$$J(\mathbf{a}(k+1)) \sim J(\mathbf{a}(k)) - \eta(k)\|\nabla J\|^2 + \frac{1}{2}\eta^2(k)\nabla J^t \mathbf{H} \nabla J.$$

From this it follows (Problem 12) that $J(\mathbf{a}(k+1))$ can be minimized by the choice

$$\eta(k) = \frac{\|\nabla J\|^2}{\nabla J^t \mathbf{H} \nabla J}, \tag{14}$$

where $\mathbf{H}$ depends on $\mathbf{a}$, and thus indirectly on k. This then is the optimal choice of $\eta(k)$ given the assumptions mentioned. Note that if the criterion function $J(\mathbf{a})$ is quadratic throughout the region of interest, then $\mathbf{H}$ is constant and η is a constant independent of k.

NEWTON'S ALGORITHM

An alternative approach, obtained by ignoring Eq. 12 and by choosing $\mathbf{a}(k+1)$ to minimize the second-order expansion, is *Newton's algorithm* where line 3 in Algorithm 1 is replaced by

$$\mathbf{a}(k+1) = \mathbf{a}(k) - \mathbf{H}^{-1}\nabla J, \tag{15}$$

leading to the following algorithm:

Algorithm 2 (Newton descent)

```
1 begin initialize  a, criterion θ
2         do
3             a ← a − H^{-1}∇J(a)
4         until H^{-1}∇J(a) < θ
5   return a
6 end
```

Simple gradient descent and Newton's algorithm are compared in Fig. 5.10.

Generally speaking, Newton's algorithm will usually give a greater improvement *per step* than the simple gradient descent algorithm, even with the optimal value of $\eta(k)$. However, Newton's algorithm is not applicable if the Hessian matrix $\mathbf{H}$ is singular. Furthermore, even when $\mathbf{H}$ is nonsingular, the $O(d^3)$ time required for matrix inversion on each iteration can easily offset the descent advantage. In fact, it often takes less time to set $\eta(k)$ to a constant η that is smaller than necessary and make a few more corrections than it is to compute the optimal $\eta(k)$ at each step (Computer exercise 1).

5.5 Minimizing the Perceptron Criterion Function

5.5.1 The Perceptron Criterion Function

Consider now the problem of constructing a criterion function for solving the linear inequalities $\mathbf{a}^t\mathbf{y}_i > 0$. The most obvious choice is to let $J(\mathbf{a};\ \mathbf{y}_1, ..., \mathbf{y}_n)$ be the number of samples misclassified by $\mathbf{a}$. However, because this function is piecewise constant, it is obviously a poor candidate for a gradient search. A better choice is the *Perceptron criterion function*

$$J_p(\mathbf{a}) = \sum_{\mathbf{y}\in\mathcal{Y}} (-\mathbf{a}^t\mathbf{y}), \tag{16}$$

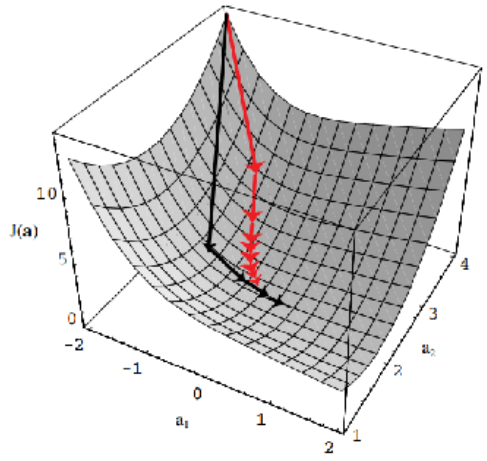

Figure 5.10: The sequence of weight vectors given by a simple gradient descent method (red) and by Newton's (second order) algorithm (black). Newton's method typically leads to greater improvement per step, even when using optimal learning rates for both methods. However the added computational burden of inverting the Hessian matrix used in Newton's method is not always justified, and simple descent may suffice.

where $\mathcal{Y}(\mathbf{a})$ is the set of samples *misclassified* by $\mathbf{a}$. (If no samples are misclassified, $\mathcal{Y}$ is empty and we define J_p to be zero.) Since $\mathbf{a}^t\mathbf{y} \leq 0$ if $\mathbf{y}$ is misclassified, $J_p(\mathbf{a})$ is never negative, being zero only if $\mathbf{a}$ is a solution vector, or if $\mathbf{a}$ is on the decision boundary. Geometrically, $J_p(\mathbf{a})$ is proportional to the sum of the distances from the misclassified samples to the decision boundary. Figure 5.11 illustrates J_p for a simple two-dimensional example.

Since the jth component of the gradient of J_p is $\partial J_p/\partial a_j$, we see from Eq. 16 that

$$\nabla J_p = \sum_{\mathbf{y}\in\mathcal{Y}} (-\mathbf{y}), \tag{17}$$

and hence the update rule becomes

$$\mathbf{a}(k+1) = \mathbf{a}(k) + \eta(k) \sum_{\mathbf{y}\in\mathcal{Y}_k} \mathbf{y}, \tag{18}$$

where $\mathcal{Y}_k$ is the set of samples misclassified by $\mathbf{a}(k)$. Thus the Perceptron algorithm is:

Algorithm 3 (Batch Perceptron)

1 **begin** **initialize** $\mathbf{a}, \eta(\cdot)$, criterion $\theta, k = 0$
2 **do** $k \leftarrow k + 1$
3 $\mathbf{a} \leftarrow \mathbf{a} + \eta(k) \sum_{\mathbf{y}\in\mathcal{Y}_k} \mathbf{y}$
4 **until** $\eta(k) \sum_{\mathbf{y}\in\mathcal{Y}_k} \mathbf{y} < \theta$
5 **return** $\mathbf{a}$
6 **end**

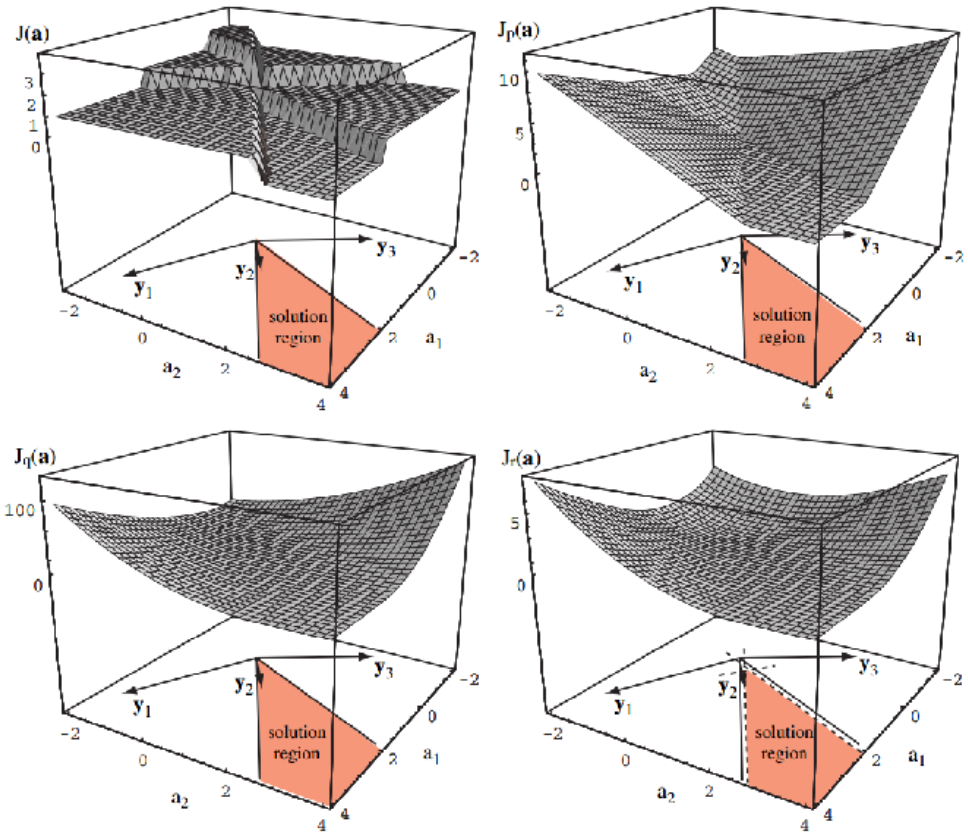

Figure 5.11: Four learning criteria as a function of weights in a linear classifier. At the upper left is the total number of patterns misclassified, which is piecewise constant and hence unacceptable for gradient descent procedures. At the upper right is the Perceptron criterion (Eq. 16), which is piecewise linear and acceptable for gradient descent. The lower left is squared error (Eq. 32), which has nice analytic properties and is useful even when the patterns are not linearly separable. The lower right is the square error with margin (Eq. 33). A designer may adjust the margin b in order to force the solution vector to lie toward the middle of the $b = 0$ solution region in hopes of improving generalization of the resulting classifier.

BATCH TRAINING

Thus, the batch Perceptron algorithm for finding a solution vector can be stated very simply: the next weight vector is obtained by adding some multiple of the sum of the misclassified samples to the present weight vector. We use the term "batch" to refer to the fact that (in general) a large group of samples is used when computing each weight update. (We shall soon see alternate methods based on single samples.) Figure 5.12 shows how this algorithm yields a solution vector for a simple two-dimensional example with $\mathbf{a}(1) = \mathbf{0}$, and $\eta(k) = 1$. We shall now show that it will yield a solution for any linearly separable problem.

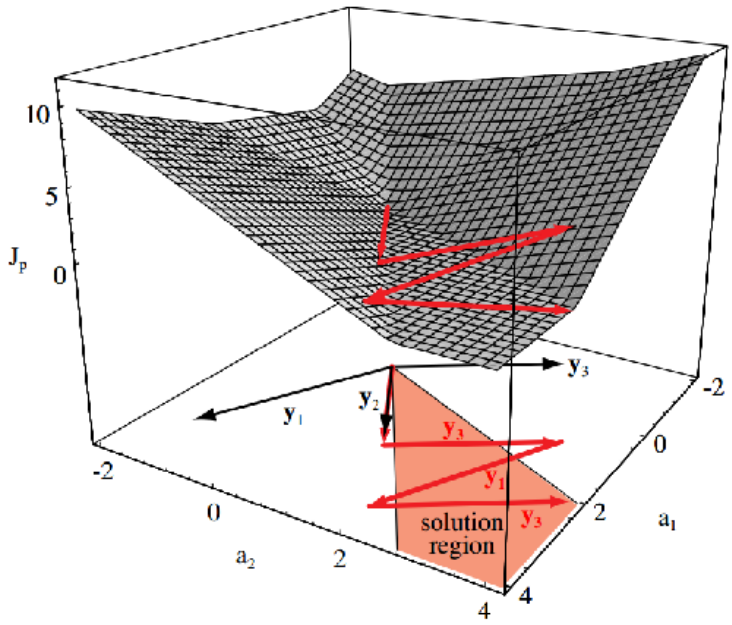

Figure 5.12: The Perceptron criterion, J_p is plotted as a function of the weights a_1 and a_2 for a three-pattern problem. The weight vector begins at $\mathbf{0}$, and the algorithm sequentially adds to it vectors equal to the "normalized" misclassified patterns themselves. In the example shown, this sequence is $\mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_1, \mathbf{y}_3$, at which time the vector lies in the solution region and iteration terminates. Note that the second update (by $\mathbf{y}_3$) takes the candidate vector *farther* from the solution region than after the first update (cf. Theorem 5.1. (In an alternate, batch method, *all* the misclassified points are added at each iteration step leading to a smoother trajectory in weight space.)

5.5.2 Convergence Proof for Single-Sample Correction

We shall begin our examination of convergence properties of the Perceptron algorithm with a variant that is easier to analyze. Rather than testing $\mathbf{a}(k)$ on all of the samples and basing our correction of the set $\mathcal{Y}_k$ of misclassified training samples, we shall consider the samples in a sequence and shall modify the weight vector whenever it misclassifies a *single* sample. For the purposes of the convergence proof, the detailed nature of the sequence is unimportant as long as every sample appears in the sequence infinitely often. The simplest way to assure this is to repeat the samples cyclically, though from a practical point of view random selection is often to be preferred (Sec. 5.8.5). Clearly neither the batch nor this single-sample version of the Perceptron algorithm are on-line since we must store and potentially revisit all of the training patterns.

Two further simplifications help to clarify the exposition. First, we shall temporarily restrict our attention to the case in which $\eta(k)$ is constant — the so-called *fixed-increment* case. It is clear from Eq. 18 that if $\eta(t)$ is constant it merely serves to scale the samples; thus, in the fixed-increment case we can take $\eta(t) = 1$ with no loss in generality. The second simplification merely involves notation. When the samples

FIXED INCREMENT