# Linear discriminant functions: Perceptron

### The two-category linearly separable case

Let's suppose we have a set of *n* samples $\mathbf{y}_1,\ldots,\mathbf{y}_n$, some labeled $\omega_1$ and some $\omega_2$. We want to use these samples to determine the weights **a** in a linear discriminant function $g(x) = \mathbf{a}^t\mathbf{y}$. If there exists a weight vector that is able to classify all the samples correctly, the samples are said to be *linearly separable*.

If $\mathbf{y}_i$ is labeled $\omega_1$ and $\mathbf{a}^t\mathbf{y}_i > 0$, or if $\mathbf{y}_i$ is labeled $\omega_2$ and $\mathbf{a}^t\mathbf{y}_i < 0$, then the sample is classified correctly. To simplify things, we want to "normalize" the input samples so that we could forget the labels and look for a vector such that $\mathbf{a}^t\mathbf{y}_i > 0$ for all of the samples. Such a weight vector is called a *separating vector* or more generally a *solution vector*. The "normalization" could be done by multiplying the input samples with their class labels, i.e., replace all samples from class 2 by their negatives.

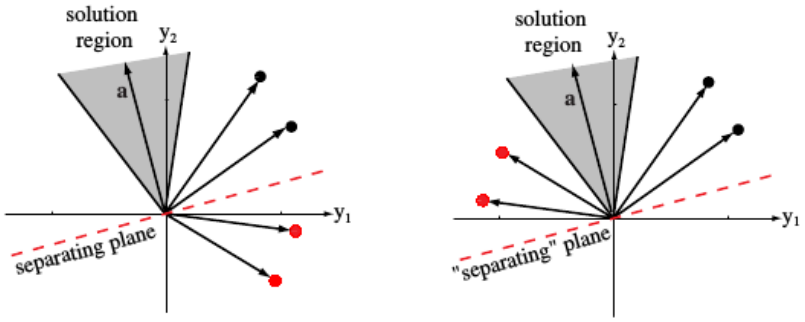


Figure 5.8: Four training samples (black for $\omega_1$, red for $\omega_2$) and the solution region in feature space. The figure on the left shows the raw data; the solution vectors leads to a plane that separates the patterns from the two categories. In the figure on the right, the red points have been "normalized" — i.e., changed in sign. Now the solution vector leads to a plane that places all "normalized" points on the same side.

Solution vector, if it exists in not unique. There are several ways to impose additional requirements to constrain the solution vector. One possibility is to seek the minimum-length weight vector satisfying $\mathbf{a}^t\mathbf{y}_i \geq b$ for all i, where b is a positive constant called the *margin*. The motivation behind these attempts to find a solution vector closer to the "middle" of the solution region is the natural belief that the resulting solution is more likely to classify new test samples correctly.

### Perceptron criterion function

Out goal if then to find a weight vector **a** such that $\mathbf{a}^t\mathbf{y}_i>0$ for all the samples (assuming one exists). Mathematically this can be expressed as finding a weight vector a that minimizes the no. of samples misclassified. However, the function is piecewise constant (discontinuous, and hence nondifferentiable) and is difficult to optimize. A better choice is the *perceptron criterion function*

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

where $\mathcal{Y}(\mathbf{a})$ is the set of samples missclassified by **a**. The criterion is proportional to the sum of distances from the misclassified samples to the decision boundary. Now, the minimization is mathematically tractable, and hence it is a better criterion function than no. of misclassifications.

**Gradient descent**

Perceptron criterion function J(a) can be minimized using gradient descent. The basic procedure is as follows:
- Start with an arbitrarily chosen weight vector a(1)
- Compute the gradient vector $\nabla J(\mathbf{a}(1))$
- The next value a(2) is obtained by moving in the direction of the steepest descent, i.e. along the negative of the gradient.
- In general, the k+1-th solution is obtained by

$$\mathbf{a}(k+1) = \mathbf{a}(k) - \eta(k)\nabla J(\mathbf{a}(k))$$

where $\eta$ is a positive scale factor or *learning rate* that sets the step size. If *η(k)* is too small, convergence is needlessly slow, whereas if *η(k)* is too large, the correction process will overshoot and can even diverge.

**Batch perceptron**

We can now use gradient descent algorithm to find **a**. Since the j[th] component of the gradient of $J_p$ is $\partial J_p/\partial a_j$, we see from the perceptron criterion function that

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

and hence the update rule becomes

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

where $\mathcal{Y}_k$ is the set of samples misclassified by **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** $\left|\eta(k) \sum_{\mathbf{y}\in\mathcal{Y}_k} \mathbf{y}\right| < \theta$
5 **return** **a**
6 **end**

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.

The perceptron algorithm is also termed the *single-layer perceptron*, to distinguish it from a *multilayer perceptron*, which is (a misnomer for) a more complicated neural network. As a linear classifier, the single-layer perceptron is the simplest feedforward neural network