Machine Learning: Collectives of local decision makers

Construction Of A Boundary Hunter

Daniel Braithwaite

May 8, 2017

The goal of this project is to create neurons which insted of searching for a global minumum they will choose a specific part of the data to place their hyperplane. Consider the folowing 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 horizonal line through the plane however this isnt telling us anything interesting. Prehaps 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 paramaterised our perceptron. We want to find a way which peforms the same as a standard perceptron before we start making it into a boundary hunter

1.1 Point & Gradient Paramaterisation

We are trying to optimise 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 folowing 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 Comparason 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 dosnt quite peform as well as the standard version. For both models below they where trained for 50000 iterations of the training data.

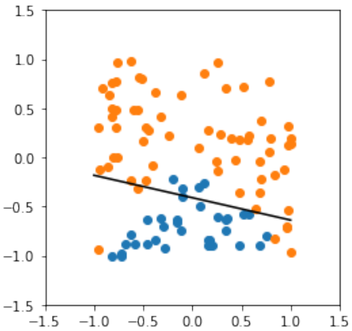

Figure 1: Standard Perceptron (SSE = 4.90)

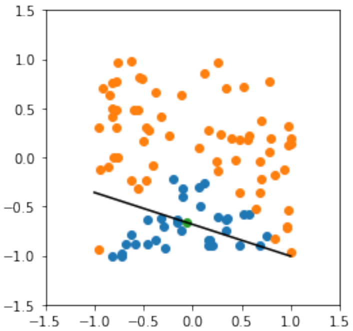

Figure 2: Modified Perceptron (SSE = 7.56)

I propose that this way of constructing our perceptron is less powerful than the standard way. Consider when we are optimising 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 origonal 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 folowing result when we run that experement over 50000 iterations.

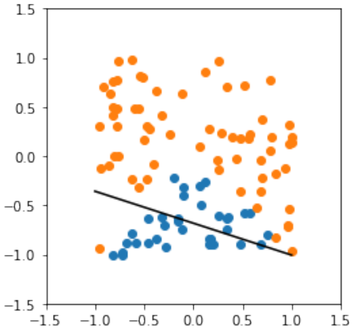

Figure 3: Standard Perceptron with C = 1 (SSE = 7.56)

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 increaes the number of paramaters 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 neighbourhood in which we care about and to do this we need to define it around some point on our hyperplane. So consider the folowing 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 actvation.

1.2.1 Comparason To Perceptron

Like before we compare the two paramaterisations 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.

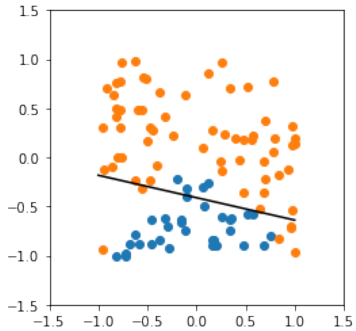

Figure 4: Standard Perceptron (SSE = 3.90)

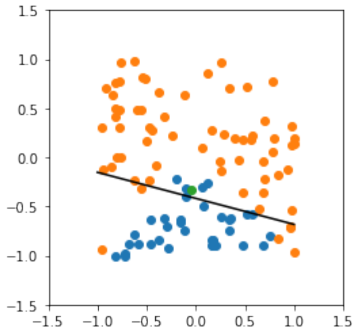

Figure 5: Modified Perceptron (SSE = 3.90)

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

2 Boundary Hunter

Using the **Normal & Point Paramaterisation** from before we construct boundary hunters, first we have a simple case where the area of intererest for each boundary hunter is simpley 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 suitible 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 uncertuanty of the classes for each of the data points. As the datapoints get further away from our radius of expertees we become more uncertain about the class of the datapoint. 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 folowing propertys 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 folowing function seems like a good place to start. The responsiblity for any point inside the area of interest is 1, and as the data points get further away the responsiblity decreases

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

However we note that this functions isnt 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 sutible 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 neglible inpact on the responsiblity. 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 steppness of 10 thus giving us the folowing function

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

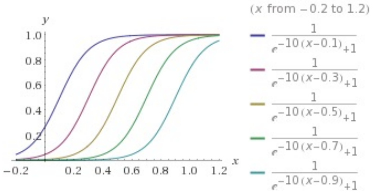

Figure 6: Graph of responsibility function with r = 0.1, 0.3, 0.5, 0.7, 0.9

While this does not meet our origonal criteria we see that for most points inside our area of interest the responsibility will be ”close” to 1 and certianly 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 Experemental 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

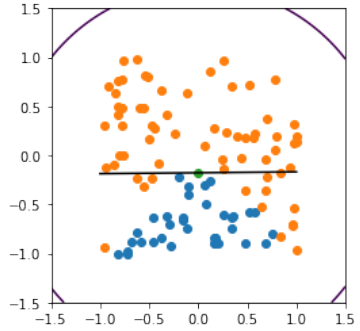

Figure 7: Boundary Hunter with (1) as loss and (3) as responsibiliy

As demonstraited 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

Esentially 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 Funcction Design: Attempt 2

Based on our previous findings a more in depth consideration of a suitable loss function is required. Consider the folowing 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 simplyfy 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 penalised 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 refering to r_n multiplyed by the standard cross entropy, restricted to points inside our area of interest. Then items (2) + (3) are refering to the oposite of our cross entropy multiplyed by r_n restricted to all points outside our area of interest.

$$-\big(\frac{1}{|I|}\big[\sum_{i \in I}^{n} H(t_i, y_i)\big] + \frac{1}{|O|}\big[\sum_{i \in O} H(\cancel{t}_i, y_i)\big]\big) \tag{4}$$

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 Experemental Results

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

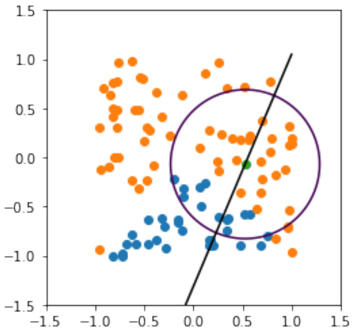

Figure 8: Boundary Hunter with (1) as loss and (3) as responsibiliy

2.3.2 Reflection

The conditions used to construct this loss are prehaps to harsh, currently we are trying to maximise our error outside while minimizing our error inside. Where we dont actually care how incorrect we are outside our area.

2.4 Loss Function Design: Attempt 3

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 paramater 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 folowing

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

Namely the quantity $\frac{\partial L}{\partial r}$ dosnt actually tell us how changing the radius changes the error it simplys tells us which direction to move the radius to increase the responsitility. **CHECK THIS CLAME**.

Returning to the loss function design it would seem that simply differentating the loss with respect to the radius wont give us what we want. So consider a naive approach of using the cross entropy loss scaled by the responsibility,

$$frac1n \sum_{i=0}^{n} r_i(C_i log(y_i) + (1 - C_i) log(1 - y_i)) \qquad (5)$$

However now when computing δr we insted peterb the radius and compute the gradient numerically, not ideal but somewhere to start.

2.4.1 Experemental Results

Using the same datasets as before we train a Boundary Hunter with this new training method and find the folowing results

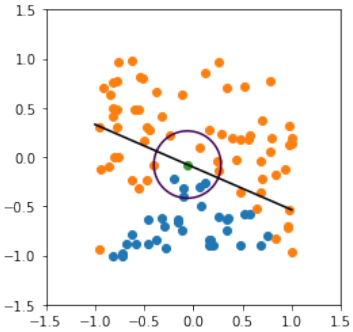

Figure 9: Boundary Hunter with (1) as loss and (3) as responsibiliy

From this method we can observe some interesting results, seeing that we indeed learn to classify our area of interest perfectly.

2.4.2 Reflection

While this method does show the promise of boundary hunters we would like to avoid using such a technequic as it requires us to set how much to peterb our radius by, this is not idea as it depends very much on the density of the data.