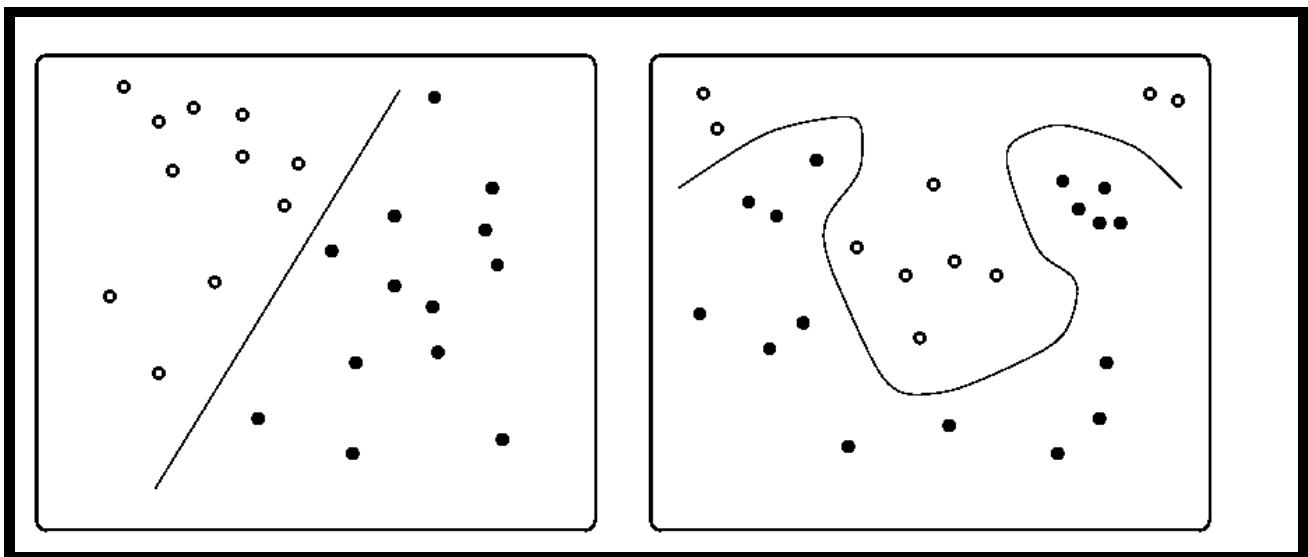


Classification

Discriminants

Neural networks can also be used to classify data. Unlike regression problems, where the goal is to produce a particular output value for a given input, classification problems require us to label each data point as belonging to one of n classes. Neural networks can do this by learning a **discriminant** function which **separates** the classes. For example, a network with a single linear output can solve a two-class problem by learning a discriminant function which is greater than zero for one class, and less than zero for the other. Fig. 6 shows two such two-class problems, with filled dots belonging to one class, and unfilled dots to the other. In each case, a line is drawn where a discriminant function that separates the two classes is zero.



6)

(Fig.

On the left side, a straight line can serve as a discriminant: we can place the line such that all filled dots lie on one side, and all unfilled ones lie on the other. The classes are said to be **linearly separable**. Such problems can be learned by neural networks without any hidden units. On the right side, a highly non-linear function is required to ensure class separation. This problem can be solved only by a neural network with hidden units.

Binomial

To use a neural network for classification, we need to construct an equivalent function approximation problem by assigning a target value for each class. For a **binomial** (two-class) problem we can use a network with a single output y , and binary target values: 1 for one class, and 0 for the other. We can thus interpret the network's output as an estimate of the probability that a given pattern belongs to the '1' class. To classify a new pattern after training, we then employ the **maximum likelihood** discriminant, y

> 0.5.

A network with linear output used in this fashion, however, will expend a lot of its effort on getting the target values *exactly* right for its training points - when all we actually care about is the correct positioning of the discriminant. The solution is to use an activation function at the output that saturates at the two target values: such a function will be close to the target value for any net input that is sufficiently large and has the correct sign. Specifically, we use the [logistic sigmoid](#) function

$$f(u) = \frac{1}{1 + e^{-u}} \quad f'(u) = f(u)(1 - f(u))$$

Given the probabilistic interpretation, a network output of, say, 0.01 for a pattern that is actually in the '1' class is a *much* more serious error than, say, 0.1. Unfortunately the sum-squared loss function makes almost no distinction between these two cases. A loss function that is appropriate for dealing with probabilities is the **cross-entropy** error. For the two-class case, it is given by

$$E = -t \ln y - (1 - t) \ln(1 - y)$$

When logistic output units and cross-entropy error are used together in backpropagation learning, the error signal for the output unit becomes just the difference between target and output:

$$\frac{\partial E}{\partial net} = \dots = y - t$$

In other words, implementing cross-entropy error for this case amounts to nothing more than omitting the $f'(net)$ factor that the error signal would otherwise get multiplied by. This is not an accident, but indicative of a deeper mathematical connection: cross-entropy error and logistic outputs are the "correct" combination to use for binomial probabilities, just like linear outputs and sum-squared error are for scalar values.

Multinomial

If we have multiple *independent* binary attributes by which to classify the data, we can use a network with multiple logistic outputs and cross-entropy error. For **multinomial** classification problems (1-of-n, where $n > 2$) we use a network with n outputs, one corresponding to each class, and target values of 1 for the correct class, and 0 otherwise. Since these targets are not independent of each other, however, it is no longer appropriate to use logistic output units. The correct generalization of the logistic sigmoid to the multinomial case is the **softmax** activation function:

$$f(net_i) = \frac{e^{net_i}}{\sum_o e^{net_o}}$$

where o ranges over the n output units. The cross-entropy error for such an output layer is given by

$$E = - \sum_o t_o \ln y_o$$

Since all the nodes in a softmax output layer interact (the value of each node depends on the values of all the others), the derivative of the cross-entropy error is difficult to calculate. Fortunately, it again simplifies to

$$\frac{\partial E}{\partial net} = \dots = y - t$$

so we don't have to worry about it.

[\[Top\]](#)

[\[Next: Non-Supervised Learning\]](#)

[\[Back to the first page\]](#)