X^2 REPORT: Homework 1 --- Artificial Neural Networks

Gage Heeringa
CS 4811
2/12/2016

1. The software

1. A pdf printout of your program

The relevant files containing code are included at the end of the document: XSquaredNetwork.java , MultilayerPerceptron.java , and TrainingSetGenerator.java.

2. Describe the design of your program in essay format. Also include

1. Did you fully implement the pseudocode provided?

2. Does every component work?

3. How can the number of layers or nodes be changed?

The XSquaredNetwork class extends MultilayerPerceptron. Every computation relevant to a neural network is inside the MultilayerPerceptron class, and XSquaredNetwork is used simply for (1) providing examples (provided by the TrainingSetGenerator class) to learn and (2) observing learning. The network uses back-propagation learning.

XSquaredNetwork consists of three layers: input, hidden, and output. For my configuration, in the input layer, there are 2 units. In the hidden, there are 5. In the output, there's 1. Since my network is tested for y >= x^2 for -4 <= x < 4, TrainingSetGenerator randomly generates 4*50 = 200 tests with -4 <= x < 4. The network is tested using the random restart method. That is, if the network doesn't successfully learn the y >= x^2 classification function after a specified number of rounds (15,000 in my configuration), it has a few chances (4 in my configuration) to attempt learning the function again (classification >= 88% accuracy) with a fresh network (the weights are initialized randomly again).

I fully implemented the pseudocode provided. The back-propagation algorithm was implemented following the algorithm in the textbook and notes from class.

Every component of the network works for a limit range of x.

XSquaredNetwork extends the MultilayerPerceptron class, in which the constructor determines the number of units in each layer. The network only consists of three layers: input, hidden, and output, but the number of units in each layer is variable.

2. The training process

1. How did you decide on the network configuration to use?

I generated what I thought could be a good training set, then would start with 3 hidden units and train the network with increasingly more units. I suppose with one hidden layer, the network requires many hidden units to classify y >= x^2 for even double digit x values. I decided for x in range [-4, 4), 5 hidden units is the optimal choice.

2. How were the weights initialized?

Randomly with equal probability between (-2.0, 2.0).

3. How many iterations were needed to converge (or stop)?

For 5 hidden units and x in range [-4,4), convergence is difficult to achieve. 15,000 rounds will usually give about 88% accuracy, the threshold for accepting the network state. Decent accuracy is difficult to achieve for x > 4.

4. What was the training data used (provide in text)? How did you create the training data?

The TrainingSetGenerator class randomly generates training examples within the specified range x:[-4, 4). Since the maximum range is set to 4, 4*50 = 200 training examples are generates, where y is slightly above, slightly below, or equal to x^2.

3. The end results

1. Tell how far the network converged. What percentage of the training data is classified correctly?

The network completely converges with 100% correct classification most of the time with 3 hidden units and x in range [-2, 2) or [-3, 3). However, with x in range [-4, 4), the network hardly converges with 100% correct classification. For the 15,000 round limit, the network is most of the time within 88%-93% accuracy.

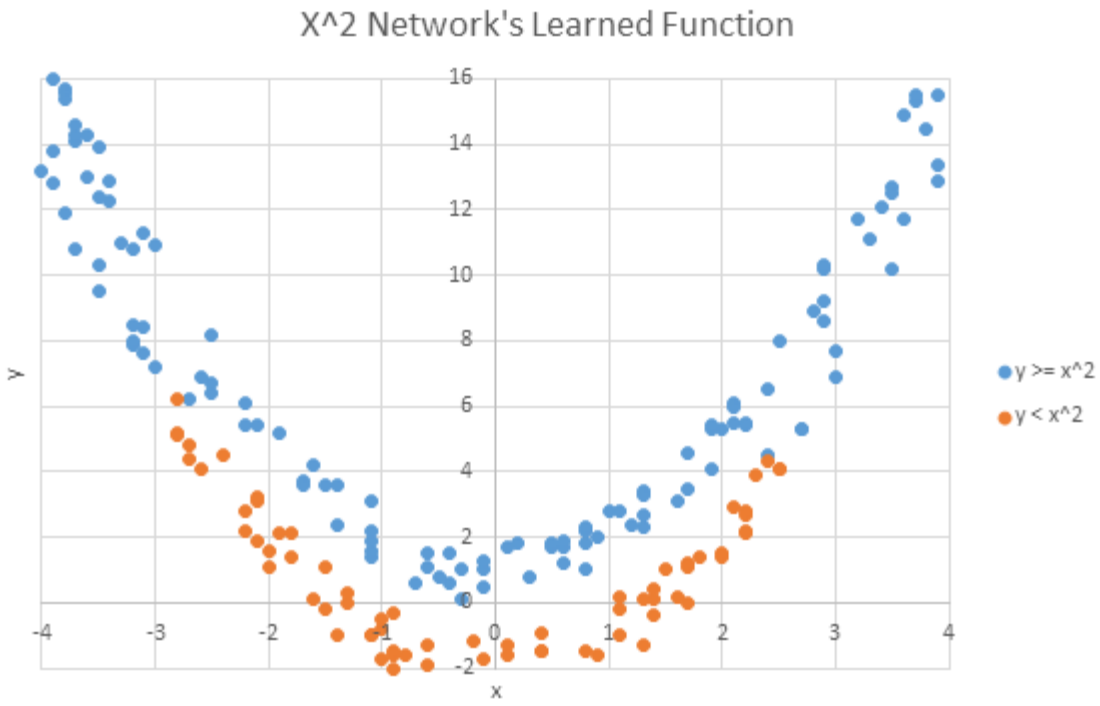
2. Show the following

- **The trained networks (nodes and weights)**

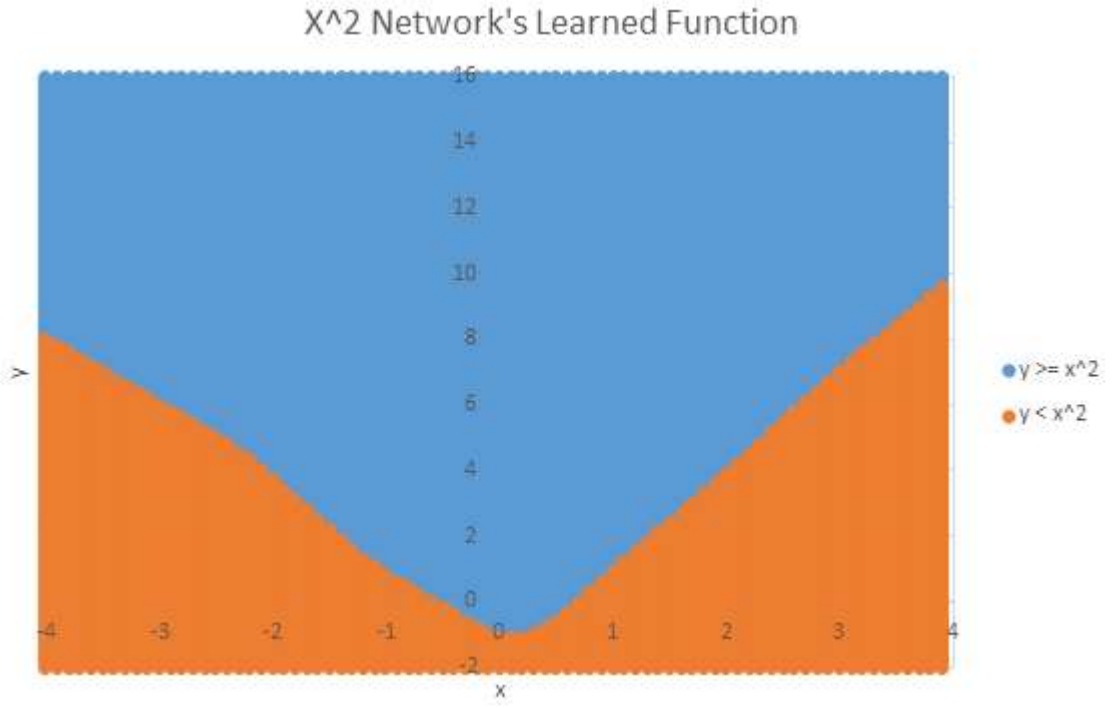
A typical final configuration for the network with 5 hidden units:

```
Network HALTED LEARNING after 15000 rounds.
      Weights between input and hidden units:
W[0][2]=2.8014  W[0][3]=-1.8545  W[0][4]=-0.4688  W[0][5]=-3.2004  W[0][6]=-
1.9777
W[1][2]=5.5024  W[1][3]=-8.9497  W[1][4]=-5.1428  W[1][5]=8.6115
W[1][6]=7.4297
      Weights between hidden and output units:
W[2][7]=5.4412
W[3][7]=-5.0257
W[4][7]=-1.6476
W[5][7]=-4.8951
W[6][7]=-4.5601
*** 0.930 accuracy (186/200 examples classified correctly) ***
*** 0.930 >= 0.88 classification accuracy. Accepting network state. ***
```

- **A graph of the training data**



- **A graph of the function learned by using more inputs than the training data**



MultilayerPerceptron.java

```
import java.io.OutputStream;
import java.io.PrintStream;

/**@author Gage Heeringa
 * 2/6/2016
 * CS 4811 - Program 1
 *
 * A structure for a feed-forward neural network that consists of an input, a
hidden, and an output layer of units.
 * Back-propagation is used for learning.
 *
 * Only difference with this MLP and the one turned in with XorNetwork is
that this MLP class allows for training
 * with input vector "x" containing doubles (rather than just integers).
 */
public class MultilayerPerceptron{

	/* Globals */
	PrintStream ps; //System.out

	final int BIAS = 1; //"magnitude of the threshold"
	final int ALPHA = 1; //= learning rate = step size
	int inputSize; //# units in input layer
	int hiddenSize;
	int outputSize;
	int nrUnits; //# units in network
	double a[]; //activation values a.k.a. unit values
	double W[][]; //weights (lower index points to higher, so
W[lower][higher] will have the weight
	//W[i][i] = unit bias
	double D[]; //delta values for each unit. they're used for updating the
weights

	/* Constructor */
	MultilayerPerceptron(int inputSize, int hiddenSize, int outputSize){
		ps = System.out;

		this.inputSize = inputSize;
		this.hiddenSize = hiddenSize;
		this.outputSize = outputSize;
		nrUnits = inputSize + hiddenSize + outputSize;
		a = new double[nrUnits];
		W = new double[nrUnits][nrUnits];
		D = new double[nrUnits];
		//assign initial random weights
		for(int i = 0; i < W.length; i++){
			for(int j = 0; j < W[i].length; j++){
				if( (int)(Math.random()*2)/*[0,1]*/ == 1){
					W[i][j] = (int)(Math.random()*2); /*[0,2]*/
				}
				else{
					W[i][j] = -1*(int)(Math.random()*2); /*[-2,0]*/
				}
			}
		}
```

```
            /*
            //Dr. Onder's NXOR example:
            W[0][2] = 2; //W-I1-H1
            W[0][3] = 1; //W-I1-H2
            W[1][2] = -2; //W-I2-H1
            W[1][3] = 3; //W-I2-H2
            W[2][4] = 3; //W-H1-T1
            W[3][4] = -2; //W-H2-T1
            W[0][0] = 1;
            W[1][1] = 1;
            W[2][2] = 0; //H1
            W[3][3] = -1;
            W[4][4] = -1; //T1
             */
      }

      /**Given an input vector "x" and correct hypothesis vector "y", perform
one round of back-propagation learning.
       */
      void training(int[] x, int[] y){
            forwardpropagate(x, y); //major step 1
            backpropagate(); //major step 2
      }

      /**Given an input vector "x" and correct hypothesis vector "y", perform
one round of back-propagation learning.
       */
      void training(double[] x, int[] y){
            forwardpropagate(x, y); //major step 1
            backpropagate(); //major step 2
      }

      /**Given input vector "x", propagate x forward to compute a_j = g(in_j)
for each unit.
       * Given correct hypothesis vector "y", compute the error for the
output layer to update its delta values.
       *
       * Return the output vector.
       * (a_j = the unit j's value) (g = the activation function) (in = the
unit input function)
       */
      double[] forwardpropagate(int x[], int y[]){
            disablePrinting();
            /* input layer ( a_i = x_i ) */
            for(int i = 0; i < inputSize; i++){
                  a[i] = x[i];
            }

            /* hidden layer ( a_j = g(in_j) ) */
            for(int j = inputSize; j < inputSize + hiddenSize; j++){
                  System.out.println("I-H j=" + j);
                  double in_j = BIAS*W[j][j]; /* unit input function for unit
j = Sum(W_i,j * a_j) over i
                  = sum of weights*activations (between unit j and units in
previous layer) */
                  for(int i = 0; i < inputSize; i++){
                        in_j += W[i][j]*a[i];
```

```
                  }
                  System.out.println("   in_j=" + in_j);
                  a[j] = g(in_j); //apply activation function "g" to unit
input function for j
                  System.out.println("   sigmoid=" + a[j]);
            }

            /* output layer ( a_j = g(in_j) ) */
            for(int j = inputSize + hiddenSize; j < nrUnits; j++){
                  System.out.println("H-T j=" + j);
                  double in_j = BIAS*W[j][j]; /* unit input function for unit
j = Sum(W_i,j * a_j) over i
                  = sum of weights*activations (between unit j and units in
previous layer) */
                  for(int i = inputSize; i < inputSize + hiddenSize; i++){
                        in_j += W[i][j]*a[i];
                  }
                  System.out.println("  in_j=" + in_j);
                  a[j] = g(in_j); //apply activation function "g" to unit
input function for j
                  System.out.println("  sigmoid=" + a[j]);
            }

            /* sanity check: for input layer, a_i = x_i
                                    for hidden/output layers, a_j = g(in_j),
and they are now proper hypothesis vectors */

            /* output vector */
            double ret[] = new double[outputSize];
            for(int i = nrUnits - outputSize; i < nrUnits; i++){
                  ret[i - (nrUnits - outputSize)] = a[i];
            }

            /* compute error and deltas at output layer */
            double[] err = new double[nrUnits]; //y_j - a_j = correct vector
- hypothesis vector = error vector for the output layer
            for(int k = inputSize + hiddenSize; k < nrUnits; k++){
                  err[k] = y[k - (inputSize + hiddenSize)] - a[k];
                  //NOTE: our activation function has the property that g'(x)
= g(x)*( 1 - g(x) )
                  D[k] = a[k]*(1- a[k])*err[k]; //g'(in_k) * (correct -
hypothesis) = modified error = delta_k
                  System.out.printf("D[%d] = %f\n", k, D[k]);
            }

            enablePrinting();
            return ret;
      }

      /**Given input vector "x", propagate x forward to compute a_j = g(in_j)
for each unit.
       * Given correct hypothesis vector "y", compute the error for the
output layer to update its delta values.
       *
       * Return the output vector.
       * (a_j = the unit j's value) (g = the activation function) (in = the
unit input function)
```

```
	 */
	double[] forwardpropagate(double x[], int y[]){
		disablePrinting();
		/* input layer ( a_i = x_i ) */
		for(int i = 0; i < inputSize; i++){
			a[i] = x[i];
		}

		/* hidden layer ( a_j = g(in_j) ) */
		for(int j = inputSize; j < inputSize + hiddenSize; j++){
			System.out.println("I-H j=" + j);
			double in_j = BIAS*W[j][j]; /* unit input function for unit
j = Sum(W_i,j * a_j) over i
			= sum of weights*activations (between unit j and units in
previous layer) */
			for(int i = 0; i < inputSize; i++){
				in_j += W[i][j]*a[i];
			}
			System.out.println("   in_j=" + in_j);
			a[j] = g(in_j); //apply activation function "g" to unit
input function for j
			System.out.println("   sigmoid=" + a[j]);
		}

		/* output layer ( a_j = g(in_j) ) */
		for(int j = inputSize + hiddenSize; j < nrUnits; j++){
			System.out.println("H-T j=" + j);
			double in_j = BIAS*W[j][j]; /* unit input function for unit
j = Sum(W_i,j * a_j) over i
			= sum of weights*activations (between unit j and units in
previous layer) */
			for(int i = inputSize; i < inputSize + hiddenSize; i++){
				in_j += W[i][j]*a[i];
			}
			System.out.println("  in_j=" + in_j);
			a[j] = g(in_j); //apply activation function "g" to unit
input function for j
			System.out.println("  sigmoid=" + a[j]);
		}

		/* sanity check: for input layer, a_i = x_i
					for hidden/output layers, a_j = g(in_j),
and they are now proper hypothesis vectors */

		/* output vector */
		double ret[] = new double[outputSize];
		for(int i = nrUnits - outputSize; i < nrUnits; i++){
			ret[i - (nrUnits - outputSize)] = a[i];
		}

		/* compute error and deltas at output layer */
		double[] err = new double[nrUnits]; //y_j - a_j = correct vector
- hypothesis vector = error vector for the output layer
		for(int k = inputSize + hiddenSize; k < nrUnits; k++){
			err[k] = y[k - (inputSize + hiddenSize)] - a[k];
			//NOTE: our activation function has the property that g'(x)
= g(x)*( 1 - g(x) )
```

```
                  D[k] = a[k]*(1- a[k])*err[k]; //g'(in_k) * (correct -
hypothesis) = modified error = delta_k
                  System.out.printf("D[%d] = %f\n", k, D[k]);
            }

            enablePrinting();
            return ret;
      }

      /**Back-propagate the delta values from output layer -> hidden layer ->
input layer.
       */
      void backpropagate(){
            disablePrinting();

            //"for l = L - 1 to 1 do"
            //hidden layer
            for(int i = inputSize; i < inputSize + hiddenSize; i++){
                  // at this point, a[j] = g(in_j)
                  double sum = 0; //Sum( W_i,j * Delta_j ) over j, so
concerns units i is pointing to
                  for(int j = inputSize + hiddenSize; j < nrUnits; j++){
                        sum += W[i][j]*D[j];
                  }
                  //recall: g'(x) = g(x)*( 1 - g(x) )
                  D[i] = a[i]*(1- a[i])*sum; //D_i <- g'(in_i)*Sum( W_i,j *
Delta_j )
            }

            //input layer
            for(int i = 0; i < inputSize; i++){
                  // at this point, a[j] = g(in_j)
                  double sum = 0; //Sum( W_i,j * Delta_j ) over j, so
concerns units i is pointing to
                  for(int j = inputSize; j < inputSize + hiddenSize; j++){
                        sum += W[i][j]*D[j];
                  }
                  //recall: g'(x) = g(x)*( 1 - g(x) )
                  D[i] = a[i]*(1- a[i])*sum; //D_i <- g'(in_i)*Sum( W_i,j *
Delta_j )
            }

            for(int i = 0; i < D.length; i++){
                  System.out.printf("D[%d] = %f\n", i, D[i]);
            }

            /* adjust all the weights */
            for(int i = 0; i < W.length; i++){
                  for(int j = 0; j < W[i].length; j++){
                        W[i][j] += ALPHA * a[i] * D[j];
                  }
            }

            //print weights - also prints irrelevant weights (between
input/output, units in same layer, etc.)
            for(int i = 0; i < W.length; i++){
                  for(int j = i; j < W.length; j++){
```

```
                        if(i != j && W[i][j] != 0){
                              System.out.printf("W[%d][%d]=%.4f\n", i, j,
W[i][j]);
                        }
                  }
            }

            //print bias
            for(int i = 0; i < W.length; i++){
                  System.out.printf("bias[%d][%d]=%.4f\n", i, i, W[i][i]);
            }

            enablePrinting();
      }

      /**Once the network has learned, use it with this function.
       */
      int[] applyKnowledge(int x[]){
            /* input layer ( a_i = x_i ) */
            for(int i = 0; i < inputSize; i++){
                  a[i] = x[i];
            }

            /* hidden layer ( a_j = g(in_j) ) */
            for(int j = inputSize; j < inputSize + hiddenSize; j++){
                  double in_j = BIAS*W[j][j]; /* unit input function for unit
j = Sum(W_i,j * a_j) over i
                  = sum of weights*activations (between unit j and units in
previous layer) = vector w . vector x */
                  for(int i = 0; i < inputSize; i++){
                        in_j += W[i][j]*a[i];
                  }

                  /* apply hard threshold function to unit input function for
j */
                  a[j] = g(in_j);
            }

            /* output layer ( a_j = Threshold(in_j) ) */
            int ret[] = new int[outputSize]; //output vector
            int reti = 0;
            for(int j = inputSize + hiddenSize; j < nrUnits; j++){
                  double in_j = BIAS*W[j][j]; /* unit input function for unit
j = Sum(W_i,j * a_j) over i
                  = sum of weights*activations (between unit j and units in
previous layer) = vector w . vector x */
                  for(int i = inputSize; i < inputSize + hiddenSize; i++){
                        in_j += W[i][j]*a[i];
                  }

                  /* apply hard threshold function to unit input function for
j (the hypothesis vector,
                   * given input vector "x", is 1 if the weight vector dot
the input vector is >= 0, else 0 */
                  if(in_j >= 0){
                        ret[reti++] = 1;
                  }
```

```
                else{
                        ret[reti++] = 0;
                }
        }

        return ret;
    }

    /**Once the network has learned, this function is useful for generating
data points
     * to determine the shape of the network's classification boundary.
     */
    int[] applyKnowledge(double x[]){
        /* input layer ( a_i = x_i ) */
        for(int i = 0; i < inputSize; i++){
                a[i] = x[i];
        }

        /* hidden layer ( a_j = g(in_j) ) */
        for(int j = inputSize; j < inputSize + hiddenSize; j++){
                double in_j = BIAS*W[j][j]; /* unit input function for unit
j = Sum(W_i,j * a_j) over i
                = sum of weights*activations (between unit j and units in
previous layer) = vector w . vector x */
                for(int i = 0; i < inputSize; i++){
                        in_j += W[i][j]*a[i];
                }

                a[j] = g(in_j); //apply activation function "g" to unit
input function for j
        }

        /* output layer ( a_j = Threshold(in_j) ) */
        int ret[] = new int[outputSize]; //output vector
        int reti = 0;
        for(int j = inputSize + hiddenSize; j < nrUnits; j++){
                double in_j = BIAS*W[j][j]; /* unit input function for unit
j = Sum(W_i,j * a_j) over i
                = sum of weights*activations (between unit j and units in
previous layer) = vector w . vector x */
                for(int i = inputSize; i < inputSize + hiddenSize; i++){
                        in_j += W[i][j]*a[i];
                }

                /* apply hard threshold function to unit input function for
j (the vector component a_j,
                 * given input vector "x", is 1 if the weight vector dot
the input vector is >= 0, else 0 */
                if(in_j >= 0){
                        ret[reti++] = 1;
                }
                else{
                        ret[reti++] = 0;
                }
        }

        return ret;
```

```
	}

	/**The sigmoid activation function.
	 */
	double g(double x){
		return 1/ (1 + Math.exp(-1*x)); //= 1/ (1 + e^(-x))
	}

	/**The network is reborn.
	 */
	void randomRestart(){
		a = new double[nrUnits];
		W = new double[nrUnits][nrUnits];
		D = new double[nrUnits];
		//assign initial random weights
		for(int i = 0; i < W.length; i++){
			for(int j = 0; j < W[i].length; j++){
				if( (int)(Math.random()*2)/*[0,1]*/ == 1){
					W[i][j] = (int)(Math.random()*2); /*[0,2]*/
				}
				else{
					W[i][j] = -1*(int)(Math.random()*2); /*[-2,0]*/
				}
			}
		}
	}

	/**Redirect PrintStream so don't have to comment out everything for
debugging.
	 * Source: http://stackoverflow.com/questions/8363493/hiding-system-
out-print-calls-of-a-class
	 */
	void disablePrinting(){
		PrintStream theVoid = new PrintStream(new OutputStream(){
			public void write(int b) {
			}
		});
		System.setOut(theVoid);
	}

	/**Reenable printing.
	 */
	void enablePrinting(){
		System.setOut(ps);
	}
}
```

XSquaredNetwork.java

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.ArrayList;

/**@author Gage Heeringa
 * Completed 2/12/2016
 * CS 4811 - Program 1
 *
 * A feed-forward neural network of an input, a hidden, and an output layer
of units.
 * The network uses back-propagation to classify (x,y) coordinates as (1) on
or above the y = x^2 parabola,
 * OR (2) below the parabola.
 */
public class XSquaredNetwork extends MultilayerPerceptron{

	final int restartLimit = 4; // number of random restarts the network is
allowed
	final int roundLimit = 15000; // numbers of rounds of learning the
network is allowed to learn the function

	/**Main
	 */
	public static void main(String args[]){
		XSquaredNetwork x = new XSquaredNetwork(2, 5, 1); // make network
and specify the number of units in each layer
		XSquaredExample[] examples = new XSquaredExample[]{ // examples
for learning

				// y >= x^2 ?  (Points generated using
TrainingSetGenerator)
				//x:[-4,4), 200 tests, 5 units
				new XSquaredExample(2.7,2.2, 0), new
XSquaredExample(1.4,-1.8, 0), new XSquaredExample(1.7,0.5, 1), new
XSquaredExample(-1.3,1.3, 0),
				new XSquaredExample(-2.4,0.1, 0), new
XSquaredExample(-1.0,1.1, 0), new XSquaredExample(3.1,-2.1, 0), new
XSquaredExample(16.4,-4.0, 1),
				new XSquaredExample(1.9,-2.1, 0), new
XSquaredExample(16.3,3.8, 1), new XSquaredExample(18.0,4.0, 1), new
XSquaredExample(2.4,-1.4, 1),
				new XSquaredExample(3.6,-1.4, 1), new
XSquaredExample(5.1,-2.8, 0), new XSquaredExample(14.5,3.8, 1), new
XSquaredExample(1.4,2.0, 0),
				new XSquaredExample(8.6,2.9, 1), new
XSquaredExample(2.8,1.0, 1), new XSquaredExample(8.4,-3.1, 0), new
XSquaredExample(0.1,1.3, 0),
				new XSquaredExample(0.6,-0.4, 1), new
XSquaredExample(15.3,3.7, 1), new XSquaredExample(-1.9,-0.6, 0), new
XSquaredExample(3.9,2.3, 0),
				new XSquaredExample(10.8,-3.2, 1), new
XSquaredExample(1.4,-1.1, 1), new XSquaredExample(12.7,3.5, 1), new
XSquaredExample(3.4,1.3, 1),
```

```
                        new XSquaredExample(1.0,-0.3, 1), new
XSquaredExample(-1.5,0.4, 0), new XSquaredExample(4.6,1.7, 1), new
XSquaredExample(1.6,-2.0, 0),
                        new XSquaredExample(0.1,-1.6, 0), new
XSquaredExample(10.2,3.5, 0), new XSquaredExample(-1.3,0.1, 0), new
XSquaredExample(4.4,-2.7, 0),
                        new XSquaredExample(1.2,0.6, 1), new
XSquaredExample(12.5,3.5, 1), new XSquaredExample(2.0,0.9, 1), new
XSquaredExample(0.8,-0.5, 1),
                        new XSquaredExample(6.7,-2.5, 1), new
XSquaredExample(0.4,1.4, 0), new XSquaredExample(1.8,0.8, 1), new
XSquaredExample(14.1,-3.7, 1),
                        new XSquaredExample(0.1,1.4, 0), new
XSquaredExample(8.2,-2.5, 1), new XSquaredExample(4.1,1.9, 1), new
XSquaredExample(0.0,1.7, 0),
                        new XSquaredExample(7.9,-3.2, 0), new
XSquaredExample(0.2,1.1, 0), new XSquaredExample(1.1,1.7, 0), new
XSquaredExample(14.3,-3.6, 1),
                        new XSquaredExample(-1.6,-0.9, 0), new
XSquaredExample(16.8,-3.9, 1), new XSquaredExample(1.0,0.8, 1), new
XSquaredExample(-1.0,-1.1, 0),
                        new XSquaredExample(-1.0,-1.1, 0), new
XSquaredExample(8.9,2.8, 1), new XSquaredExample(2.8,1.1, 1), new
XSquaredExample(1.9,-1.1, 1),
                        new XSquaredExample(3.6,-1.5, 1), new
XSquaredExample(5.3,1.9, 1), new XSquaredExample(2.2,-2.2, 0), new
XSquaredExample(6.1,2.1, 1),
                        new XSquaredExample(13.9,-3.5, 1), new
XSquaredExample(1.4,1.8, 0), new XSquaredExample(2.2,0.8, 1), new
XSquaredExample(1.8,0.5, 1),
                        new XSquaredExample(3.1,-1.1, 1), new
XSquaredExample(4.5,-2.4, 0), new XSquaredExample(8.5,-3.2, 0), new
XSquaredExample(-1.5,0.8, 0),
                        new XSquaredExample(1.0,-0.1, 1), new
XSquaredExample(6.2,-2.8, 0), new XSquaredExample(-0.2,-1.5, 0), new
XSquaredExample(14.9,3.6, 1),
                        new XSquaredExample(1.0,1.5, 0), new
XSquaredExample(13.8,-3.9, 0), new XSquaredExample(7.2,-3.0, 0), new
XSquaredExample(-2.0,-0.9, 0),
                        new XSquaredExample(11.7,3.2, 1), new
XSquaredExample(13.0,-3.6, 1), new XSquaredExample(5.3,2.7, 0), new
XSquaredExample(5.4,-2.2, 1),
                        new XSquaredExample(2.1,-1.9, 0), new
XSquaredExample(9.5,-3.5, 0), new XSquaredExample(3.7,-1.7, 1), new
XSquaredExample(12.9,3.9, 0),
                        new XSquaredExample(2.1,2.2, 0), new
XSquaredExample(12.1,3.4, 1), new XSquaredExample(1.3,-0.1, 1), new
XSquaredExample(0.5,-0.1, 1),
                        new XSquaredExample(-1.3,-0.6, 0), new
XSquaredExample(15.7,-3.8, 1), new XSquaredExample(15.6,-3.8, 1), new
XSquaredExample(6.1,-2.2, 1),
                        new XSquaredExample(2.3,1.3, 1), new
XSquaredExample(-1.6,0.1, 0), new XSquaredExample(10.2,2.9, 1), new
XSquaredExample(12.9,-3.4, 1),
                        new XSquaredExample(10.9,-3.0, 1), new
XSquaredExample(2.3,0.8, 1), new XSquaredExample(6.9,-2.6, 1), new
XSquaredExample(14.6,-3.7, 1),
```

```
                        new XSquaredExample(-1.7,-1.0, 0), new
XSquaredExample(1.6,-1.1, 1), new XSquaredExample(5.2,-1.9, 1), new
XSquaredExample(0.1,-0.3, 1),
                        new XSquaredExample(2.7,1.3, 1), new
XSquaredExample(-1.5,-0.9, 0), new XSquaredExample(2.1,-1.8, 0), new
XSquaredExample(2.8,-2.2, 0),
                        new XSquaredExample(-1.6,0.9, 0), new
XSquaredExample(2.2,2.2, 0), new XSquaredExample(1.5,-0.6, 1), new
XSquaredExample(1.1,-2.0, 0),
                        new XSquaredExample(0.8,0.3, 1), new
XSquaredExample(4.1,-2.6, 0), new XSquaredExample(2.8,2.2, 0), new
XSquaredExample(-1.5,0.4, 0),
                        new XSquaredExample(11.1,3.3, 1), new
XSquaredExample(-2.4,0.5, 0), new XSquaredExample(1.7,0.6, 1), new
XSquaredExample(16.0,-3.9, 1),
                        new XSquaredExample(10.3,-3.5, 0), new
XSquaredExample(16.3,3.8, 1), new XSquaredExample(-0.3,-0.9, 0), new
XSquaredExample(8.0,2.5, 1),
                        new XSquaredExample(4.1,2.5, 0), new
XSquaredExample(12.4,-3.5, 1), new XSquaredExample(13.2,-4.0, 0), new
XSquaredExample(-0.8,-1.0, 0),
                        new XSquaredExample(-2.9,-0.0, 0), new
XSquaredExample(-0.4,1.4, 0), new XSquaredExample(-0.9,0.4, 0), new
XSquaredExample(5.3,2.7, 0),
                        new XSquaredExample(2.9,2.1, 0), new
XSquaredExample(10.8,-3.7, 0), new XSquaredExample(13.4,3.9, 0), new
XSquaredExample(4.8,-2.7, 0),
                        new XSquaredExample(1.2,1.7, 0), new
XSquaredExample(-2.6,-0.4, 0), new XSquaredExample(0.2,1.6, 0), new
XSquaredExample(-2.6,-0.2, 0),
                        new XSquaredExample(5.3,2.0, 1), new
XSquaredExample(5.4,-2.1, 1), new XSquaredExample(1.1,-0.6, 1), new
XSquaredExample(5.5,2.1, 1),
                        new XSquaredExample(15.5,3.7, 1), new
XSquaredExample(11.7,3.6, 0), new XSquaredExample(6.2,-2.7, 0), new
XSquaredExample(-1.0,-1.4, 0),
                        new XSquaredExample(6.5,2.4, 1), new
XSquaredExample(-1.2,-0.2, 0), new XSquaredExample(7.6,-3.1, 0), new
XSquaredExample(4.2,-1.6, 1),
                        new XSquaredExample(11.3,-3.1, 1), new
XSquaredExample(10.3,2.9, 1), new XSquaredExample(4.1,2.5, 0), new
XSquaredExample(-0.2,1.1, 0),
                        new XSquaredExample(-2.2,0.2, 0), new
XSquaredExample(5.4,2.2, 1), new XSquaredExample(4.3,2.4, 0), new
XSquaredExample(11.0,-3.3, 1),
                        new XSquaredExample(5.5,2.2, 1), new
XSquaredExample(6.9,3.0, 0), new XSquaredExample(0.3,-1.3, 0), new
XSquaredExample(9.2,2.9, 1),
                        new XSquaredExample(7.7,3.0, 0), new
XSquaredExample(1.5,-0.4, 1), new XSquaredExample(8.0,-3.2, 0), new
XSquaredExample(-0.5,-1.0, 0),
                        new XSquaredExample(-1.6,-0.8, 0), new
XSquaredExample(3.2,-2.1, 0), new XSquaredExample(6.0,2.1, 1), new
XSquaredExample(3.5,1.7, 1),
                        new XSquaredExample(-1.7,-0.1, 0), new
XSquaredExample(14.3,-3.7, 1), new XSquaredExample(12.3,-3.4, 1), new
XSquaredExample(1.9,0.6, 1),
```

```
                        new XSquaredExample(1.1,-1.5, 0), new
XSquaredExample(6.4,-2.5, 1), new XSquaredExample(3.3,1.3, 1), new
XSquaredExample(5.2,-2.8, 0),
                        new XSquaredExample(1.5,2.0, 0), new
XSquaredExample(2.2,-1.1, 1), new XSquaredExample(12.8,-3.9, 0), new
XSquaredExample(5.4,1.9, 1),
                        new XSquaredExample(11.9,-3.8, 0), new
XSquaredExample(0.6,-0.7, 1), new XSquaredExample(1.8,0.2, 1), new
XSquaredExample(4.5,2.4, 0),
                        new XSquaredExample(2.4,1.2, 1), new
XSquaredExample(15.5,3.9, 1), new XSquaredExample(3.6,-1.7, 1), new
XSquaredExample(0.0,-1.3, 0),
                        new XSquaredExample(15.4,-3.8, 1), new
XSquaredExample(16.1,-3.9, 1), new XSquaredExample(3.1,1.6, 1), new
XSquaredExample(1.7,0.1, 1)


            };

            x.randomRestartTesting(examples); // learning and checking
            //x.generatePoints();
            x.generatePoints(examples); //only graph the training set
      }

      /* Constructor */
      XSquaredNetwork(int inputSize, int hiddenSize, int outputSize){
            super(inputSize, hiddenSize, outputSize);
      }

      /**Test the network.
       * Do a random restart if the network does not successfully learn XOR
by the specified number of rounds.
       */
      void randomRestartTesting(XSquaredExample[] examples){
            int success = -1; // see if the network learns all the examples
            for(int i = 0; i <= restartLimit; i++){ // give it some number of
chances
                  if(i > 0){
                        System.out.printf("---------------- (RANDOM RESTART
#%d) ----------------\n", i);
                  }
                  success = test(examples);
                  // if the network reaches the round limit for learning, do
random restart
                  if(success != 0){
                        randomRestart();
                  }
                  else{
                        return;
                  }
            }

            if(success != 0){
                  System.out.printf("The network could not learn with %d
rounds and %d random restarts.\n",
                              roundLimit, restartLimit);
            }
```

```
	}

	/**Test the network.
	 * Return 0 if the network successfully learned, else -1.
	 * Print details on the network's final configurations.
	 */
	int test(XSquaredExample[] examples){

		/* train the network by giving examples of correct
classifications */
		for(int i = 0; i < roundLimit; i++){
			double a = examples[i % examples.length].a; //cycling
through the tests
			double  b = examples[i % examples.length].b;
			int answer = examples[i % examples.length].c;
			double[] x = new double[]{a, b}; //input vector
			int[] y = new int[]{answer}; //correct hypothesis vector
			training(x, y);

			/* learning check: confirm the network produces correct
output for all examples */
			boolean done = true;
			for(int j = 0; j < examples.length; j++){
				a = examples[j].a;
				b = examples[j].b;
				answer = examples[j].c;
				x = new double[]{a, b}; //input vector

				int results[] = applyKnowledge(x); //hard threshold
function is used to classify answer as 0 or 1
				if(results[0] != answer){
					done = false;
					break;
				}
			}
			if(done){
				//print information about the successful network
				System.out.printf("Network completed learning after
%d rounds. (%d tests and %d hidden units)\n",
							i + 1, examples.length, hiddenSize);
				System.out.println("\tWeights between input and
hidden units:");
				for(int k = 0; k < inputSize; k++){
					for(int j = inputSize; j < inputSize +
hiddenSize; j++){
						System.out.printf("W[%d][%d]=%.4f\n", k,
j, W[k][j]);
					}
				}
				System.out.println("\tWeights between hidden and
output units:");
				for(int k = inputSize; k < inputSize + hiddenSize;
k++){
					for(int j = inputSize + hiddenSize; j <
nrUnits; j++){
						System.out.printf("W[%d][%d]=%.4f\n", k,
j, W[k][j]);
```

```
                        }
                    }
                    break;
                }
                if(i == roundLimit - 1){
                    int c0 = 0; //those with answer 0/1 that were
classified right
                    int  c1 = 0;

                    System.out.printf("Network HALTED LEARNING after %d
rounds.\n", i + 1);
                    for(int j = 0; j < examples.length; j++){
                        a = examples[j].a;
                        b = examples[j].b;
                        answer = examples[j].c;
                        x = new double[]{a, b}; //input vector

                        int results[] = applyKnowledge(x); //hard
threshold function is used to classify answer as 0 or 1

                        //classified wrong
                        //if(results[0] != answer){
                        //System.out.printf("\tIs %.1f >= %.1f^2?
Result: <%d>  Answer: <%d>\n", a, b, results[0], answer);
                        //}

                        if(results[0] == answer && answer == 1){
                            c1++;
                        }
                        if(results[0] == answer && answer == 0){
                            c0++;
                        }
                    }
                    //print information about the failed network
                    System.out.println("\tWeights between input and
hidden units:");
                    for(int k = 0; k < inputSize; k++){
                        for(int j = inputSize; j < inputSize +
hiddenSize; j++){

                            System.out.printf("W[%d][%d]=%.4f  ", k,
j, W[k][j]);

                        }
                        System.out.println();
                    }
                    System.out.println("\tWeights between hidden and
output units:");
                    for(int k = inputSize; k < inputSize + hiddenSize;
k++){
                        for(int j = inputSize + hiddenSize; j <
nrUnits; j++){

                            System.out.printf("W[%d][%d]=%.4f  ", k,
j, W[k][j]);

                        }
                        System.out.println();
                    }
```

```
                        /* accept some percentage of accurate classification
*/
                        double perc = (double)(c1 + c0) /
(double)examples.length;
                        System.out.printf("*** %.3f accuracy (%d/%d examples
classified correctly) ***\n",
                                    perc, c0 + c1, examples.length);
                        if(perc >= .88){
                              System.out.printf("*** %.3f >= %.2f
classification accuracy. Accepting network state. ***\n",
                                          perc, .88);
                              return 0;
                        }
                        return -1;
                  }
            }

            return 0;
      }

      /**This generates (x,y) coordinates for graphing data points to depict
the function that
       * the network has learned.
       *
       * NOTE: The network is currently configured to do learning with x
values in range [-4,4],
       * therefore points generated are for x in range [-4,4) with .1
increments and
       * y in range [-2, 4^2=16] with .1 increments.  It's tested rather the
(x,y) points are
       * classified above or on the parabola (y >= x^2), OR below it (y <
x^2).
       * So this is 14,480 points.
       *
       * Two files are generated in the same directory as where the program
is run:
       * "generatedGEQPoints.txt" contains coordinates (x,y) such that y >=
x^2.
       * "generatedLPoints.txt" contains coordinates (x,y) such that y < x^2.
       */
      void generatePoints(){
            ArrayList<String> geqPoints = new ArrayList<String>();
            ArrayList<String> lPoints = new ArrayList<String>();
            for(double i = -4.0; i <= 4.0; i += 0.1){
                  for(double j = -2.0; j <= 16.0; j += 0.1){
                        int results[] = applyKnowledge(new double[]{j, i});
//y, x format when computed...
                        if(results[0] == 1){
                              geqPoints.add(String.format("%.1f, %.1f", i,
j)); //(x,y) coordinate
                        }
                        else{
                              lPoints.add(String.format("%.1f, %.1f", i, j));
//(x,y) coordinate
                        }
                  }
            }
```

```
            // y >= x^2
            PrintWriter out = null;
            try{
                  out = new PrintWriter(new File("generatedGEQPoints.txt"));
            } catch(FileNotFoundException e){
                  e.printStackTrace();
            }
            for(int i = 0; i < geqPoints.size(); i++){
                  out.write(geqPoints.get(i) + "\n");
            }
            out.close();

            // y < x^2
            try{
                  out = new PrintWriter(new File("generatedLPoints.txt"));
            } catch(FileNotFoundException e){
                  e.printStackTrace();
            }
            for(int i = 0; i < lPoints.size(); i++){
                  out.write(lPoints.get(i) + "\n");
            }
            out.close();

            System.out.println(geqPoints.size() + lPoints.size() + "
points");
      }

      /**This generates (x,y) coordinates for graphing the training set to
depict the function that
       * the network has learned.
       *
       * Two files are generated in the same directory as where the program
is run:
       * "generatedGEQPoints.txt" contains coordinates (x,y) such that y >=
x^2.
       * "generatedLPoints.txt" contains coordinates (x,y) such that y < x^2.
       */
      void generatePoints(XSquaredExample[] examples){
            ArrayList<String> geqPoints = new ArrayList<String>();
            ArrayList<String> lPoints = new ArrayList<String>();
            for(int i = 0; i < examples.length; i ++){
                  int results[] = applyKnowledge(new double[]{examples[i].a,
examples[i].b}); //y, x format when computed...
                  if(results[0] == 1){
                        geqPoints.add(String.format("%f, %f", examples[i].b,
examples[i].a)); //(x,y) coordinate
                  }
                  else{
                        lPoints.add(String.format("%f, %f", examples[i].b,
examples[i].a)); //(x,y) coordinate
                  }
      }

            // y >= x^2
            PrintWriter out = null;
            try{
```

```
                out = new PrintWriter(new File("generatedGEQPoints.txt"));
            } catch(FileNotFoundException e){
                e.printStackTrace();
            }
            for(int i = 0; i < geqPoints.size(); i++){
                out.write(geqPoints.get(i) + "\n");
            }
            out.close();

            // y < x^2
            try{
                out = new PrintWriter(new File("generatedLPoints.txt"));
            } catch(FileNotFoundException e){
                e.printStackTrace();
            }
            for(int i = 0; i < lPoints.size(); i++){
                out.write(lPoints.get(i) + "\n");
            }
            out.close();

            System.out.println(geqPoints.size() + lPoints.size() + "
points");
      }
}

/**Example for training the network. (y,x) = inputs, passed = 1 IF y >= x^2,
ELSE 0
 */
class XSquaredExample{
      double a, b;
      int c;
      XSquaredExample(double y, double x, int passed){
            a = y;
            b = x;
            c = passed;
      }
}
```

TrainingSetGenerator.java

```
import java.math.BigDecimal;
import java.math.RoundingMode;

/**@author Gage Heeringa
 * Completed 2/12/2016
 * CS 4811 - Program 1
 *
 * Generate training examples for an XSquaredNetwork.
 */
public class TrainingSetGenerator {

	public static void main(String[] args){
		int c = 0;
		int range = 4; //x:[ -range, range )
		int total = 50*range;

		for(int i = 0; i < total; i++){
			double y, x = 0.0;
			int b = 0;

			if( (int)(Math.random()*2) == 1){ //50/50 chance
				x = round(Math.random()*range, 1); // [0, range]
rounded to 1 decimal place
			}
			else{
				x = -1*round(Math.random()*range, 1); // [-range, 0]
rounded to 1 decimal place
			}

			if( (int)(Math.random()*2) == 1){ //50/50 chance
				y = x*x + Math.random()*x; //y >= x^2
			}
			else{
				y = x*x - Math.random()*x; // y < x^2
			}
			y = round(y, 1);

			if(y >= x*x){
				c++;
				b = 1;
			}

			System.out.printf("new XSquaredExample(%.1f,%.1f, %d), ",
y, x, b);
			if(Math.abs(i)%4 == 3)
				System.out.println();
		}

		System.out.printf("%d/%d had result 1", c, total);
	}

	/**Source: http://stackoverflow.com/questions/2808535/round-a-double-
to-2-decimal-places
	 */
	public static double round(double value, int places) {
```

```
            if (places < 0) throw new IllegalArgumentException();

            BigDecimal bd = new BigDecimal(value);
            bd = bd.setScale(places, RoundingMode.HALF_UP);
            return bd.doubleValue();
        }
}
```