JAVAWORLD
SOLUTIONS FOR JAVA DEVELOPERS

This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/jw-05-2005/jw-0516-lego.html

A neural network for Java Lego robots

Learn to program intelligent Lego Mindstorms robots with Java

By Julio César Sandria Reynoso, JavaWorld.com, 05/16/05

Developers can build intelligent robots with Java, as it provides APIs for programming systems that can see, hear, speak, move, and even learn, using neural networks, which are algorithms that mimic our brain. (Please see "Futurama: Using Java Technology to Build Robots That Can See, Hear, Speak, and Move," by Steve Meloan (Sun Developer Network, July 2003).)

This article shows how to develop a robot that can learn by using the backpropagation algorithm, a basic neural network, and implementing it on a Lego Roverbot. Using both the algorithm and Java, the Roverbot—a Lego robot vehicle—can learn some basic rules for moving forward, backward, left, and right.

In this article, we use the Lego Mindstorms Robotics Invention System 2.0 for building the Lego robot; leJOS 2.1.0, a little Java operating system for downloading and running Java programs inside the Roverbot; and J2SE for compiling the Java programs under leJOS.

Lego robots

The Lego Mindstorms Robotics Invention System (RIS) is a kit for building and programming Lego robots. It has 718 Lego bricks including two motors, two touch sensors, one light sensor, an infrared tower, and a robot brain called the RCX.

The RCX is a large brick that contains a microcontroller and an infrared port. You can attach the kit's two motors (as well as a third motor) and three sensors by snapping wire bricks on the RCX. The infrared port allows the RCX to communicate with your desktop computer through the infrared tower.

In this article, we use a Roverbot as it is constructed in the Lego Mindstorms Constructopedia, the guide for constructing robots. This Roverbot, as shown in Figure 1, has been configured to use all three sensors and two motors included in Lego Mindstorms RIS 2.0.

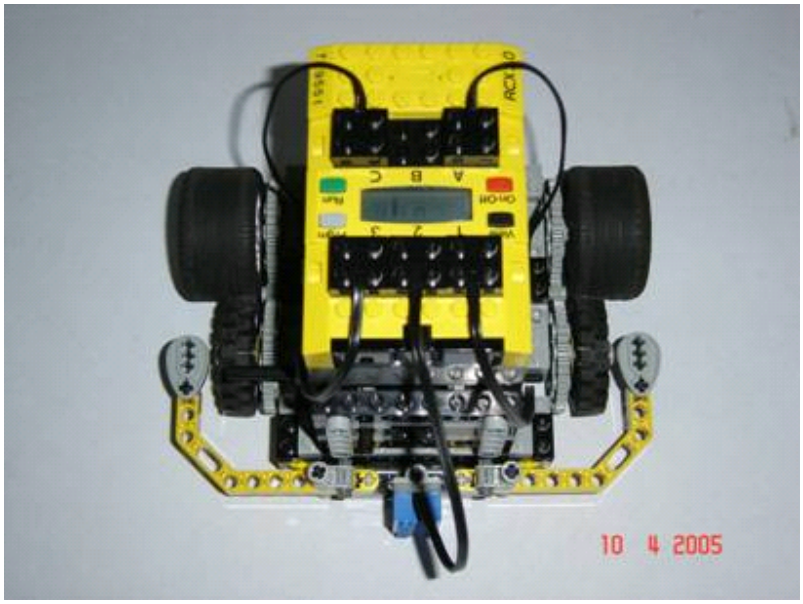

Figure 1. A Lego Roverbot with two touch sensors, one light sensor, and two motors

leJOS

leJOS is a small Java-based operating system for the Lego Mindstorms RCX. Because the RCX contains just 32 KB of RAM, only a small subset of the JVM and APIs can be implemented on the RCX. leJOS includes just a few commonly used Java classes from `java.lang`, `java.io`, and `java.util`, and thus fits well on the RCX.

You must load the RAM with the Lego firmware, or, in our case, with the leJOS firmware, and your programs. The firmware contains a bytecode interpreter, which can run programs downloaded from RCX code.

For setting up your leJOS installation, please take a look at Jonathan Knudsen's article "Imaginations Run Wild with Java Lego Robots," (*JavaWorld,* February 2001), *Programming Lego Mindstorms with Java* (Syngress Publishing, 2002), or the leJOS readme file contained in the leJOS zip file, which you can download from the leJOS homepage.

Neural networks

If we want to build intelligent machines, we should model the human brain. Early in the 1940s, the neurophysiologist Warren McCulloch and the mathematician Walter Pitts began working on the idea of building an intelligent machine out of artificial neurons. One of the earliest neural network models was the perceptron, an invention of F. Rosenblat in 1962. A perceptron can learn; it models a neuron by taking a weighted sum of its inputs and sending an output of 1 if the sum is greater than some adjustable threshold value, otherwise it sends 0. If a perceptron can compute, it can learn to compute. Figure 2 shows a neuron and Figure 3 shows a perceptron.

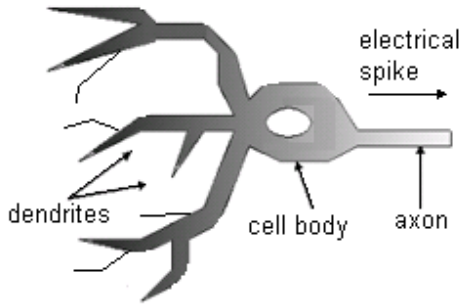

Figure 2. A neuron

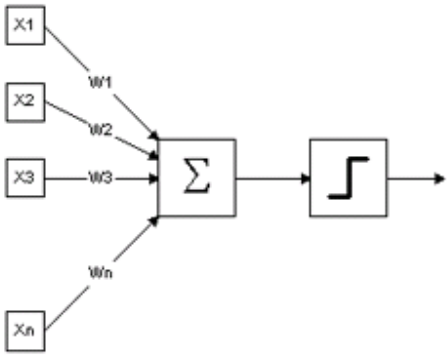

Figure 3. A perceptron

The inputs (x_1, x_2, x_3, ..., x_n) and connection weights (w_1, w_2, w_3, ..., w_n) in the figure are typically real values. If the presence of some feature x_i tends to cause the perceptron to fire, the weight w_i will be positive; if the feature x_i inhibits the perceptron, the weight w_i will be negative. As Elaine Rich and Kevin Knight note in their book *Artificial Intelligence (McGraw-Hill, 1990), "the perceptron itself consists of the weights, the summation processor, and the adjustable threshold processor. Learning is a process of modifying the values of the weights and the threshold." The authors recommend implementing the threshold as another weight w_0 because this weight can be thought of as the propensity of the perceptron to fire irrespective of its input.*

Backpropagation networks

A backpropagation network is a fully connected, layered, and feed-forward neural network (see Figure 4). Network activation flows in one direction only: from the input layer to the output layer, passing through the hidden layer. Each unit in a layer is connected in the forward direction to every unit in the next layer. Weights between units encode the network's knowledge.

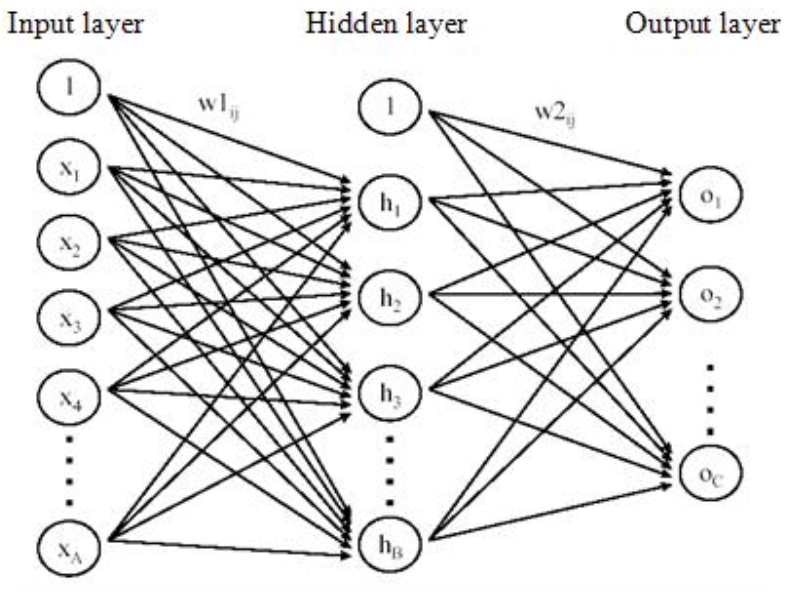

Figure 4. A backpropagation network

A backpropagation network usually starts with a random set of connection weights. The network adjusts its weights based on some learning rules each time it sees a pair of input-output vectors. Each pair of vectors goes through two stages of activation: a forward pass and a backward pass. The forward pass involves presenting a sample input to the network and letting activations flow until they reach the output layer. During the backward pass, the network's actual output (from the forward pass) is compared with the target output, and errors are computed for the output units. The weights connected to the output units can be adjusted to reduce those errors. The error estimates of the output units are then used to derive error estimates for the units in the hidden layers. Finally, errors are propagated back to the connections stemming from the input units.

After each round of forward-backward passes, the system "learns" incrementally from the input-output pair and reduces the difference (error) between the network's predicted output and the actual output. After extensive training, the network will eventually establish the input-output relationships through the adjusted weights on the network.

The backpropagation algorithm

Given a set of input-output vector pairs, you can compute a set of weights for a neural network that maps inputs onto corresponding outputs.

Let A be the number of units in the input layer, as determined by the length of the training input vectors. Let C be the number of units in the output layer. Now choose B, the number of units in the hidden layer. As shown in Figure 4, the input and hidden layers each have an extra unit used for thresholding; therefore, the units in these layers will sometimes be indexed by the ranges (0,...,A) and (0,..., B). We denote the activation levels of the units in the input layer by x_j, in the hidden layer by h_j, and in the output layer by o_j. Weights connecting the input layer to the hidden layer are denoted by $w1_{ij}$, where the subscript **i** indexes the input units and **j** indexes the hidden units. Likewise, weights connecting the hidden layer to the output layer are denoted by $w2_{ij}$, with **i** indexing hidden units and **j** indexing output units.

The backpropagation algorithm has the following steps:

1. Initialize the network weights. Initially, all connection weights are set randomly to numbers between -0.1 and 0.1:

   $w1_{ij}$ = random(-0.1, 0.1) for all **i = 0, ..., A, j = 1, ..., B**
   $w2_{ij}$ = random(-0.1, 0.1) for all **i = 0, ..., B, j = 1, ..., C**

2. Initialize the activations of the threshold units. For each layer, its threshold unit is set to 1 and should never change:

   x_0 = 1.0
   h_0 = 1.0

3. Choose an input-output pair. Suppose the input vector is x_i and the target output vector is y_i. Assign activation levels to the input units.
4. Propagate the activations from the units in the input layer to the units in the hidden layer using the activation function:

$$h_j = \frac{1}{1 + e^{-\sum_{i=0}^{A} w1_{ij} h_i}}$$

   for all j = 1, ..., B

   Note that i ranges from 0 to A. $w1_{oj}$ is the thresholding weight for hidden unit j. x_0 is always 1.0.

5. Propagate the activations from the units in the hidden layer to the units in the output layer:

$$o_j = \frac{1}{1 + e^{-\sum_{i=0}^{B} w2_{ij} h_i}}$$

   for all j = 1, ..., C

   Again, the thresholding $w2_{oj}$ for output units j plays a role in the weighted summation. h_0 is always 1.0.

6. Compute the errors of the units in the output layer, denoted $\delta 2_j$. Errors are based on the network's actual output (o_j) and the target output (y_j):

   $\delta 2_j = o_j (1 - o_j) (y_j - o_j)$ for all j = 1, ..., C

7. Compute the errors of the units in the hidden layer, denoted $\delta 1_j$:

$$\delta 1_j = h_j(1-h_j)\sum_{i=1}^{C}\delta 2_i w2_{ji}$$

for all j = 1, ..., B

8. Adjust the weights between the hidden layer and output layer. The learning rate is denoted η; its function is the same as in perceptron learning. A reasonable value of η is 0.35:

   $\Delta w2_{ij} = \eta\ \delta 2_j\ h_i$ for all **i = 0, ..., B, j = 1, ..., C**

9. Adjust the weights between the input layer and the hidden layer:

   $\Delta w1_{ij} = \eta\ \delta 1_j\ x_i$ for all **i = 0, ..., A, j = 1, ..., B**

10. Go to Step 4 and repeat. When all the input-output pairs have been presented to the network, one epoch has been completed. Repeat Steps 4 to 10 for as many epochs as desired.

The activation function has a sigmoid shape. Since infinite weights would be required for the actual outputs of the network to reach 0.0 and 1.0, binary target outputs (the y_j's of Steps 4 and 7 above) are usually given as 0.1 and 0.9 instead. The sigmoid is required by backpropagation because the derivation of the weight update rule requires that the activation function be continuous and differentiable.

The Lego Mindstorms backpropagation network

We want to model a backpropagation network for our Roverbot (Figure 5). The robot has three inputs (two touch sensors and one light sensor) and two outputs (the two motors). So, we can use a three-layer backpropagation network as shown in Figure 6, where we change the unit index to begin with 0; recall the use of index in Java arrays.

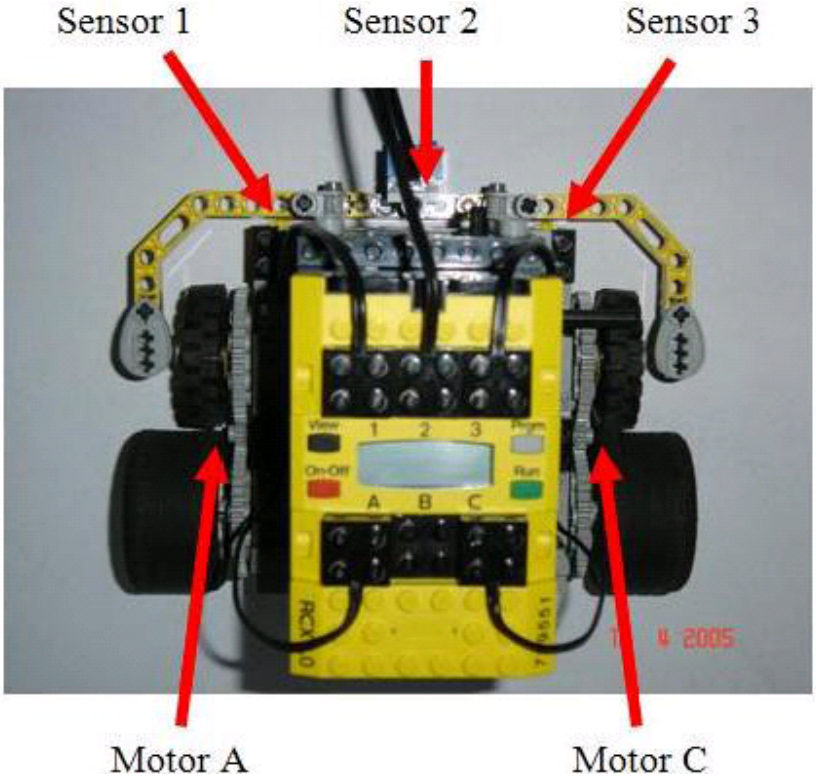

Figure 5. The roverbot to model

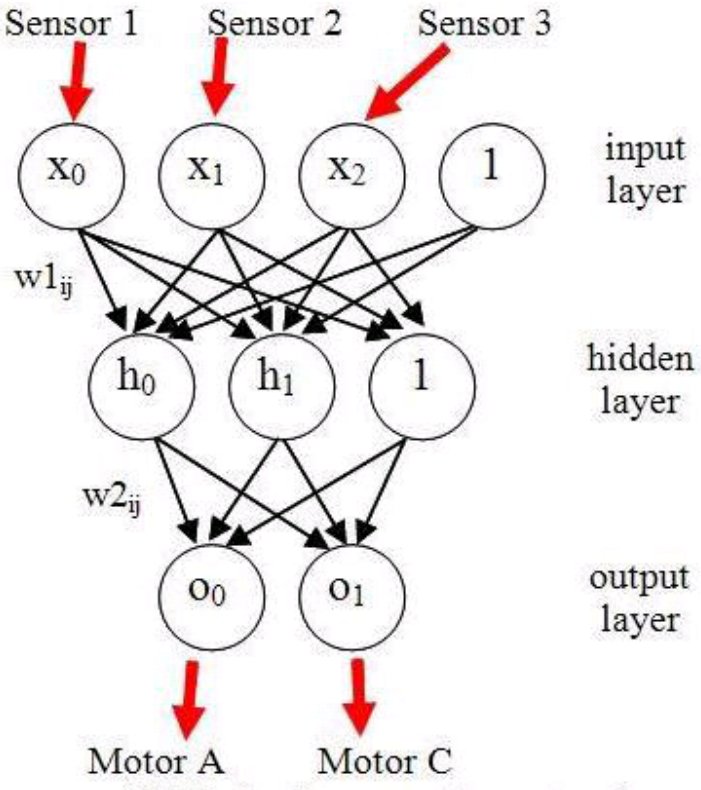

Figure 6. The backpropagation network

Note: The use of a three-layered network and three units in the hidden layer is just an arbitrary decision influenced by teaching purposes.

To define input-output vector pairs for use in the backpropagation network, from the

robot input-output (sensor-motor), we must identify what the robot is going to learn. We define four basic behavior rules:

- Moving forward: If Sensor 1 is off, and Sensor 2 is over a white floor, and Sensor 3 is off, then Motor A and Motor C go forward (Roverbot goes forward)
- Moving right: If Sensor 1 is on, then Motor A goes forward, and Motor C goes backward (Roverbot turns right)
- Moving left: If Sensor 3 is on, then Motor A goes backward, and Motor C goes forward (Roverbot turns left)
- Moving backward: If Sensor 2 is over a black floor, then Motor A and Motor C go backward (Roverbot goes backward)

We translate these rules to training examples for the backpropagation network as shown in Figures 7 and 8, where S1 = Sensor 1, M-A = Motor A, and so on.

	Rules to learn				
Rule	Sensor 1	Sensor 2	Sensor 3	Motor A	Motor C
1	Off	White	Off	Forward	Forward
2	On	White	Off	Forward	Backward
3	Off	White	On	Backward	Forward
4	Off	Black	Off	Backward	Backward
	Physical inputs			Physical outputs	

Figure 7. Rules

Training examples				
S1	S2	S3	M-A	M-C
0	0	0	1	1
1	0	0	1	0
0	0	1	0	1
0	1	0	0	0
Input vectors			Output vectors	

Figure 8. Training examples

The input-output vector pairs are the examples we use to train the backpropagation network. So, based on its sensor states, our robot will learn to move forward, right, left, and backward. But what would happen if both touch sensors were on? The robot would not learn that case (rule or example), but the backpropagation network would give it an emergent behavior.

What emergent behavior? Will the robot go forward, backward, left, or right? You will get the answer by compiling the program, downloading it to the RCX, and pressing the Run

button to run the program and see the robot behavior.

The Java classes

To implement the backpropagation algorithm, divide the code into two Java classes:

1. Class `LMpbn`, where you encapsulate all features of the backpropagation algorithm:
    - Properties, like public arrays:

```
input []
 hidden []
 output []
 w1 [][]
 w2 [][]
```

    - Methods:

```
train(...)
 test(...)
```

Note that this is a generic class, so, you can use it in any leJOS program and in any other Java program.

Listing 1. The LMbpn class

```
 /**


 * <p>Title: Lego Mindstorms Neural Networks</p>
 *
 * @author Julio César Sandria Reynoso
 * @version 1.0
 *
 * Created on 1 de abril de 2005, 06:09 PM
 */


import java.lang.Math;


/**
 * LMbpn: Lego Mindstorms Back Propagation Network
 */
class LMbpn {
    public static int data1[][] = {{0,0,0}, {1,1}};
    public static int data2[][] = {{1,0,0}, {1,0}};
    public static int data3[][] = {{0,0,1}, {0,1}};
    public static int data4[][] = {{0,1,0}, {0,0}};


    public static double input[] = {0,0,0,1};
    public static double w1[][] = {{0,0,0}, {0,0,0}, {0,0,0}, {0,0,0}};
    public static double hidden[] = {0,0,1};
    public static double w2[][] = {{0,0}, {0,0}, {0,0}};
```

```
public static double output[] = {0,0};
public static double delta2[] = {0,0};
public static double delta1[] = {0,0,0};


public static int trainedEpochs = 0;


LMbpn() {
    byte i, j;
    // Initialize weights randomly between 0.1 and 0.9
    for(i=0; i<w1.length; i++)
        for(j=0; j<w1[i].length; j++)
            w1[i][j] = Math.random()*0.8+0.1;


    for(i=0; i<w2.length; i++)
        for(j=0; j<w2[i].length; j++)
            w2[i][j] = Math.random()*0.8+0.1;
}


public static void train(int e) {
    for(int i=0; i<e; i++) {
        // Call method learn with training data
        learn( data1[0], data1[1] );
        learn( data2[0], data2[1] );
        learn( data3[0], data3[1] );
        learn( data4[0], data4[1] );
        trainedEpochs++;
    }
}


public static void learn( int inp[], int out[] ) {
    int i, j;
    double sum, out_j;


    // Initialize input units
    for(i=0; i<inp.length; i++)
        input[i] = inp[i];


    // Calculate hidden units
    for(j=0; j<hidden.length-1; j++) {
        sum = 0;
        for(i=0; i<input.length; i++)
            sum = sum + w1[i][j]*input[i];


        hidden[j] = 1 / ( 1 + Math.exp(-sum));
    }


    // Calculate output units
    for(j=0; j<output.length; j++) {
        sum = 0;
        for(i=0; i<hidden.length; i++)
            sum = sum + w2[i][j]*hidden[i];


        output[j] = 1 / (1 + Math.exp(-sum));
    }
```

```
        // Calculate delta2 errors
        for(j=0; j<output.length; j++) {
            if( out[j] == 0 )
                out_j = 0.1;
            else if( out[j] == 1 )
                out_j = 0.9;
            else
                out_j = out[j];
            delta2[j] = output[j]*(1-output[j])*(out_j-output[j]);
        }


        // Calculate delta1 errors
        for(j=0; j<hidden.length; j++) {
            sum = 0;
            for(i=0; i<output.length; i++)
                sum = sum + delta2[i]*w2[j][i];


            delta1[j] = hidden[j]*(1-hidden[j])*sum;
        }


        // Adjust weights w2
        for(i=0; i<hidden.length; i++)
            for(j=0; j<output.length; j++)
                w2[i][j] = w2[i][j] + 0.35*delta2[j]*hidden[i];


        // Adjust weights w1
        for(i=0; i<input.length; i++)
            for(j=0; j<hidden.length; j++)
                w1[i][j] = w1[i][j] + 0.35*delta1[j]*input[i];
    }


    public static void test(int inp[], int out[]) {
        int i, j;
        double sum;


        // Initialize input units
        for(i=0; i<inp.length; i++)
            input[i] = inp[i];


        // Calculate hidden units
        for(j=0; j<hidden.length-1; j++) {
            sum = 0;
            for(i=0; i<input.length; i++)
                sum = sum + w1[i][j]*input[i];


            hidden[j] = 1 / ( 1 + Math.exp(-sum));
        }


        // Calculate output units
        for(j=0; j<output.length; j++) {


            sum = 0;
            for(i=0; i<hidden.length; i++)
                sum = sum + w2[i][j]*hidden[i];
```

```
            output[j] = 1 / (1 + Math.exp(-sum));
        }


        // Assign output to param out[]
        for(i=0; i<output.length; i++)
            if( output[i] >= 0.5 )
                out[i] = 1;
            else
                out[i] = 0;
    }
}
```

2. Class `LMbpnDemoRCX`: A demo program for the RCX, where you implement the use of the `LMbpn` class:

```
  ...
   main() {
      LMbpn bpn = new LMbpn();
      ...
      bpn.train(...);
      ...
      bpn.test(...);
      ...
   }
```

Listing 2. The LMbpnDemoRcx class

```
 import josx.platform.rcx.LCD;
import josx.platform.rcx.TextLCD;
import josx.platform.rcx.Sound;
import josx.platform.rcx.Sensor;
import josx.platform.rcx.SensorConstants;
import josx.platform.rcx.Motor;
import josx.platform.rcx.Button;


public class LMbpnDemoRcx {
    public static LMbpn bpn = new LMbpn();




    public static void main(String args[]) throws InterruptedException
    {


        int i, white;
        int inp[] = {0,0,0};
        int out[] = {0,0};


        Sound.beep();
        TextLCD.print( "Train" );


        // Train bpn 500 epochs, sit down and wait about 5 minutes!
```

```
for(i=0;i<500;i++) {
    bpn.train(1);
    LCD.showNumber( bpn.trainedEpochs );
}


Sensor.S1.setTypeAndMode ( SensorConstants.SENSOR_TYPE_TOUCH,
                           SensorConstants.SENSOR_MODE_BOOL );


Sensor.S2.setTypeAndMode ( SensorConstants.SENSOR_TYPE_LIGHT,
                           SensorConstants.SENSOR_MODE_RAW );


Sensor.S3.setTypeAndMode ( SensorConstants.SENSOR_TYPE_TOUCH,
                           SensorConstants.SENSOR_MODE_BOOL );


Sound.twoBeeps();
Sensor.S2.activate();
white = Sensor.S2.readRawValue();


Motor.A.setPower(1);
Motor.C.setPower(1);


Sound.twoBeeps();


while( !Button.PRGM.isPressed() ) {


    LCD.showNumber( Sensor.S2.readRawValue() );


    if( Sensor.S1.readBooleanValue() )
        inp[0] = 1; // Sensor 1 on
    else
        inp[0] = 0; // Sensor 1 off


    if( Sensor.S2.readRawValue() > white + 50 )
        inp[1] = 1; // Sensor 2 over black floor
    else
        inp[1] = 0; // Sensor 2 over white floor


    if( Sensor.S3.readBooleanValue() )
        inp[2] = 1; // Sensor 3 on
    else
        inp[2] = 0; // Sensor 3 off


    bpn.test( inp, out );


    if( out[0] == 1 )
        Motor.A.forward();
    else
        Motor.A.backward();


    if( out[1] == 1 )
        Motor.C.forward();
    else
        Motor.C.backward();
```

```
                Thread.sleep( 500 );

            } // while()

            Sensor.S2.passivate();
            Motor.A.stop();
            Motor.C.stop();
            Sound.beep();

        } // main()

    } // class LMbpn
```

First, you must instantiate an object from class `LMbpn`. After that, you must train the backpropagation network with a certain number of epochs. Finally, you test the network with current sensor states.

Compile and run

To compile the classes and run your program, first, install leJOS and some environment variables in a command window. Compile classes with the commands:

```
 lejosc LMbpn.java
lejosc LMbpnDemoRcx.java
```

Download your program to the RCX using:

```
 lejos LMbpnDemoRcx
```

And run your program by pressing the RCX's Run button.

This example trains the backpropagation network in 500 epochs, which takes about five minutes in the RCX. You could train the network on a personal computer (that takes less than five seconds), save the weights calculated, and assign these weights instead of initializing them randomly.

Conclusion

Lego Mindstorms robots are cool toys used by hobbyists all around the world. They prove suitable for building mobile robots and programming them with artificial intelligence. The backpropagation network described in this article was implemented as a Java class to

build an intelligent Lego robot that can learn a basic behavior. With some more work, you can program more complex behavior with this neural network. Finally, such a class is a reusable Java class that can be modified and used in any other Java-based system.

About the author

Julio C�sar Sandria Reynoso is software developer at the Instituto de Ecolog�a, A.C., and professor of computer programming and artificial intelligence at the Universidad de Xalapa, in Xalapa City, Mexico. He has a master's of science in artificial intelligence and has worked with Java since 1998.