NeSR - Neuroevolución de Sistemas de Reescritura

Germán L. Osella Massa[1], Esteban A. García[2], Laura Lanzarini[3]

Instituto de Investigación en Informática LIDI[4]
Facultad de Informática. Universidad Nacional de La Plata.

Resumen

Las redes neuronales artificiales evolutivas representan una mejora de las redes neuronales artificiales convencionales donde su capacidad de adaptación se ve favorecida por la aplicación de procesos evolutivos.

El presente artículo describe un nuevo método evolutivo denominado **NeSR** (NeuroEvolución de Sistemas de Reescritura) que permite obtener redes neuronales artificiales representadas a través de codificación indirecta.

Para realizar dicha codificación se propone utilizar sistemas de reescritura ya que poseen la capacidad de generar arquitecturas complejas a partir de una representación relativamente pequeña. En base a esta representación, se ha realizado una cuidadosa definición de los operadores de crossover y mutación.

El método propuesto ha sido aplicado a dos tipos de problemas diferentes (clasificación y control) como forma de mostrar la capacidad de resolución de la estrategia planteada. Los resultados alcanzados a través de las mediciones realizadas son satisfactorios.

Finalmente se presentan las conclusiones y se plantean algunas líneas de trabajo futuras.

Palabras Claves: Redes Neuronales Artificiales Evolutivas, Algoritmos genéticos, Codificación indirecta, Sistemas de reescritura L

[1] *Auxiliar Docente. Facultad de Informática. UNLP. gosella@freedom.net.ar*
[2] *Alumno de la Licenciatura en Informática. Facultad de Informática. UNLP. garcia77@topmail.com.ar*
[3] *Profesora Titular DE. III-LIDI. Facultad de Informática. UNLP. laural@info.unlp.edu.ar*
[4] *III-LIDI miembro del Instituto de Investigación en Ciencia y Tecnología Informática (IICyTI) - Facultad de Informática. UNLP - Calle 50 y 115 1er Piso, (1900) La Plata, Argentina. TE/Fax +(54) (221) 422-7707. http://lidi.info.unlp.edu.ar*

1 Introducción

Las redes neuronales artificiales (RNA) basan su proceso de adaptación en técnicas de entrenamiento que, habitualmente, sólo modifican los pesos de una arquitectura predefinida. Esto implica la necesidad de poseer un conocimiento a priori del problema que se desea resolver para establecer la topología de la red a entrenar, pudiendo resultar insuficiente si no se dispone de la información necesaria.

Las redes neuronales artificiales evolutivas (RNAE) mejoran este proceso de adaptación a través de la incorporación de métodos evolutivos como forma de obtener los pesos de conexión, el diseño de la arquitectura, el valor de los parámetros iniciales, las reglas de aprendizaje, etc. [11].

Cuando se trabaja con RNAE se presentan dos posibilidades para la representación del genotipo: la *codificación directa* que incluye toda la información necesaria para definir la RNA y la *codificación indirecta* que sólo utiliza los parámetros más importantes. Esta última permite reducir la longitud de la representación del genotipo ya que sólo se codifican algunas características de la arquitectura obteniendo las restantes a través de la evolución.

El presente artículo describe un nuevo método evolutivo denominado ***NeSR*** *(NeuroEvolución de Sistemas de Reescritura)* que permite obtener RNA basadas en codificación indirecta. Su aplicación permite generar tanto la estructura como los pesos de las conexiones de la red. Para la definición del genotipo se propone la utilización de sistemas de reescritura conocidos como Sistemas L [6].

2. El método NeSR

Se propone utilizar RNAE obtenidas a través de la combinación de sistemas de reescritura con un algoritmo genético simple [1].

Cada elemento de la población se encuentra definido por un sistema de reescritura L, de la forma en que se describe en la sección 2.1. A partir de un sistema se obtiene una secuencia de comandos que permiten construir la red neuronal, como se describe en la sección 2.2. Dicha red será evaluada en un problema dado y su desempeño determina el valor de adecuación (fitness) del sistema de reescritura utilizado. En la sección 2.3 se detalla el proceso evolutivo junto a los operadores de crossover y mutación utilizados.

2.1. Sistemas de reescritura

Los Sistemas L son un tipo de sistemas de reescritura, introducidos en 1968 por Aristid Lindenmayer como un formalismo para simular el desarrollo de organismos multicelulares [6].

Estos sistemas han sido utilizados para modelar un amplio espectro de problemas que van desde el desarrollo de plantas [8], pasando por fractales [5], la morfología de robots [3] hasta incluso la estructura de mesas [4].

Presentan una forma similar a las gramáticas formales [2] pero la reescritura se realiza aplicando en paralelo, sobre una cadena a reescribir, todas las reglas. El siguiente ejemplo ilustra lo dicho.

Sea un sistema L de la forma:

```
P1 → P1 P2
P2 → P2 P1
```

Si se toma a P1 como módulo de inicio, luego de tres pasos de reescritura se obtiene:

```
                P1
              P1 P2
           P1 P2 P2 P1
      P1 P2 P2 P1 P2 P1 P1 P2
```

Las reglas pueden tener parámetros, dando así lugar a un sistema de reescritura paramétrico. El lado derecho de la regla, llamado *sucesor*, está formado por *módulos*. Los módulos son símbolos asociados a expresiones formadas por números, parámetros de la regla, operadores aritméticos, lógicos y/o de comparación. Además, cada regla puede tener *guardas* que permiten aplicarla solo si la condición indicada en la guarda es verdadera. Se considera un *módulo terminal* a aquel que no es reescrito por ninguna regla. De esta forma se puede obtener una cadena de módulos terminales como resultado de una reescritura.

La forma completa de una regla es:

```
Predecesor : Condición → Sucesor
```

Ejemplo de reescritura en un Sistema L:

Sea un sistema L de la forma:

```
P1(N0) : N0 > 1 → P1(N0-1) P2(N0)
P1(N0) : N0 <= 1 → P1(0)
P2(N0) : N0 > 2 → P2(N0 / 2 + 0,5) P1(N0 - 1)
P2(N0) : N0 <= 2 → P2(0)
```

Comenzando la reescritura con `P1(4)` se obtiene:

```
                                        P1(4)
                                    P1(3) P2(4)
                                P1(2) P2(3) P2(2.5) P1(3)
                        P1(1) P2(2) P2(2) P1(2) P2(1.75) P1(2) P1(2) P2(3)
                    P1(0) P2(0) P2(0) P1(1) P2(2) P2(0) P1(1) P2(2) P2(2) P1(2)
            P1(0) P2(0) P2(0) P1(0) P2(0) P2(0) P1(0) P2(0) P2(0) P1(1) P2(2)
            P1(0) P2(0) P2(0) P1(0) P2(0) P2(0) P1(0) P2(0) P2(0) P1(0) P2(0)
```

Los sistemas pueden ser libres de contexto, como los descriptos hasta el momento, o sensibles al contexto, en donde se aplica una regla solo si coinciden tanto el predecesor como los módulos de ambos contextos. Las reglas tienen la siguiente forma:

```
Contexto izquierdo < Predecesor > Contexto derecho : Condición → Sucesor
```

Los contextos pueden ser cadenas vacías. Si no posee contextos, el sistema se denomina **P0L**, si posee uno solo de los contextos, se lo denomina **P1L** y si posee ambos, **P2L**.

En particular, en NeSR se utilizan sistemas **P0L**, con las siguientes restricciones:

- El conjunto de módulos utilizado para definir los predecesores de las reglas es único y común a todos los sistemas.
- La cantidad de reglas de producción en cada sistema es la misma.
- Todos los sistemas comienzan su reescritura a partir del mismo módulo, pero cada sistema puede contener expresiones distintas en dicho módulo.
- El sistema se reescribirá un máximo de veces o hasta que la cadena de módulos resultante supere una determinada longitud.
- Cada regla tiene un número mínimo y máximo de sucesores con sus correspondientes condiciones.
- Cada sucesor de regla tiene un número mínimo y máximo de módulos.
- Cada módulo tiene un número mínimo y máximo de parámetros.

2.2. Construcción de la red neuronal

Es posible representar una red neuronal artificial con un sistema de reescritura interpretando el resultado como las órdenes para construir dicha red. De esta forma, cada módulo terminal posee una semántica asociada y se puede pensar en ellos como comandos que indican la manera de construir la red neuronal. Cualquier otro módulo no terminal que hubiera quedado luego de la reescritura será ignorado.

Por ejemplo, si se da a los módulos de la forma **LINK(N1, N2, W)** el significado de "*establecer una conexión desde la neurona **N1** hasta la neurona **N2** con peso **W***", alcanzaría para representar cualquier red neuronal describiendo con ellos todas sus conexiones.

Otro conjunto de módulos puede ser utilizado para permitir la construcción de redes que cumplan determinadas restricciones (por ejemplo, que no sean recurrentes) o que posean una forma preestablecida.

Antes de describir los comandos utilizados, es necesario analizar una serie de condiciones que se cumplen en el momento de la construcción de la red neuronal:

- La cantidad de entradas y salidas son conocidas a priori ya que dependen del problema.
- Las neuronas tienen un número entero como identificador. A la primer neurona de entrada se le asigna el número 0, la segunda el 1 y así sucesivamente.
- Los identificadores de las neuronas de salidas son continuos a los de las de entradas, de manera que si la última entrada tiene identificador 3, la primera neurona de salida será la 4.
- A las neuronas ocultas se les asignan identificadores seguidos de los de la última neurona de salida.

De esta forma, a las neuronas de una RNA con 3 entradas, 1 salida y 2 ocultas se les asigna los siguientes identificadores.

Entrada 1	Entrada 2	Entrada 3	Salida 1	Oculta 1	Oculta 2
0	1	2	3	4	5

En este método, se utilizó el siguiente conjunto de comandos para construir las redes neuronales:

- **Neuron(id)**: Se toma como referencia a la neurona indicada por el identificador **id** para la aplicación de los comandos siguientes. Esta neurona la llamaremos *NEURONA_ACTUAL*. El identificador es relativo a la primera neurona de salida, pudiendo seleccionarse neuronas ocultas y de salida.
- **FromI(id, w):** Establece una conexión con peso **w** desde la neurona de entrada indicada por (**id mod** cantidad de entradas) hasta *NEURONA_ACTUAL*.
- **FromO(id, w):** Establece una conexión con peso **w** desde la neurona de salida indicada por (**id mod** cantidad de salidas) hasta la *NEURONA_ACTUAL*.
- **ToO(id, w):** Establece una conexión con peso **w** desde *NEURONA_ACTUAL* hasta la neurona de salida indicada por (**id mod** cantidad de salidas).
- **FromH(id, w):** Establece una conexión con peso **w** desde la neurona oculta indicada por **id** hasta *NEURONA_ACTUAL*.
- **ToH(id, w):** Establece una conexión con peso **w** desde *NEURONA_ACTUAL* hasta la neurona oculta indicada por **id**.
- **Rec(w):** Establece una conexión con peso **w** desde *NEURONA_ACTUAL* hacia sí misma.

Dado que los parámetros que denotan identificadores pueden ser expresiones negativas, se toma su valor absoluto antes de determinar a que neurona corresponde.

Cuando alguno de los comandos intenta generar una conexión ya creada, el peso de la nueva conexión es sumado al de la conexión existente.

Debido a que el valor de los parámetros correspondientes a los pesos no está acotado a ningún rango, se define una constante positiva MaxValue para ajustar los pesos como indica la siguiente función ε :

$$\varepsilon(w) = \begin{cases} -MaxValue, & \text{si } w \leq -MaxValue \\ w, & \text{si } abs(w) < MaxValue \\ MaxValue, & \text{si } w \geq MaxValue \end{cases}$$

La aplicación de esta función evita generar conexiones con pesos fuera del rango deseado.

2.3 Evolucionando sistemas de reescritura

El método aquí presentado se basa en un algoritmo genético simple, de la forma descripta en [1], aplicado a poblaciones de individuos representados por cromosomas que codifican Sistemas L.

Para evolucionar sistemas de reescritura es necesario imponer restricciones a su forma conjuntamente con un diseño cuidadoso de los operadores genéticos, ya que de lo contrario el espacio de búsqueda se torna tan amplio que dificulta la convergencia de la población en una buena solución.

El algoritmo genético parte de una población inicial de Sistemas L generados al azar que cumplen con las restricciones impuestas a su forma. Para la selección de los individuos a reproducirse utiliza el método de la ruleta. El reemplazo de la población se realiza aplicando elitismo, es decir, los n mejores individuos pasan intactos a la siguiente generación. El resto es reemplazado por nuevos individuos obtenidos a partir de la aplicación de operadores genéticos sobre pares de cromosomas seleccionados dentro del total de la población.

Pseudocódigo del algoritmo genético simple:

```
T = 0
P(T) = Población inicial aleatoria de N
Evaluar P(T)
Mientras T < Máxima generación y no alcance el fitness máximo
   P(T + 1) = Población con los m mejores individuos de P(T)
   Mientras Cantidad de individuos de P(T + 1) < N
         Seleccionar Padre1 y Padre2 dentro de P(T)
         Hijo = Crossover(Padre1, Padre2)
         Mutación(Hijo)
         Agregar Hijo a P(T + 1)
   Evaluar P(T + 1)
   T = T + 1
```

En NeSR, se proponen dos operadores genéticos: el crossover entre dos sistemas, de manera de combinar las reglas que poseen el mismo predecesor, y la mutación, que realiza cambios en las reglas, agregando, reemplazando o quitando módulos, o alterando las expresiones que estos contienen.

El crossover recorre las reglas de dos sistemas, intercambiando los casos de las reglas comunes a ambos. Esto minimiza los efectos adversos de este operador, ya que es de esperar que ambas reglas tengan un significado similar dentro de sistemas de reescritura parecidos.

```
Algoritmo de crossover entre dos sistemas de reescritura S1 y S2:

Con probabilidad p_cross →
  Nuevo Sistema = Sistema vacío con el módulo inicial elegido al azar
                  entre los módulos iniciales de S1 y S2

  Para cada predecesor i de S1
    Para cada regla j con predecesor i de S1
      Obtener regla j con predecesor i del S2
      Si existe →
        Nueva Regla = Regla vacia con predecesor i común a S1 y S2
        Para cada caso k de la regla j de S1 →
          Obtener caso k de la regla j de S2
          Si existe →
            Nuevo Caso =  copia del caso k de la regla j de S1 o S2,
                          eligiéndose el sistema en forma aleatoria
          Sino →
            Nuevo Caso = copia del caso k de la regla j de S1

          Agregar Nuevo Caso a la Nueva Regla.
      Sino →
        Nueva Regla = copia de la regla j con predecesor i de S1

      Agregar Nueva Regla al Nuevo Sistema
Sino →
  Nuevo Sistema = copia de S1
```

El operador de crossover, tal cual se lo definió, permite explorar las combinaciones de los casos de reglas disponibles. Sin embargo, no altera los sucesores de las reglas.

Para introducir cambios en dichos sucesores, se define un operador de mutación, de forma que se introduzca diversidad en los sistemas evolucionados.

Como el operador de mutación puede actuar sobre las expresiones de los módulos de un sistema o bien sobre la estructura del mismo, se optó por separarlo en dos partes:

- *Mutación de expresiones*: se altera con cierta probabilidad las expresiones del sistema L, tanto en las condiciones de las reglas como en las expresiones contenidas en los módulos de los sucesores.
- *Mutación de estructura:* altera a un sistema insertando, eliminando o reemplazando módulos en los sucesores de las reglas de reescritura.

El criterio de aplicación de uno u otro es aleatorio: el 50% de las veces se aplica la versión que afecta a las expresiones y el otro 50% la versión que afecta a la estructura.

En ambas versiones del operador de mutación se recorren las reglas de un sistema alterándolas, como se describe a continuación.

Algoritmo de mutación de expresiones de un sistema de reescritura:

```
Para cada regla i del sistema
  Con probabilidad p_mexp →
    Alterar Condición de la regla i

  Para cada módulo j del sucesor de la regla i
    Para cada expresión k del módulo j
      Con probabilidad p_mexp →
        Alterar expresión k
```

Ejemplo de mutación de expresiones en una regla de producción:

Dada la expresión N0 > 10 perteneciente a una regla con predecesor P1(N0, N1), puede mutarse en cualquiera de las siguientes opciones:

- 2 > 10 (Reemplazó N0 por 2)
- N0 + 1 > 10 (Reemplazó N0 por N0 + 1)
- N0 > 1 + N1 (Reemplazó 10 por 1 + N0)
- N0 + N1 > 10 (Reemplazó N0 por N0 + N1)
- Not (N1 * 2 = 5) (Reemplazó toda la expresión por una nueva)

La subexpresión que se inserta siempre es del mismo tipo que la que se seleccionó para el reemplazo, es decir, si el resultado de la subexpresión es un entero, se reemplaza por otra subexpresión de valor entero. Análogamente ocurre para valores reales y booleanos.

Algoritmo de mutación de estructura de un sistema de reescritura:

```
Para cada regla i del sistema
  Con probabilidad p_mest →
    En el sucesor de la regla i
      Con determinadas probabilidades →
        Se insertan módulos aleatorios
        Se reemplazan módulos existentes por módulos aleatorios
        Se eliminan módulos
      de manera de respetar las restricciones impuestas.
```

Los módulos generados al azar se construyen empleando símbolos comunes a todos los sistemas o símbolos de comandos. Sus expresiones, al igual que las generadas en la mutación de expresiones, pueden contener referencias a los parámetros de la regla que las contendrá.

Ejemplo de mutación de estructura en una regla de producción:

Sea la siguiente regla, en la que se selecciona al azar un segmento el cual va a ser sometido a la mutación:
P1(N0, N1): N0 + 1 > 3 → Neuron(N0) **FromH(N0, N1) Rec(0.5) P2(N1)** P1(N0-1, N1 * 2)

Hay tres acciones posibles:
- Eliminación del segmento:
  P1(N0, N1): N0 + 1 > 3 → Neuron(N0) P1(N0-1, N1 * 2)
- Reemplazo por un segmento creado con módulos y expresiones al azar:
  P1(N0, N1): N0 + 1 > 3 → Neuron(N0) **P1(N+1, 0.2) ToO(-10, N1)** P1(N0-1, N1 * 2)
- Inserción de un segmento creado con módulos y expresiones al azar al comienzo del segmento elegido:
  P1(N0, N1): N0 + 1 > 3 → Neuron(N0) **P1(N+1, 0.2) ToO(-10, N1)** FromH(N0, N1) Rec(0.5) P2(N1) P1(N0-1, N1 * 2)

Los tamaños aleatorios de los segmentos seleccionados y creados al azar deben ser ajustados para cumplir con las restricciones de las cantidades mínimas y máximas de módulos en los sucesores de reglas.

2.4. La red neuronal

Las redes neuronales obtenidas son típicamente recurrentes debido a que no se imponen restricciones en su topología. Por tal motivo, cada neurona actualiza su valor de activación de la siguiente manera:

$$x_i(t+1) = \sigma\left(\sum_{j=1}^{N} a_{ij} x_j(t) + \sum_{j=1}^{M} b_{ij} u_j(t) + c_i \right), i = 1..N$$

donde M es la cantidad de entradas y N es el número de neuronas, ocultas y de salida, que conforman la red. Los elementos a_{ij} representan los pesos de las conexiones entre las neuronas ocultas y de salida, los b_{ij} representan las conexiones desde las neuronas de entrada hacia el resto de las neuronas y los c_i representan los términos de tendencia de cada neurona. Por lo tanto, la activación de cada neurona es actualizada en función de las estradas u_j y de las activaciones x_j del estado anterior.

La función σ está definida de la siguiente forma:

$$\sigma(x) = \begin{cases} -1 & , \text{si } x \le -MaxValue \\ \dfrac{x}{MaxValue} & , \text{si } abs(x) < MaxValue \\ 1 & , \text{si } x \ge MaxValue \end{cases}$$

En NeSR los términos de tendencia se encuentran implementados a través de una neurona de entrada adicional, cuyo valor de activación es siempre 1.

3. Experimentos con NeSR

Para comprobar la efectividad del método evolutivo, se seleccionaron tres problemas frecuentemente utilizados para evaluar el desempeño en distintos métodos: el XOR (problema de clasificación), el péndulo invertido y el doble péndulo invertido (problemas de control).

Para cada uno de los experimentos se utilizaron los siguientes valores para las probabilidades y restricciones aplicadas a los operadores:

	Descripción	Valor
Pcross	Probabilidad de realizar un crossover entre dos sistemas.	0.9
Pmexp	Probabilidad de realizar una mutación sobre una expresión.	0.05
Pmest	Probabilidad de realizar una mutación estructural sobre un sucesor.	0.15
Pins	Probabilidad de insertar módulos aleatorios durante una mutación estructural.	0.30
Premp	Probabilidad de remplazar módulos existentes por aleatorios en una mutación estructural.	0.40
Pelim	Probabilidad de eliminar módulos existentes durante una mutación estructural.	0.30
InsMods	Cantidad máxima de módulos a insertar en un sucesor en una mutación estructural.	2
ElimMods	Cantidad máxima de módulos a eliminar de un sucesor en una mutación estructural.	2

También se especificaron valores para las siguientes restricciones aplicadas a los sistemas de reescritura:

	Descripción	XOR	Péndulo Simple	Péndulo doble
NR	Cantidad de reglas que posee un sistema	3	3	3
NC	Cantidad máxima de casos para una regla de un sistema.	2	2	2
NP	Cantidad máxima de parámetros que posee un módulo.	2	3	3
LMin	Cantidad mínima de módulos que puede tener un sucesor.	5	5	5
LMax	Cantidad máxima de módulos que puede tener un sucesor.	15	15	15

La reescritura de los sistemas se realizó un máximo de 3 veces o hasta que la cadena resultante haya superado los 1000 módulos de longitud.

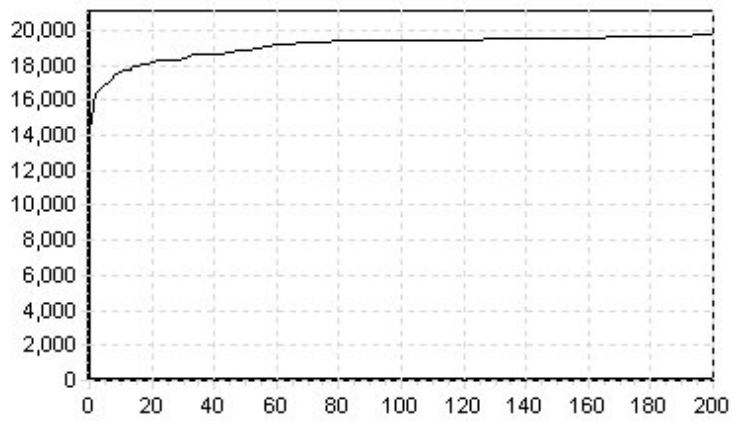
3.1. XOR

Este problema consiste en clasificar correctamente los patrones de entrada de manera que la respuesta de la red coincida con la función XOR. En este caso, la red recibe en sus dos entradas valores correspondientes a 0 y 1 y se espera como salida el resultado de aplicar la función XOR a dichas entradas. La red es evaluada para las cuatro posibles combinaciones de sus entradas.

Al trabajar con redes neuronales recurrentes es preciso definir el criterio utilizado para determinar su respuesta. Se ha optado por realizar una cantidad fija de actualizaciones y luego tomar los valores de salida como la respuesta obtenida. Esto además, permite acotar en cierta medida el tamaño de la red, puesto que si la cantidad de neuronas que deben activarse para generar una salida es mayor a la cantidad de evaluaciones realizadas, la red no arrojará respuesta alguna.

Los resultados presentados a continuación se obtuvieron de 50 corridas independientes, con una población de 100 individuos, cuyo criterio de terminación fue que se cumplieran 200 generaciones o que se alcanzara el objetivo planteado.

En promedio fueron necesarias 34 generaciones para obtener una red que resolviera el problema, como máximo se requirieron 195 generaciones y como mínimo 1. En 5 de los 50 intentos no fue posible resolver el problema.



Promedio de los mejores fitness por generación en 50 intentos

3.2. Péndulo invertido

El péndulo invertido es un problema clásico de control, que consiste en mantener balanceado un péndulo montado sobre un carrito, el cual puede moverse a la izquierda o a la derecha sobre una pista horizontal de una longitud fija. Este movimiento tiende a desbalancear el péndulo. El objetivo es mover al carrito de manera de mantener balanceado al péndulo durante un período de tiempo evitando traspasar los límites de la pista. Los detalles del modelo pueden encontrarse en [7].

La red neuronal puede recibir como entrada cualquiera de los valores que describen al estado del sistema. El caso más sencillo de resolver es cuando la red recibe el estado completo del sistema. Sin embargo, para dificultarle la tarea, sólo se le informó a la red la posición del péndulo y del carrito, debiendo la red inferir las velocidades.

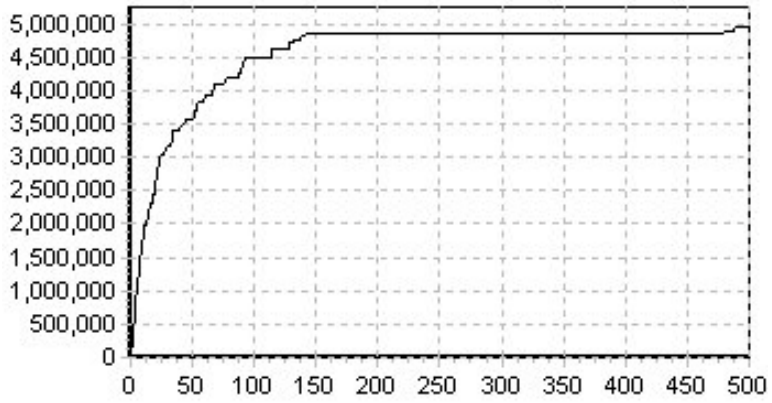
La red indica hacia que lado debe moverse el carrito mediante sus dos salidas: Si la activación de la primer salida es mayor a la de la segunda, se aplica una fuerza determinada hacia la izquierda sobre el carrito. Si es menor, se aplica hacia la derecha. Si son iguales, ninguna fuerza es aplicada.

Cada red es evaluada en 18 posiciones iniciales distintas correspondientes a las combinaciones de 3 posiciones del péndulo con 3 velocidades angulares diferentes (extremos y centro de los rangos correspondientes) y con 2 posiciones distintas del carrito (una sobre la izquierda de la pista y otra sobre la derecha). Se omitieron los casos que pueden resultar sencillos de resolver, en donde el péndulo se encuentra en equilibrio o el carrito parte desde el centro de la pista. Se observó en varios intentos que en las primeras etapas de la evolución aparecen individuos que resuelven únicamente estos casos y tienden a dominar al resto de la población, produciendo una convergencia prematura.

Cada prueba se realizó durante 50000 pasos de evaluación (equivalentes a 15 minutos), siendo el fitness resultante el promedio de pasos alcanzados en los 18 intentos, multiplicándolo por 100 y sumándole 100 por cada neurona de entrada o salida que está conectada.

Se realizaron 30 corridas independientes, con una población de 100 individuos, con un máximo de 500 generaciones, pudiendo terminar antes en caso de resolver el problema. En términos de evaluación de individuos, el problema requiere un gran tiempo de procesamiento ya que en cada simulación la red puede llegar a ser evaluada un máximo de 1.800.000 veces, lo cual aplicado a cada individuo por cada generación resulta significativo.

En promedio fueron necesarias 44 generaciones para obtener una red que resolviera el problema, como máximo se requirieron 144 generaciones y como mínimo 4. Sólo en 1 de los 30 intentos no se llegó a resolver el problema.



Promedio de los mejores fitness por generación en 30 intentos

3.3. Doble péndulo invertido

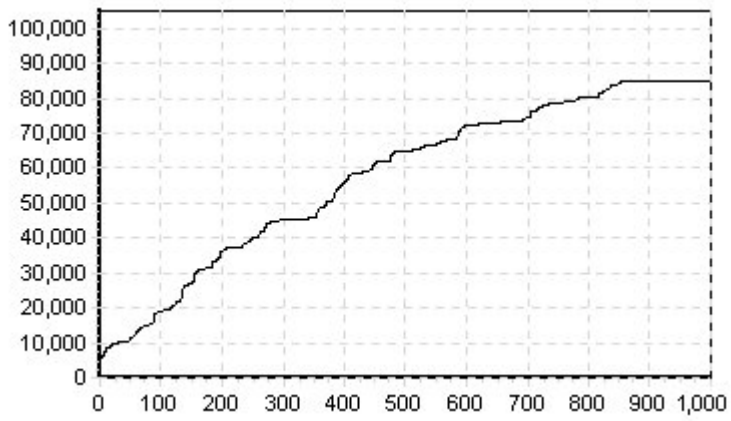
Este problema consiste en una versión de mayor dificultad del problema anterior. Se dispone de dos péndulos de diferente tamaño y peso a los cuales hay que mantener balanceados. El modelo matemático utilizado para la simulación coincide con el utilizado en [10].

Al igual que en la versión del péndulo simple, se omite la información acerca de las velocidades de los péndulos y del carrito, conociendo la red neuronal solamente las posiciones.

Por cada red se realizan los mismos 18 intentos que en el caso del problema anterior, variando las posiciones de manera similar, sólo que el péndulo más pequeño queda siempre vertical y el mayo varía su posición y velocidad. Cada intento se realiza hasta un máximo de 1000 pasos de simulación. El fitness se calcula promediando los pasos alcanzados sobre el total de los intentos.

Se realizaron 30 corridas independientes, con una población de 200 individuos, con un máximo de 1000 generaciones, pudiendo terminar antes en caso de resolver el problema. Al igual que en el caso anterior, la evaluación resulta costosa en términos de procesamiento; en cada simulación la red puede llegar a ser evaluada un máximo de 36.000 veces.

En promedio fueron necesarias 488 generaciones para obtener una red que resolviera el problema, como máximo se requirieron 855 generaciones y como mínimo 30. En 9 intentos no fue posible resolver el problema.



Promedio de los mejores fitness por generación en 30 intentos

4. Conclusiones y futuras líneas de trabajo

Se ha presentado un método para evolucionar redes neuronales, codificada indirectamente a través de sistemas de reescrituras, que ha demostrado su eficacia en resolver algunos problemas clásicos del área. En particular se destaca la flexibilidad en cuanto a las topologías de redes que pueden generarse a través de esta representación.

Se está estudiando el refinamiento del método de selección y operadores genéticos utilizados en la evolución, de manera de hacerlos más efectivos.

Otro aspecto a investigar se encuentra relacionado con los comandos utilizados para la construcción de las redes neuronales. El esfuerzo debe centrarse en el desarrollo de conjuntos específicos para determinado tipo de problemas.

Por último, se busca explotar la potencialidad que tiene el método para generar complejas estructuras de redes neuronales.

Referencias

[1]. Goldberg, D. (1989). “Genetic Algorithms in Search, Optimization, and Machine Learning”, Addison–Wesley.

[2]. John E. Hopcroft – Jeffrey D. Ullman (1993). “Introducción a la teoría de autómatas y lenguajes de computación”, Addison-Wesley.

[3] Hornby G. And Pollack J. (2001). “Evolving L-systems to generate virtual creatures”. *Computers and Graphics*, 25 : 1041-1048.

[4] Hornby G. And Pollack J. (2001). “The Advantages of Generative Grammatical Encodings for Physical Design” In Congress on Evolutionary Computation.

[5] Kókai, G, Tóth, Z and Ványi, R. (1998). “Application of Genetic Algorithms with more Populations for Lindenmayer Systems” In Proceedings of the International Symposium on Engineering of Intelligent Systems, EIS'98, pages 324{331. ICSC Academic Press.

[6]. Lindenmayer, A. (1968). “Mathematical Models for cellular interaction in development”. partes I y II. *Journal of Theoretical Biology.* 18:280-299 y 300-315.

[7] Moriarty, D. E. (1997) “Symbiotic Evolution of Neural Networks in Sequential Decision Tasks”. Dissertation.

[8] Prusinkiewiczy, P., Hammel, M., Hanan, J. & Měch, R. (1996). “L-systems: From the Theory to Visual Models of Plants”. Department of Computer Science, University of Calgary.

[9] Siegelmann, H. “Foundations of Recurrent Neural Networks”. Ph. D. Thesis.

[10]. Stanley, K. O. and Miikulainen R. (2002). “Evolving Neural Networks through augmenting topologies”. *Evolutionary Computation*, 10(2): 99-127.

[11]. Yao, X. (1999). “Evolving artificial neural networks”. *Proccedings of then IEEE*, 87(9): 1423-1447.