Artificial Neural Networks: Linear Classification (Part 2)

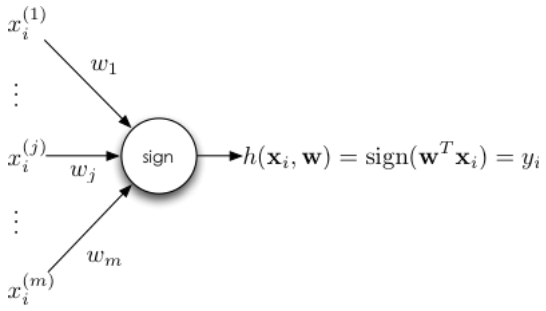
September 23, 2013 in ml primers, neural networks

So far we've covered using neural networks to perform linear regression. What if we want to perform classification using a single-layer network? In this post, I will cover two methods: the perceptron algorithm and using a sigmoid activation function to generate a likelihood. I will not cover the delta rule because it is a special case of the more general backpropagation algorithm, which will be covered in detail in Part 4.

Theory

Single-Layer Perceptron

Perhaps the simplest neural network we can define for binary classification is the single-layer perceptron. Given an input, the output neuron fires (produces an output of 1) only if the data point belongs to the target class. Otherwise, it does not fire (it produces an output of -1). The network looks something like this:



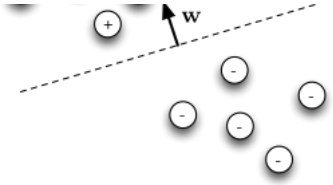
Instead of using a linear activation function like in linear regression, we instead use a sign function. Recall the definition of the sign function:

$$\text{sign}(\mathbf{w}^T\mathbf{x}_i) = \begin{cases} 1 & \text{if } \mathbf{w}^T\mathbf{x}_i > 0 \\ 0 & \text{if } \mathbf{w}^T\mathbf{x}_i = 0 \\ -1 & \text{if } \mathbf{w}^T\mathbf{x}_i < 0 \end{cases}$$

In this, we are computing the dot product of an example with our weight vector. Points with positive projections will be given a label of 1 and points with negative projections will be given a label of -1. Consequently, our decision boundary will be perpendicular to our weight vector. Why? Consider a 2-dimensional decision problem. The decision boundary is the line where it is equally probable that a point on that line belongs to either class, i.e. $h(\mathbf{x}_i, \mathbf{w}) = \text{sign}(\mathbf{w}^T\mathbf{x}_i) = 0$, or $\mathbf{w}^T\mathbf{x}_i = 0$. Then we have:

$$\begin{aligned} \mathbf{w}^T\mathbf{x}_i &= 0 \\ w_1 + w_2 x_i^{(2)} + w_3 x_i^{(3)} &= 0 \\ x_i^{(3)} &= -\frac{w_2}{w_3} x_i^{(2)} - \frac{w_1}{w_3} \end{aligned}$$

In this case, $x_i^{(1)}$ is our bias value and is always equal to 1, $x_i^{(2)}$ is "x" in the cartesian plane and $x_i^{(3)}$ is "y." The slope of our weight vector in the cartesian plane is $\frac{w_3}{w_2}$ (they "y" component of $\mathbf{w}$ is w_3, and the "x" component is w_2), while the slope of the decision boundary is $-\frac{w_2}{w_3}$ (thus making them perpendicular). Graphically, this looks something like this:



The problem we now face is that the step function is not continuously differentiable, and we cannot use standard gradient descent to learn the weights. Therefore, we will use the appropriately-named *perceptron algorithm* . This algorithm is an online method used to successively update the weights defining a linear boundary only if that boundary does not classify a training point correctly. The algorithm is as follows:

- Initialize the weight vector $\mathbf{w}$ to all zeros.
- Repeat the following:
    1. For each training example $\mathbf{x}_i$:
        - If $h(\mathbf{x}_i, \mathbf{w}) \neq y_i$, then update the weights with $\mathbf{w}' = \mathbf{w} + \eta y_i \mathbf{x}_i$ Here, η is the step size.
    2. If the stopping condition $\frac{1}{N}\sum_{j=0}^{M} |w'_j - w_j| < \delta$ is reached, then accept $\mathbf{w}$ as the final weight vector (M in this case is the number of features in the dataset).

If our problem is linearly separable, the perceptron algorithm is *guaranteed to converge*. Therefore, at the algorithm's termination, we will end up with a linear decision boundary defined by $\mathbf{w}$. However, this decision boundary is not guaranteed to be a maximum margin hyperplane as in the case of SVMs.

Finally, if we want to predict the label $\hat{y}_i$ of a test point $\mathbf{x}_i$, we use $\hat{y}_i = \text{sign}(\mathbf{w}^T \mathbf{x}_i)$.

While it is not strictly necessary to define a neural network to use the perceptron algorithm, this is a good first step towards single-layer classification.

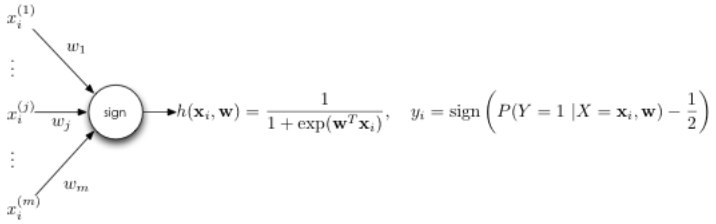
Classification with a sigmoid (softmax) activation function

Instead of an all-or-nothing classifier (like the sign function), it is helpful to come up with some way to measure the probability of assignment, that is $P(Y = y_i \mid X = \mathbf{x}_i, \mathbf{w})$. If we can calculate this likelihood, we can use as a confidence measure of our predictions.

Instead of using a sign activation function, we can instead use a sigmoid (usually called softmax in the neural net literature) to output a probability:

$$P(Y = y \mid X = \mathbf{x}_i, \mathbf{w}) = \frac{1}{1 + \exp(-y\mathbf{w}^T \mathbf{x}_i)}$$

But how do we assign a class label when given only a probability? We can simply "clamp" the probability using a sign function, so that any $P(Y = 1 \mid \mathbf{x}_i, \mathbf{w}) \geq 0.5$ is assigned a class label of 1, and any probability less than 0.5 is given a class label of -1. Our simple network now looks something like the following:



Luckily for us, this network function is identical to the likelihood used by logistic regression. Because the sigmoid is differentiable, we can use standard gradient acent to train the weights instead of the perceptron algorithm. For a derivation of the gradient for logistic regression, see the Appendix.

Implementation

To implement this theory, we'll be learning a set of weights that classify two groups of 2D data using both the perceptron algorithm and gradient descent. Let's start out by defining our 2D data (you can find this code in `ann_linear_2D_classification_perceptron.py`):

```
4  # Generate two random clusters of 2D data
5  N_c = 100
6  A = 0.3*np.random.randn(N_c, 2)+[1, 1]
7  B = 0.3*np.random.randn(N_c, 2)+[3, 3]
8  X = np.hstack((np.ones(2*N_c).reshape(2*N_c, 1), np.vstack((A, B))))
9  Y = np.vstack(((-1*np.ones(N_c)).reshape(N_c, 1), np.ones(N_c).reshape(N_c, 1)))
10 N = 2*N_c
```

This code generates two 2D clusters centered around the points (1, 1) and (3, 3). It then appends a 1 to each point, which is our bias feature. Finally, it assigns the label 1 to one cluster, and the label -1 to the other cluster. The features are put in the matrix X and the labels in the vector Y.

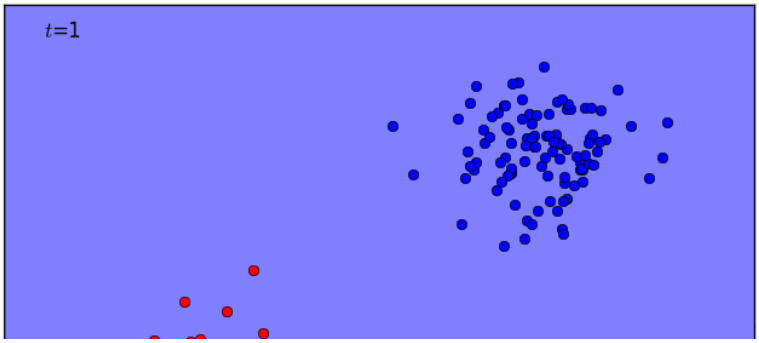
Next, we run the perceptron algorithm to learn the weights:

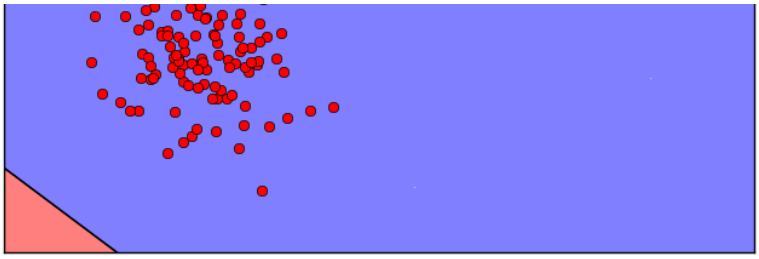
```
12 # Run perceptron
13 delta = 1E-7
14 eta = 1E-2
15 max_iter = 500
16 w = np.array([0, 0, 0])
17 w_old = np.array([0, 0, 0])
18 for t in range(0, max_iter):
19     for i in range(0, N):
20         x_i = X[i, :]
21         y_i = Y[i]
22         h = np.sign(np.dot(w, x_i))
23         if h != y_i:
24             w = w+eta*y_i*x_i
25
26     if 1/(float(N))*np.abs(np.sum(w_old-w)) < delta:
27         print "Converged in", t, "steps."
28         break
29
30     w_old = w
31
32     if t==max_iter-1:
33         print "Warning, did not converge."
34
35 print "Weights found:",w
```

This snippet follows the perceptron algorithm directly. In line 15 we initialize the weights to 0. In line 16 we also initialize a vector that stores the previous weights, which we will use to test the stopping condition. Note that we only run the algorithm for `max_iter` steps so that it doesn't loop indefinitely if the problem is not separable.

In line 21, we test to see if $h(\mathbf{x}_i) = y_i$, and if it doesn't we update the weights accordingly. Finally, we test the stopping condition in line 25.

In one particular test run, you can see that the algorithm converged in 6 steps:





What if, instead of using the perceptron, we wanted to learn a sigmoid using gradient descent? We can use the same procedure for gradient descent as detailed in Part 1 of this series. However, in the previous section, we ran gradient descent for a set amount of epochs.

Here we introduce an alternative stopping condition. The heuristic is this: if the norm of the gradient is small, we must be nearing the minimum of the function. Thus, if we set a threshold for the norm of the gradient, and it falls below this threshold at a step, we stop. The interesting part of the gradient code is as follows:

```
12 # Run gradient descent
13 delta = 1E-7
14 eta = 1E-3
15 max_iter = 1000
16 w = np.array([0, 0, 0])
17 grad_thresh = 5
18 for t in range(0, max_iter):
19     grad_t = np.array([0., 0., 0.])
20     for i in range(0, N):
21         x_i = X[i, :]
22         y_i = Y[i]
23 
24         grad_t += y_i*x_i*(np.exp(-y_i*np.dot(w, x_i)))/(1+np.exp(-y_i*np.dot(w,
25 x_i)))
26 
27     w = w + 1/float(N)*eta*grad_t
28     grad_norm = np.linalg.norm(grad_t)
29     print grad_norm
30     if grad_norm < grad_thresh:
31         print "Converged in ",t+1,"steps."
32         break
33 

print "Weights found:",w
```

In line 24, we compute the gradient according to the derivation in the Appendix. In lines 28-29, we check to see if we've reached the stopping condition. This is a simple heuristic that may not work in all cases, but works well enough for a simple problem like this. Determining how to compute the right parameters for gradient descent is an active area of research.

Code Download

Python file that learns weights using the perceptron algorithm.
Python file that learns weights using gradient ascent.

Appendix

Derivation of the gradient for logistic regression

The likelihood of our data defined by logistic regression is:

$$P(D_Y|D_X, \mathbf{w}) = \prod_i^N \frac{1}{1+\exp(-y_i \mathbf{w}^T \mathbf{x_i})}$$

Then the log likelihood is:

$$\ell(\mathbf{w}) = \log(P(D_Y|D_X, \mathbf{w})) = -\sum_i^N \log(1 + \exp(-y_i \mathbf{w}^T \mathbf{x}_i)$$

For gradient ascent, we are trying to select $\mathbf{w}$ in order to maximize the log likelihood. To do this, we must first calculate the gradient:

$$\begin{aligned}\nabla_{\mathbf{w}} \ell(\mathbf{w}) &= \frac{\partial}{\partial \mathbf{w}} \left(-\sum_i^N \log(1 + \exp(-y_i \mathbf{w}^T \mathbf{x}_i)) \right) \\ &= \sum_i^N \frac{y_i \mathbf{x}_i \exp(-y_i \mathbf{w}^T \mathbf{x_i})}{1 + \exp(-y_i \mathbf{w}^T \mathbf{x}_i)}\end{aligned}$$

Because

$$1 - P(y_i|\mathbf{x}_i, \mathbf{w}) = \frac{\exp(-y_i \mathbf{w}^T \mathbf{x_i})}{1 + \exp(-y_i \mathbf{w}^T \mathbf{x}_i)}$$

the gradient is

$$\nabla_{\mathbf{w}} \ell(\mathbf{w}) = \sum_i^N y_i \mathbf{x}_i (1 - P(y_i|\mathbf{x}_i, \mathbf{w}))$$

To avoid having to change our learning rate for different dataset sizes, we can scale by the number of examples:

$$\nabla_{\mathbf{w}} \ell(\mathbf{w}) = \frac{1}{N} \sum_i^N y_i \mathbf{x}_i (1 - P(y_i|\mathbf{x}_i, \mathbf{w}))$$

Tags: neural network, tutorial

Prev / Next

Dev A year ago · 0 Likes

Thank you for the great explanation !

@That guy: Please note this is binary classification problem. Therefore y_i will be +1 or -1. Keeping the y_i in the expression is just a compact way of writing
P(+1|x) = sigmoid(x)
P(-1|x) = sigmoid(-x)

That guy 2 years ago · 0 Likes

Can you explain the y_i in the sigmoid and logistic function. I do not see why y is part of the input?

Maudie 2 years ago · 0 Likes

Your blog is good. I am writing a thesis at http://www.essayscouncil.com/ on back propagation. thank you for sharing .

Ricky 2 years ago · 0 Likes

I think there's a mistake on the implementation code of the perceptron algorithm:
"...np.abs(np.sum(w_old-w))".
According to the previous equation, it should calculate the absolute value first, and then sum all absolute values up.