Inteligencia Artificial

Dr. Alfonso Alba Cadena
fac@galia.fc.uaslp.mx

Facultad de Ciencias
UASLP

Unidad I

Introducción a la Inteligencia Artificial

Qué es la Inteligencia Artificial?

- Es un conjunto de técnicas que se utilizan para resolver problemas cuya solución requiere, aparentemente, inteligencia humana.

- Estudia cómo lograr que las máquinas realicen tareas que, por el momento, son realizadas mejor por los humanos.

Algunos problemas en los que se aplica IA

- **Percepción:** visión computacional, reconocimiento de patrones, reconocimiento de voz.
- **Lenguaje natural:** comprensión, generación, traducción.
- **Juegos:** ajedrez, damas, etc.
- **Ingeniería:** detección de fallas, diseõ, planificación.

Algunos problemas "de juguete"

- **Jarras de agua:** Se tienen dos jarras de tres y cuatro litros de capacidad, pero sin marcas de medición. Las jarras se pueden llenar de un grifo. Se desea tener 2 litros de agua en la jarra de 4 litros.

- **Misioneros y caníbales:** 3 misioneros y 3 caníbales desean cruzar el río usando una balsa para 2 pasajeros. En ningún momento puede haber mas caníbales que misioneros en cualquier orilla del río.

- **Problema de las n reinas:** Se desea colocar n reinas en un tablero de ajedrez de $n \times n$ casillas de manera que no se amenacen entre ellas.

- **3-en-raya:** Desarrollar un programa para jugar al 3-en-raya contra la computadora de manera que ésta nunca pierda.

- **Reconocimiento de cuadriláteros:** Se tiene un dibujo en blanco y negro y se desea saber si éste representa un cuadrilátero.

Definición del problema

- El primer paso para resolver un problema mediante IA consiste en definirlo con precisión, incluyendo las situaciones finales que se aceptarían como soluciones.

- Es necesario analizar el problema para determinar:

  1. Una manera adecuada de representar y evaluar las posibles soluciones.

  2. Una manera de adecuada de representar y utilizar el conocimiento que se tiene para resolver el problema.

  3. Una técnica adecuada de IA para solucionar el problema.

Definición del problema mediante búsquedas en espacios de estados

- El **espacio de estados** de un problema es el conjunto de todas las posibles configuraciones de las variables que describen una situación (real o imposible) del problema.

- Las situaciones en las que comienza el problema se conocen como **estados iniciales**

- Los **estados objetivo** son aquellos que representan soluciones aceptables.

- Existe un conjunto de reglas que representan las acciones que se pueden llevar a cabo para ir de un estado a otro.

Ejercicios:

1. Describa el espacio de estados, así como los estados iniciales y objetivo, para los siguientes problemas:

   a. Misioneros y caníbales
   b. n-reinas
   c. 3-en-raya

2. Describa las reglas para ir de un estado a otro para los siguientes problemas:

   a. Misioneros y caníbales
   b. 3-en-raya

3. Considere el problema de reconocimiento de cuadriláteros. Una manera de representar los estados es mediante una cuádrupla $Q = (p_1, p_2, p_3, p_4)$ de puntos en dos dimensiones tomados de los puntos que forman el dibujo. Cómo se puede determinar si un estado particular Q corresponde a una solución del problema?

Sistemas de producción

- Muchos problemas de IA pueden resolverse mediante la búsqueda de la solución en el espacio de estados. Este proceso de búsqueda se realiza mediante un **sistema de producción.**

- Un sistema de producción consiste en:

  1. Un conjunto de reglas que describen los casos y la manera en que se aplican las operaciones válidas sobre los estados.

  2. Una o más bases de datos que representen de manera adecuada el conocimiento apropiado para resolver el problema.

  3. Una estrategia de control que especifique el orden en que se aplican las reglas, de acuerdo a las bases de datos.

  4. Un aplicador de reglas.

Soluciones: rutas y estados

- Una **ruta** es la secuencia de reglas que hay que aplicar para llegar de un estado a otro.

- En algunos problemas, la solución consiste en encontrar algún estado objetivo.

- En otros casos, los estados objetivos se conocen con anticipación, y la solución consiste mas bien en una ruta que lleva del estado inicial a algún objetivo.

Estrategias de control

- Una buena estrategia de control debe cumplir con los siguientes requisitos:

  1. Debe causar algún cambio en el estado actual.

  2. Debe ser sistemática, de manera que se produzca un cambio a nivel global.

Búsqueda aleatoria

- La búsqueda aleatoria puede realizarse de dos maneras:

  (a) Elegir de manera aleatoria un estado del espacio y evaluarlo para ver si corresponde a un estado objetivo. De lo contrario, repetir el proceso.

  (b) Aplicar una regla elegida al azar al estado actual y evaluarlo para ver si corresponde a un estado objetivo. De lo contrario, repetir el proceso. En caso de quedar estancado, generar un nuevo estado inicial de manera aleatoria.

- Ventajas: fácil de implementar, no puede quedarse estancado

- Desventajas: no es sistemático, no garantiza una solución

Búsqueda exahustiva

- Las técnicas de búsqueda exhaustiva recorren de manera sistemática todo el espacio de estados.

- Existen básicamente dos técnicas:
  - Búsqueda primero en anchura
  - Búsqueda primero en profundidad

- Ventajas: en teoría, siempre encuentran una solución (si existe)

- Desventajas: la búsqueda puede tomar mucho tiempo en espacios de gran tamaño

Búsqueda primero en anchura

1. Inicializar una lista llamada `LISTA_NODOS` con el estado inicial

2. Hasta encontrar un objetivo o `LISTA_NODOS` esté vacía, hacer:
   (a) Extraer el primer elemento `E` de `LISTA_NODOS`
   (b) Para cada regla que sea aplicable al estado `E`, hacer:
      i. Aplicar la regla para obtener un nuevo estado `NE`
      ii. Si `NE` es estado objetivo, terminar y devolver `NE`. De lo contrario, añadir `NE` al final de `LISTA_NODOS`

Ventajas:

- Si la solución es una ruta, encuentra la más corta.
- No queda atrapada en callejones sin salida.
- El orden en que se aplican las reglas es irrelevante.

Búsqueda primero en profundidad

1. Si el estado actual `E` es objetivo, terminar y devolverlo como solución exitosa

2. En caso contrario, hacer lo siguiente:

   (a) Aplicar la siguiente regla a `E` para generar un nuevo estado `NE`. Si no se pueden aplicar más reglas, terminar y devolver un fracaso.

   (b) Llamar recursivamente al algoritmo de búsqueda primero en profundidad con `NE` como estado actual. Si se devuelve una solución exitosa, entonces terminar y devolver la solución. De lo contrario, continuar con el ciclo.

Ventajas:

- Requiere menos memoria, ya que solo se almacenan los nodos de la ruta que se sigue en ese momento.

Ejercicios:

- Resuelva el problema de las n-reinas (e.g., para $n = 5, 6, 7$) utilizando las técnicas de búsqueda aleatoria, búsqueda primero en anchura, y búsqueda primero en profundidad. Cuál de estas técnicas funciona mejor para este problema? (en términos del número de estados generados para encontrar la solución).

- Resuelva el problema de misioneros y caníbales utilizando las técnicas de búsqueda aleatoria, búsqueda primero en anchura, y búsqueda primero en profundidad. Tome en cuenta que la aplicación de una regla puede resultar en un estado anterior y trate de evitar esta situación. Qué técnica funciona mejor para este problema?

Unidad I.5

Estructuras dinámicas utilizando la librería de plantillas estándard (STL) de C++

Introducción

- La librería de plantillas estándard (STL) de C++ implementa las estructuras de datos dinámicas (llamadas *contenedores*) más comunes mediante plantillas de clases, lo cual permite crear estructuras cuyos elementos son de cualquier tipo.

- Las principales ventajas de la STL son:
  - Evita la necesidad de manejar apuntadores
  - El manejo de memoria lo realiza la clase, de manera transparente al programador
  - Permite acceso a cualquier elemento (la eficiencia en el acceso depende del tipo de estructura)
  - Permite acceso secuencial a los elementos mediante iteradores

Algunos tipos de contenedores en la STL

Plantilla de clase	**Características**
Contenedores de secuencia	
`vector`	Acceso a cualquier elemento. Inserción y eliminación rápidas al final.
`deque`	Acceso a cualquier elemento. Inserción y eliminación rápidas al principio y al final.
`list`	Lista doblemente ligada (recorrible en ambas direcciones). Inserción y eliminación rápida en cualquier posición.
Adaptadores de contenedor	
`stack`	LIFO (último en entrar, primero en salir)
`queue`	FIFO (primero en entrar, primero en salir)
`priority_queue`	El elemento de mayor prioridad es el primero en salir

Algunas funciones de contenedores secuenciales

Método	Descripción
`back()`	Devuelve el valor del último elemento
`push_back(t)`	Inserta el elemento `t` al final
`pop_back()`	Elimina el último elemento
`front()`	Devuelve el valor del primer elemento
`push_front(t)`	Inserta el elemento `t` al principio
`pop_front()`	Elimina el primer elemento
`size()`	Devuelve el número de elementos en la estructura
`empty()`	Indica si el contenedor está vacío
`clear()`	Elimina todos los elementos

Algunas funciones de adaptadores de contenedor

- Métodos para la plantilla `stack` (pila)

Método	**Descripción**
`top()`	Devuelve el valor del elemento en el tope de la pila
`push(t)`	Inserta el elemento `t` en el tope de la pila
`pop(t)`	Extrae el elemento en el tope de la pila
`empty()`	Indica si la pila está vacía

- Métodos para la plantilla `queue` (cola)

Método	**Descripción**
`top()`	Devuelve el valor del elemento al principio de la cola
`push(t)`	Inserta el elemento `t` al final de la cola
`pop(t)`	Extrae el elemento al principio de la cola
`empty()`	Indica si la cola está vacía

Ejemplo

```
#include <iostream>
#include <stack>
#include <queue>

using namespace std;

int main() {
    int i;
    stack<int> s;
    queue<int> q;

    for (i = 0; i < 10; i++) {
        s.push(i);
        q.push(i);
    }

    cout << "Elementos en la cola: ";
    while (!q.empty()) {
          cout << q.front() << " ";
          q.pop();
    }

    cout << endl << "Elementos en la pila: ";
    while (!s.empty()) {
          cout << s.top() << " ";
          s.pop();
    }
    cout << endl << endl;

    system("pause");
    return 0;
}
```

Iteradores

- Un **iterador** es un objeto que toma el papel de un apuntador a un elemento de la estructura. De hecho, los iteradores sobrecargan los operadores para que puedan utilizarse de la misma forma que un apuntador a un elemento de un arreglo.

- Algunos de los operadores que pueden utilizarse con los iteradores son:

Operador	**Descripción**
`*p`	Dereferencia el iterador (accede al dato asociado al elemento)
`++p, p++`	Pre/Post-incrementa el iterador (pasa al siguiente elemento)
`p += n, p = p+n`	Incrementa el iterador en `n` posiciones
`--p, p--, p -= n`	Decrementan el iterador en una o más posiciones
`p1 = p,`	Asigna un iterador a otro

`p1 == p, p1 != p`
`p1 > p, p1 < p,`
etc.
	Compara dos iteradores (no los datos a los que apuntan)

- Es importante mencionar que no todas las operaciones pueden aplicarse a cualquier iterador (depende del tipo de acceso permitido por la estructura).

Funciones para el recorrido de contenedores

Método	Descripción
`begin()`	Devuelve un iterador apuntando al inicio de la secuencia
`end()`	Devuelve un iterador apuntando al final de la secuencia
`rbegin()`	Devuelve un `reverse_iterator` apuntando al primer elemento de la secuencia invertida
`rend()`	Devuelve un `reverse_iterator` apuntando al final de la secuencia invertida
`insert(pos, x)`	Inserta el elemento `x` antes del iterador `pos`
`erase(pos)`	Elimina el elemento referenciado por el iterador `pos`

Ejemplo

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    int i;
    vector<int> v;

    for (i = 0; i < 10; i++) v.push_back(i);

    cout << "Recorrido hacia adelante: ";
    vector<int>::iterator it;
    for (it = v.begin(); it < v.end(); it++) {
        cout << *it << "  ";
    }
    cout << endl << "Recorrido hacia atras: ";
    vector<int>::reverse_iterator rit;
    for (rit = v.rbegin(); rit < v.rend(); rit++) {
        cout << *rit << "  ";
    }

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Unidad II

Técnicas de búsqueda heurística

Búsqueda Heurística

- Una **heurística** es una técnica que aumenta la eficiencia de un proceso de búsqueda, posiblemente sacrificando completez.

- Una **función heurística** es una medida de la deseabilidad de un estado.

- El propósito de una función heurística es guiar el proceso de búsqueda hacia los estados más provechosos.

Escalada simple

1. Evaluar el estado inicial. Si es estado objetivo, devolverlo y terminar. En caso contrario, continuar con el estado inicial como estado actual.

2. Repetir hasta que se encuentre una solución, o hasta que no queden reglas que aplicar al estado actual:

   (a) Seleccionar una regla que no haya sido aplicada anteriormente al estado actual y aplicarla para generar un nuevo estado.

   (b) Evaluar el nuevo estado. Si es estado objetivo, devolverlo y terminar.

   (c) Si el nuevo estado es mejor que el estado actual, entonces convertirlo en el estado actual.

Escalada por la máxima pendiente

1. Evaluar el estado inicial. Si es estado objetivo, devolverlo y terminar. En caso contrario, continuar con el estado inicial como estado actual.

2. Repetir hasta que se encuentre una solución, o hasta que una iteración completa no produzca cambios en el estado actual:

   (a) Sea `SUCC` el peor estado posible.

   (b) Para cada regla aplicable al estado actual, hacer lo siguiente:

      i. Aplicar la regla para generar un nuevo estado.

      ii. Evaluar el nuevo estado. Si es objetivo, devolverlo y terminar.

      iii. Si el nuevo estado es mejor que `SUCC`, hacer `SUCC` igual al nuevo estado.

   (c) Si `SUCC` es mejor que el estado actual, hacer el estado actual igual a `SUCC`.

Algunos problemas que pueden resolverse mediante heurísticas

- **Comerciante viajero:** Dados N puntos en un espacio, encontrar la ruta de menor costo (distancia) que recorra todos y cada uno de los puntos una única vez.

- **Problema de la mochila:** Se tiene una mochila con una capacidad de carga de W unidades, y se tienen varios objetos de los cuales se conoce su utilidad u_j y su peso w_j. El problema consiste en encontrar la combinación de objetos que maximize la utilidad total, sin que el peso total sobrepase la capacidad de la mochila.

- **Problema de la partición:** Dado un conjunto de números $S = \{x_1, x_2, \ldots, x_n\}$, encontrar un subconjunto de $H \subset S$ tal que la suma de los elementos de H sea igual a la mitad de la suma de todos los elementos de S.

Ejercicios

1. Diseñe una función heurística para el problema de la mochila.

2. Considere un caso de prueba donde la capacidad de la mochila es $W = 8kg$ y las propiedades de los distintos objetos son:

$$\begin{array}{lllll} u_1 = 5 & u_2 = 3 & u_3 = 7 & u_4 = 3 & u_5 = 4 \\ w_1 = 3 & w_2 = 1 & w_3 = 5 & w_4 = 2 & w_5 = 2 \end{array}$$

   a) Resuelva el problema mediante escalada por la máxima pendiente, considerando que solo se puede incluír a lo mucho uno de cada objeto en la mochila.

   b) Resuelva el problema considerando ahora que se pueden incluír hasta dos unidades de cada objeto en la mochila.

Enfriamiento simulado

- Los algoritmos de escalada no permiten avanzar hacia estados peores que el actual, por lo que pueden quedar estancados en un máximo local.

- Una manera de superar esta desventaja es permitir la transición hacia estados peores con una cierta probabilidad p.

- Esta probabilidad depende de la diferencia en deseabilidad entre el nuevo estado y el actual, y del tiempo de búsqueda transcurrido y puede escribirse como:

$$p = \exp\left(\frac{\text{(valor del nuevo estado) - (valor del estado actual)}}{T}\right),$$

donde T es la **temperatura del sistema**, la cual decrece con el tiempo.

Enfriamiento simulado (algoritmo)

1. Inicializar `MEJOR` con el estado actual.

2. Inicializar T.

3. Repetir hasta que no queden reglas que aplicar al estado actual:

   (a) Seleccionar una regla que no haya sido aplicada al estado actual para producir un nuevo estado.

   (b) Evaluar el nuevo estado y calcular

   $$\Delta E = \text{(valor del nuevo estado) - (valor del estado actual).}$$

   (c) Si el nuevo estado es objetivo, devolverlo y terminar.

   (d) Si el nuevo estado es mejor que el estado actual, convertirlo en el estado actual y hacer `MEJOR` igual al nuevo estado.

   (e) De lo contrario, hacer el estado actual igual al nuevo estado con una probabilidad

   $$p = e^{-\Delta E/T}.$$

   (f) Actualizar T cuando sea necesario.

4. Devolver `MEJOR` como respuesta.

Optimización de rutas

- En algunos problemas en donde la solución es una ruta, se desea encontrar la ruta de menor costo. En estos casos, se utilizan dos funciones heurísticas g y h:
  - g indica el costo de ir del estado inicial al actual (e.g. la suma de los costos de las reglas aplicadas).
  - h es una estimación del costo requerido para ir del estado actual al objetivo (medida de no-deseabilidad).
- Entonces, la función $f = g + h$ es una estimación de la falta de méritos de un estado, considerando el costo de alcanzarlo.

Búsqueda en grafos

- En muchas ocasiones es posible que el algoritmo de búsqueda llegue a estados que habían sido generados previamente, pero la nueva ruta a ese estado tiene un costo menor.

- Estos casos pueden tratarse fácilmente si en lugar de realizar la búsqueda sobre un árbol se realiza sobre un grafo en el cual cada nodo conserva un enlace a su *padre*.

- Si uno encuentra un camino mas corto para llegar a un estado E previamente generado, será necesario actualizar el padre del nodo previamente generado y actualizar los costos (función g) de ese nodo y todos sus descendientes.

Búsqueda el primero mejor

- Este algoritmo combina las técnicas de búsqueda primero en anchura y primero en profundidad, junto con una función herística $f = h + g$ para encontrar una solución eficiente (i.e., de bajo costo).

- El algoritmo requiere el uso de dos listas:
  - **Abiertos:** contiene aquellos nodos que ya han sido evaluados, pero cuyos descendientes aún no han sido generados. Esta lista es una lista con prioridad.
  - **Cerrados:** contiene aquellos nodos que ya han sido evaluados y sus descendientes han sido generados.

- El algoritmo busca el mejor nodo en **Abiertos** y lo expande generando a sus descendientes, los cuales a su vez son evaluados e insertados en **Abiertos**. La lista **Cerrados** permite llevar un control de los estados repetidos y determinar la mejor ruta para llegar a ellos.

Búsqueda el primero mejor

1. Comenzar con una lista `ABIERTOS` conteniendo sólo al estado inicial y otra lista `CERRADOS` vacía.

2. Mientras no se encuentre un objetivo y `ABIERTOS` no esté vacía, hacer:
   (a) Tomar el mejor nodo de `ABIERTOS`.
   (b) Generar sus sucesores. Para cada sucesor hacer:
      i. Si no se ha generado con anterioridad, evaluarlo. Si es objetivo, devolverlo y terminar. De lo contrario, añadirlo a `ABIERTOS`, y almacenar a su padre en `CERRADOS` (si no existe ya).
      ii. Si ya se ha generado antes (es decir, si está en `CERRADOS`), cambiar al padre si el nuevo camino es mejor que el anterior. En este caso, actualizar el coste empleado para alcanzar el nodo y sus sucesores.

Ejercicios

- Diseñe una función heurística adecuada para el problema de misioneros y caníbales que cumpla con lo siguiente:

  1. Debe asignar el valor cero al estado objetivo
  2. Debe asignar valores positivos a otros estados según se alejen del objetivo
  3. Debe asignar valores muy altos a estados no deseables (donde los misioneros se vean amenazados).

- Intente resolver el problema de misioneros y caníbales mediante enfriamiento simulado. Justifique su elección de valores iniciales y curva de descenso para T.

- Implementar el algoritmo de búsqueda el primero mejor para resolver el problema de la mochila y resuelva el caso de ejemplo presentado anteriormente.

Algoritmos de búsqueda para juegos

- Los algoritmos vistos anteriormente pueden utilizarse para resolver juegos de un solo jugador, tales como: laberintos, 8-puzzle, rompecabezas, etc.

- Para juegos de dos jugadores, no es posible aplicar los algoritmos anteriores ya que las reglas que se aplican al estado actual no dependen solamente del algoritmo, sino también del otro jugador.

- Se requiere entonces un método que evalúe los movimientos plausibles considerando las posibles jugadas que pueda realizar el oponente.

Método Minimax

- El algoritmo de búsqueda **minimax** es un método de búsqueda primero en profundidad de profundidad limitada.

- El algoritmo utiliza un generador de movimientos para generar los estados descendientes del estado actual después de un número determinado de movimientos, y una función heurística para evaluarlos.

- El movimiento que conduzca al camino donde se encuentra el mejor nodo es el que devuelve como resultado.

Funciones auxiliares

- `GenMov(Estado, Jugador)` - Devuelve una lista de estados o reglas que corresponden a los movimientos plausibles que puede realizar `Jugador` a partir del `Estado` dado.

- `Estatica(Estado, Jugador)` - Devuelve un número que representa la bondad de `Estado` desde el punto de vista de `Jugador`.

- `BastanteProfundo(Estado, Profundidad)` - Devuelve verdadero si la búsqueda debe detenerse en el nivel actual y falso en caso contrario. Esta función puede depender del tiempo o de un nivel de dificultad determinado.

- `Contrario(Jugador)` - Devuelve 1 si `Jugador` $= 2$, y 2 si `Jugador` $= 1$.

El algoritmo minimax devuelve una estructura `{Valor, Camino}` con el valor heurístico del mejor nodo encontrado y la ruta para llegar a él.

Algoritmo `Minimax(Estado, Profundidad, Jugador)`

1. Si `BastanteProfundo(Estado, Profundidad)`, entonces devolver la estructura:

   `{Valor = Estatica(Estado, Jugador), Camino = nulo}`.

2. En caso contrario, generar otra capa del árbol haciendo `Sucesores = GenMov(Estado, Jugador)`.

3. Si `Sucesores` está vacía, devolver la misma estructura que se hubiera devuelto en el paso 1.

4. De lo contrario, entonces asignar a `MejorResultado` el valor mínimo que puede tomar `Estatica`, y hacer lo siguiente para cada elemento `Suc` en `Sucesores`:

   (a) Hacer `ResultSuc = Minimax(Suc, Profundidad + 1, Contrario(Jugador))`

   (b) Si `-ResultSuc.Valor > MejorResultado`, hacer lo siguiente

      i. Actualizar `MejorResultado` con `-ResultSuc.Valor`.

      ii. Hacer `MejorCamino` igual al resultado de añadir `Suc` al inicio de `ResultSuc.Camino`.

5. Una vez examinados todos los sucesores, devolver la estructura:

   `{Valor = MejorResultado, Camino = MejorCamino}`.

Unidad III

Algoritmos genéticos

Algoritmos Bio-inspirados

- Son algoritmos que simulan el comportamiento de algún proceso o evento en la naturaleza para resolver ciertos tipos de problemas.

- Algunos ejemplos son:
  - **Algoritmos Genéticos:** son técnicas de optimización basadas en la evolución.
  - **Redes Neuronales:** simulan el comportamiento de una red de neuronas y se utilizan comúnmente para clasificación o interpolación.
  - **Sistemas basados en agentes:** simulan una inteligencia colectiva para resolver tareas complejas.

Algoritmos Genéticos

- En un algoritmo genético (AG) se codifican los estados como cadenas de números llamadas *cromosomas*.

- Se tiene una *población* de cromosomas la cual se va transformando mediante operaciones de recombinación y mutación.

- La probabilidad que tiene un individuo de reproducirse en cada iteración depende de una función de bondad llamada *fitness*.

- Por lo tanto, un AG realiza una búsqueda heurística estocástica de manera paralela en el espacio de estados.

Codificación y evaluación de soluciones

- Existen dos aspectos que dependen del problema a resolver:

  1. **Codificación:** se representan las soluciones como una cadena de variables (llamadas *genes*) $c = (x_1, x_2, \ldots, x_m)$.

  2. **Función de evaluación:** es una función $f(x_1, \ldots, x_m)$ que representa la bondad de una solución particular.

- El objetivo del AG es maximizar $f(x_1, \ldots, x_m)$.

- La función de evaluación debe ser relativamente rápida en términos computacionales.

Algoritmo Genético Canónico

1. Generar una población inicial aleatoria $P_0 = \{c_i\}$ y continuar con P_0 como la población actual.

2. Iterar hasta encontrar una solución en la población actual o hasta que se hayan realizado el máximo de iteraciones permitidas:

   (a) Estimar el *fitness* $F_i = f(c_i)/\bar{f}$ para cada cromosoma c_i en la población actual, donde $\bar{f}$ es el promedio de la evaluación de todos los cromosomas en la población.

   (b) Generar una *población intermedia* utilizando algún proceso de selección basado en el fitness.

   (c) Aplicar operadores de recombinación (cruza) y mutación a la población intermedia para obtener la *población siguiente*. Esta será la población actual en la siguiente iteración.

Selección

- La población intermedia se construye mediante un proceso de selección de cromosomas tomados de la población actual.

- La probabilidad de un cromosoma de ser seleccionado es proporcional a su fitness.

- Si un cromosoma es demasiado bueno, debe ser posible colocar mas de una copia de ese cromosoma en la población intermedia.

Algoritmo de Selección

1. Hacer f_i igual al fitness del i-ésimo cromosoma de la población actual.

2. Hacer $j = 0$ y N igual al tamaño de la población.

3. Mientras $j < N$, hacer lo siguiente:
   (a) Elegir un número aleatorio uniforme entero k entre 0 y $N - 1$.
   (b) Elegir un número aleatorio uniforme real p entre 0 y 1.
   (c) Si $p < f_k$, entonces
      i. Hacer el cromosoma j de la población intermedia igual al cromosoma k de la población actual.
      ii. Hacer $f_k = f_k - 1$.
      iii. Hacer $j = j + 1$.

Recombinación (Cruza)

- La recombinación de cromosoma típicamente se realiza tomando los cromosomas de la población intermedia en parejas y generando dos descendientes por cada pareja, los cuales formarán la nueva población.

- Dada una pareja de cromosomas $c_1 = (x_1, x_2, \ldots, x_m)$ y $c_2 = (y_1, y_2, \ldots, y_m)$, la recombinación se realiza mediante una máscara binaria $(b_1, b_2, \ldots, b_m)$, $b_i \in \{0, 1\}$ de la manera siguiente:

$$d_1 = (u_1, u_2, \ldots, u_m), \quad u_i = (1 - b_i)x_i + b_i y_i,$$

$$d_2 = (v_1, v_2, \ldots, v_m), \quad v_i = b_i x_i + (1 - b_i)y_i.$$

Máscaras de recombinación

- **Uniforme equitativa:** para cada b_i se elige el valor 0 ó 1 con una probabilidad del 50

- **Uniforme elitista:** para cada b_i se elige el valor 0 ó 1, respectivamente, con una probabilidad p proporcional al fitness f_1 y f_2 de los padres, dada por $p_i = f_i/(f_1 + f_2)$, $i = 1, 2$.

- **Crossover de 1 punto:** se elige una posición k de manera aleatoria y se asigna

$$b_i = \begin{cases} 0 & \text{si } i \leq k \\ 1 & \text{si } i > k. \end{cases}$$

- **Crossover de 2 puntos:** se eligen dos posiciones k y l, $k \leq l$ de manera aleatoria y se asigna

$$b_i = \begin{cases} 0 & \text{si } k \leq i \leq l \\ 1 & \text{en otro caso.} \end{cases}$$

Mutación

- Después de la recombinación se suele aplicar un operador de mutación, el cual consiste en modificar de manera aleatoria algún gen (elegido también al azar) de los descendientes.

- La mutación es un suceso poco frecuente que se debe aplicar con una probabilidad p_m a cada cromosoma de la nueva población. Típicamente, esta probabilidad es menor al 1% (es decir, $p_m < 0.01$).

- El objetivo de la mutación es producir diversidad en la población y evitar que el AG converga de manera prematura a un máximo local.

Implementación de un AG

- Para la implementación de un AG deben tenerse en cuenta varios aspectos:
  - La codificación de las soluciones en cromosomas.
  - La función de evaluación.
  - Las técnicas de selección y recombinación.
  - La probabilidad y técnica de mutación.

Ejercicios

1. En problemas como el de las n-reinas y el comerciante viajero, las posibles soluciones se codifican como una permutación de los números $1, 2, \ldots, n$. Diseñe un operador de recombinación que sea adecuado para estos problemas; es decir, en el cual los descendientes sean también permutaciones.

2. Describa posibles funciones de evaluación para los siguientes problemas:

   (a) Comerciante viajero para n ciudades.

   (b) Ajustar una circunferencia centrada en (c_x, c_y) con radio r a un conjunto de puntos $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ dados.

Unidad IV

Redes neuronales

Redes Neuronales Artificiales

- Una *red neuronal artificial* (RNA) es un modelo computacional basado en las redes neuronales biológicas.

- Están formadas por grupos de unidades simples de procesamiento llamadas *neuronas* las cuales están interconectadas entre sí.

- A grandes rasgos, una red neuronal se utiliza para modelar una relación compleja entre las entradas y las salidas de la red.

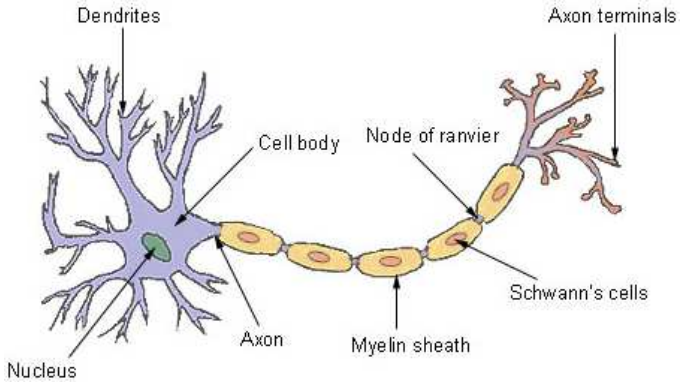
- Las RNA son modelos *adaptivos*, lo cual les permite aprender.

Aplicaciones de las RNA

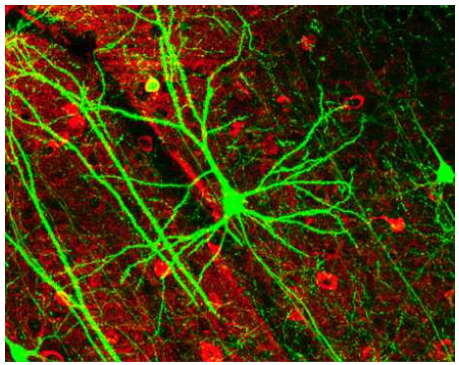
- Modelado de funciones e interpolación
- Predicción en series de tiempo
- Clasificaci'on y reconocimiento de patrones
- Procesamiento de datos (filtrado, agrupamiento, compresión)

Fundamentos biológicos de las RNA

- Una neurona es una célula que consta de un cuerpo de aproximadamente 5 a 10 μm de diámetro.
- Del cuerpo de la neurona salen una rama principal, llamada *axón*, y varias ramas mas cortas, llamadas *dendritas*. Las dendritas se conectan con los axones de otras neuronas.
- Una neurona recibe señales a través de las dendritas. La suma de los efectos excitadores e inhibidores de las señales determinan si la neurona emitirá o no un impulso a través del axón.



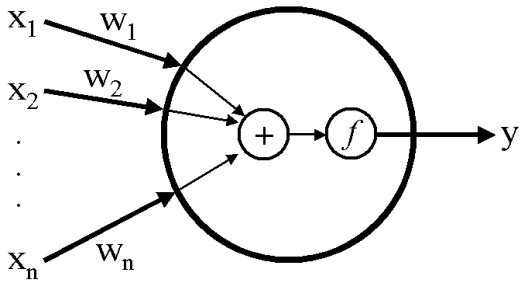
Estructura típica de una neurona



Neuronas en la corteza cerebral del ratón

El Perceptrón

- El *perceptrón* es un modelo computacional de la neurona.



- El perceptrón consta de una o mas entradas x_i y una salida y, la cual es igual a una función f (llamada *función de activación* aplicada a la suma pesada de las entradas $\sum_i w_i x_i$ (llamada *entrada neta*); es decir,

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

- Comúnmente se agrega una *entrada auxiliar* $x_0 = 1$ con un peso w_0 que corresponde al umbral de activación.

Modelo matricial del perceptrón

- Consideremos un perceptrón cuya activación está dada por

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right), \quad x_0 = 1.$$

- Podemos expresar la ecuación anterior en forma matricial:

$$y = f(\text{net}), \quad \text{net} = W^t X,$$

donde $W^t = (w_0, w_1, \ldots, w_n)$ y $X^t = (1, x_1, x_2, \ldots, x_n)$.

Funciones de activación

- Umbral: $f(z) = \begin{cases} 0 & \text{si } z < 0 \\ 1 & \text{si } z \geq 0 \end{cases}$

- Lineal: $f(z) = \begin{cases} 0 & \text{si } z < -1 \\ 1 & \text{si } z > 1 \\ (z+1)/2 & \text{si } -1 \leq z \leq 1 \end{cases}$

- Sigmoidal: $f(z) = \frac{1}{1+e^{-z}} \quad \Longrightarrow$ Notar que: $f'(z) = f(z)\,[1 - f(z)]$

Error de ajuste

- Supongamos que tenemos un vector de muestra $X = (x_1, x_2, \ldots, x_n)$ cuya clasificación (objetivo) $t(X)$ es conocida.

- Sea $y(X) = f(W^t X)$ la salida del perceptrón cuando la entrada es X, si $y(X) \neq t(X)$, entonces la salida del perceptrón es errónea.

- La magnitud del error puede calcularse como:

$$E(X) = \frac{1}{2}\left(y(X) - t(X)\right)^2.$$

Reajuste de pesos

- Si existe error en la clasificación de una muestra $(X, t(X))$, entonces pueden reajustarse los pesos W mediante un descenso de gradiente:

$$w_i \longleftarrow w_i + \Delta w_i,$$

donde

$$\begin{aligned} \Delta w_i &= -\eta \frac{\partial E}{\partial w_i} \\ &= \eta \left(t(X) - y(X)\right) x_i f'(W^t X). \end{aligned}$$

donde f' es la derivada de la función de activación, y η es la *tasa de aprendizaje*.

Entrenamiento del perceptrón

- Dado un conjunto de muestras de entrenamiento $(X_k, t(X_k))$,

  1. Iniciar con valores aleatorios para los pesos w_i.

  2. Iterar hasta que se cumpla algún criterio de convergencia.

     (a) Seleccionar una muestra de entrenamiento $(X, t(X))$.

     (b) Calcular la salida del perceptrón $y(X) = f(W^t X)$.

     (c) Actualizar los pesos w_i usando el descenso de gradiente.

Redes neuronales

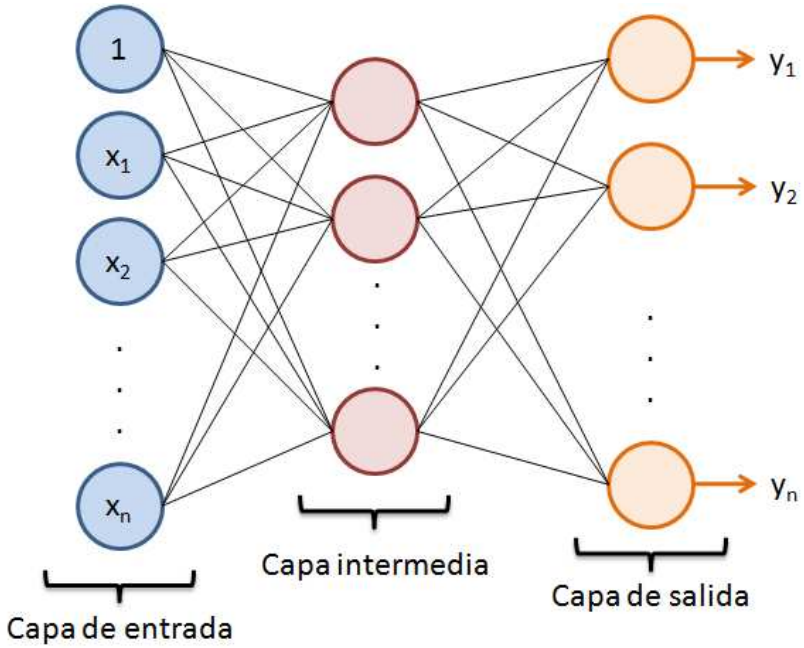
- Un perceptrón por si solo no puede modelar funciones muy complejas.

- Es posible interconectar varios perceptrones de manera que la salida corresponda a la función deseada, formando así una red neuronal.

- La arquitectura más simple de red neuronal es la *red multi-capa*, donde todas las neuronas en una capa están conectadas únicamente con todas las neuronas de la capa siguiente.

Arquitectura de las redes multicapa

- Una red multicapa tiene, por lo general, al menos tres capas: una *capa de entrada*, una *capa de salida*, y una o mas *capas ocultas o intermedias*.



Notación

- En adelante, utilizaremos las siguientes convenciones:
  - Las capas están indexadas, siendo la capa 0 la de entrada, y la capa L la de salida, para una red de $L+1$ capas.
  - La matriz W_k contiene los pesos de los enlaces entre la capa $k-1$ y la capa k. En particular, el elemento (i,j) de W_k representa el peso entre la neurona j de la capa $k-1$ y la neurona i de la capa k.
  - El vector de pesos correspondiente a la neurona artificial (sesgo) de la capa $k-1$ hacia la capa k se denota por b_k. El elemento i de b_k es el sesgo de la neurona i en la capa k.
  - El vector de entradas netas de las neuronas en la capa k se denota por net_k.
  - La función de activación de la neuronas en la capa k es f_k.
  - El vector de salidas de las neuronas en la capa k se denota por y_k.

Propagación hacia adelante

- Para calcular la salida de una capa de la red es necesario conocer la salida de la capa anterior.

- Entonces, la salida de la red se calcula comenzando en la primera capa oculta y propagando la actividad hacia las capas siguientes:

- 1. Hacer $y_0 = x$, donde x es el vector de entrada.

  2. Para k desde 1 hasta L, calcular

$$y_k = f_k(\text{net}_k), \text{ donde } \text{net}_k = W_k y_{k-1} + b_k.$$

Propagación hacia atrás del error

- Dado el vector de valores objetivos t correspondiente a la entrada de la red, el error se calcula como

$$E = \frac{1}{2}(y_L - t)^t(y_L - t) = \frac{1}{2}\sum_o (y_{L,o} - t_o)^2,$$

donde la sumatoria se realiza sobre todas las unidades en la capa de salida.

- Nos interesa entonces calcular el gradiente negativo $\Delta W_{k,i,j}$ del error con respecto al elemento (i, j) de la matriz W_k.

Cálculo del gradiente

- Sea $\delta_{k,j} = -\frac{\partial E}{\partial \text{net}_{k,j}}$. Entonces,

$$\begin{aligned} \Delta W_{k,j,i} &= -\frac{\partial E}{\partial W_{k,j,i}} \\ &= -\frac{\partial E}{\partial \text{net}_{k,j}} \frac{\partial \text{net}_{k,j}}{\partial W_{k,j,i}} \\ &= \delta_{k,j}\ y_{k-1,i}. \end{aligned}$$

- El problema se reduce a encontrar los $\delta_{k,j}$.

Cálculo del gradiente

- Para la capa de salida tenemos que

$$\delta_{L,j} = (t_j - y_{L,j}) f'_L(\text{net}_{L,j}).$$

- Para las capas ocultas aplicamos la regla de la cadena:

$$\delta_{k,i} = -\sum_j \frac{\partial E}{\partial \text{net}_{k+1,j}} \frac{\partial \text{net}_{k+1,j}}{\partial y_{k,i}} \frac{\partial y_{k,i}}{\partial \text{net} k, i},$$

donde la sumatoria se realiza sobre todas las neuronas de la capa $k+1$, y las derivadas parciales están dadas por:

$$\frac{\partial E}{\partial \text{net}_{k+1,j}} = \delta_{k+1,j}, \qquad \frac{\partial \text{net}_{k+1,j}}{\partial y_{k,i}} = W_{k+1,j,i}, \qquad \frac{\partial y_{k,i}}{\partial \text{net}_{k,i}} = f'_k(\text{net}_{k,i}).$$

Cálculo del gradiente

- Por lo tanto, $\delta_{k,i}$ puede calcularse como:

$$\delta_{k,i} = f'_k(\text{net}_{k,i}) \sum_j \delta_{k+1,j} \; W_{k,j,i},$$

donde la sumatoria corre sobre todas las neuronas en la capa $k+1$.

- Entonces, para calcular el gradiente del error en la capa k es necesario calcular primero el gradiente en las capas posteriores.

Algoritmo Back-Propagation (forma matricial)

1. Inicializar la capa de entrada: $y_0 = x$.

2. Propagar actividad hacia adelante: para $k = 1, 2, \ldots, L$ hacer

$$\text{net}_k = W_k y_{k-1} + b_k, \quad y_k = f_k(\text{net}_k).$$

3. Calcular el error en la capa de salida: $\delta_L = (t - y_L) f'_L(\text{net}_L)$.

4. Propagar el error hacia atrás: para $k = L-1, L-2, \ldots, 1$ hacer

$$\delta_k = (W_{k+1}^t \ \delta_{k+1}) \cdot f'_k(\text{net}_k),$$

donde t indica transposición y $\cdot$ indica producto elemento-por-elemento.

5. Calcular los gradientes y actualizar los pesos para $k = 1, \ldots, L$:

$$\Delta W_k = \delta_k \ y_{k-1}^t, \qquad \Delta b_k = \delta_k,$$

$$W_k \longleftarrow W_k + \eta \Delta W_k, \qquad b_k \longleftarrow b_k + \eta \Delta b_k.$$