Machine Learning: Collectives of local decision makers

Construction Of A Boundary Hunter

Daniel Braithwaite

May 31, 2017

The goal of this project is to create neurons which instead of searching for a global minimum they will choose a specific part of the data to place their hyperplane. Consider the following situation in $\mathbb{R}^2$ where we have data split into two classes and the second class is boxed into a chevron shape. A single perceptron will go and place a horizontal line through the plane however this isn't telling us anything interesting. Perhaps something better to do is to align its self to one of the sides of the chevron, then if we added another "boundary hunter" it could align its self to the other side of the chevron.

1 Simplified Situation

Before worrying about this in the general case we aim to get a better understanding of the problem by first trying to solve the case described above. We first make some changes to how we have parameterized our perceptron. We want to find a way which performs the same as a standard perceptron before we start making it into a boundary hunter

1.1 Point & Gradient Paramaterisation

We are trying to optimize a point (x,y) and a gradient. This is an alternative way to paramaterise our line. We are using sum squared error for our loss function and sigmoid as our activation. Each "boundary hunter" (BH) has the following weights vector $W = [m, x_0, y_0]$ and we consider two inputs to our BH, x and y. Our perceptron computes $z = (y - y_0) - m(x - x_0)$ and then outputs $o = f(z) = \frac{1}{1+e^{-z}}$.

1.1.1 Comparison To Perceptron

If we use the same set of data to train both our standard perceptron and our modified perceptron we see that our modified doesn't quite perform as well as the standard version. For both models below they where trained for 50000 iterations of the training data.

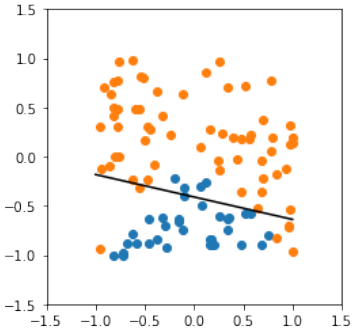

Figure 1: Standard Perceptron (SSE = 4.90)

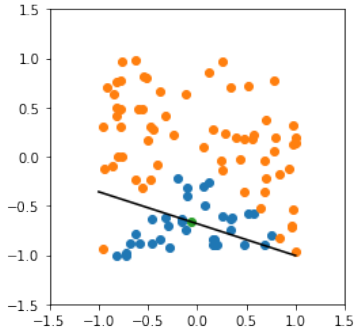

Figure 2: Modified Perceptron (SSE = 7.56)

I propose that this way of constructing our perceptron is less powerful than the standard way. Consider when we are optimizing our point-slope representation,

$$\begin{aligned} y - y_0 &= m(x - x_0) \\ y &= y_0 + mx - mx_0 \\ y &= mx + (y_0 + mx_0) \end{aligned}$$

We have just shown that (as we would expect) we can convert out point-slope representation into a intercept-slope representation. From here we see we can convert into something that could be represented by our original perceptron.

$$\begin{aligned} y &= mx + (y_0 + mx_0) \\ \Rightarrow -(y_0 + mx_0) &- mx + y \end{aligned}$$

So given the generic form of our standard perceptron $A + Bx + Cy$ we see here that our modified perceptron can only ever learn a representation where C = 1, limiting what we can achieve. An easy way to check this is to see what happens if we use a standard perceptron to learn the data but keep the third weight (i.e. C) fixed at 1. Sure enough we get the following result when we run that experiment over 50000 iterations.

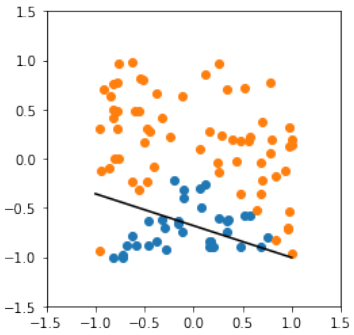

Figure 3: Standard Perceptron with C = 1 (SSE = 7.56)

So we are able to conclude that this method is unsuitable

1.2 Normal & Point Paramaterisation

This method involves learning the normal vector (which is what the standard perceptron is doing) along with a point on the hyperplane. This increases the number of parameters we have to learn by n (when we are in $\mathbb{R}^n$ but this is necessary as to implement a BH we have to define some neighborhood in which we care about and to do this we need to define it around some point on our hyperplane. So consider the following situation, we are learning the vector $n = [n_1, ..., n_n]$ normal to our hyperplane, the vector $a = [a_1, ..., a_n]$ which is on our hyperplane. We define $x = [x_1, ..., x_n]$ as our inputs making our weighted sum

$$z = \sum_{i=1}^{n} n_i * (a_i - x_i)$$

We are still using SSE for loss and Sigmoid for activation.

1.2.1 Comparison To Perceptron

Like before we compare the two parameterizations using the same data and over 50000 iterations in the 2D case. In the graph for the modified perceptron the green point is our vector on the hyperplane.

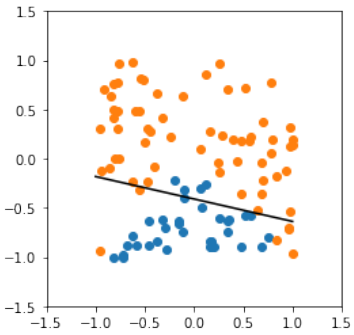

Figure 4: Standard Perceptron (SSE = 3.90)

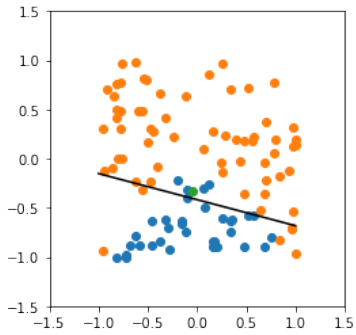

Figure 5: Modified Perceptron (SSE = 3.90)

This modified perceptron has all the quality needed to proceed with creating the boundary hunters, whats better is this also generalizes to the $\mathbb{R}^n$ case.

2 Boundary Hunter

Using the **Normal & Point Parameterization** from before we construct boundary hunters, first we have a simple case where the area of interest for each boundary hunter is simply a circle with fixed radius, our goal is to make this area of interest an ellipse which the BH will learn.

2.1 Loss Function Design: Attempt 1

We wish to convert our Perceptron into a Boundary Hunter so to begin we start with designing a suitable loss function. So firstly we want to have some notion of how responsible a boundary hunter is for a given data point, for now we will just denote the responsibility for data point n by r_n and worry about calculating this later. Consider our boundary hunter as reducing the uncertainty of the classes for each of the data points. As the data points get further away from our radius of expertises we become more uncertain about the class of the data point. So a reasonable loss function is

$$\frac{1}{n}\sum_{i=0}^{n} r_i(C_i log(y_i) + (1 - C_i) log(1 - y_i)) + (1 - r_i) log(1/2) \tag{1}$$

2.2 Responsibility Function Design

How do we construct r_i. We would like the following property's of a responsibility function

1. Making the area of interest include all points will reduce loss to standard log loss.

2. All points inside the area of interest should have the same responsibility.

3. Responsibility for points outside the area of interest should decrease as distance from area increases.

The following function seems like a good place to start. The responsibility for any point inside the area of interest is 1, and as the data points get further away the responsibility decreases

$$r_i = 1 - \frac{max(0, d - r)}{abs(d - r) + 0.1} \tag{2}$$

However we note that this functions isn't smooth, and that can cause problems when training with back prop, not to mention the hard boundary (i.e. responsibility 1 for all points inside area of interest) will also have a negative effect on the gradients. While its reasonable for us to want property (2) it seems unlikely we will find a suitable function meeting the condition. So we remove property (2) and add the condition that our function must be smooth. We now turn to the logistic function, Let $f = 1 - \frac{1}{1+e^{-S(d-r)}}$, where S is the steepness of the sigmoid. We will want our responsibility function to have a low sensitivity at each extreme, i.e. a small change in distance when the point is either very close to or far from our center point should have a negligible impact on the responsibility. How ever we want our function to be very sensitive around the border of our area of interest. By a purely arbatary decision and from inspecting the graphs I chose to have a steepness of 10 thus giving us the following function

$$r_i = 1 - \frac{1}{1 + e^{-10(x-r)}} \tag{3}$$

While this does not meet our original criteria we see that for most points inside our area of interest the responsibility will be "close" to 1 and certainly above $\frac{1}{2}$, And if the area includes all the points then our loss will not reduce to the standard log loss but it will be close to it and as $r \to \infty$ we will converge to the standard log loss.

2.2.1 Experimental Results

We train our boundary hunter over the same data for 10000 iterations and the result is somewhat uninteresting, it simply learns an area of interest which contains no points in the data set

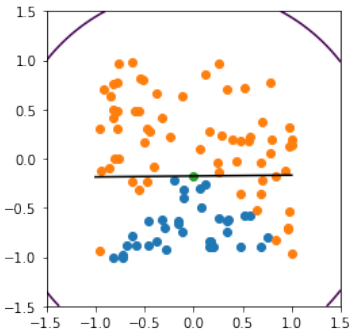

Figure 6: Boundary Hunter with (1) as loss and (3) as responsibility

As demonstrated using this loss does not produce any significant results.

2.2.2 Justification For Results

So consider our loss for a given data point with target t, prediction y and responsibility r. WILOG assume $t = 1$. So then we are left with $r\ log(y) + (1 - r)\ log(\frac{1}{2})$. We know that $log(y) \leq log(\frac{1}{2})$

2.2.3 Reflection

Essentially we want to reward our boundary hunter if it is an expert in a given region, so if it gets things wrong outside its area of interest thats good, and if it gets things correct outside its area of interest then thats bad.

2.3 Loss Function Design: Attempt 2

Based on our previous findings a more in depth consideration of a suitable loss function is required. Consider the following properties of our ideal loss function. The basic idea is that we want to become experts in a given region

1. As a data point gets further away from our area of interest we "care less" about our accuracy for classifying that point.
2. Increasing area of interest should decrease the loss iff increasing the area of interest allows us to classify more points correctly.
3. Decreasing area of interest should decrease the loss iff reducing area of interest allows us to classify less points incorrectly in our area

We can simplify these ideas into rules

1. We want to be as accurate as possible in our area of interest and care less about our accuracy based on how "responsible" we are for the data point.

2. We should be penalized for classifying things correctly out side our area of interest based on how "not responsible" we are for the data point (i.e. if we are classifying a data point correctly but it is just outside our area of interest then we should be penalised).

3. We should be rewarded for classifying things incorrectly our side our area of interest based on how "not responsible" we are for the data point (i.e. if we are classifying a data point incorrectly and its just outside our area of interest then we should be rewarded).

We now wish to convert this into a loss function. Item (1) is referring to r_n multiplied by the standard cross entropy, restricted to points inside our area of interest. Then items (2) + (3) are referring to the opposite of our cross entropy multiplied by r_n restricted to all points outside our area of interest.

$$-\big(\frac{1}{|I|}\big[\sum_{i\in I}^{n} H(t_i, y_i)\big] + \frac{1}{|O|}\big[\sum_{i\in O} H(\not{t}_i, y_i)\big]\big) \tag{4}$$

Where I is the set of points inside our area and O is the set of points not in our area of interest.

2.3.1 Experimental Results

With the same parameters as before we train a perception with this loss and find equally uninteresting results. Essentially the boundary hunter ends up positioning not doing anything useful at all

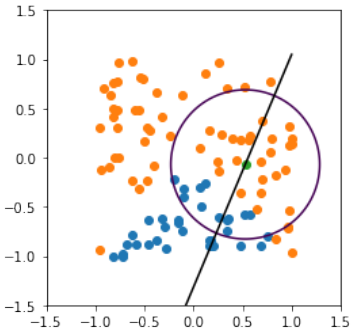

Figure 7: Boundary Hunter with (?) as loss and (3) as responsibility

2.3.2 Reflection

The conditions used to construct this loss are perhaps to harsh, currently we are trying to maximize our error outside while minimizing our error inside. Where we don't actually care how incorrect we are outside our area.

2.4 Loss Function Design: Important Observation

We take a step back and consider the basics of what we are trying to achieve. First we make an interesting observation which will change the way we approach designing this loss. Consider the simple loss function $L = \sum R(r) * CE$ where $R(r)$ is our responsibility function with radius as parameter and CE is our cross entropy. And we want to compute the quantity $\frac{\partial r}{\partial L}$ so we can train r. We observe the following

$$\begin{aligned}\frac{\partial L}{\partial r} &= \frac{\partial}{\partial r}\big[\sum R(r) * CE\big] \\ &= \sum R^{'}(r) * CE\end{aligned}$$

Namely the quantity $\frac{\partial L}{\partial r}$ doesn't actually tell us how changing the radius changes the error it simply tells us which direction to move the radius to increase the responsibility. This results in the radius constantly increasing.

2.5 Loss Function Design: Attempt 3

Based on what we just observed we need a new approach. We consider our boundary hunter as a sales man, who gets paid $\$B$ for selling something to someone who wants it (if the sales man is responsible for this individual) and gets penalized $\$C$ dollars for selling something to someone who doesn't want it. Using this model, the sales man's is to position them selves and adjust there responsibility so that they are maximizing there profit. Now we simply must quantify this. We know $y^t(1-y)^{t-1}$ is the probability we get something right (probability we are selling to someone who wants what we are selling). We arrive at the flowing quantity.

$$L = \sum_{i=0} r_i * \big[Cy_i^{1-t_i}(1-y_i)^{t_i} - By_i^{t_i}(1-y_i)^{1-t_i}\big] \tag{5}$$

While initially it seemed like this would suffer from the same problem described in the previous observation it does not. Consider the differential we know that $R^{'}(r)$ will always have the same sign but $Cy_i^{1-t_i}(1-y_i)^{t_i} - By_i^{t_i}(1-y_i)^{1-t_i}$ can be both positive and negative, so we end up in the situation where if we are losing more money that we are gaining then we will decreases our radius but if we are gaining more money than we are losing we will increase our radius

2.5.1 Experimental Results

With the same parameters as before we train a perception with this loss and find that it grows to fit all the data

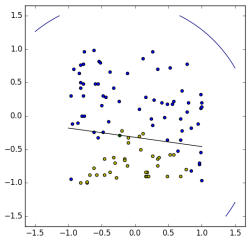

Figure 8: Boundary Hunter with (5) as loss and (3) as responsibility

Not quite the results we where looking for but this outcome makes sense as there is a disproportionate amount of points. Increasing the parameter C will allow us to tune our boundary hunter to care more about getting things incorrect. To get a good spread we use C = 2, 2.5, 3

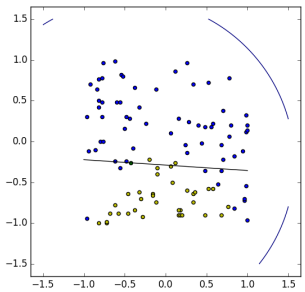

Figure 9: C = 1.3

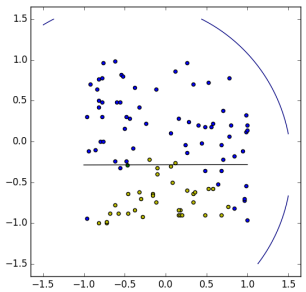

Figure 10: C = 1.6

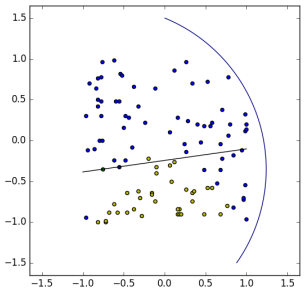

Figure 11: C = 1.9

How ever this only seems to change the positioning of the hyperplane and

not the radius of our area of interest. If we impose some reasonable restrictions on the radius (i.e. $0.3 \leq r \leq 0.8$), set our C = 2 and adjust the responsibility function to be less steep (steepness of 5) then we can achieve some results which seem interesting

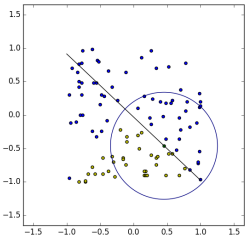

Figure 12

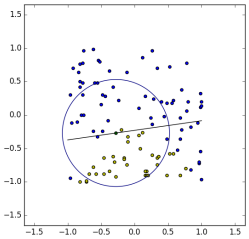

Figure 13

However more often than not we get results like following uninteresting ones

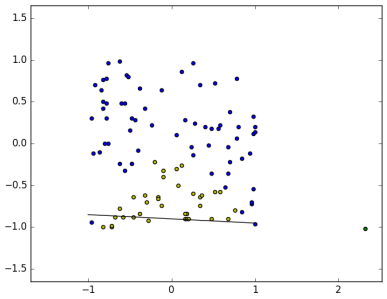

Figure 14

The issue appears to be that of local optima, the solutions looking like what we want (Figure 13 and 14) certainly have lower error than when the solutioin is outside the data however the boundary hunter appears to have trouble escaping local optima to find these more interesting solutions.

To confirm this we look to plot the landscape of the loss function. To do this we will fix the radius and and the hyperplane position while varying the x and y position of the point, The parameters we fix for hyperplane are what we would want if trying to place our self on the left side of the chevron . Using the parameters as above, (steepness of 5, C = 2) We get the following plots.

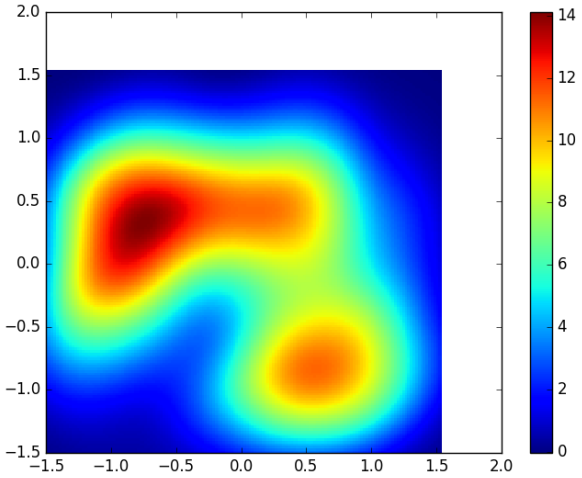

Figure 15: Radius as 0.3

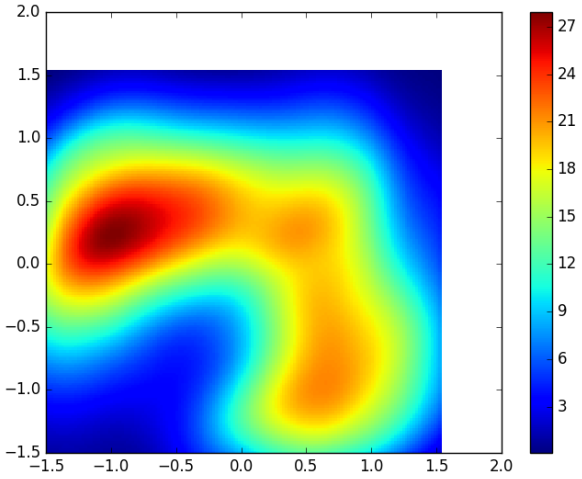

Figure 16: Radius as 0.8

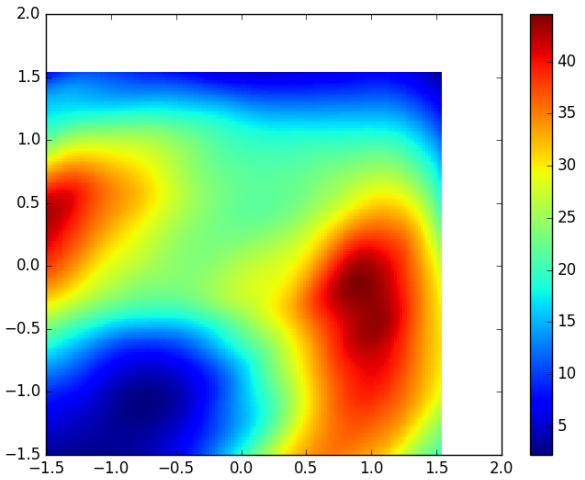

Figure 17: Radius as 2.0

As clearly demonstrated by these graphs this loss function is not condusive to finding the optimal solution we are expecting. There is a noticeable deformation in the contour where our expected solution is however it is unlikely that it will be found by the boundary hunter, more often than not we will end up on the outside of the data. For points outside our area of interest the responsibility rapidly approaches 0, this is what would cause the edge of the data to be an optimal solution. While this would indicate a responsibility function which is to steep, adjusting this has little effect.

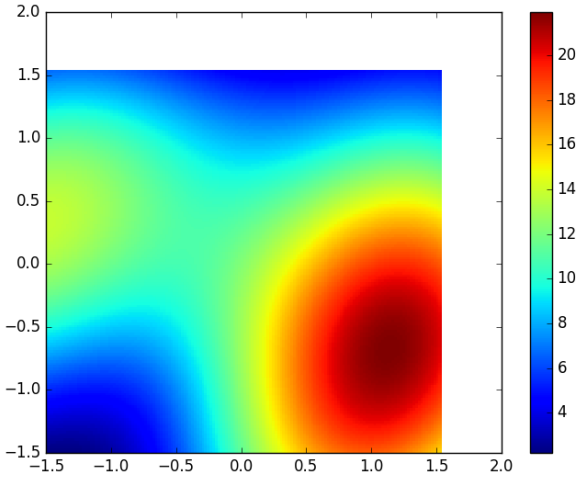

Figure 18: Radius as 0.3, with steppness as 0.5

3 Radial Basis Function Networks

3.1 Definition

We turn our attention to a class of networks which seem very closely related to what we want, these are Radial Basis Functon Networks (RBF Networks). Before we can talk about RBF Networks we will first define what an RBF is.

A **radial basis function** is a function in which the output depends only on the distance from some point we call the center of the RBF. $f(x, c) = f(\|x-c\|)$.

And then we can define an RBF Neuron as being a neuron with an activation stasfying the propertes of a radial basis function. Esentially each RBF Neuron is mesuring the similarity between its center and the input vector presented to it, the closer the input is to the center the closer the activation is to 1. Any RBF will do as an activation but we will be using one based on a the 1D Gaussian distrubution. The equation for a 1D Gussian is

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

We can make some simplifications seing as we arnt interested in the standard deviation, making our activation

$$a(x) = e^{-\beta\|x-c\|} \tag{6}$$

We use the β term to controll how quickly the activation decays.

An intuative view of an RBF network is we have a number of RBF Neurons which tells the output layer how close the input vector is to its center, the output neurons then use this information to make a decision about there activation by taking a weighted sum of the activations of the hidden neurons.

3.2 Training RBF Networks

We start with the simplist method for training an RBF Network. Simply randomly initilize all the paramaters of the network and then use gradient descent to find better paramaters. Using this method we are able to achieve good classifiers for data that a standard perceptron would have no hope at doing anything interesting.

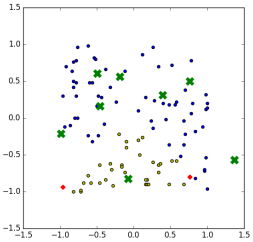

Figure 19

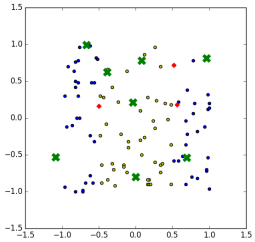

Figure 20

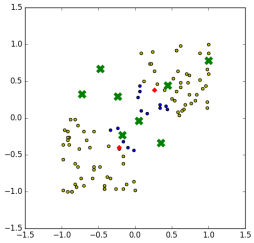

Figure 21
The green x's represent our centroids while the red x's represent the points we are getting incorrect

4 RBF Boundary Hunters

Based on these RBF Networks, is it possible to use the ideas that we had previously to construct a boundary hunter. We create the concept of a RBF Boundary Hunter Neuron and derive an activation function for such a neuron.

Consider that the output of a regular RBF Neuron is the unsertianty about wether the given point is within its radius, as the point gets further away we become less certian about our ownership and so the activation of the RBF Neuron decreases. We would like to incorpreate this into the output of our RBF Boundary Hunter as as the points move further away from the center then we become less sure about our classification. Let f be our activation function, as our beleif that the point is in class 1 increases $f \to 1$ and as it decreases we have $f \to 0$, so if $f = \frac{1}{2}$ then we are unsure about the class of the point. It seemes logical for $f \to \frac{1}{2}$ as the distance between the center of a boundary hunter and a point increases. Now we can define our RFB Boundary Hunter Neuron for binary classification.

Definition 4.1. A **RBF Boundary Hunter Neuron** (for binary classification) in $\mathbb{R}^n$ has $2n + 1$ free paramaters. $\beta \in \mathbb{R}$, $\mathbf{c} \in \mathbb{R}^n$ and $\mathbf{n} \in \mathbb{R}^n$. $\beta, \mathbf{c}$ represent decay speed and center of the RBF portion of our neuron. $\mathbf{n}$ defines a hyperplane which passes through our point $\mathbf{c}$. We define the activation as folows

$$a(x) = \frac{1}{2} + e^{-\beta\|x-c\|}\left(\frac{1}{1+e^{-(\mathbf{n}\cdot(\mathbf{c}-x))}} - \frac{1}{2}\right) \tag{7}$$

4.1 Training RBF Boundary Hunters

Using the same method as before, randomly initializing all paramaters and then training using gradient descent.

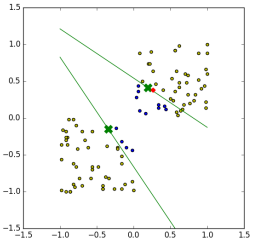

Figure 22

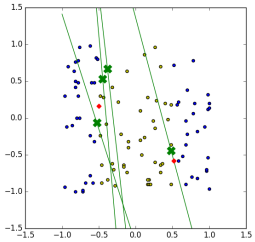

Figure 23

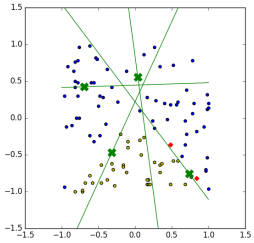

Figure 24
The green x's represent our centroids while the red x's represent the points we are getting incorrect

These resuts show very promising and consistant results. The hyperplanes being placed ezactly where we would hope, however some results are a little misterious to understand.