Now, an automatic procedure to find a $\underline{w}$ in the solution region.

Training (Learning) Algorithms – Preliminaries

General Procedure –

1. Construct a cost function J($\underline{w}$) (appropriately designed & chosen)
2. Minimize J($\underline{w}$) with respect to $\underline{w}$ (i.e., update $\underline{w}$ and check J($\underline{w}$) decreasing)
3. $\underline{w}$ at minimum of J($\underline{w}$) will be a solution weight vector $\underline{w}$

Use Gradient Descent on J($\underline{w}$)

Let $\underline{w}$(i) = solution weight vector at iteration i, then

$$\underline{w}(i+1) = \underline{w}(i) - \alpha(i)\nabla_w J[\underline{w}(i)]$$

This is the basic Gradient Descent

$-\nabla_w J[\underline{w}(i)]$ points in the direction of steepest descent of J.

$$\nabla_w J(\underline{w}) = \frac{\partial J}{\partial w_1}\hat{u}_1 + \frac{\partial J}{\partial w_2}\hat{u}_2 + ... + \frac{\partial J}{\partial w_N}\hat{u}_N \qquad \text{(N-D space)}$$

Two Considerations

1. Must choose an appropriate criterion (cost) function J($\underline{w}$). How? Coming in the next section.
2. What about choosing the learning rate parameter, $\alpha(i)$?

- There are various choices, specific to choices of J($\underline{w}$) (more to follow). These are mostly suboptimal in terms of minimizes J after each step.
- There is an optimal choice (as defined above) under certain assumptions.
- Again, coming soon.

So,

1. Set $\alpha(i)$ appropriately,
2. Then find a new $\underline{w}$ via iterations usig the gradient descent $\underline{w}$(k+1)=$\underline{w}$(k)-α(k)∇J[$\underline{w}$(k)],.

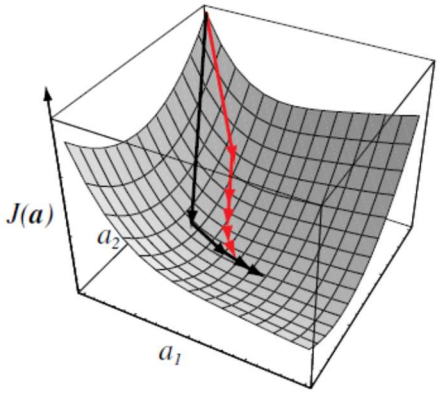


FIGURE 5.10. The sequence of weight vectors given by a simple gradient descent method (red) and by Newton's (second order) algorithm (black). Newton's method typically leads to greater improvement per step, even when using optimal learning rates for both methods. However the added computational burden of inverting the Hessian matrix used in Newton's method is not always justified, and simple gradient descent may suffice. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

- Red arrows show how a simple gradient descent method works
- Black arrows show how a Newton's method works. What is the Newton's method in optimization?

Perceptron Algorithm

General Idea –

If $\underline{y}_m^{(1)}$ gives $\underline{w}^T \underline{y}_m^{(1)} < 0$ (i.e., misclassified), then increase $\underline{w}$

General Perceptron Criterion Function

$J(\underline{w}) = \sum_{\underline{y} \in Y} (-\underline{w}^T \underline{\tilde{y}})$ (General Form)

where Y is the set of misclassified prototypes.

$J(\underline{w}) \geq 0$ always (if all prototypes are correctly classified, $J(\underline{w})=0$).

($\underline{w}^T \underline{\tilde{y}} < 0$ for misclassified)

Other Criterion (Cost) Functions

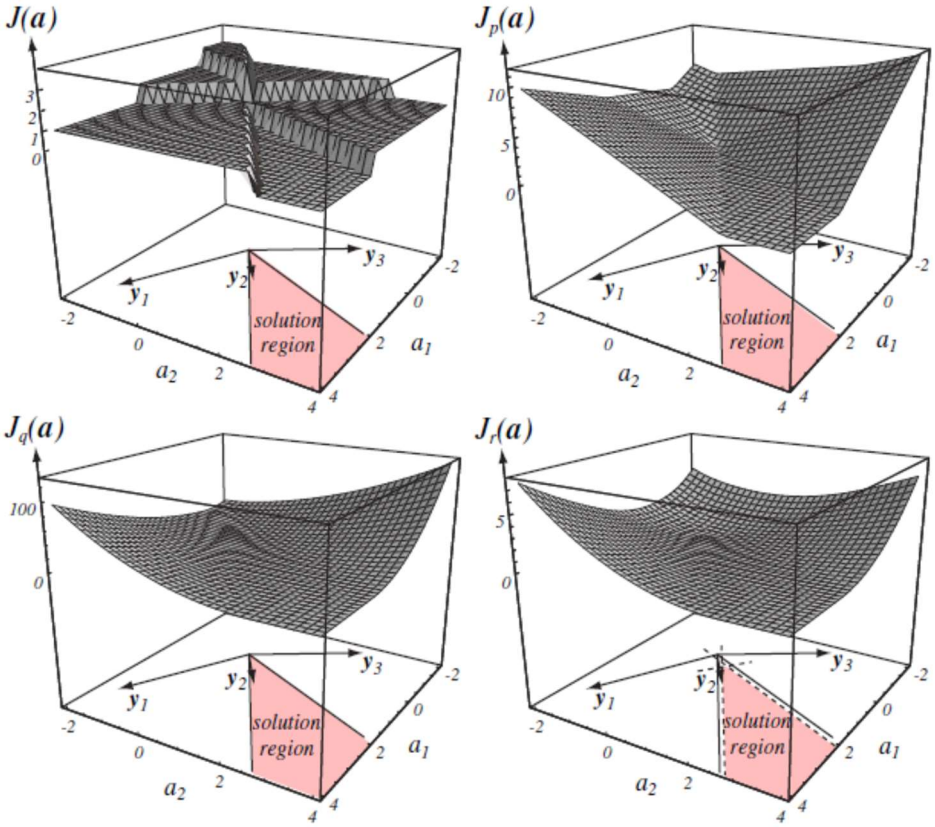


FIGURE 5.11. Four learning criteria as a function of weights in a linear classifier. At the upper left is the total number of patterns misclassified, which is piecewise constant and hence unacceptable for gradient descent procedures. At the upper right is the Perceptron criterion (Eq. 16), which is piecewise 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. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

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

General Form

$$J_q(\mathbf{a}) = \sum_{\mathbf{y}\in\mathcal{Y}} (\mathbf{a}^t\mathbf{y})^2, \qquad (32)$$

Squared From

$$J_r(\mathbf{a}) = \frac{1}{2}\sum_{\mathbf{y}\in\mathcal{Y}} \frac{(\mathbf{a}^t\mathbf{y} - b)^2}{\|\mathbf{y}\|^2}, \qquad (33)$$

with b, a margin value

Note a=w

[Perceptron Algorithm I] (One-at-a-time, Un-reflected Prototypes)

If ∃ prototype from S_1, $\underline{w}^T \underline{y}_m^{(1)} \le 0$, then increase $\underline{w}$,

If ∃ prototype from S_2, $\underline{w}^T \underline{y}_m^{(2)} \ge 0$, then decrease $\underline{w}$,

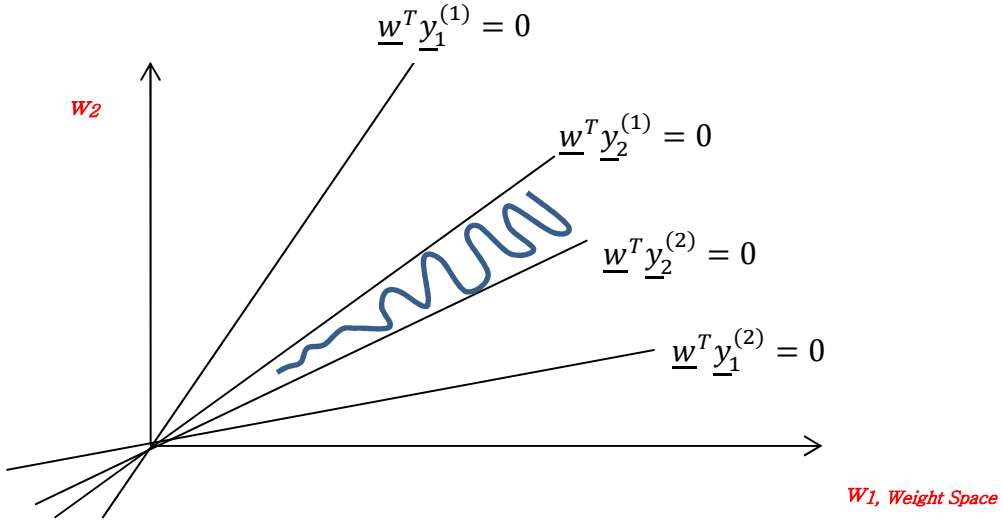
Repeat for all M_1+M_2 prototypes;
Continue cycling through all prototypes until $\underline{w}$ is no longer updated.
For the i-th iteration:

If $\underline{w}^T(i)\underline{y}_m^{(1)} \le 0$, then $\underline{w}(i+1) = \underline{w}(i) + \alpha(i)\underline{y}_m^{(1)}$

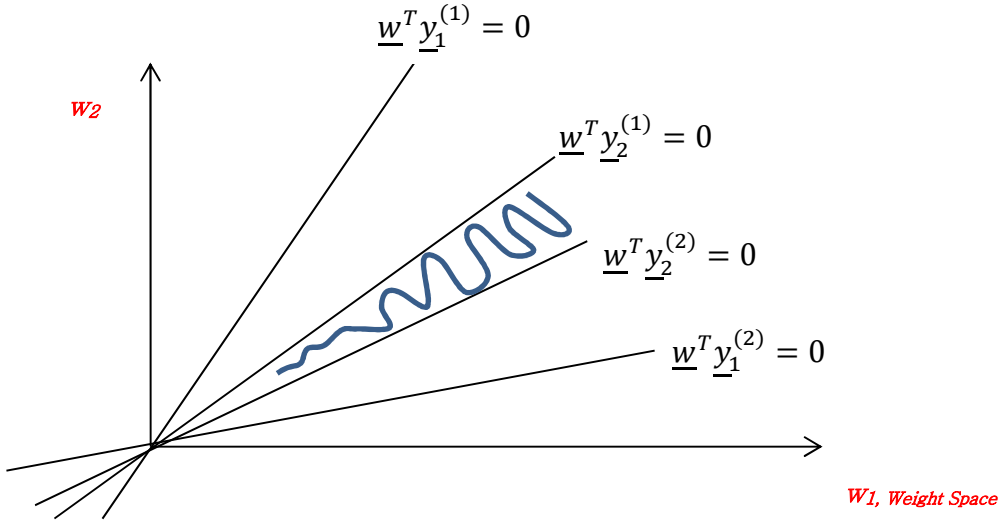
If $\underline{w}^T(i)\underline{y}_m^{(2)} > 0$, then $\underline{w}(i+1) = \underline{w}(i) - \alpha(i)\underline{y}_m^{(2)}$

Otherwise $\underline{w}(i+1)=\underline{w}(i)$
(Next prototype)
$\alpha(i) > 0$

1 pass through all prototypes = 1 epoch



What if α is very large?



[Perceptron Algorithm II] (One-at-a-time, Reflected Prototypes)

Use the reflected prototypes

$w(i+1) = w(i) + \alpha(i)\check{y}$ if the prototype $\check{y}$ is misclassified.

$w(i+1) = w(i)$ if the prototype $\check{y}$ is correctly classified.

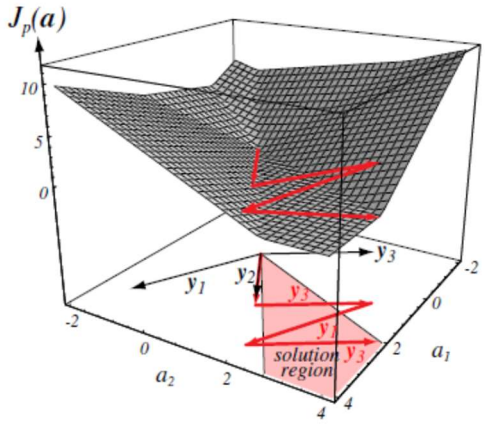


FIGURE 5.12. The Perceptron criterion, $J_p(\mathbf{a})$, is plotted as a function of the weights a_1 and a_2 for a three-pattern problem. The weight vector begins at $\mathbf{0}$, and the algorithm sequentially adds to it vectors equal to the "normalized" misclassified patterns themselves. In the example 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). From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

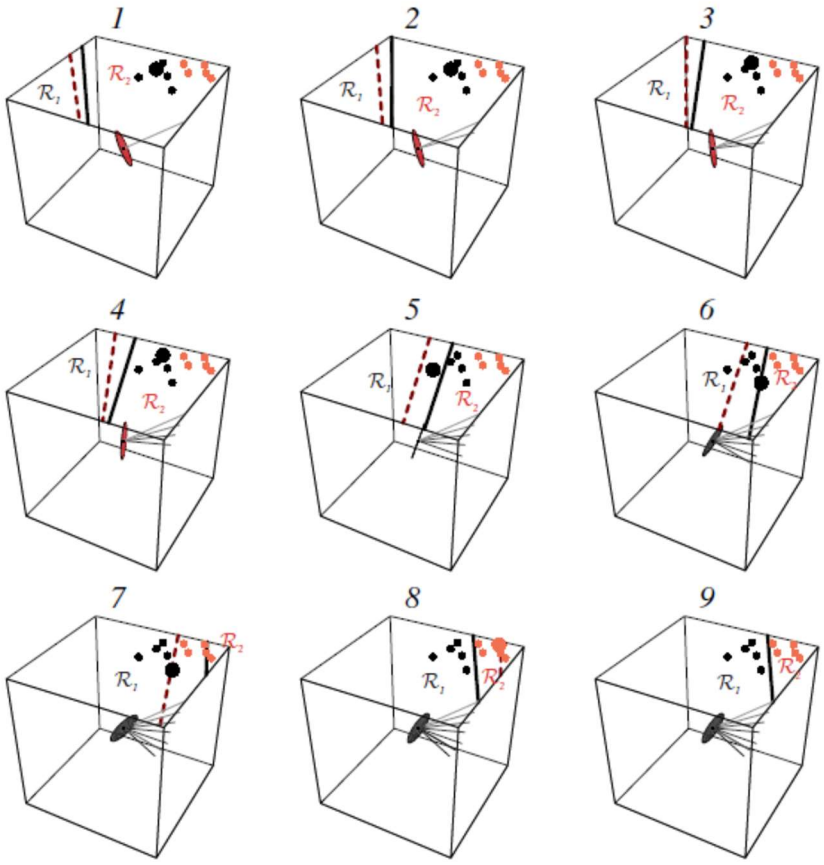


FIGURE 5.13. Samples from two categories, ω_1 (black) and ω_2 (red) are shown in augmented feature space, along with an augmented weight vector $\mathbf{a}$. At each step in a fixed-increment rule, one of the misclassified patterns, $\mathbf{y}^k$, is shown by the large dot. A correction $\Delta\mathbf{a}$ (proportional to the pattern vector $\mathbf{y}^k$) is added to the weight vector—toward an ω_1 point or away from an ω_2 point. This changes the decision boundary from the dashed position (from the previous update) to the solid position. The sequence of resulting $\mathbf{a}$ vectors is shown, where later values are shown darker. In this example, by step 9 a solution vector has been found and the categories are successfully separated by the decision boundary shown. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Choice of α(i), Two Sample Choices:

1. Fixed Increment Rule
   α(i)=constant (independent of i)
   α(i)>0

2. Absolute Correction Rule
   Choose α at each iteration to be just large enough to guarantee correct classification after weigh adjustment.

   i.e., If $\underline{w}^T(i+1)\underline{y}_m^{(1)} > 0$

   then $\underline{w}^T(i+1)\underline{y}_m^{(1)} = [\underline{w}(i) + \alpha \underline{y}_m^{(1)}]^T \underline{y}_m^{(1)} > 0$

   Satisfied if $\alpha = [\frac{\left|\underline{w}^T(i)\underline{y}_m^{(1)}\right|}{\underline{y}_m^{(1)^T}\underline{y}_m^{(1)}}]$

   [*]=smallest integer larger than *
   Guaranteed convergence.

Notes:

1. Fixed increment with α>0 is guaranteed to converge if prototypes are linearly separable.
2. Absolute correction is guaranteed to converge if prototypes are linearly separable.

Problems:

1. If prototypes are not linearly separable, perceptron will not converge
2. Perceptron terminated early may give poor classification results.

※ Frank Rosenblatt who simulated Perceptron on an IBM computer in 1957.

Frank Rosenblatt
1928–1969

Rosenblatt's perceptron played an important role in the history of machine learning. Initially, Rosenblatt simulated the perceptron on an IBM 704 computer at Cornell in 1957, but by the early 1960s he had built special-purpose hardware that provided a direct, parallel implementation of perceptron learning. Many of his ideas were encapsulated in "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms" published in 1962. Rosenblatt's work was criticized by Marvin Minksy, whose objections were published in the book "Perceptrons", co-authored with Seymour Papert. This book was widely misinterpreted at the time as showing that neural networks were fatally flawed and could only learn solutions for linearly separable problems. In fact, it only proved such limitations in the case of single-layer networks such as the perceptron and merely conjectured (incorrectly) that they applied to more general network models. Unfortunately, however, this book contributed to the substantial decline in research funding for neural computing, a situation that was not reversed until the mid-1980s. Today, there are many hundreds, if not thousands, of applications of neural networks in widespread use, with examples in areas such as handwriting recognition and information retrieval being used routinely by millions of people.

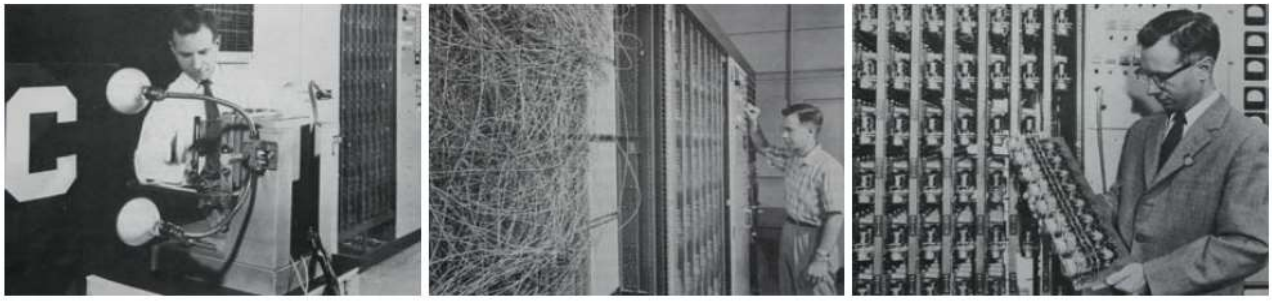


Figure 4.8 Illustration of the Mark 1 perceptron hardware. The photograph on the left shows how the inputs were obtained using a simple camera system in which an input scene, in this case a printed character, was illuminated by powerful lights, and an image focussed onto a 20×20 array of cadmium sulphide photocells, giving a primitive 400 pixel image. The perceptron also had a patch board, shown in the middle photograph, which allowed different configurations of input features to be tried. Often these were wired up at random to demonstrate the ability of the perceptron to learn without the need for precise wiring, in contrast to a modern digital computer. The photograph on the right shows one of the racks of adaptive weights. Each weight was implemented using a rotary variable resistor, also called a potentiometer, driven by an electric motor thereby allowing the value of the weight to be adjusted automatically by the learning algorithm.