Machine Learning: Collectives of local decision makers

Construction Of A Boundary Hunter

Daniel Braithwaite

April 27, 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.

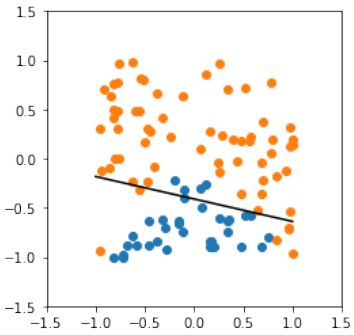

Figure 1: Standard Perceptron (SSE = 4.90)

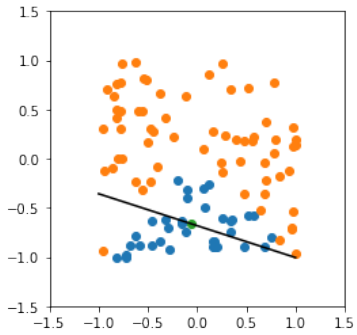

Figure 2: Modified Perceptron (SSE = 7.56)

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.

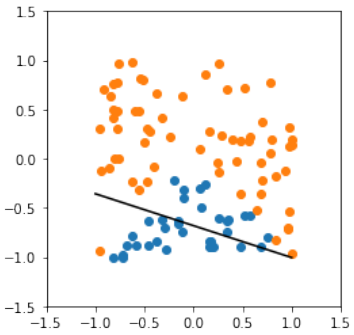

Figure 3: Standard Perceptron with C = 1 (SSE = 7.56)

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.

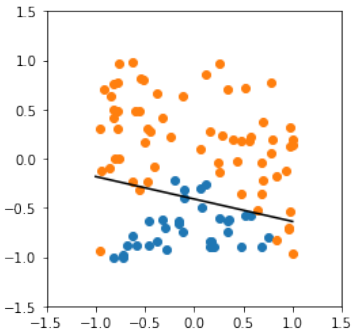

Figure 4: Standard Perceptron (SSE = 3.90)

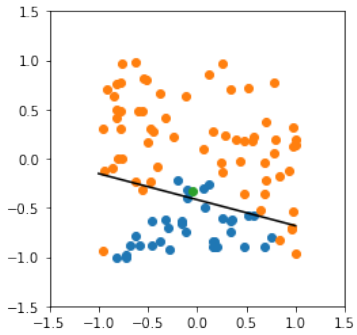

Figure 5: Modified Perceptron (SSE = 3.90)

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

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

But 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

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

2.1.1 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 the entire data set, reducing our loss function to the standard cross entropy loss function

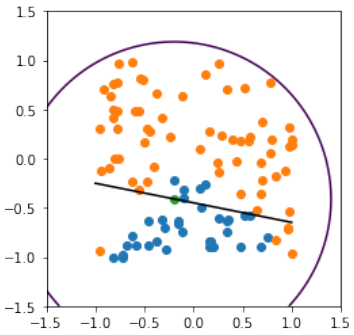

Figure 6: Boundary Hunter with (1) as loss

With corosponding line equation $y = -0.19877x - 0.44949$. As demonstraited this line is almost identical to the one learned by a normal & point perceptron (descrepencies may be caused by longer training for standard perceptron). This demonstraites that this loss is not sutible

2.1.2 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 $r\ log(y) + (1 - r)\ log(\frac{1}{2})$. And $log(y) = log(\frac{1}{2})$ in the worst case. So we can always at least obtain an equivelent solution by setting $r = 1$ for each data point. Peforming such a thing is equivlent to including all points inside the area of interest.

2.1.3 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.

2.2 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. But now by using (1) we are doing the oposite of (2) and (3). Item (2) and (3) refer to r_n multiplyed by what we call the "inverted cross entropy", here we restrict the "inverted cross entropy" to just the points that are not inside our area of interest. Where the standard cross entropy really hates to get things wrong, the inverted cross entropy really hates to get things right. So we propose the folowing as our loss where $H(t, y) = t\ log(y) + (1 - t)\ log(1 - y)$ is our standard cross entropy and we define $H^{-}(t, y) = t\ log(1-y) + (1-t)\ log(y)$ as our "inverted cross entropy" giving us the folowing loss function

$$-\big(\frac{1}{n}\big[\sum_{i=0}^{n} H(t_i, y_i)\big] + \frac{1}{|O|}\big[\sum_{i \in O} H^{-}(t_i, y_i)\big]\big) \tag{3}$$

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