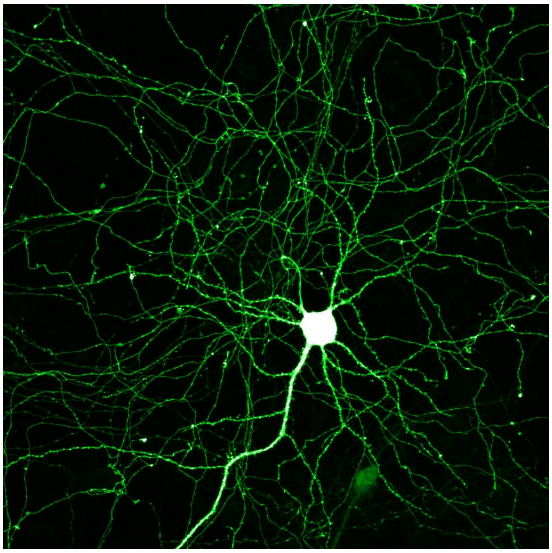
Neural Networks (using NXC on Lego Mindstorms)

Galen J. Wilkerson, Instructor
Johns Hopkins University
Center for Talented Youth, Summer 2010
Introduction to Robotics

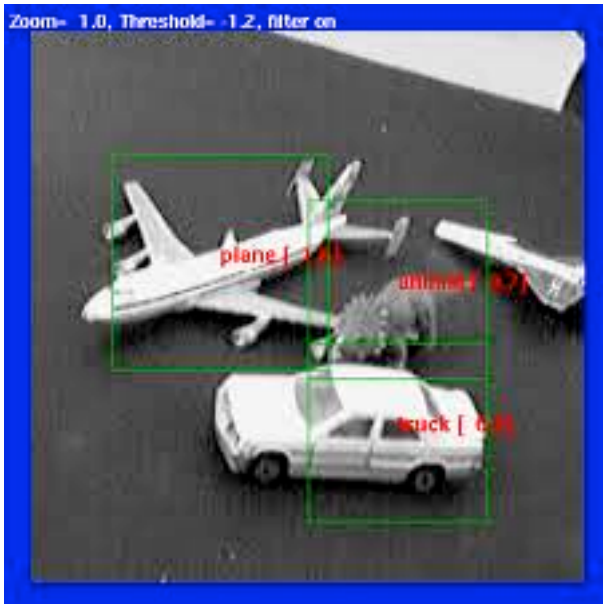
Neurons



All brains are made up of **neurons**. Neurons are cells that can connect to each other and transmit electrical signals. Although it is not understood exactly how these cells create thoughts, we know that they act like small switches in a very complicated network, determining the route that electrical signals take.

Neurons are "excitable", capable of releasing a small electrical charge. Roughly speaking, neurons receive electrical signals from other neurons, and either fire or don't fire based on how strong the input is. This then leads to other neurons firing or not firing.

Every thought we have or action we take is the result of thousands or millions of neurons firing in a very complicated cascade of electrical signals!



Classification

Classification is very important in Artificial Intelligence. For example, if we want to get a computer to recognize objects, we need to take input data - digital images, for example – and recognize that something is a chair, table, person, etc. That is, we want to be able to label the object as belonging to a ***class*** of other objects.

Once it knows what something is, a computer program can interact with that object more intelligently. For example, in the Asimo video, Asimo was able to recognize that something was a chair, and then, since it knows some things about chairs, to know that it could sit down on it, even though it had never seen it before!

Classification with neural networks can be used in many areas – voice recognition, path planning, game strategy, identifying customers for advertising, stock market analysis, and optical character recognition - to name a few!

Neural Networks

Neural networks are artificial networks that can do things like classify objects.

When setting up a neural network to classify objects, we have 2 phases:

- ***Training***, where the computer 'learns' how to classify objects by being told what each new object is.
- ***Classification,*** where the computer is ready to classify new unknown objects.

Training

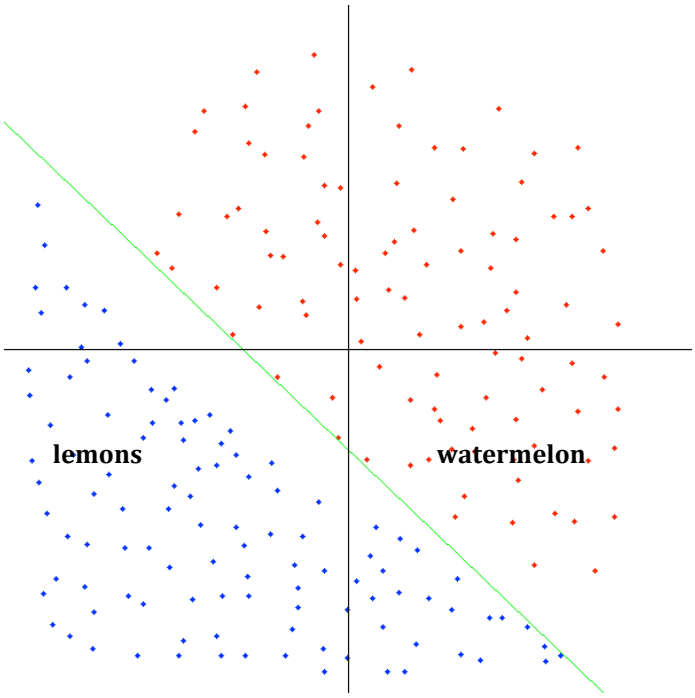
During the *training* phase, we show the computer program a new piece of fruit, then tell it which kind of fruit it is.

Suppose we hand the computer a lemon. We weigh the lemon, measure its size and weight. The computer stores this fact (one of the blue dots). We then give it a watermelon with size and weight. This creates a red dot.

The computer can now draw a line separating the blue dot (lemon) and red dot (watermelon).

Each time we enter a new piece of fruit, the computer adjusts the line to better divide the two types of fruit.

Eventually, we might end up with something like the picture to the right, with hundreds of data points (fruit), all having different sizes and weights, and with a line between them.



size vs. weight, blue = lemons, red = watermelon

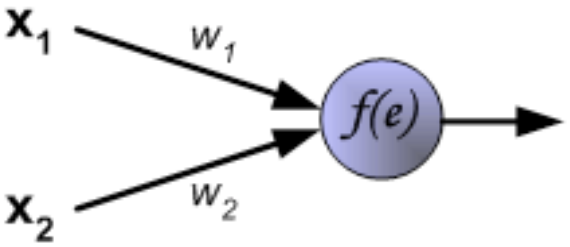
Classification

Once the computer is well-trained, we can switch to the *classification* phase. Now, when it sees a new piece of unknown fruit, it can:

- plot the point on the graph
- determine which side of the line it is on
- color the point red or blue, meaning *watermelon* or *lemon*

The computer can now recognize objects!!?!

Perceptrons



a simple perceptron

In Artificial Intelligence, the simplest model of a neuron is the **perceptron**.

Perceptrons are *virtual*. That is, they only exist in a computer program! However, they have some similar properties to neurons. They are also excitable, and can be networked together to perform more complex tasks.

A perceptron is a "binary classifier". That is, based on input, it either fires or doesn't fire, similar to a neuron. So, we can tell from the output what the input was.

[Review slope-intercept form of line: y = mx + b]

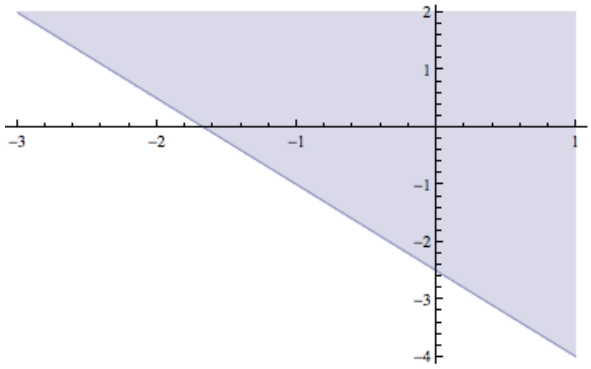
A perceptron can be represented by a very simple function:

f(x,y) = 1 if weight1 * x + weight2 * y + shift > 0

f(x,y) = 0 otherwise

It's a little hard to see what this means, but it turns out that weight1 * x + weight2 * y + shift > 0 is just the equation of a line! (weight1, weight2, and shift are constants)

For example:
3x + 2y + 5 > 0
3x + 5 > -2y
(3x + 5)/-2 < y (flip the >, since div. by -2)
y > -3x/2 -5/2 (slope-intercept form of a line! We can draw the line!)



So, generally the equation of the line becomes:
y > -1*(weight1/weight2) * x – (shift/weight2)

What this means is that a single perceptron fires *(= 1 = TRUE)* if a piece of data is on one side of a line, and doesn't fire *(= 0 = FALSE)* if it is on the other side - just like our watermelons and lemons example above.

Simply, a perceptron just divides the x-y plane with a line, and tells us which side of the line data falls in!

Neural Networks and Face-Bot

For Face-Bot, we used two sensors. Let's suppose we used the sound sensor and ultra-sound sensor.

Now suppose we use two variables:

- int volume – to store the intensity from the sound sensor
- int distance- to store the distance from the ultra-sound sensor

At first, we will have only two emotions: *happy* and *sad*.

Training

We will now "train" our robot to classify various inputs into things that make our robot either *happy* or *sad*.

Let's suppose we want our robot to be:

- *sad* if we are quiet and far away (volume is low, distance is large)
- *happy* if we are loud and close (volume is high, distance is small).

Normally, we would pass in each data point, telling our robot to be "happy" or "sad", based on how high it is. The robot would then draw a line separating the data points. The more data we entered, the better the line would be at dividing between happy and sad.

We will skip this part and simply manually enter a line to divide up our data.

Classification

Now we are ready to have our robot react to different inputs, and be happy or sad!

Here is the algorithm for what we are trying to do in a perceptron.

This is not exactly correct syntax, so we keep it within a comment block:

```
/*
int perceptron(int weight1, int weight2, int shift, int volume, int distance) {
        if (weight1 * volume + weight2 * distance + shift > 0)      {
                return 1;
        }
        else {
                return 0;
        }
}
*/
```

This function can return a 1 (=happy), or 0 (=sad), depending on the input parameters.

That was pretty simple!

Again, the perceptron 'fires' (returns 1) if (WEIGHT1 * volume + WEIGHT2 * distance + SHIFT > 0).

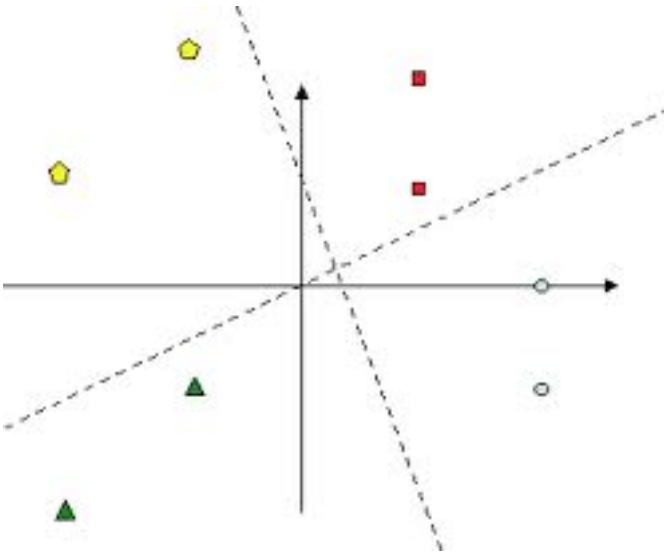
It does not 'fire' (returns 0) if (WEIGHT1 * volume + WEIGHT2 * distance + SHIFT <= 0)

So, the perceptron 'knows' which side of the line the data is on!

Linking two perceptrons using *logical operations*

Once we have one perceptron, we can add a second perceptron to our neural network.

This is like drawing another line on the graph!



Now, we can say something like this:

If we are to the right of the more vertical line, perceptron1 fires.
Else, perceptron1 does not fire.

Also -

If we are above the more horizontal line, perceptron2 fires.
Else, perceptron2 does not fire

Now we can divide our data into *4 regions* using just these two perceptrons.

We can link the two perceptrons together using simple ***logic*** operations.

/*

If perceptron1 'fires' AND perceptron2 'fires', be happy. (top-right region)

If perceptron1 'fires' AND NOT perceptron2 'fires' be sad. (top-left region)

If perceptron1 NOT 'fire' AND perceptron2 'fires', be angry. (botton-right region)

If perceptron1 NOT 'fire' AND perceptron2 NOT 'fires', be evil. (bottom-left region)

*/

Now, we can write a little neural network subroutine so our robot can express emotions based on inputs from two sensors!!

First, we have to write a subroutine for the second perceptron…

Wait! We already wrote a subroutine that lets us pass in parameters for weight1, weight2, shift, distance, and volume! Since already have a perceptron subroutine, we can re-use it as many times as we want, passing in different values!

Now, we just need a subroutine that combines the outputs of the perceptrons into a neural network using simple *logic operations*…

Again, we don't want to worry about syntax, so we put our algorithm inside a comment block:

```
/*

//NOTE: ALL OF THESE #DEFINE STATEMENTS GO AT THE VERY TOP OF THE WHOLE PROGRAM!!

#define PERCEPT1_WEIGHT1 10  //weights for first perceptron
#define PERCEPT1_WEIGHT2 10
#define PERCEPT1_SHIFT 5


#define PERCEPT2_WEIGHT1 10  //weights for second perceptron
#define PERCEPT2_WEIGHT2 10
#define PERCEPT2_SHIFT 5

//subroutine takes distance and volume as input, returns emotion as output
//output:  0 = happy, 1= sad, 2 = angry, 3 = evil
int neuralNetwork(int distance, int volume) {

        int perceptron1Output;
        int perceptron2Output;

        // put distance and volume into the first perceptron, get an answer
        perceptron1Output = perceptron(PERCEPT1_WEIGHT1, PERCEPT1_WEIGHT2,
        PERCEPT1_SHIFT, distance, volume);

        //put distance and volume into the second perceptron, get an answer
        perceptron2Output = perceptron(PERCEPT2_WEIGHT1, PERCEPT2_WEIGHT2,
        PERCEPT2_SHIFT, distance, volume);

        if (perceptron1Output) {  // perceptron1Output == TRUE == 1

                if (perceptron2Output) {  // perceptron1Output AND perceptron2Output
                return 0;  //happy
                }

                else { // perceptron1Output AND NOT perceptron2Output
                return 1; // sad
                }

        }
```

```
        else { // NOT perceptron1Output

                if (perceptron2Output) { // ! perceptron1Output AND perceptron2Output
                return 2;  //angry
                }

                else {  // ! perceptron1Output AND ! perceptron2Output
                return 3; // evil
                }

        }
}
*/
```

Once we enter the program for the perceptron subroutine and the neural network subroutine, we can use this to classify input (volume, distance) into emotions.

We can use them like this:

```
/*
task main()  {

        //declare variables
        int volume, distance;
        int emotion; //0 = happy, 1= sad, 2 = angry, 3 = evil

        while(true) {   //loop forever

                //get input
                distance = ultrasound(IN_1);
                volume = sound(IN_2);

                //run neural net on input
                emotion = neuralNetwork(distance, volume);

                //create output based on emotions from our neural network

                if (emotion == 0) {
                        happy();  //call subroutine to make face happy
                }

                else if (emotion == 1) {
                        sad();  //call subroutine to make face sad
                }

                else if (emotion == 2) {
                        angry();  //call subroutine to make face sad
                }

                else if (emotion == 3) {
                        evil();  //call subroutine to make face sad
                }

        }
}
*/
```

We are now ready to combine the above steps and make our own program intelligent!

Your tasks:

- Contruct a Face-Bot! Face-Bot should take 2 inputs (ultra-sonic, sound, or light), and make 4 facial expressions based on combinations of these inputs.

- Go through the NXC tutorial to make sure you understand the NXC language. Program your face-bot to respond with facial expressions to inputs.

- Now for the A.I. Take the above block-comment algorithms (perceptron, neuralNetwork, and main) and copy them into your program (you may have used different inputs, or different emotions). **Use them to write working subroutines. Modify them to work with your robot.**

- **Take some sample sensor readings.**

  Figure out how loud different sounds are. What is the lowest and highest reasonable inputs (do not scream!).

  If you are using the light sensor, figure out the lowest and highest reasonable values you can get for input without walking into a closet or shining a bright light.

Sensor:	High Value	Low Value

- Decide which sensor input you are using for X (ex: volume) and which input for Y (ex: distance)

  Draw an X-Y plane. Plot some possible input values on the X-Y plane. Based on these points, decide on a reasonable way to draw 2 lines to divide up the plane into 4 regions. **Draw two intersecting lines to divide the data in the plane into obvious regions.** (loud and close, soft and close, loud and far, soft and far, etc.)

- Figure out the equations of the 2 lines (slope and y-intercept):

  Remember: y = m x + b
  1:

  2:

- Now, figure out the corresponding values of weight1, weight2, and shift to make this line.

  We know from earlier that:
  y > -1 * (weight1/weight2) * x – (shift/weight2)

  Based on this equation:
  m (slope) = -1 * (weight1/weight2)
  b (y-intercept) = -(shift/weight2)

  We can also figure out that:
  - (shift/weight2) = (y + (weight1/weight2)*x)
  shift = -1*weight2*(y + (weight1/weight2)*x)

  shift = -1 * weight2*y - weight1*x
  weight2 = -shift/b
  weight1 = -1 * m*weight2

  From these last 3 formulas, figure out what shift, weight2, and weight1 are for each of your 2 lines in the x-y plane.

  Line1: Shift:

  weight2:

  weight1:

  Line2: Shift:

  weight2:

  weight1:

- You are really adjusting the lines that are in the x-y plane, so the perceptrons can decide which a particular combination of inputs (distance, volume) causes the robot to feel a certain way!

  Adjust the constants (#define statements) at the beginning of your program to correctly represent the lines you chose.

  Now your robot can choose an emotion based on inputs…!

  Run your robot and give it a try! Does it respond to input the way you thought it would? If not, make sure you have done everything correctly.

  You now have an artificially-intelligent face-bot!!