTN-FRSF.

Un framework es un conjunto de clases que encapsula un diseño abstracto para dar soluciones a una familia de problemas relacionados [6] y como tal, brinda a los alumnos una herramienta con la cual desarrollar sus trabajos prácticos de manera rápida sin perder de vista los conceptos aprendidos en clase.

La figura 1 muestra la vista estática de la arquitectura, en particular la vista de módulos con un estilo de usos [7]. La misma refleja dos componentes centrales para la resolución de problemas de búsqueda y clasificación. El componente *ANN* representa el modelado conceptual para redes neuronales artificiales, mientras que el componente *GeneticSolution* permite modelar algoritmos genéticos. Dado que las redes neuronales deben ser entrenadas, el componente *Training* abarca los elementos que permiten entrenar y validar la red. Para realizar el entrenamiento existen diferentes métodos, el componente *TrainingMethod* engloba las diferentes estrategias a usarse. Dado que los algoritmos genéticos en este trabajo son usados como métodos de entrenamiento de redes neuronales artificiales,

[1] http://www.javahispano.org/contenidos/es/joone java object oriented neural engine /
[2] http://www.iit.upcomillas.es/pfc/resumenes/44a2672e19490.pdf

se establece la relación de uso entre los componentes *GeneticSolution* y *TrainingMethod*. Ya que tanto las redes neuronales artificiales como los algoritmos genéticos utilizan diferentes tipos de funciones tales como función identidad, sigmoidea, error cuadrático medio, entre otras, para completar el modelo se las ha representado con el componente *SupportFunction*.

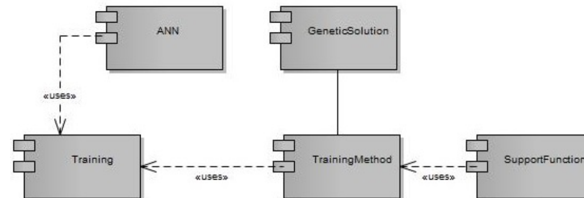

Fig. 1. Arquitectura de FACI

3 Descripción del Framework

En esta sección se comenta la estructura de los módulos del framework y su uso en el ejemplo planteado en secciones anteriores.

3.1 Componente ANN de la Arquitectura

En la figura 2 se observa la estructura de clases del componente *ANN* que se mostró en la sección 2.

En dicha figura, puede observarse que la herramienta propuesta provee de dos tipos de topología de redes neuronales, definidas como subclases de *ANNTopology*: *MapTopology* y *NetworkTopology* representando mapas autoorganizativos y redes neuronales respectivamente. A su vez, *NetworkTopology* tiene dos subclases: *MLPNetwork* y *RBFNetwork* representando redes MLP y RBF, respectivamente. Para permitir la flexibilidad en la implementación y uso de diferentes topologías, se utilizó el patrón Abstract Factory, donde la clase *NetworkTopology* representa la fábrica abstracta y *NetworkFactoryClient* el cliente de la misma.

Al crear una red, debe definirse la cantidad de capas, la cantidad de neuronas y la función de activación de éstas. Las capas se hallan en implementadas a través de la clase *Layer*, la cual es una generalización de los tipos de capas que forman parte de una red: *InputLayer*, *HiddenLayer* y *OutputLayer* representando respectivamente a las capas de entrada, oculta y salida. Una característica importante de la implementación de las capas en el framework es que cada tipo de capa tiene asociado una *función de entrada* a la capa, representada por la asociación entre la clase *Layer* y la interface *Function*.

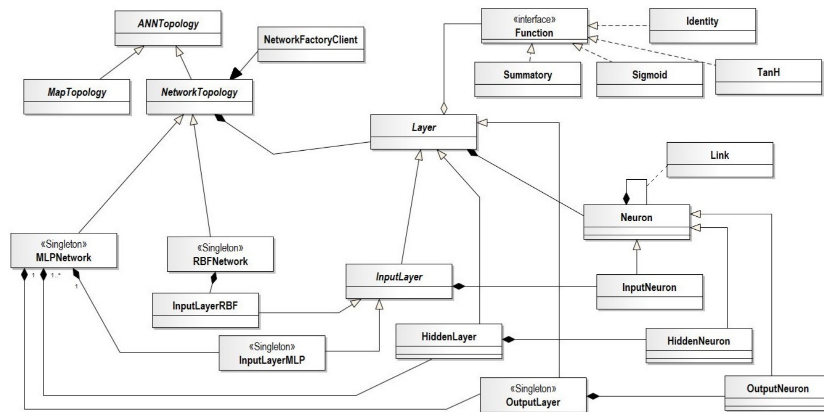

Fig. 2. Implementación de Red Neuronal

Las clases que representan cada tipo de capa componente de la red, no necesitan ser extendidas a menos que se desee algún comportamiento nuevo no previsto en sus implementaciones.

Cabe destacar que el framework también provee la implementación de ciertas funciones de activación y de entrada a las capas, como ser *Identity* (función Identidad), *Sigmoid* (función Sigmoide), *Summatory* (función Sumatoria) y *TanH* (función Tangente Hiperbólica). Si el usuario lo desea, puede crear otras funciones como implementaciones de la interface *Function*.

Las capas, asimismo, están compuestas por neuronas que funcionan de acuerdo al tipo de capa. En la implementación, esto se ve reflejado en las clases *InputNeuron*, *HiddenNeuron* y *OutputNeuron* que representan las neuronas de entrada, ocultas y de salidas respectivamente.

Las neuronas están relacionadas unas con otras a través de la sinapsis (uniones entre neuronas), dado que esta relación es una relación n a n, están representadas en el framework por la clase asociativa *Link* que modela la asociación de una neurona con las neuronas que se encuentran relacionadas a ésta en la capa anterior.

En este framework, cuando es instanciada la clase *MLPNetwork* se construye una estructura de red que esta formada por una capa de entrada, n capas ocultas y una capa de salida restringiendo de esta forma la estructura que una MLP puede adoptar. Para el ejemplo que estamos tratando, el valor de n se estableció en uno, ya que la solución propuesta es una red tipo MLP de tres capas.

Adicionalmente, el framework provee de una clase denominada *TestData*. La misma no es en sí parte de la estructura de una red neuronal, pero fue creada con el objetivo de poder representar los casos de entrenamiento y validación como un objeto, al cual se le pueda solicitar la información en forma clara a través de sus interfaces. Este objeto tiene dos atributos, x y t, que son el valor de entrada a la red y la salida esperada respectivamente. Estructurando de esta forma los datos, es más sencilla su manipulación.

3.2 Entrenamiento supervisado de la red

Dado que las redes neuronales para cumplir con su función deben ser entrenadas, en la arquitectura se ha incluído un componente llamado *Training*. La figura 3 muestra el diagrama de clases que describe este componente. El mismo cuenta con una clase denominada *Trainer* que es el contexto o cliente donde se aplica la estrategia concreta de entrenamiento de la red. Es un componente de soporte, es decir, que no es parte de la definición de una red neuronal, sino más bien presta un servicio a la misma. Esta clase cuenta con dos métodos abstractos: *training(int repeat, double splitPercent)* y *saveBestRepresentation()*. Éstos deberán ser implementados para realizar específicamente el entrenamiento y salvado de la red según el tipo de red con el que se esté tratando.

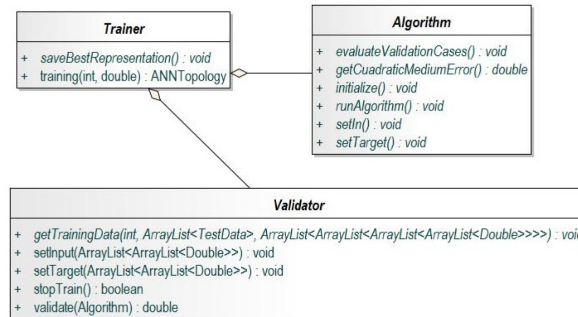

Fig. 3. Implementación de componente *Training*

El método *training* realiza la llamada al método *runAlgorithm()* de la clase *Algorithm* para que el entrenamiento se lleve a cabo.

La clase *Validator* provee la funcionalidad de validación de la red. Ésta es una clase abstracta, que permite que se puedan crear otros método de validación que se desee utilizar para validar la red resultante luego del entrenamiento. En el caso de la clase *Validator*, las subclases sólo deben implementar el método denominado *getTrainingData()*, cuya función es separar los datos que se reciben como parámetro de entrenamiento, en datos de entrenamiento y validación. Ésta división, se hace de acuerdo al criterio que utilice la subclase particular de validación que se ha definido para el problema.

La figura 4 muestra el diagrama de secuencia de entrenamiento de la red neuronal. Para este ejemplo, se define una subclase de la clase *Algorithm* llamada *BackPropagation* que implementa el método de entrenamiento back propagation más comúnmente usado en entrenamientos de redes neuronales del tipo MLP.

Cuando la instancia de *TrainingMLP* recibe el mensaje *training()*, invoca al método *runAlgorithm()* de la instancia de *BackPropagation* que representa el método de entrenamiento para la red. Luego, este último envía a la instancia de red neuronal *MLPNetwork* el mensaje *evaluate(inputCase)* para que el caso

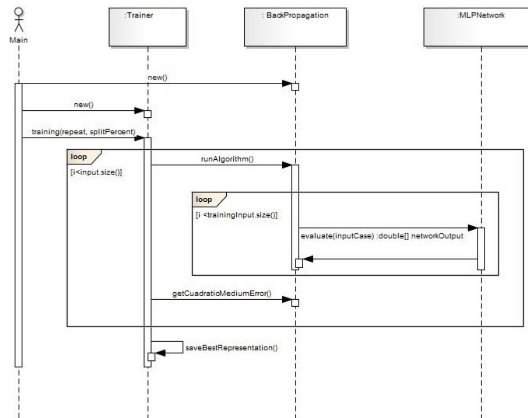

Fig. 4. Secuencia General de entrenamiento de la Red Neuronal

enviado como argumento *inputCase* sea evaluado por la red. Una vez obtenidas las salidas de la red, se comparan con los valores *target* para el caso que se está tratando y se actualiza el valor del error de la red. Este proceso se repite por tantos valores como haya para entrenamiento. Cuando el algoritmo ha finalizado, retorna la ejecución a la instancia de la clase *TrainerMLP*, quien solicita al objeto *:BackPropagation* el valor del error de la red a través del mensaje *getCuadraticMediumError()*. Finalmente, la red ya entrenada es persistida por el objeto instancia de *TrainerMLP* mediante la ejecución del método *saveBestRepresentation()*.

3.3 Ejemplo. Creación de una red MLP de tres capas: Entrada, oculta y de salida.

Este ejemplo consiste en el diseño de una red tipo MLP utilizada para reconocer desde un patrón de entrada un dígito del 0 al 9. Se ha definido que la misma sea entrenada utilizando el algoritmo Back Propagation. Las capas de la red utilizan las funciones de activación identidad en el caso de las capas de entrada y de salida y sigmoide en el caso de la capa oculta. Adicionalmente, dado que tanto la capa oculta como la capa de salida deben contar con una función que permita unificar las entradas, se utiliza en este ejemplo la función más utilizada en la práctica, que es la sumatoria. Ésta se encuentra también implementada por la clase *Summatory*.

Para poder entrenar la red, se debe hacer una extensión de la clase *Trainer* que implemente las responsabilidades necesarias. En este caso, se creó la clase *TrainerMLP*.

Para comenzar con la implementación, se crea la clase *Main* que representa la solución al problema. La misma debe crear la topología de la red y entrenarla. La figura 5 muestra el método *main()* definido en esta clase, encargado de generar la solución.

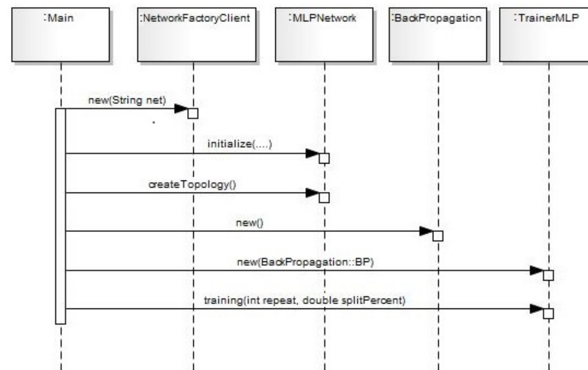

Fig. 5. Interacciones de la clase *Main*

La clase *Main* crea una instancia de la clase *NetworkFactoryClient* que se encargará de generar la instancia de la topología correspondiente, en este caso *MLPNetwork*.

Luego, se le envía al objeto de clase *MLPNetwork*, el mensaje *initialize(...)* cuyos parámetros identifican la cantidad de neuronas en cada capa, la cantidad de capas y las tres funciones que se utilizarán en las distintas capas.

Posteriormente, *Main* envía el mensaje *createTopology()* para que la topología sea creada (este método se muestra con más detalle en la figura 6). Si se deseara crear una red con más capas, sólo es necesario cambiar de tres a la cantidad deseada de capas en el parámetro que recibe éste método. Finalmente, se crean instancias de la clase *BackPropagation* y *TrainerMLP*, esta última recibe el mensaje *training* que utilizará la estrategia back propagation para entrenar la red creada. Es importante destacar que una vez que se crea la instancia de *BackPropagation*, ésta es enviada como argumento del método *new()* de la clase *TrainerMLP* de manera de asociar ambas clases.

En cuanto al método *createTopology()*, éste es provisto en la herramienta y su invocación genera las instancias de las capas y sus conexiones.

3.4 Componente GeneticSolution de la Arquitectura.

Dado que en la teoría de algoritmos genéticos nos podemos encontrar con dos tipos de poblaciones, de acuerdo a cómo se hacen los reemplazos dentro de ellas, en la implementación de la herramienta (representada en la figura 7), se han considerado dos tipos de poblaciones: generacional y estacional. Como se observa, esto aparece plasmado en la arquitectura a través del uso del patrón *Abstract Factory* donde la clase abstracta es la denominada *Population* y las clases concretas son las clases *GenerationalPopulation* y *StationalPopulation*. El cliente de Abstract Factory está representado por la clase *Genetic* que es la clase que efectivamente ejecuta el algoritmo. En este punto, es importante destacar que la clase *Genetic* tiene dos subclases, que se han definido para poder

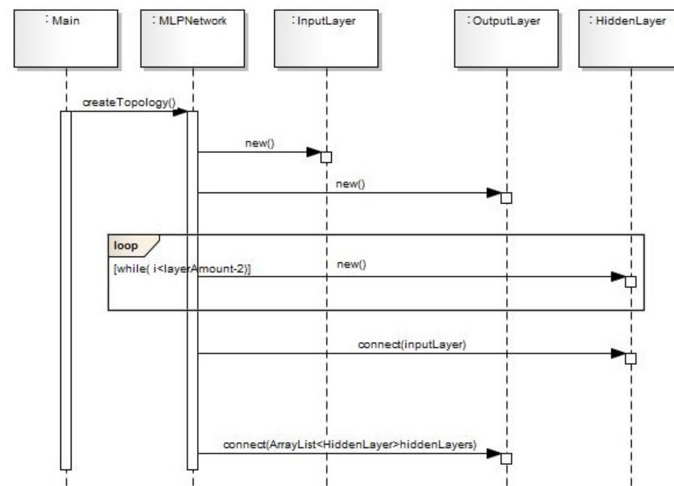

Fig. 6. Interacciones del método *createTopology()*

afrontar la clasificación de poblaciones antes mencionada: *GeneticStational* y *GeneticGenerational*.

Respecto de los tipos de reemplazo, para el tipo de población estacionaria, es necesario definir la forma en que los nuevos individuos se insertan en la población, y de la misma forma, el criterio por el cual se retiran individuos de ella para mantener siempre estable el tamaño poblacional. A raíz de esto, el framework provee estrategias de reemplazo de la siguiente manera: se ha creado la clase *Replacement* y sus subclases (*BothParentsReplacement*, *WeakParentReplacement* y *RandomReplacement*), donde *Replacement* es una clase abstracta, que cuenta con los siguientes métodos también abstractos: *makeReplacement(...)* y *getReplacementStrategy()*. El contexto de la estrategia de tipos de reemplazo es la clase concreta *StationalPopulation*. Las subclases de *Replacement* proveen la funcionalidad completa de la estrategia de reemplazo que representan.

La población está compuesta de individuos denominados Cromosomas, quienes representan una solución perteneciente al espacio de búsqueda del problema en cuestión. En la arquitectura, los cromosomas están representados por la clase abstracta *Chromosome*. Dado que cada individuo en la población es evaluado según una función de fitness, en la arquitectura se cuenta con la clase abstracta *GeneticFitnessFunction*, que es una implementación de la interfase *Function*. Ésta agrega a la especificación de *Function* un método específico de las funciones de fitness que son aplicables al modelo genético. Este método se denomina *compareBestFitness(Double evaluationValue,Double parameterValue)* y su función es evaluar cuál de los valores que ha recibido como parámetro es entendido como el mejor respecto al fitness que se esté evaluando.

Los cromosomas están compuestos por Genes (instancias de clase *Gene*), los que identifican una característica particular del individuo. Los valores de un gen

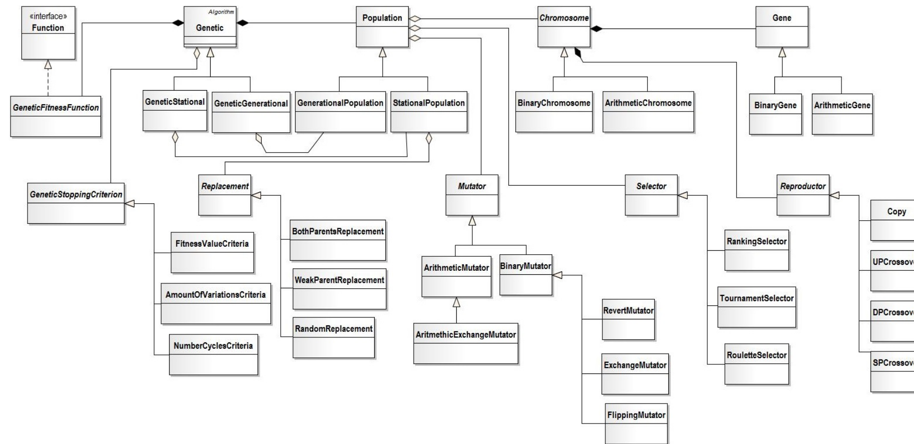

Fig. 7. Implementación de Algoritmo Genético

pueden representarse tanto como números (enteros, decimales o representaciones binarias), como por caracteres alfanuméricos.

Cada uno de los pasos comentados anteriormente (selección, cruza y mutación) involucra en sí mismo una estrategia, por lo cual se utilizó el patrón *Strategy* para definir cada una de ellas. Para definirlas, se cuenta con la implementación de las siguientes clases:

- Estrategia de selección: Está representada por la clase abstracta *Selector* y sus subclases: *RouletteSelector* (selección por Ruleta), *RankingSelector* (selección por Ranking) y *TournamentSelector* (selección por Torneo). Con sólo instanciar una de las subclases, el usuario dispone de la funcionalidad de selección.
- Estrategia de cruza: Representada por la clase abstracta *Reproductor* y sus derivadas. Éstas son las clases *SPCrossover* (cruza en un punto), *DPCrossover* (cruza en dos puntos), *UPCrossover* (cruza uniforme) y *Copy*. La clase *Reproductor* contiene un método abstracto denominado *getCrossing()*, que es la acción de cruza misma entre dos individuos. Como todo método abstracto, debe ser implementado en cada subclase de *Reproductor*. Igualmente que las subclases de *Selector*, a través de la implementación del método abstracto mencionado, las subclases de *Reproductor* proveen funcionalidad completa de cruza sin necesidad de extensiones.
- Estrategia de mutación: En este caso, la clase principal es la denominada *Mutator*. Al igual que las clases anteriores, es una clase abstracta, de la cual derivan los principales de operadores de mutación.

Otro aspecto a tener en cuenta en una solución basada en algoritmos genéticos es cuándo los individuos de una población representan soluciones válidas. En el framework se ha previsto el uso de las diferentes estrategias de parada del algoritmo, a través de la implementación de la clase abstracta *GeneticStoppingCri-*

terion. Se han definido además los siguientes criterios: Cantidad de generaciones transcurridas (clase *AmountOfGenerationsCriteria*), valor de fitness deseable alcanzado (clase *FitnessValueCriteria*) y cantidad de ciclos transcurridos (clase *NumberCyclesCriteria*).

3.5 Continuación del Ejemplo. Entrenamiento de red MLP de tres capas con Algoritmos Genéticos.

En esta sección se utilizará un algoritmo genético para representar la red neuronal presentada en la sección 3.3.

En la definición del algoritmo genético la determinación de la estructura del cromosoma es fundamental para el éxito del mismo. En la definición del cromosoma se debe determinar cuáles son los genes que forman parte de él y el alelo de cada uno. Dado que el cromosoma que se debe representar codifica los pesos de una red neuronal del tipo que se muestra en la Figura 8, se propone un cromosoma que contenga $x * h + h * y$ genes, donde cada gen representa un valor sináptico de una de las matrices $W1$ y $W2$, como se muestra gráficamente en la Figura 9.

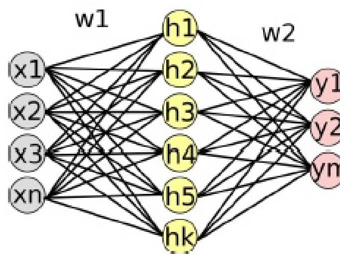

Fig. 8. Red Neuronal utilizada en el modelo: MLP de 3 capas

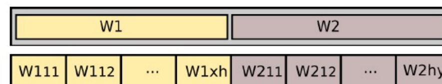

Fig. 9. Cromosoma de Algoritmo Genético para entrenamiento de una red neuronal MLP

El siguiente paso es definir la función de aptitud del cromosoma teniendo en cuenta que nuestro objetivo es minimizar el error de la salida de la red. Por lo tanto, una solución posible es evaluar la red neuronal y calcular el error cuadrático medio:

$$fitness = f(y)\frac{1}{2}(t-y)^2, \tag{1}$$

donde y es el valor obtenido de evaluar la red neuronal y t es el valor objetivo o deseado.

Para entrenar esta red utilizando el framework propuesto, se procede la siguiente forma:

Se crea una clase *Main* que representa la solución al entrenamiento de la red neuronal, de acuerdo a los parámetros del algoritmo que se han seleccionado. El método *main()* de la clase *Main* se define para instanciar las clases que son necesarias utilizar en el algoritmo genético. En este caso: el método de selección (*WindowSelector*), método de cruza (*DPCrossover*), método de mutación (*FlippingMutator*), función de fitness (*CuadraticMediumErrorFunction*), criterio de parada (*FitnessValueCriteria*) y estrategia de reemplazo (*WeakParentReplacement*). Seguido, se crea también la instancia de *GeneticStational* y se le envía el mensaje *runAlgorithm()*, el cual ejecuta la secuencia de pasos de un algoritmo genético. La figura 10 muestra la interacción entre las clases cuando se crea la instancia de *GeneticStational*. Como se observa, una vez que la instancia es creada, invoca al constructor de su superclase, el cual crea la población y origina la primera generación con la cual se comienza la ejecución del algoritmo genético.

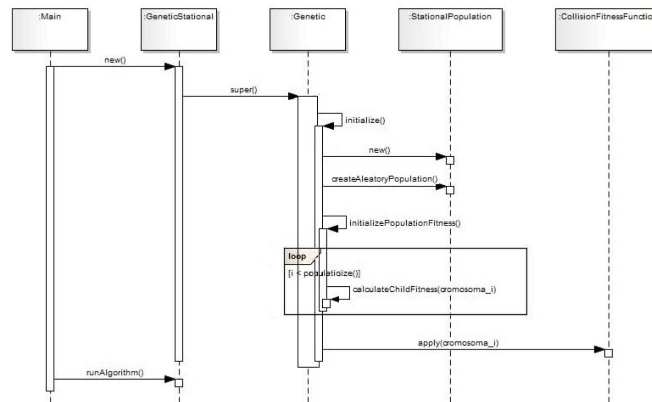

Fig. 10. Secuencia de creación de componentes para el ejemplo de entrenamiento de red neuronal MLP de 3 capas

La ejecución del algoritmo genético comienza cuando la clase *GeneticStational* recibe el mensaje *runAlgorithm()*. El mismo se ejecuta hasta que es alcanzado el criterio de parada. Para esto, se ejecuta un ciclo **while**, que implica en cada iteración el envío de un mensaje *reached()* a la clase *FitnessValueCriteria*.

4 Materiales y métodos

El framework tal como se mostró en los ejemplos de las secciones previas, fue utilizado para resolver el problema de detectar dado un patrón que codifica números enteros del 0 al 9, a qué dígito refiere el mismo. Para lograr resolverlo, se utiliza la estrategia de Redes Neuronales. En este trabajo red es entrenada utilizando dos métodos diferentes de entrenamiento. El de más amplio uso, Back Propagation y por otro lado, Algoritmos Genéticos.

Aquí se describen las métricas utilizadas para realizar las comparaciones respecto de cada algoritmo de entrenamiento utilizado y el formato de los datos que se utilizaron para entrenar y probar la red. También se muestra el criterio por el cual se determina qué red se considera la mejor entrenada.

4.1 Datos

Los datos utilizados para entrenar la red neuronal son dígitos del 0 al 9 representados cada uno por una matriz de 7 x 9, donde un valor 1 representa un *cuadro* de la grilla que está pintado y un valor 0 es un *cuadro* en blanco, como se observa en la figura 11. Además se incluyeron en el conjunto de datos de entrada, 9 versiones con distorsiones. Estas versiones, contienen un error de entre 1,6% y 4% (difieren entre 1 y 3 bits) respecto de la versión original.

El conjunto de datos está conformado por 100 valores de entrada y sus respectivas salidas. Estos se particionan en dos conjuntos, el primer 30% destinado a datos de prueba y el 70% restante a datos de entrenamiento.

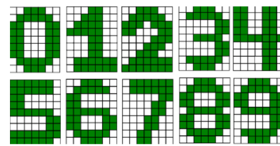

Fig. 11. Datos de entrada (sin ruido)

5 Resultados y Discusión

En esta sección se muestra cómo se obtuvo una configuración específica de la red que permite resolver el problema planteado utilizando los algoritmos BP y AG para entrenar la red. Además, se especifican los parámetros necesarios para definir concretamente estos algoritmos.

5.1 Implementación y configuración del algoritmo de Retropropagación del error (Back Propagation)

En el caso de BP, para encontrar una configuración satisfactoria, se modificaron varios aspectos de la red neuronal, entre ellos, la cantidad de neuronas en la capa

de salida, el número de neuronas de la capa oculta, la cantidad de iteraciones de los algoritmos e incluso el conjunto de datos de entrenamiento. De las diferentes configuraciones, se observó que una única red no es capaz de resolver el problema. Por este motivo, se definió un esquema eel que se utilizan 10 redes neuronales iguales, donde cada una se entrena para interpretar un dígito, es decir, una red para el 0, otra para el 1, etc. El resultado obtenido con este esquema es aceptable, ya que la exactitud de la red se aproxima a un 70 % sobre los datos de prueba.

Finalmente, la configuración de cada una de las 10 redes neuronales se fijó de la siguiente forma: capa de entrada, 63 neuronas; capa oculta, 12 neuronas; capa de salida, 1 neurona; función de activación: sigmoide; número máximo de iteraciones del algoritmo de BP: 200; error máximo admisible: 0.001 y constante del algoritmo de BP: 0.1.

5.2 Implementación y configuración del algoritmo genético

Los parámetros más importantes para el algoritmo genético son *población*, *bits*,*probabilidad de cruza* y *probabilidad de mutación*. Para encontrar una configuración correcta, se procedió de la misma forma que en el caso del algoritmo BP, es decir, se modificaron los parámetros correspondientes y se selecionó la combinación de éstos que mejores resultados arrojó. Un detalle importante a destacar es que, al igual que en el algoritmo anterior, se utilizó una red por cada posible dígito a reconocer, cada una configurada de la siguiente forma: 63 neuronas en la capa de entrada, 12 en la capa oculta y una única neurona en la capa de salida, con función de activación sigmoide.

Utilizando la configuración mencionada previamente, la red arrojó un error de entrenamiento próximo al 20% y una exactitud cercana al 100%. Además, se establecieron los siguientes parámetros: número máximo de iteraciones del algoritmo: 200; población inicial:100; cantidad de bits utilizados para representar cada peso sináptico: 18; probabilidad de cruza: 0.6 y probabilidad de mutación: 0.01.

6 Comparación de algoritmos.

6.1 Tiempo promedio de entrenamiento[ms].

Ésta métrica es obtenida con cinco corridas de los algoritmos de entrenamiento. Los resultados de la tabla 1 muestran que el tiempo de entrenamiento de Back Propagation es más corto que el del Algoritmo Genético.

6.2 Comparación de las salidas.

En la tabla 2 se evaluó la salida de la red entrenada para cada algoritmo y se vió que el algoritmo genético dio mejores resultados. Si bien el algoritmo genético es mucho más lento, se puede concluir que es de mayor utilidad para el entrenamiento de la red propuesta, ya que tuvo un porcentaje de aciertos mucho mayor que BP.

Tabla 1. Tiempo promedio de entrenamiento[ms]

Neurona	AG	BP
0	2257.4	6.04
1	2147.28	1.66
2	27.66	0.88
3	1959.18	4.2
4	2214.18	1.94
5	1675.76	0.9
6	2118.62	0.88
7	2321.08.4	13.32
8	1593.94	0.86
9	1593.94	0.86
Promedio de entrenamiento	18189.6	31.54

Tabla 2. Salida de los algoritmos

Red Neuronal	Acierto BP	Acierto AG
0	Si	Si
1	Si	Si
2	No	Si
3	Si	Si
4	Si	Si
5	No	Si
6	Si	Si
7	Si	Si
8	No	Si
9	No	Si
Exactitud	60%	100%

7 Conclusiones

En este trabajo se presentó un framework para el diseño e implementación de redes neuronales del tipo perceptrón multicapa utilizando distintos algoritmos de entrenamiento. En particular, se compararon los algoritmos genéticos y Back Propagation, teniendo en cuenta el tiempo de entrenamiento y exactitud de la red entrenada.

El primer criterio favoreció a Back Propagation ya que éste es considerablemente más rápido para realizar el entrenamiento. En cambio, la exactitud lograda por la red entrenada es un punto positivo para el Algoritmo Genético, ya que no sólo logra un mayor porcentaje de aciertos, sino que obtiene una mejor distinción entre las neuronas que se activan y las que no. Otra diferencia importante entre los algoritmos es que, en el problema planteado, el Algoritmo Genético resultó más simple de configurar para lograr la convergencia del mismo.

Finalmente, en base a los resultados obtenidos se puede afirmar que el Algoritmo Genético es apto para entrenar una red neuronal MLP. Además, en determinadas condiciones puede obtener mejores resultados que el algoritmo Back Propagation. Es de destacar, que esta conclusión es válida únicamente para este dominio y configuración de la red.

Por otro lado, el framework, al ser una solución parcial, permite a los alumnos probar diferentes estrategias con muy pocos cambios. En la experiencia de uso de esta herramienta en la cátedra de Inteligencia Computacional, los alumnos han destacado la facilidad para probar diferentes configuraciones como ser: cambios en la función de activación, el método de entrenamiento y cantidad de capas, ofreciendo así, la oportunidad de aprender a partir de la elaboración de pruebas.

References

1. Werbos, P.: The roots of the backpropagation: from ordered derivatives to neural networks and political forescasting. New York: John Wiley and Sons, inc (1993).
2. Rumelhart DE., McClelland JL. In: Parallel distributed processing: Explorations in the theory of cognition, vol. 1 Cambridge, Ma: MIT press (1986).
3. Curry B., Morgan, P.: Neural Networks: a need for caution. Omega, International Journal of Management Sciences, (1997) 25:123–133.
4. Gupta, J., Sexton, r.: Comparing backpropagation with a genetic algorithm for neural network training. The international journal of management science. (1999) 27: 679–684.
5. Stepniewski, S., Keane, A.: Topology design of feedforward neural networks by genetic algorithms. Parallel Problem Solving from Nature - PPSN IV; Lecture Notes in Computer Science 1141. Springer. (1996): 771–780.
6. Johnson, R.; Foote, B.: "Designing reusable classes". Journal of Object-Oriented Programming, 1(2), 22-35, (1988)
7. Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R.; Stafford, J.: "Documenting Software Architectures: Views and Beyond", Second Edition, Addison Wesley, (2011)
8. Barnij Schifitto, J.E.: Proyecto Final de Carrera "Framework Académico para Inteligencia Computacional". UTN FRSF, (2013)