Práctica 3 - Introducción a los Modelos de Computación Conexionistas

RED DE HOPFIELD

Profesor responsable: Javier Rodriguez Gonzalez
Fecha tope de corrección: 6 de noviembre del 2002
Objetivos:

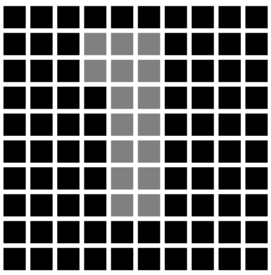
1.- Diseñar y programar una red de Hopfield binaria con las siguiente propiedades.

- Memoria auto-asociativa.
- Resolución minima de 10X10
- Capaz de almanenar al menos 5 patrones de aprendizaje
- Convergencia partiendo de un patrón de testeo hacia algunos de los patrones memorizados.

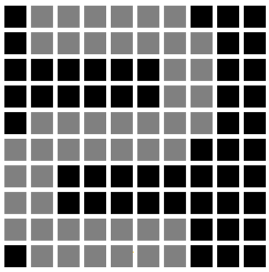
2.- La practica se corregirá en el aula de prácticas y es necesario adjuntar un breve informe sobre la realización de la misma.

Ejemplo de patrones a memorizar.

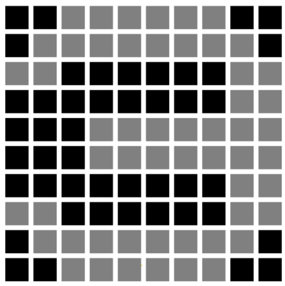
PATRON



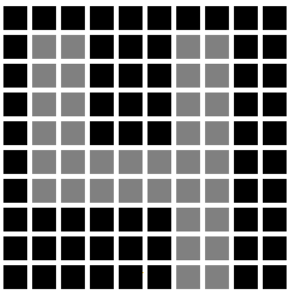
PATRON



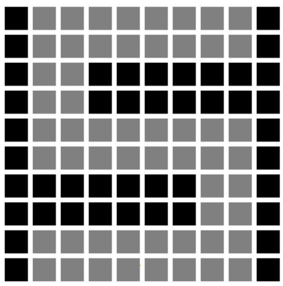
PATRON



PATRON



PATRON



Ver más información en: http://www.geocities.com/ResearchTriangle/Facility/7620/

Código Fuente Java de un simulador de Red Binaria de Hopfield.

```
//  A Simple Binary Hopfield Artificial Neural Network
//  by Matt Hill, written for fun, to learn Java, ie, no guarantees
//  or stylistic endrosements.
//  Based on an a-life program by A.F. Slater
(http://www.cee.hw.ac.uk/~afs/)
//

import java.applet.*;
import java.awt.*;
import java.lang.*;

class SettleThread extends Thread {
        PatternClass pat;
        HopMem          hop;
        final int itermax = 10;
        final int thresh  = 0;

  SettleThread(HopMem Hmem, PatternClass c_p) {
        pat = c_p;
        hop = Hmem;
        //outp.init(c_p);
  }

  public void run() {
        int iter=0;

        PatternClass lastpat = new PatternClass();

        lastpat.init(pat);

        // System.out.println("About to settle");
        while ((iter < itermax) && (hop.ham(pat, lastpat) >
thresh)) {
                lastpat = pat.myclone();
                pat     = hop.iterate(pat);
                iter++;
        }
        // System.out.println("Done settling");

  }
}       // End class SettleThread


class HopMem {
        boolean settling;
        int size;
        int weights[][];
        SettleThread sthread;
        HopfieldApp happ;

  public void debug(PatternClass pat) {
        int x, y;

        System.out.println("Current Pattern: ");
        for (x=0;x<pat.length;x++) {
                System.out.println(x+": "+pat.pattern[x]);
        }

        System.out.println("Weights: ");
        for (x=0;x<pat.length;x++) {
          for (y=0;y<pat.length;y++) {
                System.out.print(x+","+y+":
"+weights[x][y]+"\t");
          }
        System.out.println("");
        }
  }     // End debug

 public void init(PatternClass pattern) {
        settling = false;
        size       = pattern.length;
        weights  = new int[pattern.length][pattern.length];
 }

 public int ham(PatternClass p1, PatternClass p2) {
        int x, total=0;

        // Returns number of elts with different signs

        for (x=0;x<p1.length;x++) {
                if (p1.pattern[x]*p2.pattern[x] <= 0) total++;
        }
        return(total);
 }

 public void settledown(PatternClass pattern){
        sthread = new SettleThread(this, pattern);
        sthread.start();
 }

 public PatternClass stopsettle() {
        if (sthread != null) {
                sthread.stop();
                return(sthread.pat);
        } else {
                return(null);
        }
        //repaint();
 }

 public PatternClass iterate(PatternClass pat) {
        PatternClass new_state;
        int x, y;

        new_state = new PatternClass();
        new_state.init(pat);
        // System.out.println("One iteration");
        for (x=0; x<new_state.length; x++) {
          for (y=0; y<new_state.length; y++) {
           new_state.pattern[x] = new_state.pattern[x] +
pat.pattern[y]*weights[x][y];
           }
     }

        new_state = new_state.normalize();

        happ.current_pattern = new_state;
        happ.repaint();

        return(new_state); // return the new state of the net

 }      // End iterate

 public void impose(PatternClass pat) {
        int x, y;

        for (x=0;x<pat.length;x++) {
                for (y=0;y<pat.length;y++) {
                  if (x!=y) {
                        weights[x][y] = weights[x][y] +
pat.pattern[x]*pat.pattern[y];
                        }
                  }
                }
        // System.out.println("Done with impose");
 }      // End impose

}       // End class HopMem

class PatternClass {
```

```
	int length, blocksize, xsize, ysize;
	int pattern[];

  public PatternClass myclone() {
	PatternClass tmp = new PatternClass();
	int x;
	tmp.pattern = new int[this.length];

	for (x=0;x<this.length;x++) {
		tmp.pattern[x] = this.pattern[x];
		}

	tmp.length = this.length;
	tmp.blocksize = this.blocksize;
	tmp.xsize = this.xsize;
	tmp.ysize = this.ysize;

	return(tmp);
  }

  public void init() {
	int x;

	for (x=0;x<length;x++) {
		this.pattern[x] = -1;
		}
  }

  public void init(PatternClass pat) {
	this.length = pat.length;
	this.blocksize = pat.blocksize;
	pattern = new int[length];
	this.init();
  }

  public void init(int xs, int ys) {

	xsize   = xs/blocksize;
	ysize   = ys/blocksize;
      length  = (int)(xs/blocksize) * (int)(ys/blocksize);
	pattern = new int[length];
	this.init();
  }

  public void toggle(int loc) {
	if (loc<length) pattern[loc] = pattern[loc] * -1;
	// System.out.println("toggle: loc "+loc+" val
"+pattern[loc]);
	}

  public PatternClass normalize() {
	int x;

	for (x=0;x<this.length;x++) {
		if (this.pattern[x] != 0) {
			this.pattern[x] =
(int)(this.pattern[x]/Math.abs(this.pattern[x]));
		}
	}

	return(this);
  }

  public void randomize() {
	int x;
	int y;

    for(x=0;x<length;x++) {
	pattern[x] = (int)(2*Math.round(Math.random()) -1 );
	  }
  }
}
```

```
public class HopfieldApp extends Applet {

  HopMem mem;
  PatternClass current_pattern;
  int xsize, ysize;
  Color bgrnd=Color.lightGray;
//  Image offscreen;

  public void init() {
	current_pattern = new PatternClass();
	mem = new HopMem();

	String w=getParameter("wid");
	String h=getParameter("ht");
	String ps=getParameter("size");

	if((w!=null)&&(h!=null)) {
		xsize=Integer.parseInt(w);
		ysize=Integer.parseInt(h);
	} else {
		xsize=200;
		ysize=200;
	}

	if (ps!=null)
	    current_pattern.blocksize=Integer.parseInt(ps);

	current_pattern.init(xsize, ysize);

	mem.init(current_pattern);
	mem.happ = this;

	setLayout(new BorderLayout());
	Panel p = new Panel();

//	Unfinished section, to replace key presses with buttons
//	p.add(new Button("Randomize"));
//	p.add(new Button("Settle One Step"));
//	p.add(new Button("Settle Until Stable"));
//	p.add(new Button("Clear Display"));
//	p.add(new Button("Clear Memory"));
//	p.add(new Button("Memorize This Pattern"));

//	add("East", p);

//	System.out.println("Hopfield: supplied width="+xsize);
//	System.out.println("Hopfield: supplied height="+ysize);

  }	// init

public boolean action(Event evt, Object arg) {
		if ("Randomize".equals(arg)) {
			current_pattern.randomize();
			repaint();
		} else {
		if ("Settle Until Stabble".equals(arg)) {
			toggle_settle();
		} else {
		if ("Settle One Step".equals(arg)) {
		   current_pattern = mem.iterate(current_pattern);
		} else {
		if ("Clear Display".equals(arg)) {
			current_pattern.init();
			repaint();
		} else {
		if ("Clear Memory".equals(arg)) {
			mem.init(current_pattern);
		} else {
		if ("Memorize This Pattern".equals(arg)) {
			mem.impose(current_pattern);
		}}}}}}
```

```
            return true;
    }

//  public void start() {
//
//  /* while (!mem.settling) {
//            mem.iterate(current_pattern);
//            }  */
//  }         // start

  public void stop() {
    if (mem.settling) {
            current_pattern = mem.stopsettle();
            }
    mem.settling=false;
  }

  public void paint(Graphics g) {
    int loc, xpos, ypos;

    setBackground(bgrnd);
    g.clearRect(1, 1, xsize - 2, ysize - 2);
    g.setColor(Color.red);

    for (loc=0;loc<current_pattern.length; loc++) {
      ypos =
(int)(loc/(xsize/current_pattern.blocksize))*current_pattern.blocksi
ze;
      xpos = (loc % (xsize/current_pattern.blocksize)) *
current_pattern.blocksize;
      if (current_pattern.pattern[loc] > 0) {
      // System.out.println("Hopfield: paint: loc "+loc+" xpos
"+xpos+" ypos "+ypos);
            g.fillRect(xpos, ypos, current_pattern.blocksize,
current_pattern.blocksize);
            }
     } // End for loc

    //g.drawRect(1, 1, xsize - 3, ysize - 3);
    g.setColor(Color.black);
    g.drawRect(0, 0, xsize, ysize - 1);
  }

  /**
   * Mouse methods
   */

  public boolean mouseDown(Event evt, int x, int y) {
    Graphics hop_graphics;

    // getFocus();

    // System.out.println("Mouse Down!");

    int click_loc = (int)((x/current_pattern.blocksize) +
                              ((int)(y/current_pattern.blocksize) *
(int)(xsize/current_pattern.blocksize)) );

    if ((x <= xsize) && (y <=ysize)) {

      current_pattern.toggle(click_loc);

      hop_graphics = mem.happ.getGraphics();
      if (current_pattern.pattern[click_loc] > 0)      {
            hop_graphics.setColor(Color.red);
            hop_graphics.fillRect(x-(x %
current_pattern.blocksize), y-(y%current_pattern.blocksize),
                        current_pattern.blocksize,
current_pattern.blocksize );
      } else {
            hop_graphics.clearRect(x-(x %
current_pattern.blocksize), y-(y%current_pattern.blocksize),
                        current_pattern.blocksize,
current_pattern.blocksize);
      }
    }
    return true;
  }

/*  public void mouseDrag(int x, int y) {
    mouseDown(x,y);
  } */

  public void toggle_settle() {
            if (!mem.settling) {
                        mem.settling=true;
                        mem.settledown(current_pattern);
                        mem.settling=false;
                        } else {
                        current_pattern = mem.stopsettle();
                        mem.settling=false;
                        }
  }

  public boolean keyDown(Event evt, int key) {
    char c=(char)key;

            switch (c) {
                        case 's' :  toggle_settle();
                                    break;
                        case ' ' :  current_pattern =
mem.iterate(current_pattern);
                                    //repaint();
                                    //System.out.println("mack was 2.
is now "+mack);
                                    break;
                        case 'r' :  current_pattern.randomize();
                                    //mem.init(current_pattern);
                                    repaint();
                                    break;
                        case 'c' :  current_pattern.init();
                                    repaint();
                                    break;
                        case 'n' :  mem.init(current_pattern);
                                    break;
                        case 'i' :  mem.impose(current_pattern);
                                    break;
                        case 'd' :  mem.debug(current_pattern);
                                    break;
                        case 'q' :  repaint();
                                    break;
            }           // End switch (c)

            return true;

  }         // End keyDown
}           // End applet HopfieldApp
```