International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 02, February-2022

A Simple Additive Neural Network Algorithm for Point Classification

Patrick Mcdowell, Kuo-Pao Yang, Paulo Regis, John Burris
Department of Computer Science
Southeastern Louisiana University
Hammond, LA 70402 USA

***Abstract*— This paper describes a method to generate and run a Radial Basis Function network designed to classify unstructured/unknown data. The growR3 algorithm is introduced and described along with results of testing on two data sets; a proof on concept and a more complex spiral data set using x and y inputs – versus more dimensions. The work here is preliminary but shows promise because of the algorithm's self-organizing, deterministic nature, and relatively low training time requirements.**

Keywords- Neural Network Algorithm, Radial Basis Function

I. INTRODUCTION

This paper describes a deterministic approach for classifying points for arbitrary patterns of data. The work is in its early stages, yet it has shown good results in classifying non-linearly separable patterns using a single layer of radial basis neurons. This paper shows results for basic low neuron count clustering and high neuron count swirl pattern data. The algorithm can be characterized as a self-organizing radial basis network because it "grows" in order to accommodate the complexity of the presented classification problem.

The motivation of this work lies in the desire to classify data points in complex patterns, such as the swirl pattern, with simple networks. As a general rule, simple networks require less training iterations, and the networks presented here follow this rule. Furthermore, it was felt that an ideal solution would be deterministic so the generated network that solved the problem at hand could easily be replicated, as long as the data set was the same. Another goal was that the solution should be analysis friendly; that is, it should be possible to select a point that the network accepted and know which neuron in the network made the decision to accept it. The work presented in this paper fulfills these goals because the generated solution is a single layer network, thus making it possible to pair accepted points with the neuron that accepted them.

This paper is arranged as follows. The Background section provides some basic information on the perceptron and uses that as a build up to the radial basis neuron. It also provides some information on other methods of classification, most notably the multi-layer feed forward network. The Approach section describes the growR3 algorithm that was developed for this work. In the Results section two examples are discussed, the first being a basic test of the system, the second being the swirl pattern. The takeaway from this section is that the growR3 algorithm can generate a radial basis network that will classify simple or complex data. Simpler problems require fewer neurons, complex requires more, but the programmer does not have to specify the size of the network, the growR3 algorithm builds it accordingly. The Conclusion and Future Work sections provides a summary of the work, and highlights areas in which the growR3 algorithm can be enhanced.

II. BACKGROUND

In this section we discuss the basics of two basic neural units, the perceptron and the radial basis function. We provide detail on the perceptron in order to bolster understanding of the radial basis function, and the need for the radial basis function in order to realize the goals of this research. The treatment of these subjects is an overview at best; an abundance of information is available in textbooks and literature.

A. The Perceptron

The basic perceptron provides the ability to classify linearly separable data in a supervised manner. It creates a hyperplane between the desired data points (the ones deemed as acceptable) and the undesired points; remember in supervised learning, each point is labeled as desired (value 1) or not desired (value -1). We say hyperplane to indicate multi-dimensional input, but put more plainly, for data with two components, x and y, the perceptron creates a line between the two classes of data. Detailed information is available from multiple sources including [1]. Here we present the basics:

w – weights
x – input vector

$$v = \sum w_i x_i$$

i = 1 to n, where n is the number of inputs and weights. The v parameter is the dot product of the input vector and the weights.

The perceptron uses a hard limiting function that behaves as follows:

```
if (v ≥ 0)
  y = 1
else
  y = -1
```

The diagram in Fig. 1 below illustrates the data flow through the perceptron.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 02, February-2022

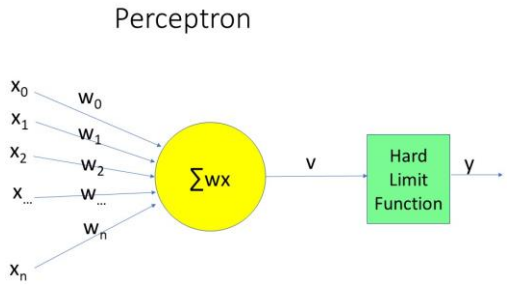

Fig. 1. The x values are the input vector. It is multiplied by the weights, w, resulting in v. The v parameter is passed through the hard limit function resulting in y. If y is 1 the perceptron classifies the input vector x as accepted, otherwise it is rejected.

The updating of the weights is handled by the following process:

Initially all weights are set to 0. At the presentation of each input vector, desired value pair, the weights are updated.

We include an extra input, x_0 which is always -1, and an extra weight w_0. This is referred to as the bias. Its purpose is the shift the hyperplane (line in 2 dimensions) away from the origin. It is the y intercept if using inputs with 2 dimensions.

```
η - the learning rate
x0 = -1

if (y == desired)
  do not update
else if (y == 1) and (desired == -1) {
        for each j from 0 to n {
           weights[j] = weights[j] - (η * x[j])
        }
}
else if (y == -1) and (desired == 1) {
         for each j from 0 to n {
           weights[j] = weights[j] + (η * x[j])
         }
}
```

To get an intuitive feel for how a perceptron works consider a 2 input (x1, x2) system that has been trained. Fig. 2 below illustrates the situation. The shaded portion shows the area in which the perceptron has been trained to accept inputs; in the non-shaded portion the vectors are not accepted.

For two inputs (x_1, x_2) we get:

$$x_1w_1 + x_2w_2 + x_0w_0 = 0$$

Remembering that x_0 is always -1, we get:

$$x_1w_1 + x_2w_2 - w_0 = 0$$

Now, let's rename our variables:

$$x = x_1$$
$$y = x_2$$
$$b = w_0$$

Our equation is now:

$$xw_1 + yw_2 - b = 0$$

Rearranging, we get:

$$y = (-w_1/w_2)*x + b$$

The $(-w_1/w_2)$ term can be thought of as the slope of the line, and the b term is the y intercept, resulting in:

$$y = mx + b$$

Notice that this is for the decision boundary line. This line is perpendicular to the ideal value of the weights, so the slope of the line is the negative reciprocal of the weights.

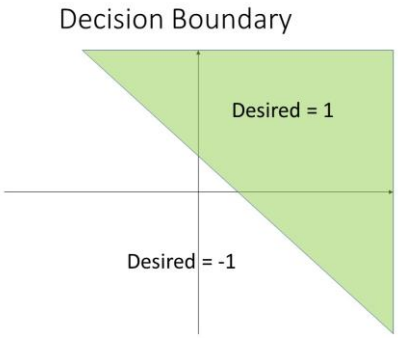

Fig. 2. Illustration of the decision boundary for a perceptron. In general, a perceptron creates a hyperplane between the desired (desired = 1) and not desired (desired = -1) input vectors. In data that uses 2 inputs (2 dimensions) the hyperplane is a line.

This makes intuitive sense when you consider the inputs as unit vectors and the weights as a unit vector. In this case, if the best input would be parallel to the weights, resulting in an output of 1, the worst would be the total opposite of the weights, resulting in an output of -1 (and putting the vector firmly into the area of the undesired values), and an input vector right on the decision boundary would result in a value of 0, perpendicular to the weights.

The perceptron with its learning rule works with only linearly separable data, which is fine, as long as the person using the perceptron as a decision tool knows that their data is linearly separable. This is a serious limitation. Much has been written about this, most notably by Minsky and Papert as indicated by Haykin [1].

B. The Radial Basis Function

The Radial Basis Function (RBF) [2] is similar to a perceptron in that it is a single neural unit. However, they are very different in operation. The learning process of a perceptron finds the location of the hyperplane that divides the desired inputs from the undesired inputs. The RBF learning process finds the center of the zone of acceptance for the inputs. If using two dimensions (x, y), the zone is a circle or ellipse depending on the calculation of the v parameter. Because of these differences, the calculation of the v parameter, the limiting function, and the update function are all changed somewhat. The general expression for the v parameter is shown below:

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 02, February-2022

$$v = (x - w)^T * A * (x - w)$$

This equation is calculating the square of the distance of the input vector x from the weight vector w. The A quantity can be thought of as a shape modifier, meaning that if it is left as 1 when using a 2-dimensional system, v describes the square of the radial distance from the center identified as w, i.e., a circle.

The limiting function actuates if v is less than some threshold. So, if an input is within the distance described by the threshold, an input is accepted, otherwise it is rejected. The area within the threshold, centered at the value of the weights is a circle (if the A parameter is set to 1) The logic of the limiting function is shown below.

```
if (v <= threshold)
  y = 1
else:
  y = -1
```

If we want to think of the limiting function as accepting all inputs within the circle centered at the weights, then the threshold is set to the radius of the acceptance circle squared (because v is the square of the distance from the tip of the vector w).

The weight update function is shown below:

```
η - the learning rate
error = desired - y

for each j from 0 to n-1 {
  weights[j] = weights[j] + (2* η *error)*(xInput[j] -
               weights[j])
}
```

Notice that there is no bias used in the radial basis function. Fig. 3 below illustrates the radial basis function situation. This update function's logic is slightly different than that of perceptron, but functionally similar. Notice that error will be 0 when desired and y are the same, resulting in no update to the weights; this is functionally similar to what happens in the perceptron.

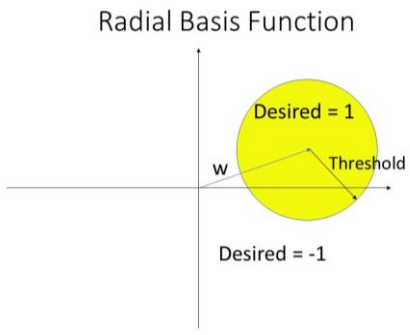

Fig. 3. This figure shows the acceptance zone of a radial basis function. The weight vector w points to the center of the acceptance zone.

Of particular importance for the purposes of this paper is that the RBF defines a finite zone of acceptance that can be arbitrarily placed within a solution space.

As part of this work, the Tensorflow Playground application [3] was used to solve/classify the points in their spiral points set. An ad-hoc solution, striving for minimal network architecture complexity, resulted in a solution with 4 input nodes, 8 nodes in the first hidden layer, 6 nodes in the second hidden layer, and 2 output nodes. As for the input nodes, they were the x, y coordinates, and the coordinates squared. The transfer function was the hyperbolic tangent function, and number of epochs was in the upper 300's.

As powerful as the feedforward network is, as stated earlier, one of the goals of this work was to minimize network complexity/number of layers, and autogenerate the number of neurons needed to solve a given problem. The growR3 algorithm described in the next section of this paper relies on being able to programmatically place multiple RBFs in a solution space in order to "cover" a given pattern of acceptance. This work relies on circular zones of acceptance; it is recognized that tuning of the A parameter and threshold, would sharpen/enhance the boundary condition accuracy.

III. APPROACH

The general idea behind the growR3 algorithm is to use a patchwork of RBF neurons in order to "cover" a solution space. In Fig. 4 below we see some examples of decision boundaries. This is different than the traditional approach used by RBF networks [2] in which the number of nodes and layers is fixed from the beginning. Others [3] use K means or Kohonen networks to find the centers of the RBF functions. In this work, the network grows to the size needed by the data, so no number of clusters or hidden node count is selected. The two smaller diagrams on the left of the figure illustrate classic examples that lend themselves to a perceptron and an RBF function. However, the larger figure on the right is more complex, with curved and concave surfaces. For the data points that lie within this spiral-like figure, usually a multi-layered feed forward network /tool is used. Examples of problems being solved by networks with 1 or more hidden layers can be explored using available tools such as tensorflow [4] and tensorflow playground [5].

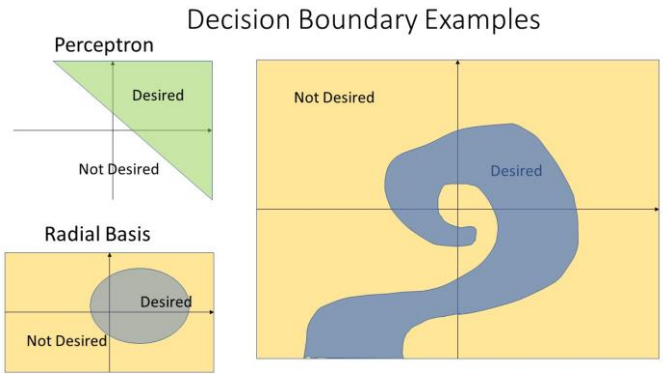

Fig. 4. The two smaller diagrams to the left illustrate problems that a basic perceptron or RBF neuron can easily solve. However, neither of these can solve a more complex problem such as the one depicted in the diagram on the right. For problems of this nature, a multilayer network is commonly used.

The growR3 algorithm approaches the problem by locating multiple RBF neurons in the area of the desired data. Fig. 5 below illustrates how the growR3 algorithm would tackle the spiral diagram from Fig. 3 above.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 02, February-2022

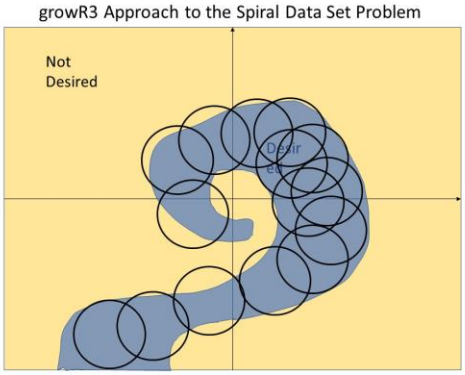

Fig. 5. This figure shows how the growR3 algorithm tries to solve problems in which the desired data points form complex shapes. Each of the dark circles represents an area of acceptance of an RBF neuron. In Fig. 5 it can be seen that some of the circles are too big for the blue area, some overlap, etc. These phenomena can be tuned by the threshold and associated multipliers in the algorithm and are addressed in a later section.

The pseudo code for the growR3 algorithm is presented below:

```
growR3(n, inputs, desired, threshold, nEpochs) {
   // Initialize parameters.
   myRadials = {}
   nRadials = 0
   η = 0.01

   # Run and Train on inputs
   for (j = 0; j < nEpochs; j++) {

      for(k = 0; k < n; k++) {
         x = inputs[k]
         d = desired[k]
```

closeOne, v = find the closest radial in myRadials to the input x. Return its index and the distance v from x to the center of the selected radial. If the myRadials set is empty, return -1

```
if (closeOne == -1) {
    // We only create a new radial for the desired data class.
    if (d == 1) {
        rNew = radialBasis with the desired number of inputs
        Initialize the weights in rNew to the value of the
                current input x
        Add rNew to myRadials
        nRadials = nRadials + 1
        }
}
else {
    // If the input is within the range of the closest radial,
    // update that radials weights.
    if (v < threshold*spacing multiplier) {
        y = myRadials[closeOne].hardLimitRadial(v,
threshold)
        myRadials[closeOne].updateMe(d, y, η, x)
    }
    else if (d == 1) {
       // We only create a new radial for the desired data class.
        rNew = radialBasis with the desired number of inputs
        Initialize the weights in rNew to the value of the
                current input x
        Add rNew to myRadials
        nRadials = nRadials + 1
        }
    }
  } // End k
}   // End j
return nRadials, myRadials
}  // End growR3
```

Using the RBF network contained in the myRadials array is done using the runR function. It works by presenting each radial with an input. If any single RBF neuron accepts the point, it is classified as desired. The logic of the runR function is shown in the pseudo code below.

```
runR(numRadials, radials, nPoints, radiusThresh, points{}) {
   classification = {}
   for (j = 0; j < nPoints; j++ ) {
      inVec = points[j]
      accepted = -1
      // Present the input to each radial that was generated.
      for (k = 0 ; k < numRadials; k++) {
         // runMe is the method that runs a radial neuron.
         y = radials[k].runMe(inVec,
             radiusThresh*radiusThresh)
         if (y == 1):
            accepted = 1
      }
      Add accepted to the classification set
   }  // End for
   return points, classification
} // End runR
```

IV. RESULTS

Early tests of growR3 shows promise. For a simple test, one with desired points randomly centered around (-4, 4) and (4, -4). The growR3 algorithm finds the centers quickly, albeit the centers are not precisely at (-4, 4), (4, -4) because of the random generation of the points favors generating points more densely around the origin, which slightly shifts the centers closer to the origin. The results of the test are shown in Fig. 6 below.

One thing of note is that growR3 generated 3 radials for this test, instead of 2. This is due to the tuning parameters used.

The next test was the spiral test. Once again a random group of 1000 points was generated with a spiral pattern embedded in it. The spiral itself is the desired data, as denoted by the blue data points in Fig. 7 below. The diagram on the left of Fig. 7 shows the training data, the figure to the right shows the test data.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 02, February-2022

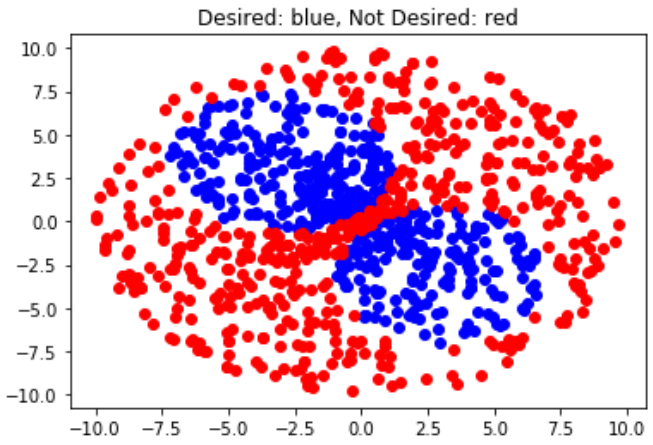

Fig. 6. Basic test of the growR3 algorithm. The training data was a random set of data centered around the points (-4, 4) and (4, -4). There were 1000 points in the data set.

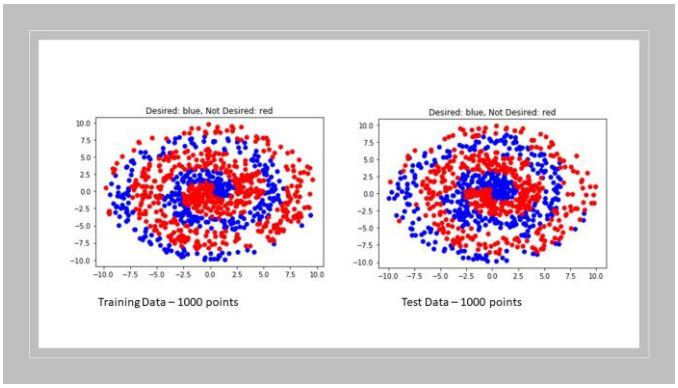

Fig. 7. This figure shows spiral training data and spiral test data. It is clear that the growR3 algorithm is finding the shape of the data, but it is also clear that some points are being misclassified as evidenced by the thicker spirals in the test data.

The spiral test shown in the Fig. 7 caused the growR3 algorithm to create an RBF network with 81 radials. As referred to in the caption, the growR3 algorithm's placement of the radials causes errors in acceptance of undesired points as can be seen in the width of the band of blue dots in the right figure vs the left. This can be due to placement of the radial from training, but also due to the non-adaptive nature of the radius (threshold value) of the radial.

One thing of note is that this result was obtained with a training regimen of 7 epochs. In the scheme of things, this is quite low; the earlier referenced tensorflow example required upwards of 300 epochs. Another point that can be recognized is that if we were to select one of the points marked by the RBF network as desired, we can trace that point to the RBF neuron that accepted it.

V. CONCLUSION AND FUTURE WORK

The RBF network generated by the growR3 algorithm and used by runR shows promise for being able to learn to classify points as desirable or not in arbitrary arrangements by virtue of its self organizing/additive nature. In its current form it can generate a good but somewhat rough solution in very few epochs. It also has the benefit of being a very simple single layer system which allows direct tracing of the path of acceptance of each point marked as desired. The solution is repeatable from run to run as long as the data is the same because the weights are initialized to 0, not to random values. The algorithm needs to be improved by implementing adaptive thresholding and possible modification of the A parameter so that radials can be tuned to the area in which they are responsible.

REFERENCES

[1] S. Haykin, Neural Networks A Comprehensive Foundation, Macmillan College Publishing, 1994.
[2] M. Orr, "Introduction to Radial Basis Function Networks," Centre for Cognitive Science, University of Edinburgh, April 1996, retrieved from https://faculty.cc.gatech.edu/~isbell/tutorials/rbf-intro.pdf.
[3] D. Touretzky, "Radial Basis Functions," 15-486/782 Artificial Neural Networks, Fall 2006, retrieved from http://www.cs.cmu.edu/afs/cs/academic/class/15883-f19/slides/rbf.pdf.
[4] Tensorflow, January 2022, retrieved from https://www.tensorflow.org.
[5] Tensorflow Playground, January 2022, retrieved from https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=spiral&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=8,6,2&seed=0.84393&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=true&ySquared=true&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false&showTestData_hide=false.