**CHAPTER** **6:** **UNDERSTANDING** **GENETIC** **ALGORITHMS**

* Presentación del algoritmo genético
* Entender la estructura de un algoritmo genético
* Entender cómo funciona un algoritmo genético
* Implementación del problema del vendedor ambulante
* Red neuronal juega Tic-Tac-Toe

La backpropagation no es la única manera de entrenar una red neuronal. Este capítulo introducirá algoritmos genéticos (GAs), que se pueden utilizar para resolver muchos tipos diferentes de problemas. Comenzaremos explorando cómo utilizar un algoritmo genético para resolver un prob-lem independiente de una red neuronal. A continuación, el algoritmo genético se aplicará a una red neuronal feedforward.

# Algoritmos genéticos

Tanto los algoritmos genéticos como el recocido simulado son procesos evolutivos que pueden ser utilizados para resolver problemas de espacio de búsqueda y optimización. Sin embargo, los algoritmos genéticos difieren sustancialmente del recocido simulado.

El recocido simulado se basa en un proceso evolutivo hemodinámico, mientras que los algoritmos genéticos se basan en los principios de la teoría de la evolución de Darwin y el campo de la biología. Dos características introducidas por los GAs, que los distinguen del simu- recocido latado, son la inclusión de una población y el uso de un operador genético llamado "crossover" o recombinación. Estas características se discutirán con más detalle más adelante en este capítulo.

Un componente clave de la evolución es la selección natural. Los organismos poco adecuados para su environment tienden a morir, mientras que los organismos más adecuados para su entorno actual son más propensos a sobrevivir. Los organismos sobrevivientes producen descendencia que tienen muchas de las mejores cualidades que poseen sus padres. Como resultado, estos niños tienden a ser "más adecuados" para su entorno, y son más propensos a sobrevivir, aparearse y producir generaciones futuras. Este proceso es análogo a la teoría de Darwin de "supervivencia del más apto", un proceso continuo de evolución en el que la vida sigue mejorando con el tiempo. Los mismos conceptos que se aplican a la selección natural se aplican también a los algoritmos genéticos.

Al hablar de evolución, es importante tener en cuenta que a veces se hace una distinción entre microevolución y macroevolución. La microevolución se refiere a pequeños cambios que ocurren en la composición genética general de una población durante un período relativamente corto de tiempo. Estos cambios son generalmente pequeñas adaptaciones a una especie existente, y no la introducción de una especie completamente nueva. La microevolución es causada por factores como la selección natural y la mutación. La macroevolución se refiere a cambios significativos en una población durante un largo período de tiempo. Estos cambios pueden dar lugar a una nueva especie. Los conceptos de algoritmos genéticos son consistentes con la microevolución.

## Antecedentes de algoritmos genéticos

John Holland, profesor de la Universidad de Michigan, desarrolló los conceptos asociados con algoritmos genéticos a través de la investigación con sus colegas y estudiantes. En 1975, publicó un libro, Adaptación en sistemas naturales y artificiales, en el que presenta la teoría detrás de los algoritmos genéticos y explora su aplicación práctica. Hollanda es considerada el padre de los algoritmos genéticos.

Otro contribuyente significativo al área de algoritmos genéticos es David Goldberg. Goldberg estudió con Hollanda en la Universidad de Michigan y ha escrito una colección de libros, incluyendo Algoritmos genéticos en búsqueda, optimización y aprendizaje mecanico (1989), y más recientemente, El diseño de la innovación (2002).

## Usos para algoritmos genéticos

Los algoritmos genéticos son algoritmos de búsqueda adaptables, que se pueden utilizar para muchos propósitos en muchos campos, como la ciencia, los negocios, la ingeniería y la medicina. Los GAs son expertos en buscar espacios de búsqueda grandes y no lineales. Un problema de espacio de búsqueda no lineal tiene un gran número de soluciones potenciales y la solución óptima no se puede resolver por medios iterativos convencionales. Los GAs son más eficientes y apropiados para situaciones in que:

* El espacio de búsqueda es grande, complejo o no se entiende fácilmente;
* No hay ningún método programático que se pueda utilizar para restringir el espacio de búsqueda;
* Los métodos de optimización tradicionales son inadecuados. La Tabla 6.1 enumera algunos ejemplos.

### Tabla 6.1: Usos comunes para algoritmos genéticos

|  |  |
| --- | --- |
| **propósito** | **Usos** comunes |
| optimización | Programación de producción, enrutamiento de llamadas para centros de llamadas, enrutamiento para el transporte, determinación de diseños de circuitos eléctricos |
| diseño | Aprendizaje automático: diseño de redes neuronales, diseño y control de robots |
| Aplicaciones empresariales | Utilizado en el comercio financiero, evaluación de crédito, alocapresupuesto- ción, detección de fraude |

Muchos problemas de optimización no son lineales en el comportamiento y son demasiado complejos para los métodos tradicionales. El conjunto de posibles soluciones para un problema de este tipo puede ser enorme (por ejemplo, determinar la ruta óptima para un vendedor ambulante o determinar el design óptimo para un diseño de circuito eléctrico). Un algoritmo genético posee la capacidad de buscar espacios de búsqueda grandes y complejos para determinar eficientemente soluciones casi óptimas en un plazo razonable mediante la simulación de la evolución biológica. Ahora que tiene been introducido a algunos de los usos para algoritmos genéticos, debemos examinar cómo construir uno realmente.

# Comprensión de algoritmos genéticos

Los algoritmos genéticos se asemejan mucho al modelo biológico de cromosomas y genes. Los organismos individuales en un algoritmo genético generalmente consisten en un solo cromosoma. Estos cromosomas están compuestos de genes. Al manipular los genes, se recrean nuevos cromosomas, que poseen rasgos diferentes. Estas manipulaciones ocurren a través del cruce y la mutación, tal como ocurren en la naturaleza. El cruce es análogo al proceso biológico de apareamiento, y la mutación es una forma en la que se puede introducir nueva información en una población existente.

## Comprensión de los genes

En un algoritmo genético, los genes representan componentes individuales de una solución. Este es un concepto muy importante en el análisis de un problema para el que se va a utilizar un algoritmo genético. Para resolver eficazmente un problema, debe determinar una manera de dividirlo en sus componentes o genes relacionados. A continuación, los genes estarán unidos para formar un cromosoma. Las formas de vida en esta simulación consisten en un solo cromosoma; por lo tanto, cada cromosoma representará una posible solución a un problema.

Más adelante en este capítulo, examinaremos el problema de los vendedores ambulantes. Se le mostrará cómo descompone la solución para este problema en genes individuales. Además, en este capítulo se le mostrará cómo hacer que los pesos individuales en una red neuronal representen los genes en el cromosoma.

Es importante tener en cuenta que hay un número finito de genes que se utilizan. Individ- los genes uales no se modifican a medida que los organismos evolucionan. Son los cromosomas los que evolucionan cuando se cambia el orden y la composición de sus genes.

## Comprensión de los cromosomas

Como se explicó anteriormente, cada organismo en nuestro algoritmo genético contiene un cromosoma. Como resultado, cada cromosoma se puede considerar como un "individuo" o una solución compuesta por una colección de parámetros a optimizar. Estos parámetros son genes, que aprendiste en la sección anterior. Cada cromosoma es inicialmente una solución aleatoria o una colección de genes. Esta solución se utiliza para calcular un nivel de "aptitud", que determina la idoneidad del cromosoma o la "aptitud" para sobrevivir, como en la teoría de darwin de la selección natural. Si un crisoma tiene un alto nivel de "aptitud", tiene una mayor probabilidad de aparearse y mantenerse con vida.

# Cómo funcionan los algoritmos genéticos

Un algoritmo genético comienza creando una población inicial de cromosomas que reciben una colección aleatoria de genes. A continuación, continúa de la siguiente manera:

1. Cree una población inicial de cromosomas.
2. Evaluar la aptitud o "idoneidad" de cada cromosoma que conforma la población.
3. En función del nivel de aptitud de cada cromosoma, seleccione los cromosomas que se aparearán, o aquellos que tengan el "privilegio" de aparearse.
4. Cruce (o aparee) los cromosomas seleccionados y produzca descendencia.
5. Mutar aleatoriamente algunos de los genes de los cromosomas.
6. Repita los pasos de tres a cinco hasta que se cree una nueva población.
7. El algoritmo termina cuando la mejor solución no ha cambiado para un numero preestablecido de generaciones.

Los algoritmos genéticos se esfuerzan por determinar la solución óptima a un problema mediante la utilización de tres operadores genéticos. Estos operadores son selección, crossover y mutación. Los GAs buscan la solución óptima hasta que se cumplan criterios específicos y finalice el proceso. Los resultados del proceso incluyen buenas soluciones, en comparación con una solución "optma", para problemas complejos(como problemas "np-hard"). NP-hard se refiere a problemas que no se pueden resolver en tiempo polinomio. La mayoría de los problemas resueltos con las computadoras hoy en día no son np-hard y se pueden resolver en tiempo polinomio. Un polinomial es una expresión matemática que involucra exponentes y variables. Un problema P, o problema polinomio, es un problema para el cual el número de pasos para encontrar la respuesta está limitado por un polinomio. Un problema np-hard no aumenta exponencialmente, pero a menudo increases a un ritmo mucho mayor, descrito por el operador factorial (n!). Un ejemplo de un problema difícil de NP es el problema de los vendedores ambulantes, que será desencantado más adelante en este capítulo.

## Initial Population

Para resumir, la población de un algoritmo genético se compone de organismos, cada uno de los cuales generalmente contiene un solo cromosoma. Los cromosomas se componen de genes, y estos genes generalmente se inicializan a valores aleatorios basados en boundar definidos - es decir. Cada cromosoma representa una solución completa al problema definido. El algoritmo genético debe crear la población inicial, que se compone de múltiples cromosomas o soluciones.

Cada cromosoma de la población inicial debe evaluarse. Esto se hace mediante la evaluación de su "aptitud" o la calidad de su solución. La aptitud se determina mediante el uso de una función especificada para el problema que el algoritmo genético está diseñado para resolver.

## Idoneidad y el privilegio de aparearse

En un algoritmo genético, el apareamiento se utiliza para crear una población nueva y mejorada. La "idoneidad" para aparearse se refiere a si los cromosomas están calificados o no para aparearse, o si tienen el "privilegio" de aparearse.

Determinar los cromosomas específicos que se aparearán se basa en la aptitud de cada cromosoma individual. Los cromosomas se seleccionan de la población antigua, se aparean y se producen niños, que son nuevos cromosomas. Estos nuevos niños se añaden a la población existente. La población actualizada se utiliza para la selección de cromosomas para el acoplamiento posterior.

## apareamiento

Ahora examinaremos el proceso de cruce utilizado en algoritmos genéticos para lograr el apareamiento. El apareamiento se logra seleccionando a dos padres y tomando un "empalme" de cada una de sus secuencias genéticas. Estos empalmes dividen eficazmente las secuencias genéticas de los cromosomas en tres partes. Los niños se crean a partir de genes de cada una de estas tres secciones.

El proceso de apareamiento combina genes de dos padres en nuevos cromosomas descendientes. Esto es útil en que permite que los nuevos cromosomas hereden rasgos de cada padre. Sin embargo, este método también puede conducir a un problema en el que no se introduce ninguna nueva relación genética- rial en la población. Para introducir nuevo material genético, se utiliza el proceso de mutación.

## mutación

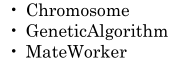
La mutación se utiliza para introducir nuevo material genético en una población. La mutación puede considerarse como experimentos naturales. Estos experimentos introducen una nueva secuencia, algo aleatoria, de genes en un cromosoma. Se desconoce completamente si esta mutación producirá o no un atributo deseable, y realmente no importa, ya que la selección natural determinará el destino del cromo mutado algunos.

Si la aptitud del cromosoma mutado es mayor que la población general, sobrevivirá y es probable que se le permita aparearse con otros cromosomas. Si la mutación genética produce una característica indeseable, entonces la selección natural asegurará que el cromosoma no viva para aparearse.

Una consideración importante para cualquier algorit genéticohm es el nivel de mutación que se utilizará. El nivel de mutación generalmente se expresa como un porcentaje. El programa de ejemplo que se examinará más adelante en este capítulo utilizará un nivel de mutación del 10%. La selección de un nivel de mutación tiene muchas ramificaciones. Por ejemplo, si eliges un nivel de mutación demasiado alto, no realizarás nada más que una búsqueda aleatoria. No habrá adaptación; en su lugar, se probará una solución completamente nueva hasta que no se pueda encontrar una mejor solución.

# Implementación de un algoritmo genético genérico

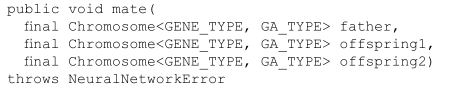
Ahora que entiendes la estructura general de un algoritmo genético, examinaremos un problema común al que a menudo se aplican. Para implementar un algoritmo genético genérico, se han creado varias clases :

* cromosoma
* GeneticAlgorithm
* CompañeroWorker

La clase **Chromosome** implementa un solo cromosoma. La clase **GeneticAlgorithm** se utiliza para controlar el proceso de entrenamiento. Implementa la interfaz **Train** y se puede utilizar para entrenar redes neuronales feedforward.

## apareamiento

El acoplamiento se realiza en la clase **Cromosome** base o en una subclase. Las sub-clases pueden implementar sus propios métodos de relación de **posición** si se necesita especialización. Sin embargo, los ejemplos de este libro no requieren nada más allá del método **de relación de posición** de la clase **Cromosome** base. Ahora examinaremos cómo funciona el método **mate.** La signa- ture para el método **mate** de la clase **cromosómica** se muestra aquí:

público void mate(

cromosoma final<GENE\_TYPE, GA\_TYPE> padre, cromosoma final<GENE\_TYPE, GA\_TYPE> descendencia1, cromosoma final<GENE\_TYPE, GA\_TYPE> descendencia2)

lanza NeuralNetworkError

El proceso de apareamiento trata los genes de un cromosoma como una larga gama de elementos. Para esta red neuronal, estos elementos serán variables **dobles** tomadas de la matriz de peso. Para el problema de los vendedores ambulantes, estos elementos representarán cit- es decir, en el que el vendedor se detendrá; sin embargo, estos elementos pueden representar cualquier cosa. Los genes Some serán tomados de la madre y algunos serán tomados del padre. Se crearán dos crías. La Figura 6.1 muestra cómo el material genético es empalmado por el método **mate.**

### Figura 6.1: Apareamiento de dos cromosomas.

**cortar punto 1 cortar punto 2**

|  |  |  |
| --- | --- | --- |
|  |  |  |
| madre | madre | madre |
|  |  |  |
| padre | padre | padre |
|  |  |  |
| padre | madre | padre |
|  |  |  |
| madre | padre | madre |
|  |  |  |

**ADN** de la madre



**ADN** de la madre



**ADN** **de la descendencia 1**



**ADN** **de Offspring 2**



Como pueden ver, se crean dos puntos de corte entre el padre y la madre. Parte del material genético se extrae de cada región, definida por los puntos de corte, para crear el material genético para cada descendencia.

En primer lugar, se determina la longitud de la matriz genética.

último gen intLength = getGenes().length;

A continuación, deben establecerse dos puntos de corte. El primer punto de corte se elige aleatoriamente. Debido a que todos los "cortes" serán de una longitud constante, el primer punto de corte no se puede elegir hasta ahora en la matriz que no queda una sección suficientemente larga para permitir que se tome la longitud de corte completa. El segundo punto de corte es simplemente la longitud de corte añadida alprimer punto de corte.

punto final de corte int1 = (int) (Math.random() \* (geneLength - getGeneticAlgorithm(). getCutLength()));

punto de corte final int2 = punto de corte1 + getGeneticAlgorithm(). getCutLength();

A continuación, se asignan dos matrices que son lo suficientemente grandes como para contener a los dos descendientes.

set final<GENE\_TYPE> taken1 = new HashSet<GENE\_TYPE>(); conjunto final<GENE\_TYPE> taken2 = nuevo HashSet<GENE\_TYPE>();

Hay tres regiones de material genético que ahora deben ser consideradas. El primero es el área entre los dos puntos de corte. Las otras dos son las áreas antes y después de los puntos de corte.

para (int i = 0; i < geneLength; i++) {

si ((i < cutpoint1) || (i > punto de corte2)) {

} else {

offspring1.setGene(i, father.getGene(i)); offspring2.setGene(i, this.getGene(i)); taken1.add(offspring1.getGene(i)); taken2.add(offspring2.getGene(i));

}

}

Ahora que se ha manejado la sección central, las dos secciones externas deben ser ad-

vestido.

manejar secciones externas

para (int i = 0; i < geneLength; i++) {

si ((i < cutpoint1) || (i > punto de corte2)) { if (getGeneticAlgorithm().

isPreventRepeat()) { offspring1.setGene(i,

getNotTaken(esto, tomado1)); offspring2.setGene(i,

getNotTaken(padre, tomado2));

} else {

offspring1.setGene(i, this.getGene(i));

offspring2.setGene(i, father.getGene(i));

}

}

}

Copie los resultados de las dos crías.

copiar resultados offspring1.calculateCost(); offspring2.calculateCost();

Se utiliza un número aleatorio para see si la primera descendencia debe mutarse.

if (Math.random() < this.geneticAlgorithm.getMutationPercent()) { offspring1.mutate();

}

Del mismo modo, se considera mutar la segunda descendencia.

if (Math.random() < this.geneticAlgorithm.getMutationPercent()) {

offspring2.mutate();

}

El proceso exacto para la mutación varía dependiendo del problema que se esté resolviendo. Para

una aplicación de red neuronal, algún valor de matriz de peso se escala en un porcentaje aleatorio de edad. Para el problema del vendedor viajero, se intercambian dos paradas aleatorias en el viaje.

## Problemas multiproceso

La mayoría de los ordenadores nuevos que se compran hoy en día son multinúcleo. Multicore proces- sors puede ejecutar programas considerablemente más rápido si están escritos para ser multithread- ed. El entrenamiento de la red neuronal,en particular, puede beneficiarse de la multiproceso. Mientras que la retropropagación puede ser algo difícil de multiprocesar, los algoritmos genéticos son bastante fáciles. La clase **GeneticAlgorithm** proporcionada en este libro está diseñada para usar un grupo de subprocesos, si se proporciona uno. Una discusión completa de las capacidades integradas de agrupación de subprocesos de Java está fuera del alcance de este libro; sin embargo, los conceptos básicos estarán cubiertos.

Para utilizar el grupo de subprocesos integrado de Java, se debe crear una clase que procese una unidad de trabajo. El verdadero desafío al escribir código que se ejecuta bien en un procesador multinúcleo es dividir el trabajo en paquetes pequeños que se pueden dividir entre los subprocesos.

Las clases que realizan estos paquetes de trabajo pequeños deben implementar la interfaz **del callable.** Para el algoritmo genético genérico, los paquetes de trabajo son realizados por la clase **MateWorker.** Los objetos **MateWorker** se crean durante cada iteración de la formación. Estos objetos se crean para cada acoplamiento que debe producirse. Una vez creados todos los objetos **MateWorker,** se entregan a un grupo de subprocesos. La clase **MateWorker** se muestra en el listado 6.1.

### Listado 6.1: Clase MateWorker (MateWorker.java)

paquete com.heatonresearch.book.introneuralnet.neural.genetic; importar java.util.concurrent.Callable;

* MateWorker: Esta clase se utiliza junto con un subproceso
* piscina. Esto permite que el algoritmo genético descargue todos los
* esos cálculos a un grupo de subprocesos.

\* @author Jeff Heaton

* @version 2,1

clase pública MateWorker<CHROMOSME\_TYPE extiende cromosoma<?, ?>> implementa Callable<Integer> {

final privado CHROMOSME\_TYPE madre; final privado CHROMOSME\_TYPE padre; chr final privadoOMOSME\_TYPE niño1; final privado CHROMOSME\_TYPE niño2;

MateWorkerpúblico(madre CHROMOSME\_TYPE final, padre CHROMOSME\_TYPE final,

último CHROMOSME\_TYPE niño1, último CHROMOSME\_TYPE niño2) { este.mother = madre;

this.father = padre; this.child1 = child1; this.child2 = child2;

}

@SuppressWarnings("sin marcar")

public Integer call() produce Exception { this.mother.mate((Cromosoma)this.father,

(Cromosoma) this.child1, (Cromosoma)this.child2);

retorno nulo;

}

}

Como se puede ver en el listado anterior, la clase se proporciona con todo lo que

necesita aparear dos cromosomas y producir dos crías. El trabajo real se realiza dentro del método **de ejecución.** La obra consiste en nada más que llamar al método **mate** de uno de los padres.

# El problema del vendedor ambulante

En esta sección se le presentará el problema del vendedor ambulante (TSP). Los algoritmos ge-netic se utilizan comúnmente para resolver el problema de los vendedores ambulantes, porque el TSP es un problema difícil de NP que generalmente no puede ser resuelto por algoritmos iterativos tradicionales.

## Comprensión del problema del vendedor ambulante

El vendedor ambulante problem involucra a un "vendedor ambulante" que debe visitar un cierto número de ciudades. La tarea es identificar la ruta más corta para que el vendedor viaje entre las ciudades. El vendedor puede comenzar y terminar en cualquier ciudad, pero debe visitar cada ciudad una vez. El vendedor puede no visitar una ciudad más de una vez.

Esto puede parecer una tarea fácil para un programa iterativo normal, sin embargo, considerar la velocidad con la que el número de combinaciones posibles crece a medida que aumenta el número de cit- es decir. Si hay una o dos ciudades, solo se requiere un paso. Tres aumenta las rutas posibles a seis. La Tabla 6.2 muestra la rapidez con la que crecen estas combinaciones.

### Tabla 6.2: Número de pasos para resolver TSP con un programa convencional

|  |  |
| --- | --- |
| **Número de** **ciudades** | **Número de** **pasos** |
| 1 | 1 |
| 2 | 1 |
| 3 | 6 |
| 4 | 24 |
| 5 | 120 |
| 6 | 720 |
| 7 | 5,040 |
| 8 | 40,320 |
| 9 | 362,880 |
| 10 | 3,628,800 |
| 11 | 39,916,800 |
| 12 | 479,001,600 |
| 13 | 6,227,020,800 |
| ... | ... |
| 50 | 3.041 \* 10^64 |

La fórmula detrás de la tabla anterior es la factorial. El número de ciudades, n, es calculado utilizando el operador factorial (!). El factorial de algún valor arbitrario n es dado por n \* (n – 1) \* (n – 2) \* ... \* 3 \* 2 \* 1. Como se puede ver en la tabla anterior, estos valores se vuelven increíblemente grandes cuando un programa debe hacer una búsqueda de "fuerza bruta". El programa de muestra que examinaremos en la siguiente sección encuentra una solución a un problema de 50 ciudades en cuestión de minutos. Esto se hace mediante el uso de un algoritmo genético, en lugar de un enfoque normal de fuerza bruta.

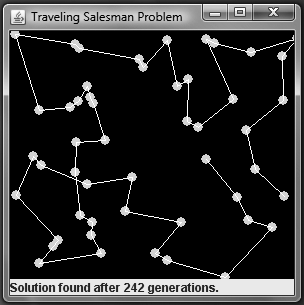
# Implementación del problema del vendedor ambulante

Hasta ahora, hemos discutido los principios básicos de los algoritmos genéticos y cómo se utilizan. Ahora es el momento de examinar un ejemplo de Java. En esta sección, se le mostrará una aplicación completa que es capaz de encontrar soluciones para el TSP. A medida que se examina este programa, se le mostrará cómo se construye la interfaz de usuario, y también cómo se implementa el algoritmo genético en sí.

## Uso del Programa de Vendedores Ambulantes

El programa de vendedores ambulantes en sí es muy fácil de usar. Este programa muestra el cities, que se muestra como puntos, y la mejor solución actual. También se muestra el número de generaciones y el porcentaje de mutación. A medida que se ejecuta el programa, estos valores se actualizan. La salida final del programa se muestra en la Figura 6.2.

### Figura 6.2: El programa de vendedores ambulantes.



A medida que el programa se está ejecutando, verá que las líneas blancas cambian entre las ciudades verdes. Con el tiempo, comenzará a surgir un camino. La ruta que se muestra actualmente está cerca de la ruta más corta de toda la población.

Cuando el programa esté casi terminado, notarás que no se introducen nuevos patrones; el programa parece estabilizar. Sin embargo, también notará que todavía se están calculando generaciones adicionales. ¡Esta es una parte importante del algoritmo genético — saber cuándo se hace! No es tan sencillo como podría parecer. Usted no sabe cuántos pasos se required, ni usted sabe la distancia más corta.

Se deben especificar criterios de terminación, por lo que el programa sabrá cuándo parar. Este programa en particular se detiene cuando la solución óptima no cambia para 100 generaciones. Una vez que esto ha sucedido, el programa indica que ha encontrado una solución después del número de generaciones indicadas, que incluye las 99 generaciones que no cambiaron la solución. Ahora que ha visto cómo funciona este programa de GA, examinaremos cómo se construyó. Comenzaremos examinando la interfaz de usuario.

## Estructura general

El programa de vendedores ambulantes utiliza cinco clases Java. Es importante mantener la relación entre las clases individuales que componen el programa de ventas ambulantes. Estas clases, y sus funciones, se resumen en la Tabla 6.3.

### Tabla 6. 3: Clases utilizadas para la versión GA del vendedor ambulante

|  |  |
| --- | --- |
| **clase** | **propósito** |
| ciudad | Esta clase almacena coordenadas de ciudad individuales. También contiene métodos que se utilizan para calcular la distancia entre ciudades. |
| GenéticaTravelingSalesman | Esta clase implementa la interfaz de usuario y la inicialización general por formularios. |
| Cromosoma TSP | Esta clase implementa el cromosoma. Es la clase más compleja del programa, ya que implementa la mayor parte de la funcionalidad del algoritmo genético |
| TSPGeneticAlgorithm | Esta clase implementa el algoritmo genético. Se utiliza para realizar el entrenamiento y procesar el chro- gemidos. |
| Mapa del mundo | Esta clase ayuda a la clase GeneticTravelingSalesman dibujando el mapa de las ciudades. |

La mayor parte del trabajo es realizado por la clase **TSPChromosome.** Esta clase se cubre en la siguiente sección.

## Cromosomas vendedor ambulantes

Al implementar un algoritmo genético utilizando las clases proporcionadas en este libro, generalmente debe crear su propio método de cálculo de costos, denominado **calculateCost**, así como su propia función de mutación, denominada **mutate**. La firma para el método **calculateCost** del problema del vendedor de viajes se muestra aquí:

el vacío público calculateCost() lanza NeuralNetworkError

Calcular el costo del problema de los vendedores ambulantes es relativamente fácil; se resume la des-tance entre cada una de las ciudades. El programa comienza inicializando una variable de coste de ejecución a cero y recorriendo toda la lista de ciudades.

costo doble = 0.0;

para (int i = 0; i < this.cities.length - 1; i++) {

Para cada ciudad, se calcula la distancia entre esta ciudad y la siguiente.

último doble dist = this.cities[getGene(i)]

.proximity(this.cities[getGene(i + 1)]);

La distancia se añade al coste total.

costo += dist;

}

Por último, el coste se guarda en una variable de instancia.

setCost(costo);

También se proporciona un método mutado. La firma para el método **mutado** se muestra aquí:

vacío público mutado()

En primer lugar, se obtiene la longitud y se eligen dos ciudades aleatorias para ser intercambiadas.

longitud final int = this.getGenes().longitud;

final int iswap1 = (int) (Math.random() \* longitud); final int iswap2 = (int) (Math.random() \* longitud);

Las dos ciudades son entonces intercambiadas.

temperatura entera final = getGene(iswap1); setGene(iswap1, getGene(iswap2)); setGene(iswap2, temp);

Elcódigo anterior muestra cómo se amplió el algoritmo genético genérico proporcionado en este libro para resolver un problema general como el vendedor ambulante. En la siguiente sección, se proporcionarán clases que le permitirán utilizar el algoritmo genérico para entrenar una red neuronal utilizando conjuntos de entrenamiento en lugar de backpropagation.

# Operador XOR

En el último capítulo, se utilizó backpropagation para resolver el problema del operador XOR. En esta sección, verá cómo se puede utilizar un algoritmo genético, la clase **TrainingSetNeuralG** **eneticAlgorithm,** para entrenar para el operador XOR. Esta versión del solucionador XOR se puede ver en el listado 6.2.

### Listado 6.2: XOR con un algoritmo genético (GeneticXOR.java)

paquete com.heatonresearch.book.introneuralnet.ch6.xor;

importar com.heatonresearch.book.introneuralnet.neural.feedforward.

FeedforwardLayer;

importar com.heatonresearch.book.introneuralnet.neural.feedforward.

FeedforwardNetwork;

importar com.heatonresearch.book.introneuralnet.neural.feedforward.

train.genetic. TrainingSetNeuralGeneticAlgorithm;

/\*\*

* Capítulo 6: Entrenamiento usando un algoritmo genético

\*

* XOR: Aprende el patrón XOR con una red neuronal feedforward
* que utiliza un algoritmo genético.

\*

* @author Jeff Heaton
* @version 2,1

\*/

clase pública GeneticXOR {

doble estática pública XOR\_INPUT[][] = { { 0.0, 0.0 },

{ 1.0, 0.0 },

{ 0.0, 1.0 }, { 1.0, 1.0 } };

doble estática pública XOR\_IDEAL[][] = { { 0.0 }, { 1.0 },

{ 1.0 }, { 0.0 } };

public static void main(final String args[]) { FeedforwardNetwork network = new FeedforwardNetwork(); network.addLayer(nuevo FeedforwardLayer(2)); network.addLayer(nuevo FeedforwardLayer(3)); network.addLayer(nuevo FeedforwardLayer(1)); network.reset();

entrenar la red neuronal

formación finalSetNeuralGeneticAlgorithm tren = nuevo TrainingSetNeuralGeneticAlgorithm(

red, cierto, XOR\_INPUT, XOR\_IDEAL, 5000, 0,1, 0,25);

int epoch = 1;

hacer {

train.iteration();

System.out

. println("Epoch #" + epoch + " Error:" + train.getError());

epoch++;

} mientras que ((epoch < 5000) && (train.getError()

> 0.001));

red = train.getNetwork();

probar la red neuronal

System.out.println ("Resultados de la red neuronal:"); para (int i = 0; i < XOR\_IDEAL.length; i++) {

final doble real[] = red. computeOutputs(XOR\_INPUT[i]);

System.out.println(XOR\_INPUT[i][0] + ","

+ XOR\_INPUT[i][1]

+ ", real=" + real[0] + ", ideal=" + XOR\_IDEAL[i][0]);

}

}

}

La lista anterior es muy similar al solucionador XOR del último capítulo. La principal diferencia está en las siguientes líneas:

formación finalSetNeuralGeneticAlgorithm tren = nuevo TrainingSetNeuralGeneticAlgorithm(

red, cierto, XOR\_INPUT, XOR\_IDEAL, 5000, 0,1, 0,25);

Como puede ver, varios parámetros se pasan al constructor de la clase de entrenamiento. En primer lugar, se pasa la red a entrenar. El valor de **true** especifica que todos los formularios de vida iniciales deben inicializar aleatoriamente sus matrices de peso. También se proporcionan las matrices de entrada e ideales, tal como están con la backpropagation al-gorithm. El valor de 5000 especifica el tamaño de la población. La valoracióne 0,1 indica que el 10% de la población será elegida para aparearse. El 10% será capaz de aparearse con cualquier forma de vida en el 25% superior.

**Calcular coste**

A continuación, se utiliza un método especializado para calcular el coste del patrón XOR recog- nition. La firma para el método de cálculo de costes se muestra aquí:

public void calculateCost() lanza NeuralNetworkError {

En primer lugar, el contenido del cromosoma se copia de nuevo en la red neuronal.

this.updateNetwork();

El error cuadrado medio raíz de la red neuronal se puede utilizar como un costo. El error RMS se calculará tal como estaba en capítulos anteriores. Las matrices **de entrada** e **ideales** se copian en variables locales.

entrada doble final[][] = this.getGeneticAlgorithm(). getInput();

doble ideal final[][] = this.getGeneticAlgorithm(). getIdeal();

La función **calculateError** de la red neural se utiliza para calcular el error de la red neuronal dada la **entrada** y las matrices ideales.

setCost(getNetwork(). calculateError(entrada, ideal));

Aunque el programa XOR utiliza un método de cálculo de costes especializado, el pro-gram utiliza el mismo método de acoplamiento que se había discutido anteriormente.

## mutar

La mutación es manejada por una función **mutada** especializada. La firma de esta función mutada se muestra aquí:

vacío público mutado()

El método de mutación comienza obteniendo el número de genes en un cromosoma.

Cada gen se escalará utilizando un porcentaje aleatorio.

longitud final int = getGenes().length;

La función recorre todos los genes.

para (int i = 0; i < longitud; i++) {

Obtiene el valor de un gen y lo multiplica por una proporción aleatoria dentro de un rango especificado.

doble d = getGene(i);

relación doble final = (int) ((RANGE \* Math.random()) - RANGE);

d\*=relación; setGene(i,d);

}

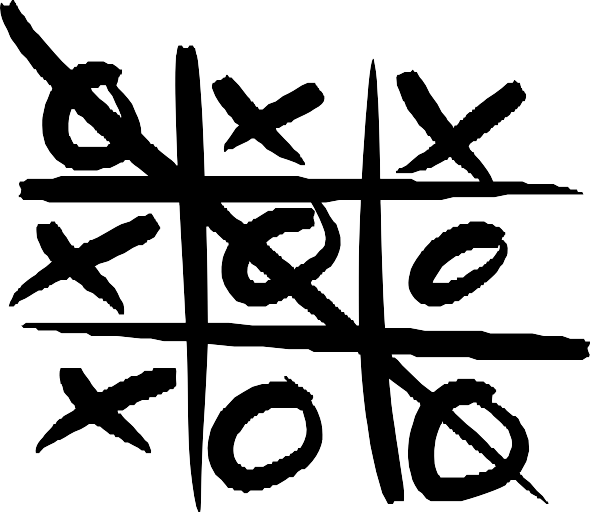
El resultado es que cada elemento de matriz de peso se cambia aleatoriamente.

# Tic-tac-teo

El juego de tic-tac-toe, también llamado círculos y cruces en muchas partes del mundo, puede ser una aplicación interesante de redes neuronales entrenadas por algoritmos genéticos. Tic-tac-toe tiene reglas muy simples. La mayoría de los jugadores humanos rápidamente se aburren con el juego, ya que es tan fácil de aprender. Una vez que dos personas han dominado el dedo del pie tic-tac, los juegos casi siempre resultan en un empate.

Las reglas simples para tic-tac-toe son las siguientes: Un jugador juega X y otro juega O. Se turnan para colocar a estos personajes en una cuadrícula 3x3 hasta que un jugador obtiene tres seguidos. Si la cuadrícula se rellena antes de que uno de ellos obtenga tres en una fila, entonces el juego es un empate. La Figura 6.3 muestra un juego de tic-tac-toe en progreso.

### Figura 6.3: El juego de tic-tac-toe.



Hay muchas implementaciones de tic-tac-toe en Java. Este libro hace uso de uno de Thomas David Baker, que fue lanzado como software de código abierto. Proporciona varios jugadores que se pueden emparejar para los juegos:

* + Boring: Sólo tiene que elegir el siguiente lugar abierto.
  + Human: Permite que un humano juegue.
  + Logical : utiliza la lógica para jugar un juego casi perfecto.
  + MinMax: Utiliza el algoritmo min-max para jugar un juego perfecto.
  + Random: se mueve a ubicaciones aleatorias.

Cada jugador puede jugar con cualquier otro jugador. Una clase simplemente tiene que implementar la interfaz **del jugador** y puede jugar contra los demás. Algunos de los jugadores están más avanzados. Esto le da a la red neuronal varios niveles de jugadores contra los que jugar.

El jugador más advanced es el jugador min-max. Este reproductor utiliza un min-max algo- rithm. Un algoritmo min-max utiliza un árbol para trazar todos los movimientos posibles. Esta técnica se puede utilizar en un juego tan simple como tic-tac-toe; sin embargo, no sería eficaz para un juego con un gran número de combinaciones, como el ajedrez o ir.

El ejemplo de este capítulo crea una clase denominada **PlayerNeural**. Esta clase utiliza una red neuronal para jugar contra los otros jugadores. El jugador neural no jugará un juego perfecto. Sin embargo, jugará razonablemente bien contra algunos de los jugadores proporcionados.

El objetivo no es producir un jugador perfecto utilizando sólo una red neuronal, sino demostrar el uso de redes neuronales. Muchos juegos utilizan un enfoque híbrido para opti- resultados incorrectos. El algoritmo min-max, junto con una red neuronal para podar ramas enteras del árbol de búsqueda puede ser una combinación muy eficaz. Sin embargo, implementaremos un enfoque de red neuronal pura.

El ejemplo tic-tac-toe ilustra un concepto muy importante. Hasta ahora, sólo hemos entrenado redes neuronales usando conjuntos de entrenamiento. El ejemplo tic-tac-toe no utilizará conjuntos de entrenamiento. Más bien, la red neuronal se emparejará con cualquiera de los otros jugadores y se utilizará un algoritmo genético para entrenarla. Dado que un algoritmo genético utiliza la "selección natural" es particularmente adecuado para aprender a jugar un juego.

**Uso de la** implementación de **Tic-Tac-Toe de** **muestra**

Hay muchas opciones disponibles con esta versión de tic-tac-toe. Hay varias opciones de línea de comandos que le permiten especificar qué dos jugadores jugarán entre sí. El formato general de un comando es el siguiente:

NeuralTicTacToe [Comando] [Jugador 1] [Jugador 2]

El comando también especifica el modo en el que debe ejecutarse el programa. Los modos fol- lowing están disponibles:

* + juego – Jugar un solo juego.
  + partido : juega 100 juegos.
  + tren – Entrenar (y salvar) una red neuronal.

Debes elegir a los dos jugadores. Están disponibles las siguientes opciones:

* + aburrido
  + Humano
  + lógica
  + MinMax
  + NeuralBlank
  + Carga neuronal
  + aleatorio

Por ejemplo, para reproducir una coincidencia entre el algoritmo MinMax y el random player, utilice el siguiente comando:

NeuralTicTacToe coinciden con MinMax random

Si desea entrenar una nueva red neuronal en blanco contra el jugador aleatorio, utilizaría el siguiente comando:

NeuralTicTacToe Tren NeuralBlank Random

Esto entrenará una red neuronal en blanco; pero ten en cuenta que puede tomar una cantidad considerable de tiempo. El algoritmo genético utilizará un grupo de subprocesos, por lo que una computadora multinúcleo ayudará. Una vez completado el entrenamiento, la red neuronal se guardará en el disco. Se guardará con el nombre "tictactoe.net". La descarga de este libro contiene un ejemplo de archivo "tictactoe.net" que ya está entrenado para tic-tac-toe. Esta red neuronal tardó casi 20 horas en entrenarse en mi computadora. El Apéndice A explica cómo descargar los ejemplos de este libro.

Para reproducir una red neuronal entrenada, utilice el siguiente comando:

NeuralTicTacToe Jugar NeuralLoad Humano

La red neuronal de muestra siempre cargará y guardará una red neuronal llamada "tictactoe.net".

## Salvar y cargar redes neuronales

Es posible que se pregunte cómo cargar y guardar las redes neuronales en este libro. Estas redes se pueden cargar y guardar utilizando técnicas regulares de serialización Java. Por ejemplo, el siguiente comando guarda una red neuronal:

SerializeObject.save("filename.net", neuralNetwork);

Cargar una red neuronal es casi tan fácil. El siguiente comando carga una red neuronal desde el disco.

Resultado de FeedforwardNetwork = (FeedforwardNetwork)SerializeObject.load("neuralnet.net");

Algunas de las redes neuronales de este libro tardan una cantidad considerable de tiempo en entrenarse. Por lo tanto, es valioso guardar la red neuronal para que pueda ser recargada rápidamente más tarde.

## Estructura de la Red Neuronal Tic-Tac-Toe

Una de las preguntas más importantes que un programador de red neuronal debe considerar es cómo estructurar la red neuronal. Tal vez la forma más obvia de estructurar una red neuronal para tic-tac-toe es con nueve entradas y nueve salidas. Las nueve entradas especificarán la configuración actual de la placa. Las nueve salidas permitirán a la red neuronal especificar dónde desea moverse.

La primera versión del ejemplo utilizó esta configuración. No funcionó particularmente bien. Un problema es que la red neuronal tuvo que pasar mucho tiempo aprendiendo los movimientos válidos e inválidos. Además, esta configuración no jugó muy bien contra los otros jugadores.

La versión final de la red neuronal tiene una configuración con nueve entradas y una sola salida. En lugar de preguntarle a la red neuronal dónde moverse, esta estructura pregunta a la red neuronal si una posición propuesta de la junta es favorable. Siempre que sea el turno del jugador neural, se determina cada movimiento posible. Puede haber como máximo nueve movimientos posibles. Una tabla temporal está estructurada que indica cómo sería el tablero de juego después de cada uno de estos posibles movimientos. A continuación, cada placa se presenta en la red neuronal. La posición de la placa que recibe el valor más alto de la neurona de salida será el siguiente movimiento.

## Jugador neural

La interfaz **Player** requiere que la clase **NeuralPlayer** incluya un método **getMove.** Este método determina el siguiente movimiento que hará el jugador neural. La firma para el método **getMove** se muestra aquí:

public Move getMove(byte final[][] tablero, final Move prev, colorde byte final)

En primer lugar, se establecen dos variables locales. La **mejor variableMove** mantendrá el mejor movimiento encontrado hasta ahora. La variable **bestScore** mantendrá la puntuación de la mejor **variableMove.**

Mover bestMove = null;

double bestScore = Double.MIN\_VALOR;

A continuación, se examinan todos los movimientos potenciales en la cuadrícula de la placa.

for (int x = 0; x < board.length;x++) { for (int y = 0; y < board.length; y++) {

Se construye un tablero de ejemplo para representar este movimiento potencial.

movimiento final = nuevo Move((byte) x, (byte) y, color);

El movimiento potencial se examina para determinar si es válido.

if (Board.isEmpty(placa, mover)) {

Si el movimiento potencial es válido, se llama al método **tryMove** para ver qué puntuación proporcionaría su posición en el tablero.

final doble d = tryMove(tablero, movimiento);

Si la puntuación para esa posición de tablero supera la mejor puntuación actual, entonces este movimiento se guarda como la mejor jugada encontrada hasta ahora.

si ((d > bestScore)|| (bestMove == null)) { bestScore = d;

bestMove = mover;

}

}

}

}

Por último, se devuelve el mejor movimiento.

devolver bestMove;

El método **getMove** hace uso del método **tryMove** para crear una placa temporal para cada movimiento posible. La firma para el método **tryMove** se muestra aquí:

doble tryMoveprivado(byte final[][] tablero, movimiento final de movimiento) {

En primer lugar, se construye una matriz de **entrada** para la red neuronal. Una variable local

índice que recordará la posición actual dentro de la matriz de **entrada.**

entrada doble final[] = nuevo doble[9]; int index = 0;

A continuación, se consideran todas las posiciones de la cuadrícula de la placa.

for (int x = 0; x < board.length;x++) { for (int y = 0; y < board.length; y++) {

Cada cuadrado de la cuadrícula está marcado y la matriz de entrada está configurada para reflejar el estado de ese cuadrado.

si (board[x][y] == TicTacToe.NOUGHTS) { entrada[index] = -1;

} si (board[x][y] == TicTacToe.CROSSES) { entrada[index] = 1;

} si (board[x][y] == TicTacToe.EMPTY) { input[index] = 0;

}

Si el cuadrado contiene una "X" (cruz), se inserta un valor de –1 en la matriz de entrada. Si el cuadrado contiene una "O" (nought), a continuación, se coloca un valor de 1 en la matriz.

Si el cuadrado contiene el movimiento actual, la entrada se establece en –1 o un valor de "X", que es lo que está reproduciendo el reproductor neural.

if ((x == move.x) && (y == move.y)) { input[index] = -1;

}

A continuación, se examina el siguiente elemento de la matriz de entrada.

index++;

}

}

Por último, se calcula la salida de esta matriz de entrada.

salida doble final[] = this.network.computeOutputs(entrada); salida de retorno[0];

La salida se devuelve al llamador. Cuanto mayor sea este valor, más favorable será la posición de la placa.

## cromosomas

La red neuronal tic-tac-toe está entrenada usando un algoritmo de entrenamiento genético. Cada cromosoma se prueba jugando 100 partidos contra su oponente. Esto hace que se genere una puntuación que permita evaluar la eficacia de cada cromosoma. Cada cromosoma tiene un método **calculateCost** que realiza esta operación. El sig- naturaleza para **calculateCost** se muestra aquí:

público void calculateCost() {

En primer lugar, la red neuronal se actualiza mediante la matriz genética.

pruebe { este.updateNetwork();

A continuación, se construye un jugador de **PlayerNeural** para jugar contra el com- ponent elegido. Para mantener este ejemplo simple, el reproductor de red neuronal siempre es el jugador uno, y por lo tanto siempre se mueve primero.

jugador finalNeural1 = nuevo PlayerNeural(getNetwork());

Se crea el segundo jugador y se juega un partido.

Jugador jugador2;

player2 = (Player) this.getGeneticAlgorithm(). getOpponent()

. newInstance();

puntuación final scorePlayer = nuevo ScorePlayer(player1, player2, false);

La coincidencia es administrada por la clase **ScorePlayer.** Esta clase permite a dos jugadores jugar 100 partidos y se genera una puntuación.

setCost(score.score());

La puntuación se convierte entonces en el costo de este cromosoma.

# Resumen del capítulo

En este capítulo, se le introdujo en algoritmos genéticos. Algoritmos genéticos pro-vide un enfoque para encontrar soluciones potenciales a problemas complejos de NP-hard. Un problema difícil de NP es un problema para el que el número de pasos necesarios para resolver el problema aumenta a un ritmo muy alto a medida que aumenta el número de unidades en el programa.

Un ejemplo de un problema difícil de NP, que se examinó en este capítulo, es el problema de los vendedores ambulantes. El problema de los vendedores ambulantes intenta identificar el camino más corto para un vendedor que viaja a un cierto número de ciudades. El número de posibles rutas de acceso que un programa tiene que buscar aumenta factorialmente a medida que aumenta el número de cit- es decir.

Para resolver un problema de este tipo, se utiliza un algoritmo genético. El algoritmo genético crea una población de cromosomas. Cada uno de los cromosomas es un camino a través de las ciudades. Cada pierna en ese viaje es un gen. Los mejores cromosomas están determinados y se les permite "aparearse". El proceso de apareamiento combina los genes de dos padres. El chro- gemidos que tienen caminos más largos, menos deseables, no se permiten aparearse. Debido a que la población tiene un tamaño fijo, los cromosomas menos deseables se purgan de la memoria. A medida que el programa continúa, la selección natural hace queloscromosomas más adecuados se apareen y produzcan mejores y mejores soluciones.

El proceso real de apareamiento se produce dividiendo los cromosomas primarios en tres empalmes. Estos empalmes se utilizan entonces para construir nuevos cromosomas. El resultado de todo esto será doscromosomas de descendencia. Desafortunadamente, el proceso de apareamiento no intro- duce nuevo material genético. El nuevo material genético se introduce a través de la mutación.

La mutación cambia aleatoriamente los genes de algunas de las crías recién creadas. Esto introduce nuevos rasgos. Muchas de estas mutaciones no serán adecuadas para el particu- problema de lar y serán purgados de la memoria. Sin embargo, otros pueden ser utilizados para promover unapoblación estancada. También se introduce mutación para ayudar a encontrar una solución óptima.

Hasta ahora en este libro se le ha mostrado cómo entrenar redes neuronales con backpropagation y algoritmos genéticos. Las redes neuronales también pueden ser entrenadas por un tecnique que simula la forma en que un metal fundido se enfría. Este proceso se llama simu- recocido latado. El recocido simulado se cubrirá en el siguiente capítulo.

# vocabulario

Gen de evolución cromosómica

compañero

Algoritmo Min-Max Multicore

Mutar la piscina de hilos de población

# Preguntas para revisión

1. ¿Cuál es el papel de un cromosoma en un algoritmo genético? ¿Cuál es el papel de un gen?
2. ¿Cómo se puede expresar una red neuronal como cromosoma y genes?
3. ¿Cuál es el papel de la mutación en un algoritmo genético?
4. ¿Cómo se puede calcular el costo para una posible solución de vendedor ambulante? ¿Para una red neuronal?
5. ¿Para qué tipos de problemas son los algoritmos genéticos más adecuados? ¿Para qué tipos de problemas no son adecuados?