Entrenamiento por refuerzo de redes neuronales mediante algoritmos genéticos

Autor:

Jorge Timón Morillo-Velarde
e-mail: timon.elviejo@gmail.com

Tutores del proyecto:

Rosa M. Pérez Utrero
e-mail: rosapere@unex.es

Juan A. Gómez Pulido
e-mail: jangomez@unex.es

14 de junio de 2012

Resumen

En este trabajo se estudia un método alternativo para el entrenamiento de redes neuronales con conexión hacia delante. Se utiliza un algoritmo genético para ajustar los pesos de la red neuronal. Se evalúa el uso de diferentes tipos de neuronas (con salida real o binaria) para comparar sus rendimientos utilizando diferentes implementaciones paralelas (para el coprocesador XMM y para la arquitectura CUDA). Se prueban variaciones de los operadores genéticos y se mide su efectividad en el entrenamiento. Se enfrenta el algoritmo a diferentes tipos de problemas de aprendizaje por refuerzo y se medita sobre la idoneidad del mismo para cada problema.

Palabras clave: Redes neuronales, algoritmos genéticos, redes neuronales evolutivas, aprendizaje por refuerzo, CUDA, entornos artificiales, juegos multi-agente.

Índice

1. Introducción

El aprendizaje automático es una rama de la inteligencia artificial que trata de construir sistemas informáticos que optimicen un criterio de rendimiento utilizando datos o experiencia previa. Dentro del aprendizaje automático, las técnicas se clasifican en base a su entrenamiento como supervisado o no supervisado. En el primero, al sistema se le suministran ejemplos de entradas con sus correspondientes salidas deseadas. En el entrenamiento no supervisado, no se tienen a priori ejemplos de cómo debería comportarse el sistema. El aprendizaje por refuerzo es un caso especial de aprendizaje supervisado en que la salida deseada exacta es desconocida. Se basa sólo en información sobre si la salida actual es correcta o no. Al agente que se quiere que aprenda se le provee información sobre lo bien o lo mal que está actuando. Esta señal de recompensa puede serle indicada bien cada vez que el agente actúa, bien al final de una prueba completa durante la que realiza varias acciones.

Los algoritmos que tienen su origen en la observación de la naturaleza se denominan bio-inspirados. Entre ellos se encuentran las redes neuronales y los algoritmos genéticos.

Las redes neuronales son modelos que intentan reproducir el comportamiento del cerebro humano [Hilera y Martínez, 1995]. Una red neuronal consiste en un conjunto de elementos de procesamiento, llamados neuronas, los cuales se conectan entre sí [Koehn, 1994]. Las conexiones entre las neuronas tienen pesos asociados cuyos valores determinarán el comportamiento de la red. Existen algoritmos para determinar el valor de los pesos de una red mediante un entrenamiento supervisado, cabe destacar el de retro-propagación del error.

Los algoritmos evolutivos, dentro de los cuales los algoritmos genéticos son los más conocidos, son una familia de modelos computacionales inspirados en la evolución y la supervivencia del más apto [Bäch, et. al., 1991; Ömer, 1995; Whitley, 2001]. Se utilizan fundamentalmente en la resolución de problemas de búsqueda y de optimización [Holland, 1975]. Buscan una solución del problema reproduciendo genéticamente una población de individuos a lo largo de una serie de generaciones [Koza, 1997]. El aprendizaje es formulado como un problema de optimización, en el que cada individuo de la población es una posible solución.

Dada una topología de red fija, el entrenamiento de una red neuronal puede ser visto como un proceso de optimización cuyo objetivo es encontrar un conjunto de pesos que minimice el error que produce la red sobre el conjunto de ejemplos en el entrenamiento supervisado, o que maximice la recompensa en el aprendizaje por refuerzo.

En nuestro caso, el agente utilizará una red neuronal para decidir sus acciones (salida de la red) a partir de los datos que pueda recoger de su entorno (entrada a la red). Se utiliza un algoritmo genético para decidir los pesos adecuados para la red, utilizando una población de agentes con redes diferentes. Se utiliza la recompensa acumulada por el individuo en una o varias pruebas para medir su adaptación al medio. Se utilizan estas medidas y la población actual para construir la siguiente generación. Cuando algún individuo demuestra estar lo suficientemente adaptado al medio para cumplir con las expectativas del entrenamiento (o cuando se supera un límite prefijado de iteraciones), éste finaliza.

En todo este proceso, El algoritmo que más se ejecuta es el que calcula la salida de la red a partir de la entrada. Este algoritmo se puede llamar varias veces por cada prueba. Además, las pruebas pueden repetirse varias veces por individuo y generación para disminuir el ruido producido por los factores aleatorios que pueda haber en las pruebas. Por ello, esta función se ha optimizado mediante su paralelización en dos arquitecturas ampliamente extendidas y económicas: el coprocesador XMM, presente en todos los PCs de reciente fabricación y la arquitectura CUDA[Nota al pie: CUDA forma parte de lo que se conoce como GPGPU, que consiste en utilizar unidades de procesamiento gráfico

(GPUs) para cálculos generales que no tienen por qué ser gráficos], compatible con la mayoría de tarjetas gráficas NVIDIA a partir de la serie 8000. Además, para incrementar aún más el rendimiento, se estudia la viabilidad de entrenar redes con una versión mínima de neurona con activación de tipo escalón y con pesos con tamaño de un byte (en la sección 4.1 se describe con más detalle). Llamaremos a las redes que utilizan estas estructuras redes discretas.

Los problemas en los que aplicamos nuestro desarrollo, aunque no son realistas, se consideran interesantes para la robótica y/o para la inteligencia artificial. Los tipos de problemas tienen que ver con la clasificación, los juegos considerados como deportes mentales, el control automático, la toma de decisiones en tiempo real y la colaboración de múltiples agentes en tiempo real; pudiendo estar relacionados estos últimos con la vida artificial. En la mayoría de los problemas, el agente se enfrenta a pruebas genéricas que pueden tener factores aleatorios y/o agentes directamente programados. En algunos problemas, sin embargo, los agentes de la población pueden enfrentarse entre ellos para obtener una valoración.

En el presente documento (que contiene la documentación asociada al proyecto fin de carrera titulado «Entrenamiento por refuerzo de redes neuronales mediante algoritmos genéticos», desarrollado por Jorge Timón Morillo-Velarde para la consecución del título de ingeniería informática), primero comentaremos los fundamentos teóricos precisos para su comprensión y su correcta ubicación dentro del dominio de la neuro-evolución, diferenciándolo de otros desarrollos en éste área. Después, justificaremos las principales decisiones tomadas durante el desarrollo y explicaremos en detalle algunas partes de su implementación. En el capítulo 5, se describen los experimentos realizados y se justifica la elección de los mismos. El capítulo de resultados 6 expone los datos empíricos recogidos en los experimentos y razona unas someras conclusiones que luego se completan en el capítulo 7.

2. Base teórica

Tanto las redes neuronales como los algoritmos genéticos están inspirados en la naturaleza y han sido utilizados desde largo tiempo atrás. Las redes neuronales están inspiradas en el funcionamiento del cerebro, como un sistema de procesamiento de información distribuido. Por su parte, los algoritmos genéticos se basan en la teoría de la evolución de Darwin, por la que la evolución se produce a través de dos principios básicos: los individuos que no se adaptan suficientemente al medio perecen, mientras que los que sí lo hacen transmiten sus genes con cierta variabilidad. Esta variabilidad puede proceder de dos fuentes: la cruza de genes entre individuos ó la mutación directa de genes.

La combinación de estas dos técnicas es algo relativamente reciente. Algunos - principalmente anglosajones - han coincidido en llamar a esta síntesis neuro-evolución, otros se refieren a ella como evolución de redes neuronales artificiales, pero - en general - la mayoría de los que la usan recurren a nombres que definen con más precisión la técnica concreta que utilizan; dadas las múltiples posibilidades para combinar ambos métodos. Aceptaremos el término neuro-evolución para el resto del texto, aunque sin renunciar a los otros términos que hayan podido ser utilizados como redes neuronales evolutivas.

A continuación explicaremos más detalladamente las bases teóricas de las tres técnicas: redes neuronales, algoritmos genéticos y neuro-evolución. Nos centraremos principalmente en los algoritmos y estructuras que más se asemejan a los implementamos en nuestra librería.

2.1. Redes neuronales

Las redes neuronales constan de un conjunto de elementos de procesamiento - conocidos como nodos o neuronas - interconectados entre sí. Pueden ser descritas mediante un grafo dirigido en el que cada neurona i usa una función de activación de la forma:

$$y_i = f_i(\sum_{j=1}^{n}(w_{ij} \cdot x_j - \theta_i)). \tag{1}$$

donde y_i es la salida de la neurona i, x_j es la entrada número j a la misma, w_{ij} es el peso de la conexión entre los nodos i y j, θ_i es el umbral de activación (o Bias) y f_i es una función que puede ser, o no, lineal.

Las redes neuronales artificiales pueden clasificarse como *feed-forward* (con propagación hacia delante) o recurrentes dependiendo de su conectividad. Una red es *feed-forward* (figura 2.1) si existe un método de numeración de las neuronas que cumpla que no existan conexiones desde un nodo hacia otro nodo con un número más pequeño que el de nodo de origen. Una red es recurrente (figura 2.2) si no existe un método de numeración que cumpla tal condición. Para simplificar nuestro trabajo, nos centraremos en las redes *feed-forward*.

El aprendizaje de las redes neuronales se consigue habitualmente usando ejemplos: suelen tener un entrenamiento supervisado. Se basa en la comparación directa entre la salida de la red y la salida correcta o deseada. Normalmente se formula el entrenamiento como la minimización de una función de error como el sumatorio del cuadrado del error de la salida respecto de la salida deseada para todos los datos disponibles (que constan de pares de entradas con sus correspondientes salidas deseadas). Un algoritmo de optimización basado en el descenso del gradiente como la regla delta generalizada (también conocido como algoritmo backpropagation) puede ser usado después iterativamente para ajustar los pesos y así minimizar el error.

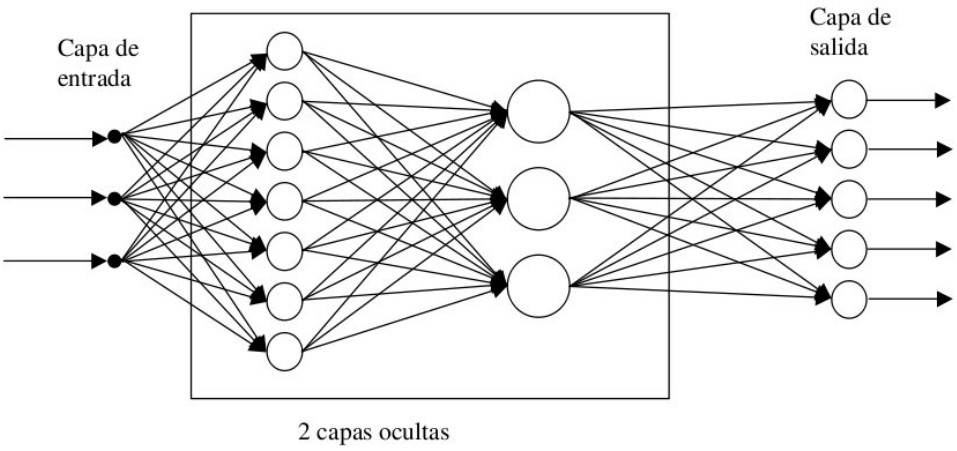

Figura 2.1: Red neuronal *feed-forward*.

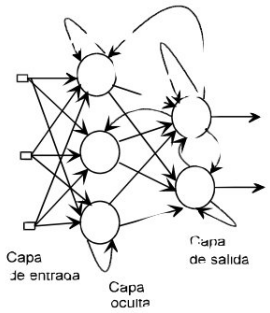

Figura 2.2: Red neuronal recurrente.

En nuestro caso, utilizamos aprendizaje por refuerzo y no necesitamos una colección de ejemplos (aunque se podría utilizar para calcular el refuerzo). Los pesos los ajustará un algoritmo genético. La estructura de la red se definirá de forma previa para cada problema y sólo evolucionarán los pesos (y umbrales).

2.2. Algoritmos genéticos

Los algoritmos genéticos son métodos sistemáticos para la resolución de problemas de búsqueda y optimización que aplican a éstos los principios de la evolución biológica: selección basada en la población, reproducción sexual y mutación.

Los algoritmos genéticos son métodos de optimización, que tratan de resolver el conjunto de problemas formulados como: hallar (xi,. . . ,xn) tales que F(xi,. . . ,xn) sea máximo. En un algoritmo genético, tras parametrizar el problema en una serie de variables (xi,. . . ,xn), se codifican en un cromosoma. Todos los operadores utilizados por un algoritmo genético se aplicarán sobre estos cromosomas, o sobre poblaciones de ellos. En el algoritmo genético va implícito el método para resolver el problema; son sólo parámetros de tal método los que están codificados - a diferencia de otros algoritmos evolutivos como la programación genética. Hay que tener en cuenta que un algoritmo genético es independiente del problema, lo cual lo hace un algoritmo robusto, por ser útil

para cualquier problema, pero a la vez débil, pues no está especializado en ninguno.

Las soluciones codificadas en un cromosoma compiten para ver cuál constituye la mejor solución (aunque no necesariamente la mejor de todas las soluciones posibles). El ambiente, constituido por las otras camaradas soluciones, ejercerá una presión selectiva sobre la población, de forma que sólo los mejor adaptados (aquellos que resuelvan mejor el problema) sobrevivan o leguen su material genético a las siguientes generaciones, igual que en la evolución de las especies. La diversidad genética se introduce mediante mutaciones y reproducción sexual. En la Naturaleza lo único que hay que optimizar es la supervivencia, y eso significa a su vez maximizar diversos factores y minimizar otros. Un algoritmo genético, sin embargo, se usará para optimizar habitualmente para optimizar sólo una función, no diversas funciones relacionadas entre sí simultáneamente. Este tipo de optimización, denominada optimización multimodal, también se suele abordar con un algoritmo genético especializado.

Por lo tanto, un algoritmo genético consiste en lo siguiente: hallar de qué parámetros depende el problema, codificarlos en un cromosoma, y se aplican los métodos de la evolución: selección y reproducción sexual con intercambio de información y alteraciones que generan diversidad. En las siguientes secciones se verán cada uno de los aspectos de un algoritmo genético.

Mediante los operadores de selección, se eligen los individuos que serán progenitores de la siguiente generación (o directamente formarán parte de ella). Con los operadores de cruza, se generan nuevos individuos mezclando los cromosomas de varios individuos (normalmente, dos). Por último, los operadores de mutación añaden cambios aleatorios a los individuos. La función de fitness nos da una aproximación de la adaptación del individuo al medio y ésta es utilizada por los operadores de selección.

En nuestro caso, el cromosoma de cada individuo lo forman los pesos de la red que utiliza ese individuo. Para calcular el fitness del individuo, se construirá la red con los pesos del cromosoma y se realizarán varias pruebas (para reducir el ruido generado por los posibles factores aleatorios de éstas) sobre el individuo, sumando las recompensas de todas y obteniendo el citado fitness.

2.3. Neuro-evolución

La evolución se ha aplicado las redes neuronales artificiales en tres niveles muy diferentes: a los pesos de las conexiones, la arquitectura de la red y a las reglas de aprendizaje. La evolución de los pesos de las conexiones introduce una aproximación global y adaptable al entrenamiento, especialmente para el aprendizaje por refuerzo o para el entrenamiento de redes recursivas, donde los métodos basados en el gradiente experimentan grandes dificultades. La evolución de las arquitecturas permite a las redes neuronales adaptar su topología a diferentes problemas sin intervención humana y con esto se consigue un diseño automático de redes neuronales, dado que tanto la arquitectura como los pesos pueden ser evolucionados. La evolución de las reglas de aprendizaje puede ser considerada como un proceso de "aprender a aprender"en redes neuronales donde la adaptación de las reglas de aprendizaje se consigue mediante la evolución. También puede ser contemplada como un proceso de descubrimiento automático de nuevas reglas de aprendizaje. Nos centraremos en la evolución de los pesos de las conexiones, por ser la evolución que utilizaremos.

La evolución de los pesos de las conexiones se puede realizar en el aprendizaje supervisado (con ejemplos) definiendo la función de fitness como el error global obtenido por la red (invirtiendo el signo), comparando las salidas de la red y la salida deseada para cada ejemplo. También puede utilizar para el aprendizaje por refuerzo definiendo una función de fitness distinta.

En general, los pasos a seguir son dos: decidir la codificación de los pesos de las conexiones (si

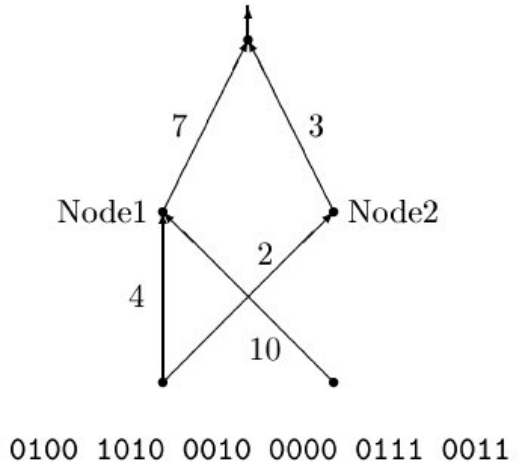

Figura 2.3: Red neuronal y su codificación binaria (asumiendo que se usan 4 bits para representar cada número real).

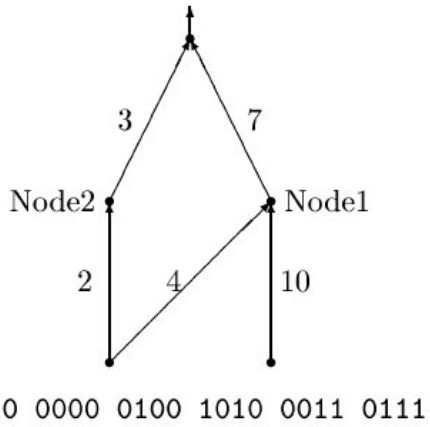

Figura 2.4: Red equivalente con codificación alternativa.

se hará mediante cadenas binarias o no) y la ejecución del algoritmo genético propiamente dicho. Para el primer paso, las opciones más extendidas son la representación binaria y la representación con números reales.

El algoritmo genético canónico siempre usa cadenas de bits para codificar las diferentes soluciones. Por ello, algunos trabajos tempranos de evolución de los pesos de las conexiones siguen esta aproximación [1, Yao99]. Las ventajas son la fácil aplicación de los operadores genéticos y su posible implementación digital. Habría que elegir la representación de los números reales. Aquí hay un compromiso para la precisión con que se quieran representar los números reales. Si se usan muy pocos bits para representar cada conexión, el entrenamiento puede fallar porque algunas combinaciones de pesos no se pueden aproximar con suficiente precisión por valores discretos. Por otra parte, si se usan demasiados bits, los cromosomas que representen a redes neuronales grandes se volverán demasiado largos y la evolución en proceso resultará muy ineficiente.

Por su parte, en la representación con números reales, los cromosomas se codifican como vectores de números reales con tantos elementos como conexiones. Los operadores genéticos no se pueden aplicar directamente sobre los bits y han de ser diseñados de nuevo. Esto puede ser una ventaja, pues, por ejemplo, el operador de mutación podría tener una distribución gaussiana (u otra función) en lugar de mutar un bit cualquiera sin tener en cuenta su peso en la construcción del número.

Uno de los problemas a los que se enfrenta la evolución de redes neuronales es el problema de la permutación. Es causado por el mapeado "muchos-a-uno" desde la representación en el cromosoma a la red que es construida. Con dos cromosomas distintos se pueden generar redes equivalentes como se muestra en las figuras 2.3 y 2.4. Se puede solucionar dando más importancia al operador de mutación que al de cruza (que es el que sufre con este problema) o con otros métodos matemáticos [2, Gomez, Miikkulainen 2003].

3. Análisis del problema

3.1. Fortalezas y deficiencias

Tanto las redes neuronales como los algoritmos genéticos tienen fortalezas que nuestro método aprovecha y debilidades que se pueden, en parte, minimizar por la combinación de ambos métodos.

3.1.1. Redes neuronales

Las redes neuronales con conexión hacia delante en general son un importante método de aproximación de funciones [Kim, 1992]. El perceptrón multicapa es un tipo de red neuronal con conexiones hacia delante. La topología de un perceptrón multicapa esta definida por un conjunto de capas ocultas, una capa de entrada y una de salida. No existen restricciones sobre la función de activación aunque en general se suelen utilizar funciones sigmoideas. Existen demostraciones teóricas [Funahashi, 1989] de que un perceptrón multicapa cuya función de activación sea no constante, acotada y monótona creciente es un aproximador universal de funciones. En [Hornik et alt, 1989] se llega a un resultado similar utilizando funciones de activación sigmoideas, no necesariamente continuas. Esto es un punto muy fuerte de las redes neuronales.

Además, constituyen buena una herramienta para la construcción de agentes pues sólo hay que codificar las entradas y las salidas de la red como las del agente y el tiempo de ejecución de la red sólo depende de la topología de ésta (para una topología dada, es constante).

Algunas deficiencias del algoritmo back-propagation son su baja adaptabilidad, la alta dependencia de los parámetros del algoritmo, el estancamiento en mínimos locales, la posibilidad de parálisis y la alta dependencia de las condiciones iniciales.

- **Adaptabilidad :** El algoritmo tiene como premisa la utilización de una función de activación derivable [Walker, 1995]. Al hacer uso de la derivada de la función de activación, es condición necesaria para la aplicación del algoritmo que la misma sea continua y derivable en todo el dominio de aplicación [Wilson, 1994]. Esto impide la utilización del método en otras topologías donde la función de activación presenta discontinuidades.

  Este problema suele encontrarse en varios métodos de entrenamiento, los cuales son desarrollados para una determinada topología y sus resultados, en general, no son extensibles directamente a otras topologías. Es necesario adaptar los métodos para aplicarlos a otras topologías.

- **Dependencia de parámetros del algoritmo :** Los algoritmos de gradiente descendente hacen uso de una tasa de aprendizaje que idealmente debería ser infinitesimal. De esta manera, mediante pequeños ajustes de los pesos sinápticos el algoritmo converge hacia un mínimo. El uso de tasas de aprendizaje muy pequeñas hace que el algoritmo tenga una convergencia estable hacia un mínimo, aunque el tiempo necesario para alcanzarlo puede llegar a ser muy alto. Como consecuencia de lo dicho anteriormente, y con el objetivo de disminuir el tiempo de convergencia del algoritmo, en la práctica se suelen utilizar tasas de aprendizajes mayores a las teóricas. El aumento de la tasa de aprendizaje disminuye el tiempo de convergencia, pero tiene un efecto contraproducente: el algoritmo comienza a oscilar en torno a un mínimo, disminuyendo la probabilidad de alcanzarlo. El efecto de oscilación puede reducirse mediante la adición de una tasa de momento, como se describió en el capítulo 3, pero no puede eliminarse.

  El algoritmo backpropagation es muy dependiente de los parámetros mencionados previamente. Dependiendo de la selección de parámetros realizadas el resultado de la aplicación del

algoritmo será exitosa o no [Liu et alt, 2004]. Pequeñas variaciones sobre los parámetros del algoritmo pueden conducir a resultados diferentes. El principal problema es que no existe un método general que permita establecer el valor de estos parámetros [Branke, 1995]. Los parámetros que aseguran la convergencia para un determinado problema pueden no ser aplicables a otro problema. De esta manera, la selección de los parámetros del algoritmo se realiza en base a la experiencia del diseñador, y se realiza un refinamiento de los mismos mediante mecanismos de prueba y error. Esto produce un aumento en el tiempo total de diseño y entrenamiento de la red.

- **Mínimos locales :** La superficie que define la función de error E (ecuación 8) en base a los parámetros de la red neuronal es compleja y esta llena de valles y colinas. Debido a la utilización del gradiente para encontrar el mínimo de dicha función de error se corre el riesgo de que el proceso de entrenamiento quede atrapado en un mínimo local [Sutton, 1986]. Esta situación no es deseable, fundamentalmente si dicho mínimo esta localizado lejos del mínimo global.

  Existen algunos mecanismos para evitar que esto suceda. Una posible solución para evitar que el entrenamiento quede atrapado en un mínimo local es aumentar el número de neuronas ocultas de la red. Este mecanismo puede ayudar en aquellos casos en los que la red tiene escaso poder de representación interna, y no es capaz de distinguir entre dos patrones diferentes, proporcionando una misma salida para ambos patrones. Al aumentar el número de neuronas ocultas la red posee mayor cantidad de parámetros libres y puede conseguir una mejor representación interna.

  Otros mecanismos que ayudan a disminuir los efectos de este problema son la adición de una tasa de momento al proceso de entrenamiento, utilizar una tasa de aprendizaje decreciente a lo largo del proceso, partir de otras configuraciones iniciales de la red, añadir ruido al método de gradiente, etc.

- **Parálisis :** El fenómeno de parálisis, también conocido como saturación, se produce cuando la entrada total a una neurona de la red toma valores muy altos, ya sean positivos o negativos. Al utilizar funciones de activación sigmoidales, la función de activación posee dos asíntotas horizontales. Si la entrada de la neurona alcanza un valor alto, la función de activación se satura y alcanza un valor de activación máximo o mínimo.

  Cuando la función de activación se satura su derivada tiende a hacerse nula, haciendo que los parámetros de la red permanezcan invariables y, como consecuencia, la suma de los errores locales permanece constante por un largo periodo de tiempo [Kröse y van der Smagt, 1993]. Aunque esta situación se suele confundir con un mínimo local, pues el error permanece invariable, en este caso es posible que después de un cierto tiempo el error comience nuevamente a decrecer.

  El fenómeno de parálisis del perceptrón multicapa ocurre fundamentalmente cuando los parámetros de la red toman valores muy altos. Un mecanismo para evitar esto consiste en partir de valores iniciales bajos.

- **Condiciones iniciales :** El conjunto de pesos iniciales de la red neuronal generalmente se selecciona de manera aleatoria. Sin embargo, el algoritmo backpropagation es muy dependiente de las condiciones iniciales seleccionadas [Kolen, 1991]. Pequeñas variaciones realizadas sobre las condiciones iniciales pueden llevar a grandes diferencias en el tiempo de convergencia del algoritmo.

A esto hay que añadir que los algoritmos de gradiente requieren entrenamiento supervisado (normalmente, no funcionan para el aprendizaje por refuerzo) y que las conexiones sean hacia delante (la retro-propagación del error no se puede aplicar en redes recurrentes).

Usando un algoritmo genético como método de entrenamiento de la red, se solucionan algunos de estos problemas y otros se mitigan en cierto grado. Con el algoritmo genético, se puede usar el aprendizaje por refuerzo y se pueden entrenar redes recurrentes sin problema. No se tienen requerimientos para la función de activación, por lo que aumenta su adaptabilidad. Se cambia la dependencia de los parámetros de ese algoritmo y ahora depende de los parámetros del algoritmo genético, estos parámetros son más flexibles y se pueden alterar en medio del entrenamiento. El algoritmo genético es mucho menos tendente a estancarse en mínimos locales porque no utiliza la información del gradiente y porque explora varios puntos (tantos como individuos tenga la población) del espacio de búsqueda simultáneamente. El fenómeno de saturación se produce cuando una neurona alcanza un máximo o un mínimo. En este caso, la derivada de la función de activación se hace nula, y los pesos de la red permanecen invariables. Como el método propuesto no hace uso de la derivada de la función de activación, el efecto de este fenómeno es completamente eliminado. Los valores iniciales de los pesos también pueden afectar al algoritmo genético, en especial si son muy altos (ya sean positivos o negativos), pero existen experimentos que permiten afirmar que el método propuesto es menos dependiente de los valores iniciales que el algoritmo backpropagation [3, Bertona2005].

3.1.2. Algoritmos genéticos

Un algoritmo genético es independiente del problema, lo cual lo hace un algoritmo robusto, por ser útil para cualquier problema, pero a la vez débil, pues no está especializado en ninguno. Hay que elegir la codificación de los cromosomas para cada caso concreto. Sin embargo, con nuestro método siempre códificaremos los cromosomas de manera similar (con una red neuronal) y sólo será necesario definir la función de fitness, elegir la topología de la red, codificar las entradas y las salidas. Aunque la codificación de éstas pueda admitir varias posibilidades (y algunas puedan ser más ventajosas que otras) la red debe aprender a interpretar las correctas relaciones entre entradas y salidas por sí misma.

3.2. Objetivos

En el presente proyecto se pretende construir una librería de programación en C++ para la utilización de redes neuronales con entrenamiento mediante algoritmos genéticos. Se quiere que sea lo más flexible posible en cuanto a la estructura de la red, para poder, en un futuro, determinar la topología también de forma genética. Por tanto, con la librería implementada debe ser posible crear una cualquier red con una arquitectura arbitraria. Deben ser posibles conexiones recurrentes y conectar capas con tipos de datos diferentes (por ejemplo, que una capa cuya salida son números en coma flotante debe poder usar como entrada una capa que tiene bits como salida).

Como los entrenamientos pueden ser costosos en tiempo de ejecución, la librería debe estar paralelizada internamente al menos para la ejecución de redes neuronales. Esta paralelización debe poder aprovecharse por los sistemas más extendidos para que pueda ser utilizada en proyectos que aprovechen la computación voluntaria.

Además, se probará la librería en casos concretos con el fin de contestar a las siguientes cuestiones:

1. ¿Qué ventajas en el rendimiento se pueden obtener gracias a la paralelización?

2. ¿Se puede simplificar la estructura de las redes neuronales para mejorar la paralelización? ¿Qué efecto tienen las funciones de tipo escalón (que permiten codificar la salida de cada neurona como un bit en vez de como un número real) tanto en el rendimiento como en el aprendizaje? ¿Qué efecto tiene la codificación de los pesos en estructuras discretas (en lugar de números reales) tanto en el rendimiento como en el aprendizaje?

3. ¿Qué operadores genéticos resultan más adecuados para el entrenamiento en diferentes problemas? ¿Qué valores de los parámetros del algoritmo genético resultan más adecuados para el entrenamiento en diferentes problemas?

4. ¿Para qué tipo de problemas resulta más adecuado el método propuesto?

4. Diseño e implementación

4.1. Implementaciones paralelas de las redes neuronales

4.1.1. Utilización del coprocesador XMM

4.1.2. Utilización de la arquitectura CUDA

4.2. Diseño del algoritmo genético

4.3. Interfaz del programador

5. Experimentación

5.1. Tareas de clasificación

5.2. Juegos de estrategia abstractos

6. Resultados

6.1. Rendimiento

En esta sección se analizan resultados de rendimiento en tiempo de ejecución.

6.1.1. Rendimiento de las implementaciones paralelas de red neuronal

En esta sección se presentan los resultados en tiempo de ejecución de los métodos que se ejecutan de forma diferente con cada implementación.

6.1.1.1. Acumulación de resultados En la figura 6.1, puede apreciarse que la implementación con instrucciones SSE2 de ensamblador para utilizar el coprocesador XMM es muy superior a la implementación de referencia en lenguaje C, especialmente para los tipos de Buffer BIT y SIGN.

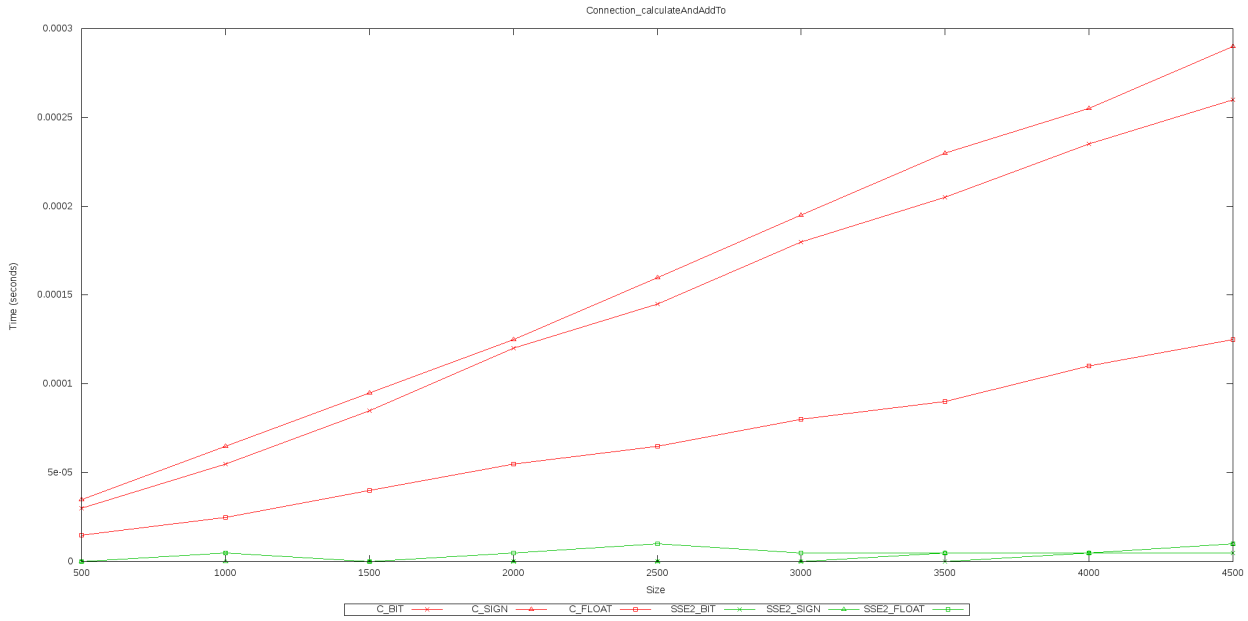

Figura 6.1: Método de acumulación de resultados (Connection::calculateAndAdd).

6.1.1.2. Activación De la figura 6.2 se extraen varias conclusiones. No mucha diferencia de rendimiento entre las implementaciones y prácticamente ninguna cuando las neuronas son de tipo FLOAT. Esto era de esperar porque ambas implementaciones son idénticas: no se han hecho optimizaciones SSE2 para la función de activación para el tipo FLOAT.

Para los tipos BIT y SIGN, la implementación en C es ligeramente más rápida que para el tipo FLOAT y la escrita en SSE2 es ligeremante más lenta que la FLOAT. En la implementación SSE2 de BIT y SIGN no se ponen los bits en el orden normal, si no en la representación especial de SSE2, para que el método de acumulación de resultados pueda ser óptimo.

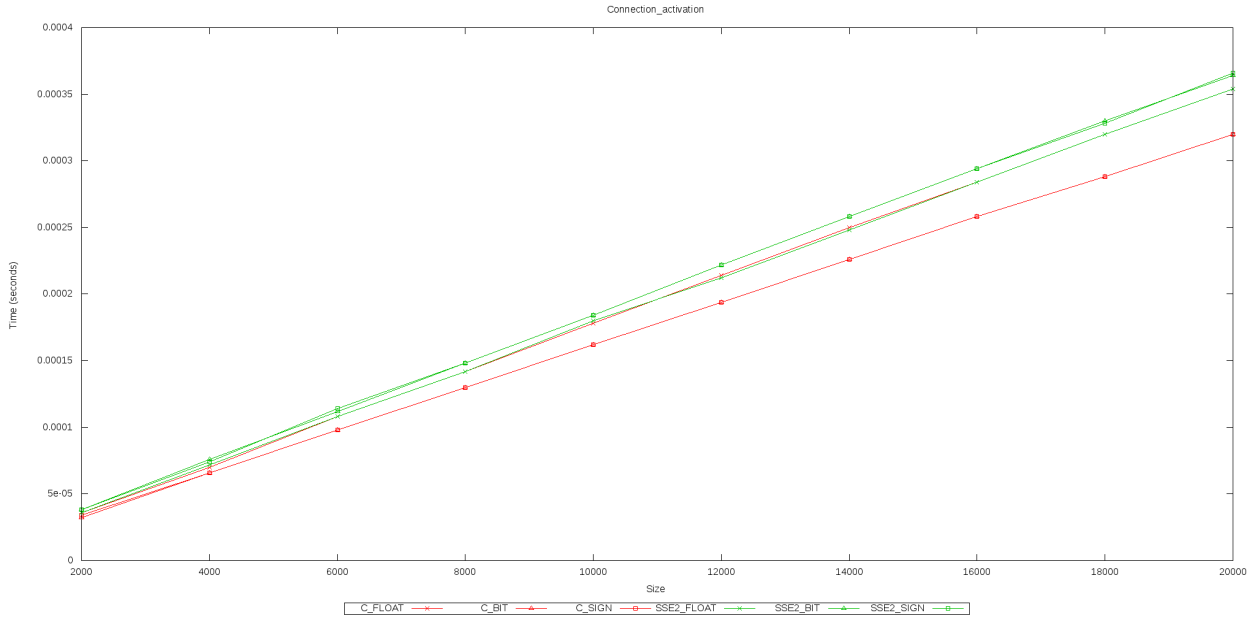

Figura 6.2: Método que ejecuta la función de activación para las salidas de una capa (Connection::activation).

6.1.1.3. Funciones de activación Si en la figura 6.2 se comparaba el rendimiento de los distintos tipos de neurona (BIT, SIGN y FLOAT con la función de activación IDENTITY), en la figura 6.3 se comparan los rendimientos de los distintos tipos de funciones para el tipo de neurona FLOAT. Con el tipo de neurona BIT, la función de activación siempre es $\text{BINARY}_{\text{STEP}}$ y con el tipo SIGN la activación siempre es $\text{BIPOLAR}_{\text{STEP}}$, por ello podemos excluir estos tipos de esta gráfica. Además, la implementación SSE2 de la activación para FLOAT es idéntica a la de C, por lo que tampoco se muestran los resultados de la activación SSE2 en este caso.

Se puede ver como, en general, la función más lenta es la sigmoide, seguida de la sigmoide bipolar. La tangente hiperbólica sólo es ligeramente más lenta que el resto, cuyo rendimiento es similar al de la función identidad (que no hace nada).

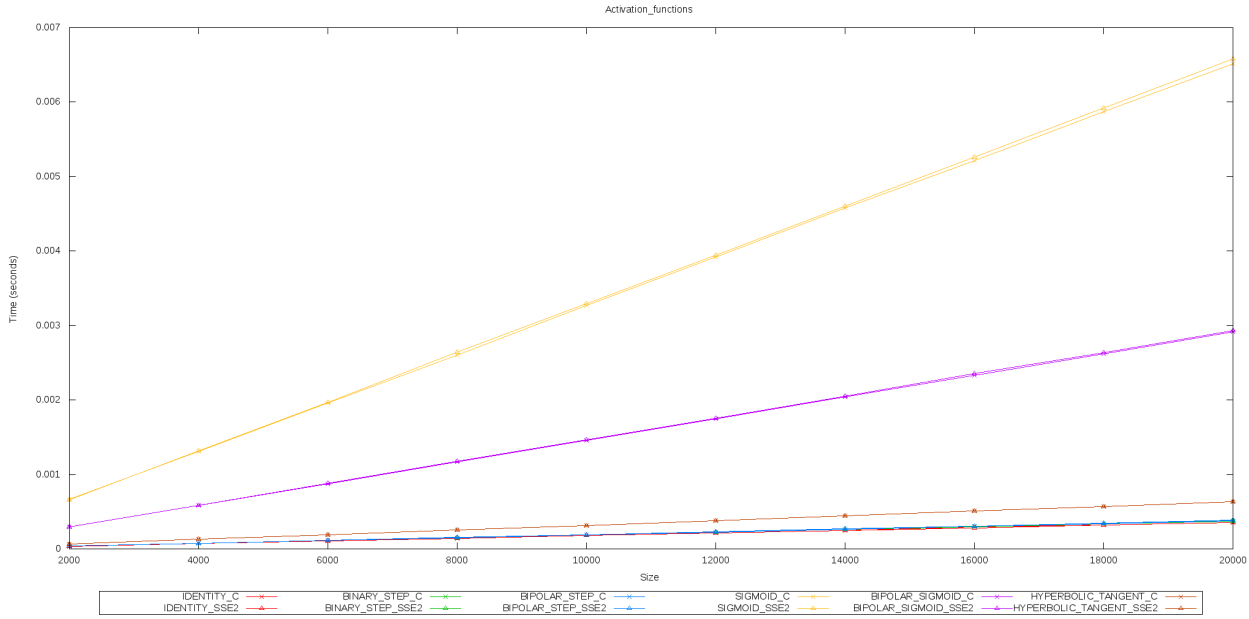

Figura 6.3: Método que ejecuta la función de activación para las salidas de una capa (Connection::activation) con tipo de Buffer FLOAT y las diferentes funciones de activación implementadas.

6.1.1.4. Cruza No hay una diferencia sustancial de rendimiento entre las diferentes implementaciones y tipos de neuronas para el operador genético de crossover, como se muestra en la figura 6.4. Como el método de activación, la cruza de SSE2 se ve ligeramente penalizada al traducir el vector de bits con los pesos a cruzar para que tenga se adapte a la posición especial de los pesos en SSE2.

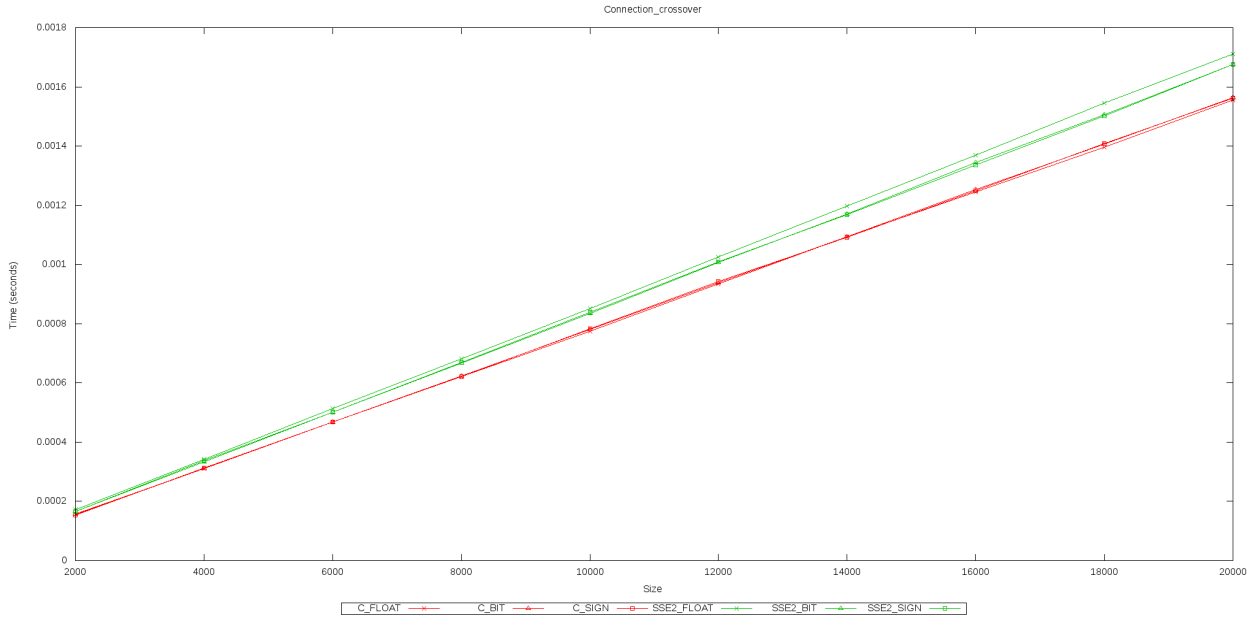

Figura 6.4: Método que implementa el operador de crossover para dos conexiones dadas (Connection::crossover).

6.1.1.5. Mutación De la figura 6.5 se puede deducir que el operador de mutación tiene un coste mínimo y constante respecto al tamaño de la conexión para todas las implementaciones.

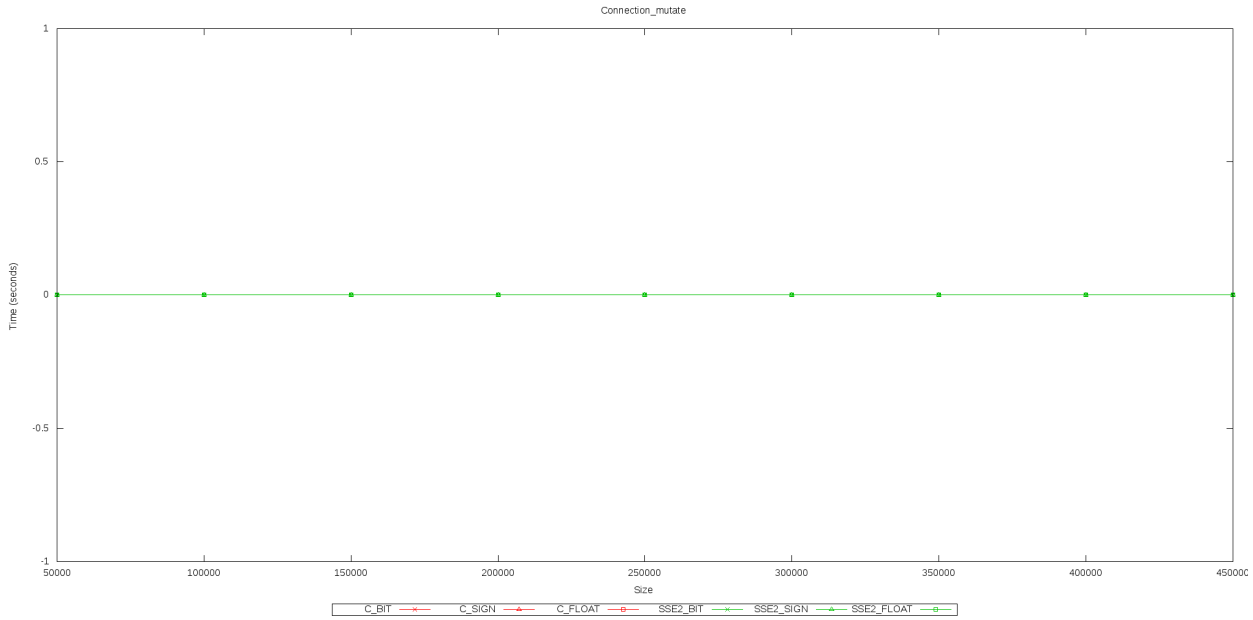

Figura 6.5: Método que implementa el operador genético de mutación (Connection::mutate).

6.1.1.6. Olvido De manera similar a la operación de mutación, el olvido (figura 6.6) tiene un coste constante y mínimo.

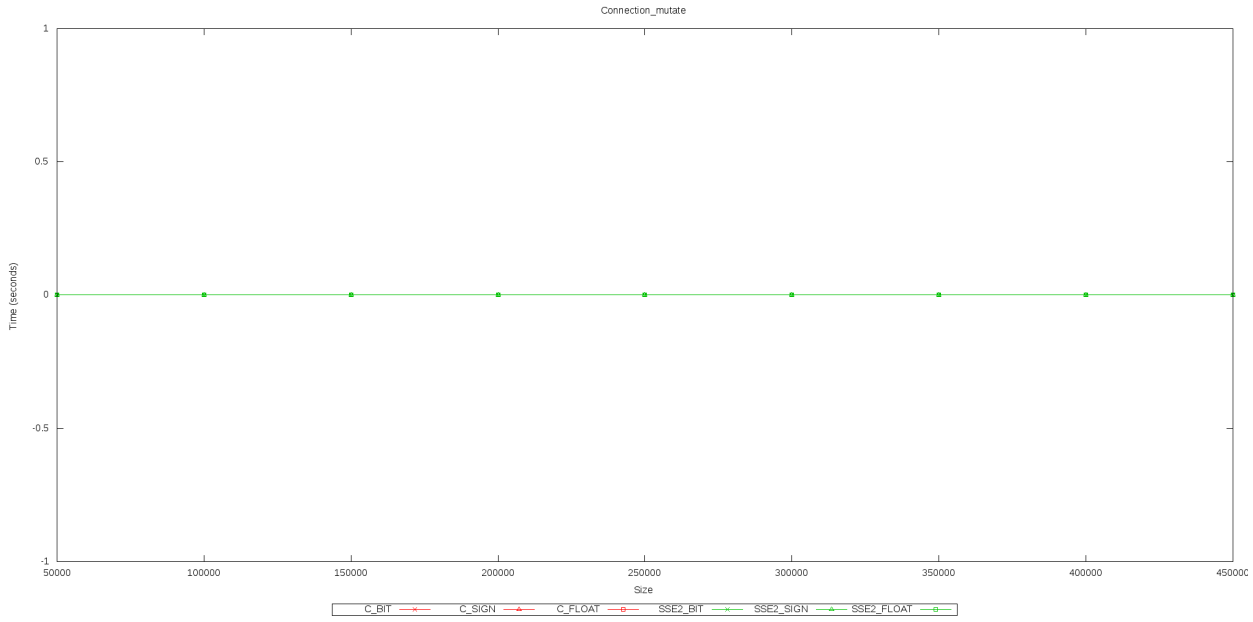

Figura 6.6: Método que implementa el operador genético de olvido (Connection::reset).

6.1.1.7. Copiar desde y hacia Interfaz Estos métodos se usan para recibir entradas y sacar salidas al exterior de la red. Las interfaces sirven para independizar la red de la implementación concreta escogida. Cada implementación debe mapear correctamente desde la representación genérica (Interfaz) hacia su propia representación interna (figura 6.7) de los datos y viversa (figura 6.8).

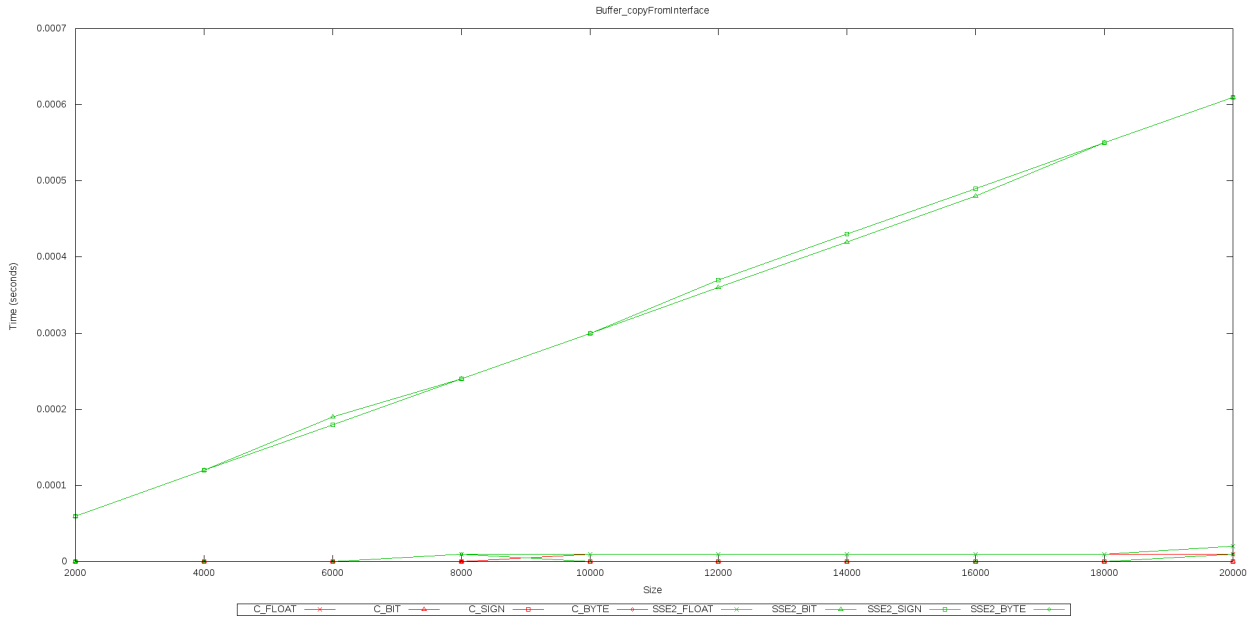

Figura 6.7: Método que implementa la copia a un Buffer con una implementación determinada desde un Buffer genérico de la clase Interfaz (Buffer::copyFromInterface).

En ambos casos se observa que el coste es muy pequeño excepto para la implementación SSE2 de los tipos BIT y SIGN, por razones similares a las de Activación.

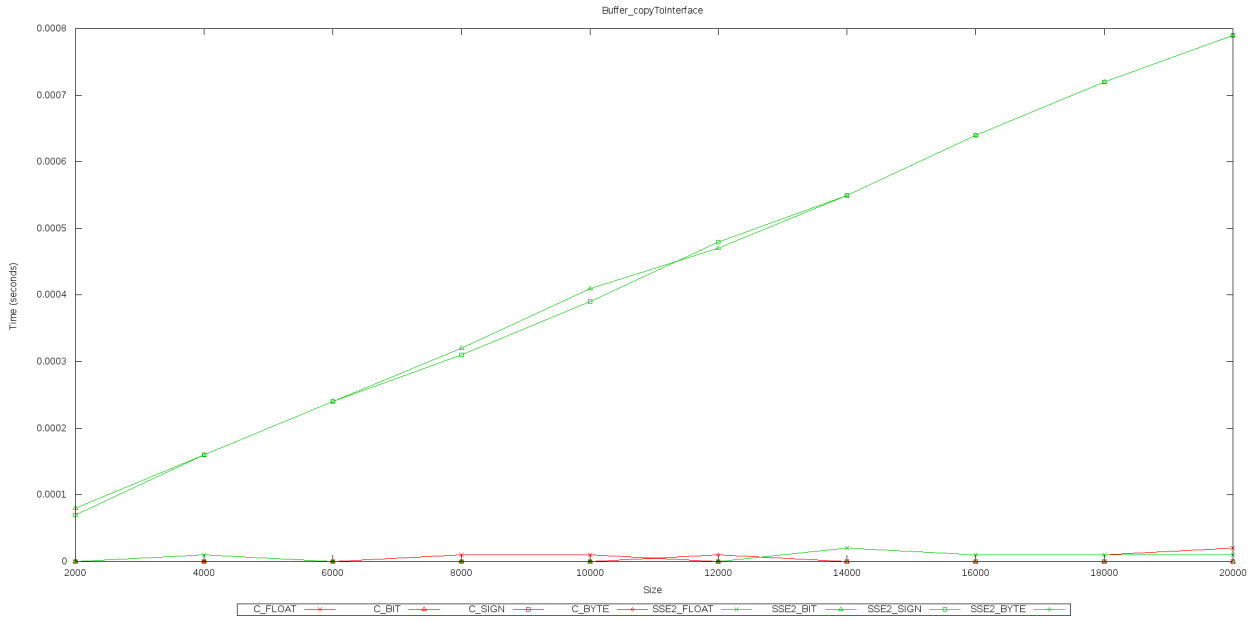

Figura 6.8: Método que implementa la copia de un Buffer con una implementación determinada a un Buffer genérico de la clase Interfaz (Buffer::copyToInterface).

6.1.2. Rendimiento de los diferentes operadores genéticos

En esta sección se cronometran los operadores genéticos. Algunos ya se habían analizado en la sección 6.1.1, pero a un nivel más bajo, del que dependen las implementaciones. Ahora lo que nos interesa es principalmente el coste de los operadores dependiendo de las diferentes formas de usarlo a más alto nivel. Nos abstraemos por ello de la implementación concreta y usamos la implementación SSE2 para todas estas pruebas.

6.1.2.1. Selección figura 6.9

Aunque no se aprecia bien en la figura 6.9, las selecciones de ruleta y truncado tienen un coste mínimo y bastante independiente tanto del tamaño total de la población como del número de individuos a seleccionar en comparación con los otros dos esquemas de selección. La selección por ranking es la más costosa y además la que más depende del tamaño total de la población. La selección por torneo es algo menos costosa y depende más del tamaño del torneo que del tamaño total de la población.

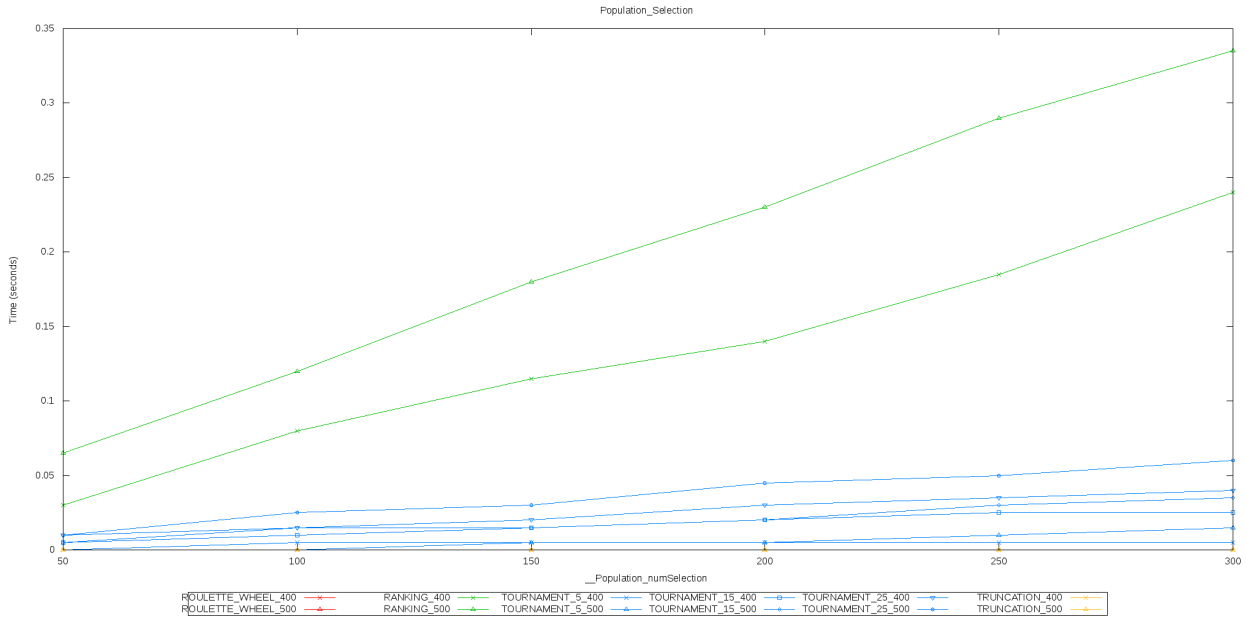

Figura 6.9: Rendimiento de los distintos operadores de selección en función del número de individuos a seleccionar. Cada operador se prueba con tamaños totales de población 400 y 500. Además, para la selección por torneo se muestran los resultados con varios tamaños de torneo.

6.1.2.2. Cruza figura 6.10

Como muestra la figura 6.10 el nivel de cruza más costoso es el propio peso, el más barato es el nivel de capa completa y los niveles de neurona tienen rendimientos intermedios y parecidos independientemente de cómo se interprete qué pesos pertenecen a una neurona (si los pesos de entrada a la neurona o los pesos de salida de la misma).

En general, parece que el cruce uniforme es más costoso que el multi-punto y también es más costoso cuanto más se acerque al 50 % la probabilidad de coger el peso de un padre o de otro. El algoritmo multi-punto, sin embargo, no presenta diferencias de rendimiento significativas para 1 punto o 6 puntos de corte.

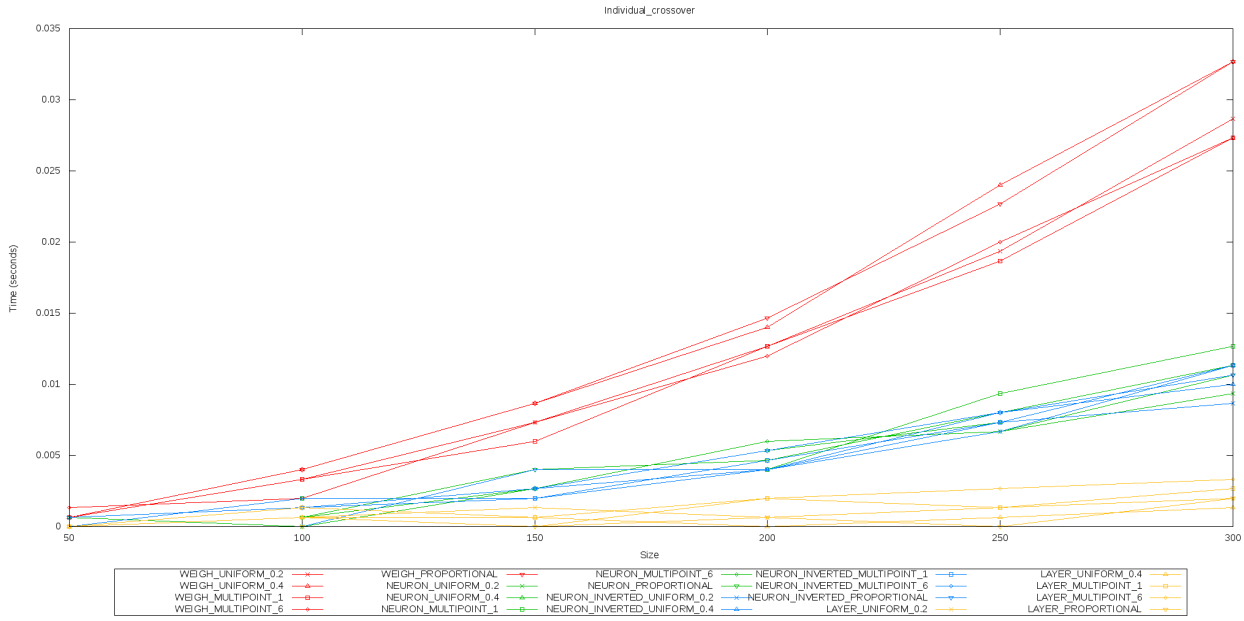

Figura 6.10: Rendimiento de los distintos operadores genéticos de cruza en función del tamaño de las capas de neuronas internas.

6.1.2.3. Mutación Según la figura 6.11, ambos tipos de mutación son más costosos cuánto mayor es la probabilidad, tal y cómo era de esperar, pues más mutaciones son más escrituras. Además, la mutación probabilística es en general más costosa que la mutación determinística (un número constante de mutaciones por individuo) salvo cuando la probabilidad es muy grande (muchas mutaciones por individuo). Aunque la interfaz determinística es más rápida, también se hace más lenta compratívamente cuanto mayor es la probabilidad. La explicación es que la opción probabilística tiene un coste fijo elevado (un número aleatorio por gen) mientras que en la determinística el coste sube con el número de mutaciones (varios números aleatorios por cada mutación a realizar).

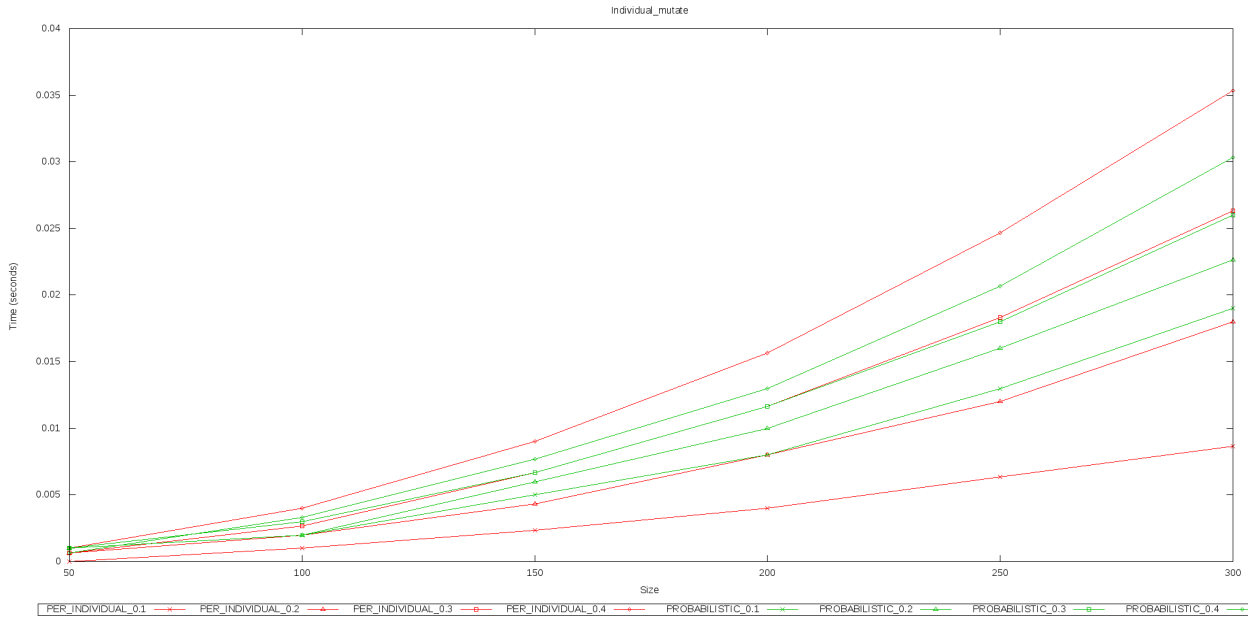

Figura 6.11: Rendimiento de los distintos operadores genéticos de mutación para diferentes probabilidades. Para el operador no probabilístico, se ha calculado el número de mutaciones por individuo multiplicando la probabilidad por el número de genes (pesos y umbrales), para poder compararlo en igualdad con el operador probabilístico en cuanto al número de escrituras en los pesos.

6.1.2.4. Olvido El nuevo operador de olvido/reset presenta otra vez unos resultados similares a los de la mutación y se pueden extraer conclusiones semejantes como muestra la figura 6.12.

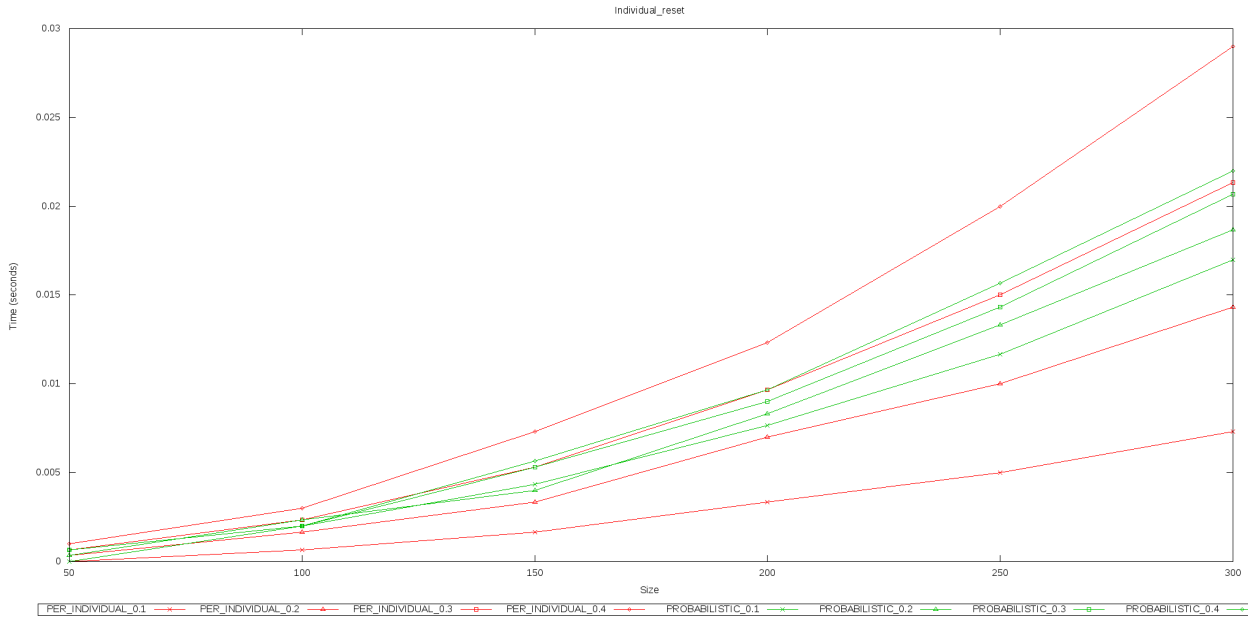

Figura 6.12: Rendimiento de los distintas interfaces del nuevo operador genético de olvido para diferentes probabilidades. Para el operador no probabilístico, se ha calculado el número de mutaciones por individuo multiplicando la probabilidad por el número de genes (pesos y umbrales), para poder compararlo en igualdad con el operador probabilístico en cuanto al número de escrituras en los pesos.

6.2. Aprendizaje

En esta sección se muestran resultados de aprendizaje en términos del mejor fitness de la población en cada generación.

6.2.1. Comparación entre distintas funciones de activación

6.2.2. Comparación entre redes discretas y lineales

A continuación compararemos los posibles efectos beneficiosos o perjudiciales al aprendizaje que se pueden obtener a partir de renunciar a las funciones derivables en favor de neuronas de tipo BIT (salida 0 ó 1) o SIGN (salida -1 ó 1) y de pesos discretos y más reducidos (1 Byte en lugar de los 4 bytes de un float).

6.2.2.1. OR

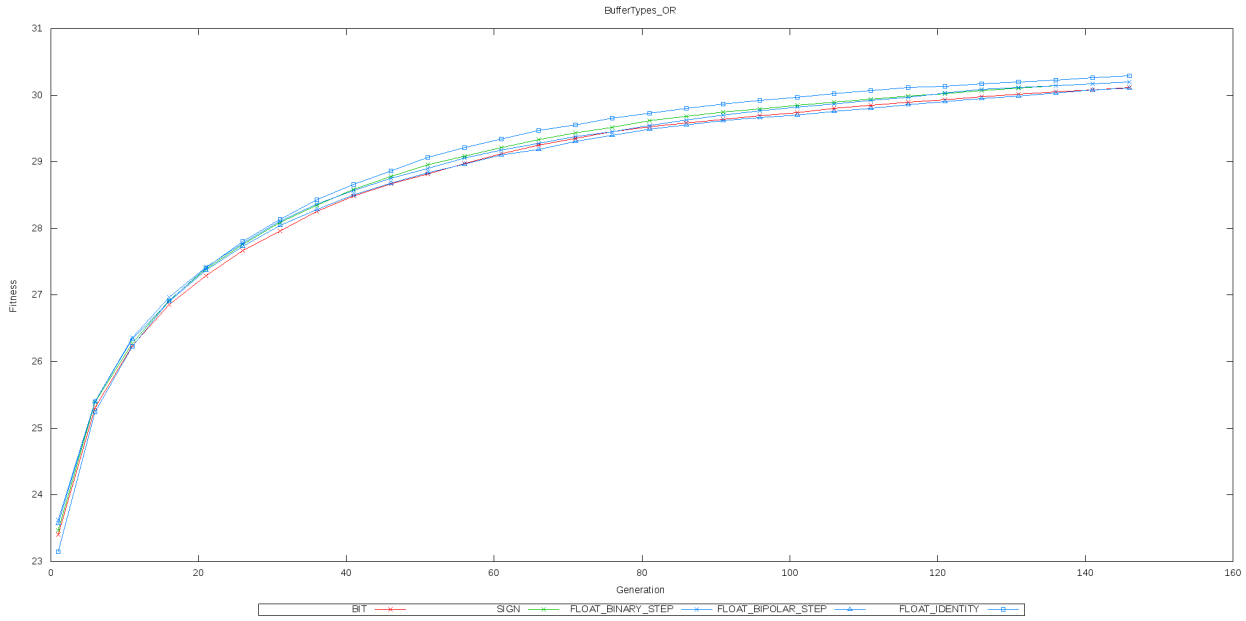

Figura 6.13: Aprendizaje de la tarea lógica Or con los distintos tipos de neuronas (Binarias, bipolares y lineales).

6.2.2.2. AND

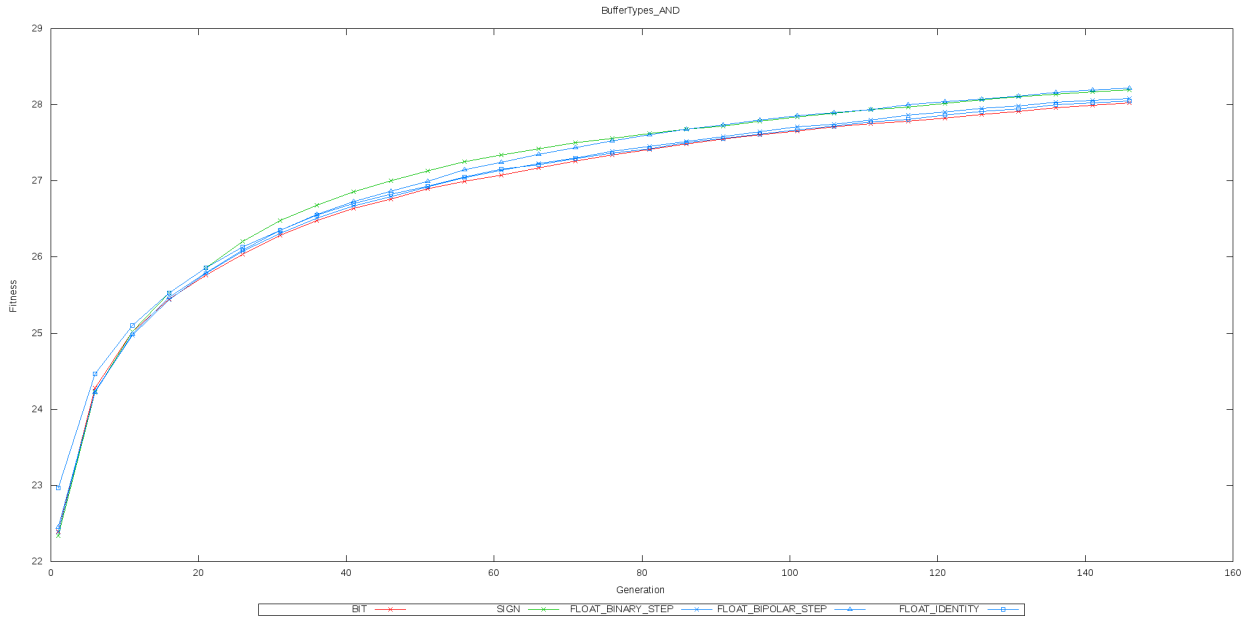

Figura 6.14: Aprendizaje de la tarea lógica And con los distintos tipos de neuronas (Binarias, bipolares y lineales).

6.2.2.3. XOR

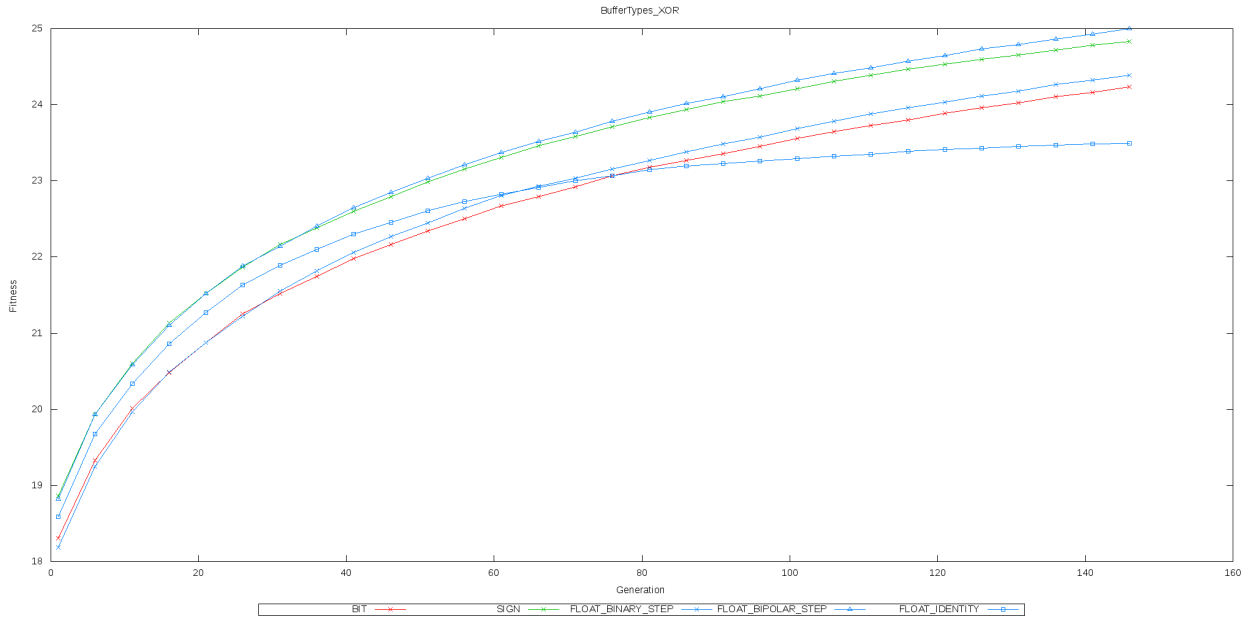

Figura 6.15: Aprendizaje de la tarea lógica Xor con los distintos tipos de neuronas (Binarias, bipolares y lineales).

6.2.2.4. Reversi

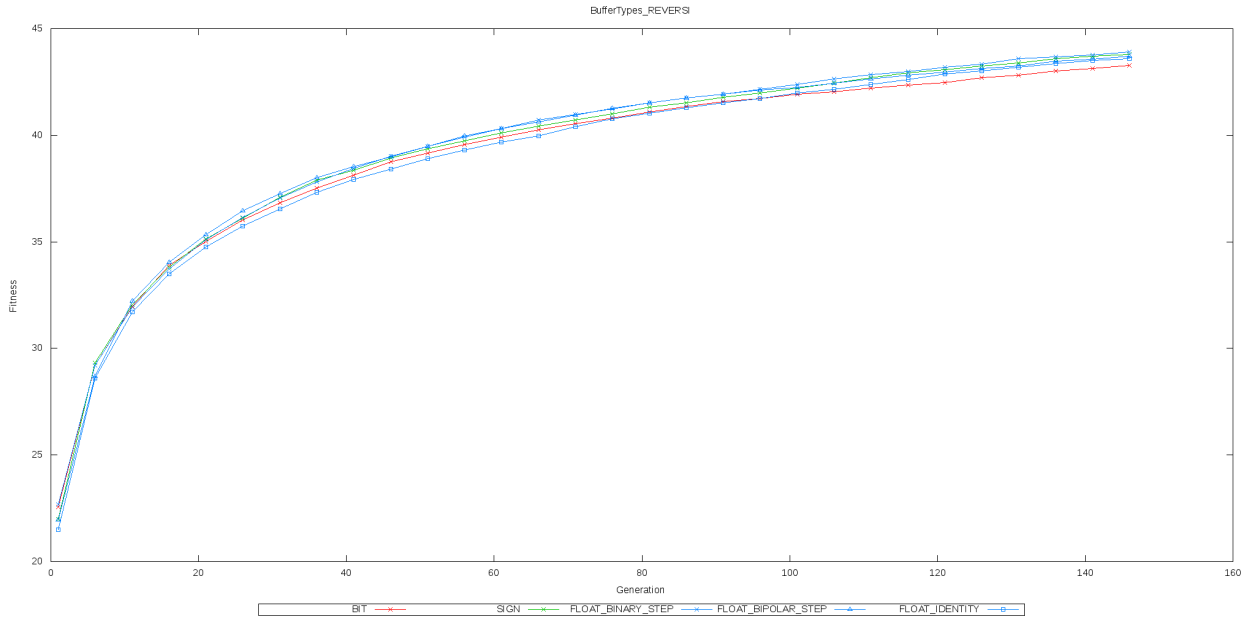

Figura 6.16: Aprendizaje de la tarea Reversi con los distintos tipos de neuronas (Binarias, bipolares y lineales).

6.2.3. Comparación de operadores genéticos

6.2.3.1. Selección

6.2.3.2. Cruza

6.2.3.3. Mutación

6.2.3.4. Operador de olvido

6.2.4. Comparación de distintos tamaños de población número de individuos conservados por generación

7. Conclusiones

Referencias

[1] Xin Yao: *Evolving Artificial Neural Networks*. School of Computer Science, The University of Birmingham (1999).

[2] Faustino J. Gómez y Risto Miikkulainen: *Robust Non-Linear Control through Neuroevolution*. Deparment of Computer Science, The University of Texas (2003).

[3] Luis Federico Bertona: *Entrenamiento de redes neuronales basado en algoritmos evolutivos*. Faculad de ingeniería, Universidad de Buenos Aires (2005).