Prof. Sergio A. Alvarez
Fulton Hall 410–B
Computer Science Department
Boston College

Chestnut Hill, MA 02467

web: http://cs.bc.edu/∼alvarez/
e-mail: alvarez@cs.bc.edu
voice: (617) 552-4333
fax: (617) 552-6790

CS345, Machine Learning
Training Perceptrons using Gradient Descent Search

Gradient descent is a search strategy used in continuous search spaces. It is the basis for the error backpropagation algorithm used to train multilayer artificial neural networks. Below I explain how gradient descent works in general and how it applies in particular to training a single layer perceptron network. I will explain error backpropagation elsewhere. Studying the single layer case discussed in these notes is a good warm–up.

Gradient Descent

In its purest form, gradient descent is a technique for function minimization. A real–valued *objective function* $V : P \to \mathrm{R}$ is assumed to be given on some continuous set of parameters P (which is usually a region in n–dimensional space). The objective function V will normally represent an error magnitude or some other "bad" quantity that we want to make as small as possible. The goal is therefore to find a point p^* in parameter space P such that the value $V(p^*)$ at that point is a minimum. Unless we're lucky and have a simple algebraic expression for the objective function V, attempting to minimize V analytically (using calculus, for example) is difficult. Instead we proceed via an iterative search process that we now describe.

A starting point p_0 in parameter space P is chosen somehow, perhaps randomly. Given our current guess p_i, the next guess is chosen by stepping away from p_i in the direction in which the objective function V decreases the fastest, that is, the direction opposite to the *gradient vector* of the objective function V. This process continues until the resulting change in the objective function is smaller than some threshold. We summarize this process below.

Gradient descent pseudocode

Inputs: (P, V, η, ϵ), where P is the parameter space, $V : P \to \mathrm{R}$ is the objective function, η is the step size, and ϵ is the stopping threshold.

Output: A point p^* in parameter space P that approximates a minimum point of V.

Process:

```
1. Randomly select a starting point $p_0$ in parameter space $P$
2. Let $i \leftarrow 0$
3. repeat {
     (a) Compute the gradient vector $\nabla V(p_i)$ of $V$ at $p_i$ if possible
         or estimate $\nabla V(p_i)$ as the direction in which $V$ increases fastest from $p_i$
     (b) Let $p_{i+1} \leftarrow p_i - \eta \nabla V$
     (c) Let $\delta V \leftarrow V(p_i) - V(p_{i+1})$
     (d) Let $i \leftarrow i + 1$
   } until $\delta V < \epsilon$
4. Output $p_i$
```

Training Perceptrons using Gradient Descent

Let's see how to apply the gradient descent search strategy outlined above to the machine learning task of training a single–layer neural network as shown in Fig. 1.

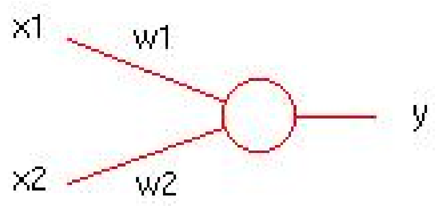

Figure 1: Single layer neural network

Gradient descent input selection

The key is to define the necessary inputs for the gradient descent algorithm here. In neural network training, the objective is to adjust the connection weights associated with pairs of processing units. Hence, we let each point of the parameter space correspond to a choice of values for all of the network weights. The objective function measures the mismatch between the desired output for given inputs and the actual network output on the same inputs. We leave the step size and stopping threshold unspecified for now. If the network is as shown in Fig. 1, we have the following selections:

- Parameter space $P = \mathrm{R}^2 = \{(w_1, w_2)|$ each w_i is a real number$\}$
- Objective function $V(w_1, w_2) = \frac{1}{2}(y - \hat{y})^2$, where:
  $y = \sigma(w_1 x_1 + w_2 x_2)$ is the actual network output value on input (w_1, w_2)
  $\hat{y}$ is the desired output value on input (w_1, w_2)

Computation of the gradient

We begin by expanding the objective function in terms of the network weights. I'll assume that the processing node of the network shown in Fig. 1 has a standard sigmoid transfer function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

The objective function then becomes:

$$V(w_1, w_2) = \frac{1}{2}(y - \hat{y})^2 \\ = \frac{1}{2}(\sigma(w_1 x_1 + w_2 x_2) - \hat{y})^2 \tag{2}$$

We will need the (calculus) derivative of the transfer function in order to compute the gradient of the objective function. I'll spare you the details, which are actually not difficult, and simply show you the last step of the computation of the derivative of the function of Eq. 1:

$$\sigma'(x) = \frac{e^{-x}}{1 + e^{-x}} = \sigma(x)(1 - \sigma(x)) \tag{3}$$

We can now compute the gradient of the objective function in Eq. 2. Using Eq. 3, we have for $j = 1, 2$:

$$\frac{\partial V}{\partial w_j}(w_1, w_2) = (y - \hat{y})\frac{\partial y}{\partial w_j} \\ = (\sigma(w_1 x_1 + w_2 x_2) - \hat{y})\,\sigma'(w_1 x_1 + w_2 x_2) x_j \\ = (y - \hat{y})\,y(1 - y) x_j \tag{4}$$

As you can see in Eq. 4, the derivative of the objective function for a perceptron with respect to a given connection weight (w_1 or w_2 in the present case) is equal to the output error $y - \hat{y}$ times a term related to the form of the nodal transfer function times the signal x_j at the "source" end of the connection. The gradient vector consists of the pair of partial derivatives:

$$\nabla V(w_1, w_2) = ((y - \hat{y})\,y(1 - y)x_1,\ (y - \hat{y})\,y(1 - y)x_2) \tag{5}$$

This expression for the gradient vector can now be used in the gradient search algorithm described earlier in pseudocode. Accomodating a larger number of connection weights is straightforward. The result is a gradient descent training algorithm for single layer neural networks. I invite you to study the above description and to implement the gradient descent perceptron training algorithm. Try experimenting with different step sizes and stopping thresholds.

Relation to classical threshold perceptron learning

An early direction in neural network research involved the study of networks built from threshold units. In a threshold unit, there are only two possible output values: -1 and $+1$, and the nodal transfer function is simply:

$$\theta(x) = \begin{cases} -1 & \text{if } x < 0 \\ +1 & \text{if } x \geq 0 \end{cases} \tag{6}$$

Contrast this with a sigmoid transfer function as in Eq. 1. Single–layer networks with threshold units were the original perceptrons as defined by Rosenblatt in 1962. The classical learning rule for threshold perceptrons is the following:

$$\Delta w_j = \begin{cases} 0 & \text{if instance classified correctly } (y = \hat{y}) \\ -\eta x_j & \text{if } -1 \text{ instance classified as } +1 \\ +\eta x_j & \text{if } +1 \text{ instance classified as } -1 \end{cases} \tag{7}$$

The classical perceptron learning rule of Eq. 7 is equivalent to the gradient descent strategy described above, assuming that the positive term $y(1 - y)$ in Eq. 5 is ignored. This shows that the classical rule follows a rational error minimization strategy.