The Knowledge Engineering Review, Vol. 15:2, 2000, 151–170
Printed in the United Kingdom

Designing neural network architectures for pattern recognition

MIROSLAV KUBAT

Center for Advanced Computer Studies, University of Louisiana at Lafayette, LA 70504–4330, USA
(email: mkubat@cacs.usl.edu)

Abstract

An appropriately designed architecture of a neural network is essential to many realistic pattern-recognition tasks. A choice of just the right number of neurons, and their interconnections, can cut learning costs by orders of magnitude, and still warrant high classification accuracy. Surprisingly, textbooks often neglect this issue. A specialist seeking systematic information will soon realize that relevant material is scattered over diverse sources, each with a different perspective, terminology and goals. This brief survey attempts to rectify the situation by explaining the involved aspects, and by describing some of the fundamental techniques.

1 Introduction

Artificial neural networks rank among the most popular pattern-recognition tools. Their widespread use was spawned by the invention of algorithms for their induction from sets of pre-classified training examples, and by the publication of several persuasive case studies in the book edited by Rumelhart & McClelland (1986). During the past two decades, scientists have addressed a host of problems pertaining to learning techniques, classification behaviors, and various application paradigms. As a result, the practical and theoretical aspects of neural networks, and of the attendant learning methods, are now fairly well understood.

This paper will focus on issues related to the design of a neural layout. The number of layers, as well as the number of interconnections of neurons, affect the network's ability to learn. Small networks are not sufficiently flexible; large networks, apart from incurring high computation costs, can overfit noisy training data. The conflicting nature of the involved trade-offs complicates the search for an "ideal" size that is neither too small nor too large for a given task.

To shun difficulties, many researchers in the 1980s resorted to trial and error. They experimented with several different networks, and then selected the one that met their requirements in the most favorable manner. Others, dissatisfied with such an *ad hoc* methodology, embarked on more systematic studies. During the next couple of years, they developed techniques that can roughly be divided in the following categories:

1. *Search-based* strategies that exploit heuristic search (including the genetic algorithm).
2. *Piecemeal algorithms* that train each neuron on a different training set (either on re-labeled examples or on different subsets of examples).
3. *Logic-based* strategies that determine the architecture using prior knowledge expressed in the form of production rules or decision trees.

Existing textbooks do not seem to devote sufficient attention to this problem. A student seeking deeper understanding of the involved issues has to be prepared for tedious combing through diverse materials that can be difficult to get hold of. To provide some help, the text below outlines the essentials of some of the most representative methods.

2 Classifers, networks and error functions

As indicated, the paper will deal with methods for the induction of systems for automated classification of data (pattern recognition). The input of the requisite learning algorithms consists of pairs [**x**, c(**x**)], where the vector **x** = (x_1, x_2,. . .,x_n) describes a training *example* and c(**x**) denotes its class. The variables x_i are referred to as *attributes*, and the cartesian space defined by these coordinates is called an *instance space*. In the sequel, the focus will be on *numeric* attributes. The class of **x** is determined by some function $c : R^n \rightarrow L$, where L is the set of *labels* (the symbols representing the classes). In applications that seek to distinguish positive examples of a particular class from negative examples, c(**x**) is a binary function, for instance $c : R^n \rightarrow \{0,1\}$.

The learner's task is to induce a *classifier*, $h : R^n \rightarrow L$, that approximates the function c, while satisfying certain user-specified criteria. One such criterion requires that the probability of h(**x**) $\neq$ c(**x**) for a randomly drawn **x** be minimized; another will restrict the learner to a specific data structure in which the function h should be expressed (such as a decision tree or a neural network). The classifier divides the instance space into two or more regions separated by a *decision surface*. All examples in a region are assigned by the classifier the same class label.

Figure 1 illustrates a relatively broad class of classifiers, a *multilayer perceptron*, consisting of two or more layers of tests. Each test performs a function $f : R^n \rightarrow S$, where S is a subset of real numbers ($S \subseteq R$). Prevailing terminology refers to the tests as *neurons*. Figure 1 contains *output* neurons that determine the class label of **x** (each output neuron usually represents one class) and *hidden* neurons, called so because they cannot be "seen" from outside. The interconnection of adjacent layers is either complete, as in Figure 1, or partial, when some links are missing. Each link has a *weight*. A neuron's input, z, is calculated as a weighted sum of the signals arriving along the incoming links. A neuron is associated with a *transfer function* that maps its input to a binary or continuous output. From the various transfer functions encountered in the literature, this paper will consider only the two depicted in Figure 2. The `hardlim` function (the name is borrowed from the programming language MATLAB) is a binary function that returns $F(z) = 1$ for $z > 0$, and $F(z) = 0$ (or sometimes $F(z) = -1$) otherwise. The `sigmoid` function is continuous, and its output is determined by the following expression:

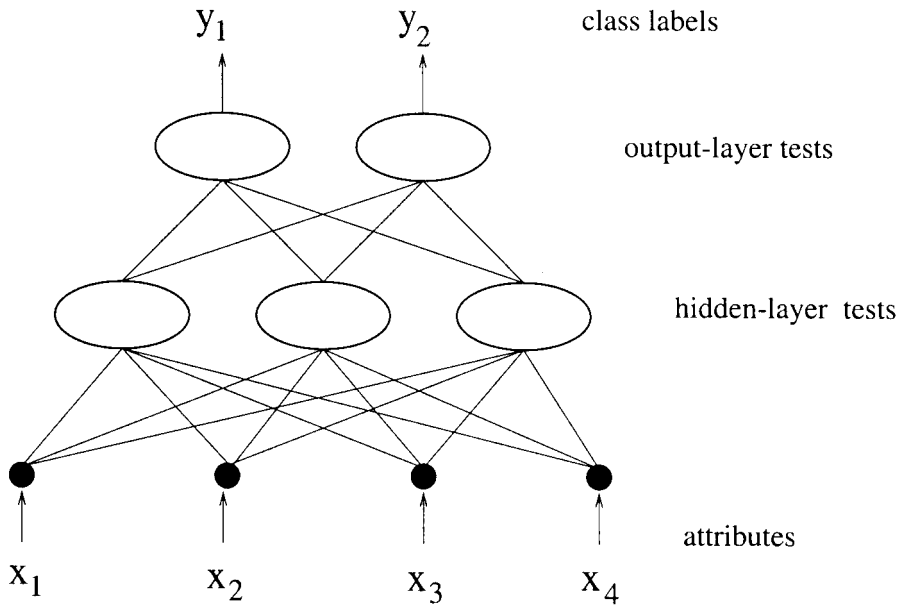

Figure 1 A multilayer perceptron consists of two or more layers of neurons that represent specific tests. The tests in adjacent layers are fully or partially interconnected. Each of the connection links is associated with a numeric value—a *weight*

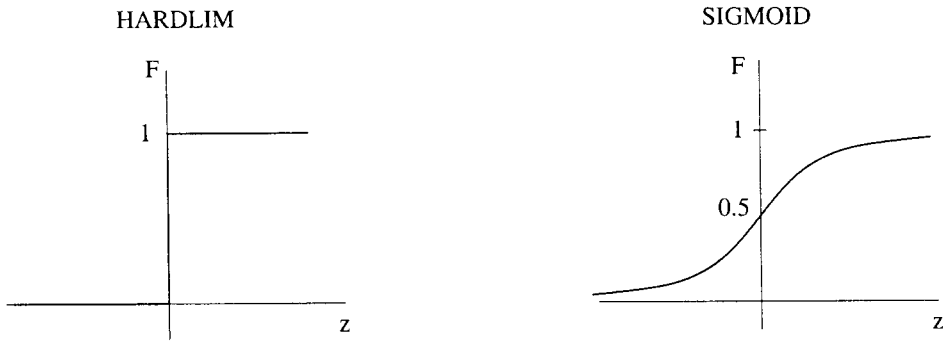

Figure 2 This paper will consider two different transfer functions: `hardlim` (binary output) and `sigmoid` (continuous output)

$$F(z) = \frac{1}{1 + e^{-z}} \tag{1}$$

The task of a neural network in pattern recognition is to predict the class of a given input vector, **x**. This is accomplished by a *forward pass*. At the beginning of the forward pass, the weighted sum of **x**'s attributes is presented at the lowest layer of neurons. If w_{ij} is the weight associated with the link leading from the ith attribute to the jth neuron, then the neuron receives $z_j = \Sigma_i\, w_{ij}x_i$. The weighted sum is then subjected to the neuron's transfer function, and the output is passed to the next layer. The process continues until the top layer is reached.

If the output neurons have `hardlim` transfer functions, then the network's output is a binary vector, $\mathbf{y} \in \{0, 1\}^m$. Each of the m output neurons represents one class. Let us describe by $\mathbf{y} = (0, 1)$ the response of a network that has two output-layer neurons, one giving 0 and the other giving 1. In this case, **x** is classified as belonging to the second class. If the output neurons have `sigmoid` transfer functions, then the network's output is a numeric vector, $\mathbf{y} \in (0, 1)^m$ and each element of **y** determines the degree of confidence in the corresponding class. For instance, a reasonable interpretation of $\mathbf{y} = (0.2, 0.6)$ will stipulate that **x** is likely to belong to the second class because $0.6 > 0.2$. Realistic implementations will label **x** with the ith class where i is the index of the output neuron giving the maximum output ($i = \text{argmax}_j\, y_j$).

How does a researcher evaluate the network's classification *performance*? Common practice takes a set of pre-classified testing examples, and attempts to re-classify them, incrementing a counter whenever the class label recommended by the network differs from the original. The status of this counter after all testing examples have been re-classified gives the number of errors committed by the network. The *error rate* is defined as the relative frequency of misclassified examples (the counter divided by the number of testing examples).

Being an integer, the error rate does not give the full picture of the classifier's behavior. Consider two different networks, each with three output neurons. For a given input vector **x**, the first network outputs $\mathbf{y}^{(1)} = (0.5, 0.2, 0.9)$ and the second network outputs $\mathbf{y}^{(2)} = (0.6, 0.6, 0.7)$. This means that both of them suggest the third class. If the correct answer is the *second* class, the error is more strongly pronounced in the first case because the second network displayed some uncertainty, indicated by the low variance of output values. Since error rate ignores this information, researchers prefer to work with the *Mean Square Error* (MSE). Let the network's output values be denoted by y_i, and let the desired output values be t_i (target). The mean square error for m outputs is defined by the following equation:

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(y_i - t_i)^2 \tag{2}$$

If **x** belongs to the second class, then the target vector is $\mathbf{t} = (0, 1, 0)$. The mean square error of the first network is $\frac{1}{3}[(0 - 0.5)^2 + (1 - 0.2)^2 + (0 - 0.9)^2] = 0.2$ and the mean square error of the second network is $\frac{1}{3}[(0 - 0.6)^2 + (1 - 0.6)^2 + (0 - 0.7)^2] = 0.3$. The fact that the former is smaller

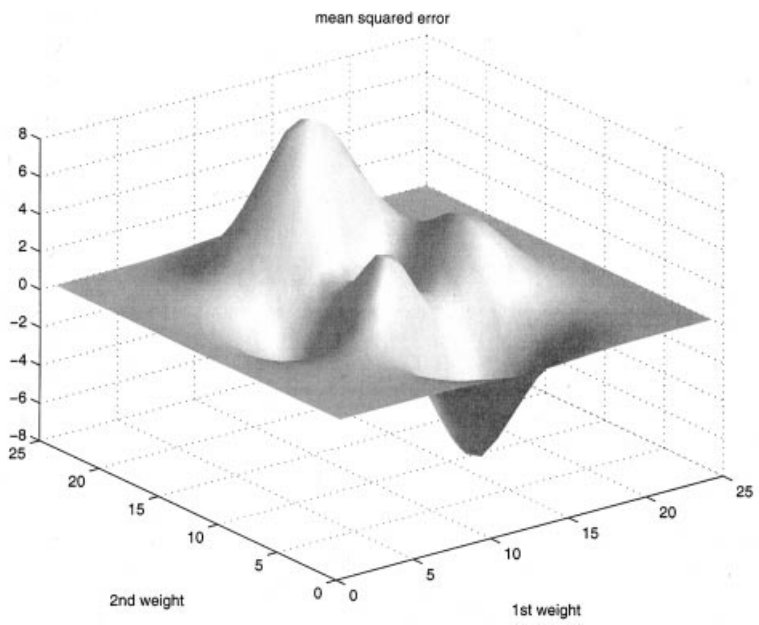

Figure 3 Each set of weights is associated with a certain value of the Mean Square Error (MSE). Gradient-descent techniques reduce the MSE by stepwise corrections of weights

complies with the intuition that, in this particular example, the first network is less wrong than the second.

3 Gradient-descent learning

Mathematically speaking, a multilayer perceptron with two layers of neurons is a physical representation of the following compound function (its generalization for a network with more than two layers is straightforward):

$$y_i = f_i\left(\sum_j w_{ij}^{(1)} g_j\left(\sum_k w_{jk}^{(2)} x_k\right)\right) \tag{3}$$

Here, f_i and g_j are the transfer functions of the output neurons and hidden neurons, respectively; $w_{ij}^{(1)}$ are the weights of the links leading to the output neurons; and $w_{jk}^{(2)}$ are the weights of the links leading to the hidden layer. In a broader sense, an algorithm for the induction of a neural network should define the number of neurons, their layout, the weights of their interconnections, and perhaps also the transfer functions. To make the paper self-contained, this section will briefly revise a popular mechanism for weight induction (the architectural issues will be discussed in subsequent sections).

The weight-induction process consists of two steps: intialization and learning. *Initialization* assigns to each weight a small random number.[1] For a given input **x**, each set of weights entails a certain value of the mean square error at the output. Figure 3 shows an error function for the extremely simplified case of a network with only two weights. Note that the weights are mapped to a single error value. The error function of a network with k weights is expressed as $e : R^k \rightarrow S$, where $S \subseteq R$. Further on, note the many "valleys" that correspond to *local minima* of the error function. The lowest minimum is called the *global minimum*.

Modification of the weights usually affects error. The essence of *learning* (or *training*) in multilayer perceptrons is to modify the weights in a direction that promises lower error for most of

[1]More sophisticated techniques for weight intialization are discussed by Nguyen & Widrow (1990) and Russo (1991).

Table 1 The principle of the *backpropagation-of-error* algorithm for a neural network with a single hidden layer

The Algorithm

1. Select a training example **x**, present it to the input neural layer, and propagate the signals through the network (*forward pass*).
2. Let **y** denote the observed output vector.
3. Let **t** denote the desired (target) output vector.
4. For each neuron, calculate its share of the overall error ("backpropagate" it from the output to the input). Upper layer: δ_i; lower layer: δ_j.
5. Make the weight corrections (upper: w_{ij}; lower: w_{jk}) and return to step 1.

Share of error:	**Weight correction:**
$\delta_i = y_i(1 - y_i)(t_i - y_i)$	$w_{ij}^{(1)} := w_{ij}^{(1)} + \eta\delta_i h_j$
$\delta_j = h_j(1 - h_j)\Sigma_i\delta_i w_{ij}$	$w_{jk}^{(2)} := w_{jk}^{(2)} + \eta\delta_j x_k$

the training examples. Ideally, this should result in finding the weights corresponding to the global minimum of the error function.

Perhaps the most famous training technique is the "backpropation of error" (Rumelhart & McClelland, 1986) which falls into the broader category of *gradient-descent* methods that seek to change the weight vector in a direction of the steepest decrease of the error function.[2] The algorithm rests on the following conjecture. Each neuron has a certain degree of "responsibility" for the overall error—a *share* of the error. Learning should modify more strongly the links from neurons with higher shares of error than the links emanating from neurons with smaller shares. Suppose that each neuron uses the `sigmoid` transfer function defined by equation (1). The formulae to calculate each neuron's *share* of the overall error, as well as the formulae for weight corrections, are summarized in Table 1. Their derivations can be found in any textbook on neural networks (e.g. Haykin, 1994). Different transfer functions imply different weight correction formulae.

The first formula in Table 1, $\delta_i = y_i(1 - y_i)(t_i - y_i)$, calculates the share of error for the ith output neuron. Note that the value, and also the sign, of δ_i primarily depends upon the difference between the real value and the desired value at the ith output: $t_i - y_i$. This difference is multiplied by the expression $y_i(1 - y_i)$ that reaches its maximum for $y_i = 0.5$, and converges to zero whenever y_i approaches either 1 or 0. The second formula, $\delta_j = h_j(1 - h_j)\,\Sigma_i\delta_i w_{ij}$, calculates the share of error for the jth hidden neuron whose output is denoted by h_j. The expression $\Sigma_i\delta_i w_{ij}$ is a weighted sum of the shares of error that have been calculated for the output neurons. The way these shares are "backpropagated" from the output to the input has given this technique its name. Generalization of this procedure for two or more hidden layers does not pose any serious problems.

Once the error shares have been determined, the weights of the links are corrected according to the corresponding formulae in Table 1. Whether a given weight is increased or decreased is decided by the sign of the corresponding δ_i or δ_j. The extent of the weight correction is also controlled by the *learning rate*, $\eta \in (0, 1)$, and by the strength of the signal arriving along the corresponding input link.

Importantly, backpropagation of error is very expensive. Upon the presentation of an example, the error share of each individual neuron has to be calculated and all weights have to be modified. An *epoch* is the time necessary to run through all training examples. Usually, many epochs are necessary before the error rate begins to converge to its minimum—sometimes even many thousands of epochs. To get an idea of the computational requirements, consider the case where 1000 weights have to be trained on 100,000 examples. If the algorithm is to run through 10,000 epochs, then 10^{12} weight updates have to be carried out.

[2]For more advanced variations of this technique, see the excellent survey by Sarkar (1995).

4 Ideal size of a multilayer perceptron

Having outlined a mechanism for weight training, let us now turn our attention to the problem how to design the network's layout and how to determine its *size*. A decade ago, experienced practitioners customarily relied on their ability to estimate the appropriate size based on their personal assessment of the complexity of the given problem. They considered such aspects as the number of attributes, the number of class labels, the size of the training set, the extent of noise, and previous experience with similar problems. When intuition failed, they simply experimented with several different options and then selected the best one.

Such subjective approach is hardly satisfactory in advanced pattern recognition where complexity is hard to predict. Serious reservations have to raised against the arbitrariness of the trial-and-error experiments that are far from systematic, and can easily incur high computational and programming costs. What is needed are techniques that would automate the process. To this end, the researcher needs to understand how the network's architecture affects classification behavior.

The most obvious weakness of the backpropagation algorithm is its hill-climbing nature that at each training step follows the direction of the steepest slope of the error function. Gradient-descent techniques can easily end up in a local minimum. Improved versions of the backpropagation algorithm have therefore been studied. For instance, a small modification of the weight-correcting formula can cause the descent into a "valley" to incur momentum that will help the learner escape from a shallow minimum. Alternatively, the learner can be permitted to make exploratory "jumps" into other regions. However, none of these improvements is a panacea.

A lot depends upon the classifier's size. The great number of free parameters in very large networks can do away with local minima. But then large networks are expensive to train, and their excessive flexibility makes them prone to *overfit* noisy training sets. Conversely, small networks are cheap to train and less vulnerable to overfitting, but their limited flexibility exposes them to local minima. While both extremes (very large versus small networks) are harmful, a compromise is necessary. A simple experiment will clarify the point. Given a classification task, and a noisy training set, suppose that a researcher experiments with several networks, each with a different size of hidden layer. For simplicity, suppose the adjacent layers are fully interconnected. Training with the gradient-descent algorithm continues until the error on the training set levels off. Each network is then tested on independent examples that were withheld from the training session.

The graph in Figure 4 illustrates a typical observation: as long as the hidden layer is small, the error rate on the testing set is high. With a larger hidden layer, the error will decrease, but only to a certain point where the trend reverses and the error starts growing again. The curve is idealized. In reality, statistical fluctuations will distort its smooth shape, and the growth of the error rate for large networks will be much less pronounced if the amount of noise in the data is moderate. Still, the general tendency will largely follow the depicted pattern. Small networks are hurt by local minima, large ones by overfitting.

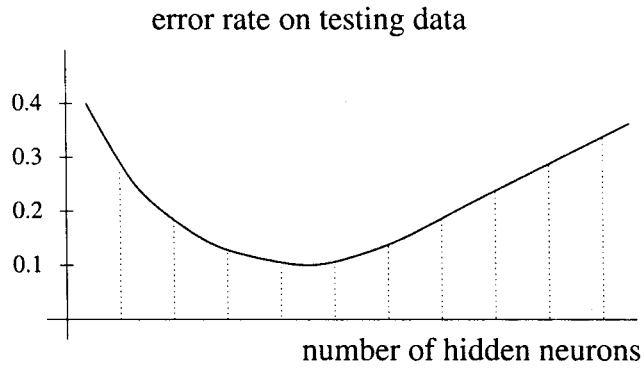

Figure 4 Small networks lack sufficient flexibility; large networks are expensive to train and are prone to overfit noisy training data. In either event, the error rate on testing examples will be high

The experiment suggests that the designer should avoid very small networks as well as very large networks. But what *is* a small network? Does it have ten hidden neurons or 200? Does a network consisting of 3000 neurons provide just the right flexibility or is it already too large? Similar questions cannot be isolated from the concrete task. In one domain, the minimum error will be achieved with five hidden neurons; some other domain will call for fifty. The designer needs techniques that will tailor the network's size, and perhaps even its general layout, to the given domain. The next three sections will describe three principal strategies to determine a reasonable neural architecture.

5 Classicism of architecture learning

One way to determine an "appropriate size" for a given task is to let the network *grow* during the learning process. At the beginning, a multilayer perceptron with no more than just two or three hidden neurons is trained. Such a small network is likely soon to end up in a local minimum, where further training does not bring about any improvement. If the error is still considerably high, the learning program adds to the hidden layer a new neuron. The number of trainable parameters thus increases, and the enlarged set of weights will entail a different error function. As a result, the network now finds itself at some neutral point of this function, most likely not in a local minimum. If the weights of the newly added links are small, the classification behavior of the network may remain virtually unchanged, while gaining the potential of further improvement with training.

The algorithm can be summarized as follows. First, train the network until the mean square error begins to level off. Then, add a new neuron (with small random weights) and resume training. Repeat these two steps as long as the new neurons keep improving the classifier's behavior. The resulting size of the hidden layer will not be far from the point where the curve from Figure 4 reaches bottom. A technique building on this principle was first suggested by Lee et al. (1990).

A fundamental limitation of this family of algorithms is *overfitting*. This can usually be mitigated by a "hold-out" technique, where a subset of the training examples is withheld from training, and is used only for occasional check-ups: if the reduction of the training-set error is accompanied by an increased error on the withheld subset, then the network probably overfits the training examples.

Instead of *growing*, some researchers suggested the opposite approach: *trimming*. Relevant algorithms were studied by Mozer & Smolensky (1989) and Le Cun et al. (1990). The principle is as follows. Train a large network until a very small error on the training set is achieved. Using an appropriate heuristic, select a neuron that has only moderate impact on the network's behavior. Check whether the network's mean square error on a withheld subset of examples remains about the same (or even drops) and re-start training. Repeat these two steps (network training and neuron removal) as long as the demise of a neuron does not increase the error on the withheld subset. Various heuristics for the selection of the neuron to be removed have been suggested. For example, if the variance of the weights of the links emanating from a neuron is small, then this neuron only weakly contributes to discrimination between classes, and can thus be deleted.

The trimming methodology works well in simple domains but is prone to fail in really difficult tasks such as recognition of complex objects in digital images. The reason is that the designer does not really know what a "large network" is. Starting from millions of neurons is unrealistic; a smaller initial network, say, 200 neurons, may be insufficient.

The fact that each of these two approaches (growth and trimming) suffers from idiosyncratic weaknesses motivates attempts to amalgamate them using search techniques from artificial intelligence. The basic procedure alternates between two steps: (1) gradient-descent training; and (2) architecture update. The latter (Figure 5) employs various heuristics to decide whether to add or remove a neuron, or rather a group of neurons, or even whether to create or destroy an entire hidden layer. The idea that a network's layout can be determined by a heuristic search led to experimentation with the popular *genetic algorithms* that are known for their potential for search in very large spaces. A more detailed discussion of these methods exceeds the scope of this paper. The interested reader is referred to the collection of papers edited by Whitley & Schaffer (1992).

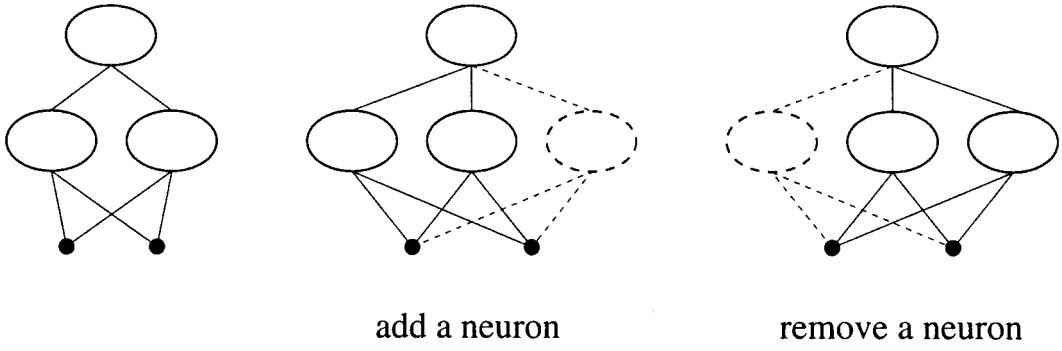

Figure 5 When the backpropagation algorithm does not seem to bring about any improvement, a *search mechanism* interrupts training, and decides whether to add/remove a hidden neuron. Then, training is re-started. In general, a group of neurons, or even an entire hidden layer, can be added or removed

One does not have to remove neurons to curtail the number of trainable parameters. Another possibility is to destroy some of the links (by assigning them zero weights) while still keeping all neurons in place. The network thus becomes *partially interconnected*. One of the earliest methods to reduce the number of links is mentioned by Hinton (1990). In his approach, the gradient-descent algorithm is modified in a way that increases the variance in the weights. Most links then fall into one of two categories: those with large weights (significant impact on the network's behavior); and those that are very small (negligible impact). Small weights are then removed without much detriment to the network's behavior. Alternative mechanisms for link elimination are discussed by Weigend et al. (1991).

Despite the fact that fairly high classification accuracy can be achieved, search-based approaches to the design of a neural network's architecture are rarely employed in real-world applications: since the expensive gradient-descent procedure has to be repeated for each interim architecture, computational costs are often prohibitive.

The next two sections discuss more sophisticated methods that achieve good classification results at significantly reduced learning costs. The first group develops the network piecemeal (one neuron at a time), focusing each neuron on a different aspect of the given task. The second approach initializes the network using machine learning techniques that are faster (albeit coarser) than neural-network training.

6 Networks built from specialized neurons

The gradient-descent algorithm considers at each step only a neuron's share of overall error, while ignoring the status of the other neurons at the same layer. Suppose that, upon the presentation of a training example, each neuron in a hidden layer has the same δ_j. Their weight vectors will then be corrected in the same direction. This is far from ideal. The reader will agree that the "members" of the hidden layer should rather behave as a good team so that imperfections of individuals can be corrected by others. However, the team behavior can hardly be achieved by the backpropagation algorithm, a technique that does not assume any communication among neurons in the same layer. If each neuron's weights are corrected irrespective of what happens to the rest of the layer, then the neurons are unlikely to focus on mutually complementary subtasks.

The techniques from the previous section *do* to some degree satisfy the requirement of neural specialization. For instance, newly added neurons rectify some imperfections of their predecessors. But the price in terms of computation costs is high because the expensive backpropagation algorithm has to be repeated many times. Some researchers therefore explore a different approach. Whereas the search-based algorithms re-train the entire network after each change in the architecture, the techniques described below update only the weights of the newly added neuron. This has two important consequences. First, the computational costs of learning are significantly

reduced. Secondly, when a new neuron is added, the behavior of the rest of the network remains intact—the results of previous training are not erased and specialization is encouraged.

To keep the exposition simple, this section will assume that the neurons have the `hardlim` transfer function. Let x_i denote the input attributes and let w_{ij} denote the weights of the jth neuron. If $\Sigma_{i=0}^{n} w_i x_i > 0$, the neuron's output is 1; otherwise, the output is 0. This does not necessarily incur a great loss in generality. After the classifier has been induced, there is always the possibility to replace the `hardlim` transfer function with the more flexible `sigmoid` function, and to fine-tune the network with the backpropagation algorithm. Further on, this section will deliberately narrow the domain of interest to two-class domains, where the classifier is to separate positive examples from negative examples. The output layer can therefore consist of a single neuron.

Two important ideas will be discussed (interestingly, both appeared at about the same time). Frean's (1990) Upstart algorithm grows the network by adding new neurons that are trained on the original, though re-labeled, examples. In contrast, Schapire's (1990) boosting and its descendants train each new neuron on a different subset of the training data.

6.1 Frean's Upstart

Frean's Upstart trains only one neuron at a time. Obviously, to train a single neuron is much faster than to train an entire network. One can use the perceptron-learning algorithm (Rosenblatt, 1958), the delta rule, closely related to the gradient-descent learning (Widrow & Hoff, 1960), or the statistical methods that use pseudo-inverse matrices or other regression techniques (Duda & Hart, 1973).

At the beginning, Upstart attempts to solve the given task with a neuron denoted by **z**. As the representation powers of a single neuron are limited, **z** will misclassify many training examples. Two types of error can be distinguished. The first, *wrongly ON*, occurs when **z**'s output is 1 for a negative example. The second, *wrongly OFF*, occurs when **z**'s output is 0 for a positive example.

Upstart seeks to reduce the frequency of these errors by appropriate corrections of **z**'s input. This is carried out by two auxiliary neurons (see Figure 6). The task of **x** is to reduce **z**'s input for wrongly ONs, and the task of **y** is to increase **z**'s input for wrongly OFFs. The output of **x** should be 1 for those (and only those) examples for which **z** is wrongly ON and **y** should output 1 for those (and only those) examples for which **z** is wrongly OFF. The link between **x** and **z** has a large *negative* weight that reduces **z**'s input; the link between **y** and **z** has a large *positive* weight, increasing **z**'s input.

The auxiliary neurons, **x** and **y**, are trained on the same training set as **z**, but now the examples are *re-labeled*. For **x**'s training, all examples for which **z** was wrongly ON are labeled with 1 and the other examples are labeled with 0. For **y**'s training, all examples for which **z** was wrongly OFF are labeled with 1 and the other examples are labeled with 0. Suppose that **z** outputs 1 for a negative example. For **x**'s training, this example will be labeled with 1; for **y**'s training, it will be labeled with 0.

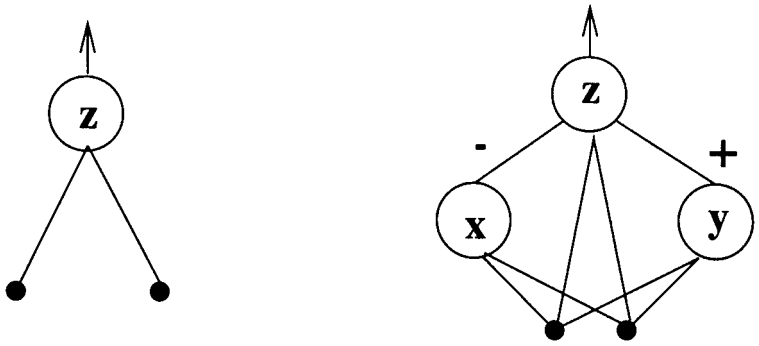

Figure 6 Upstart rectifies the behavior of **z** by two auxiliary neurons: **x**, with a large negative weight, should correct "wrongly ON" states of **z**; **y**, with a large positive weight, should correct "wrongly OFF" states of **z**

Table 2 The recursive principle of Frean's Upstart algorithm

Input: neuron **z**, trained to minimize the error on the training set

Upstart(**z**).
1. Observe **z**'s behavior on the training examples.
2. Train a new neuron, **x**, to output 1 when **z** is wrongly ON and 0 otherwise. Connect **x** to **z** with a large *negative* weight.
3. Train a new neuron, **y**, to output 1 when **z** is wrongly OFF and 0 otherwise. Connect **y** to **z** with a large *positive* weight.
4. Unless a stopping criterion is met, call *Upstart*(**x**) and *Upstart*(**y**).

Alternatively, if **z** misclassifies a positive example (assigning it 0 instead of 1), then this example will be labeled with 0 during the training of **x** and with 1 during the training of **y**. Any example classified by **z** will always be labeled with 0.

Suppose that **x** and **y** classify all re-labeled training examples as required. With sufficiently high weights assigned to the links leading from these two neurons to **z**, the network will correctly classify all training examples. However, with the `hardlim` transfer functions, **x** and **y** will hardly behave so nicely—they will most likely fail to classify many of the re-labeled examples. For this reason, Upstart attempts to correct their behaviour by recursive calls of the same routine, providing a pair of auxiliary neurons for **x** and another pair for **y**. The process continues until some reasonable termination criterion is met. This time, the re-labeling will reflect the wrongly ONs and wrongly OFFs of the lower-level neurons. Upstart keeps expanding the network in the hope that, sooner or later, all training examples will be classified correctly. The recursive principle is described in Table 2, and some of the intermediate steps are illustrated by Figure 7. In reality, the resulting network can be much larger than the one shown in the picture.

A designer intending to employ this algorithm in a real-world application should keep in mind that **x** and **y** cannot actually be guaranteed to improve **z**'s performance. Consider the situation, where **z** is wrongly ON only on one training example while correctly classifying all the remaining

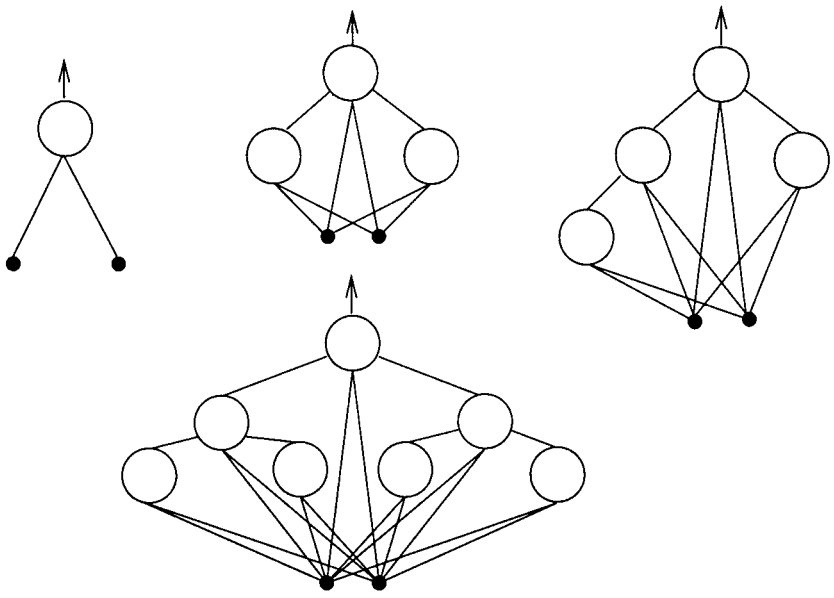

Figure 7 The network grows by recursive calls of Upstart, in an attempt to gradually decrease the error on the training set. The stopping criterion has to prevent overfitting as well as uncontrolled growth of the network and oscillation of the error rate

training set. Denote the misclassified example by e. According to Upstart, **x** should be trained to output 1 for e and 0 for all other examples. However, this behavior on the part of **x** cannot in many geometric domains be achieved by a `hardlim` neuron. The price for correct classification of e will be that many examples that were originally correctly labeled **z** will now be misclassified and the error rate observed at **z**'s output will increase.

The auxiliary neurons can improve the network's performance only if they do not fail on examples that have already been correctly classified by **z**. Under less favorable conditions, the error rate on the training set can oscillate: the newly added neurons improve the network's behavior in some parts of the instance space while degrading it in other parts. The danger of oscillation could perhaps be diminished if the corrections of **z**'s inputs were carried out by some more powerful classifiers than just simple neurons.[3] An important issue in any realistic implementation of Upstart is therefore how to control the network's growth. Even if the danger of overfitting is curtailed by the use of a withheld training subset, the major concern with oscillation persists and the stopping criterion has to be defined very carefully. For instance, the learning system should always check whether adding a new neuron actually reduces error rate.

Frean's Upstart was discussed here mainly for its instructiveness. The reader should be aware that other solutions exist. At about the same time when Upstart was published, another popular and frequently cited "piecemeal" approach to the design of neural networks was developed by Fahlman & Lebiere (1990) under the name of *cascade correlation*. The principle is to add one hidden neuron at a time, and to train its weights in a manner that maximizes the correlation between this neuron's output and the error function of the entire network. Subtraction of the two signals then improves the network's classification behavior. The interested reader is referred to the original paper.

6.2 Schapire's boosting by filtering

Whereas Upstart trains the newly added neurons on the entire training set (even if re-labeled), the *boosting* algorithms described in the sequel train each new neuron on a *different subset* of the training examples. A ground-breaking event was the paper by Schapire (1990). He proved that a powerful classifier can be constructed as an assembly of weaker subclassifiers, provided that each of them was induced from a carefully constructed subset of training examples. This section will present the boosting algorithm from the perspective of the problems of neural architectures.

Schapire carefully filters the examples to be used to train a neuron. Again, the exposition will for simplicity be limited to two-class domains. Another important assumption will request that the positive and negative examples be equally represented. Finally, the focus will be on networks with `hardlim` transfer functions where a neuron's output is either 1 or -1. The idea is easy to extend to more difficult settings.

As before, the discussion will assume the existence of a simple algorithm, called a *learner*, to train a single neuron.[4] To start with, the boosting algorithm selects a subset of the training examples, T_1, and induces from them the first neuron, L_1. If the error rate of this first neuron on the entire training set is higher than 50%, the process is repeated until less than 50% is achieved. In the next step, boosting will choose a second subset of the same size, denoted by T_2. Importantly, T_2 has to satisfy the requirement that the previous neuron, L_1, should totally fail on these examples, classifying correctly 50% and misclassifying the rest. From T_2, the learner will induce neuron L_2 that addresses those aspects of the given task that were hard for L_1. To break ties between L_1 and L_2, boosting creates a third subset, T_3, consisting of examples on which L_1 and L_2 disagree. From T_3, the learner induces the third neuron, L_3.

This simple mechanism, summarized in Table 3, creates a multilayer perceptron with three hidden

[3]More detailed analysis of this issue would deserve a research paper.

[4]Note that since we assume only two-class domains, *any* learner can induce a classifier with error rate (on the training set) below 50%. If the error rate is higher, then the induced classifier can just be reverted, assigning the positive class instead of the negative and vice versa.

Table 3 The use of Schapire's boosting. A multilayer perceptron with three hidden neutrons is trained in a two-class domain

1. Choose m random examples from which the learner induces the first neuron. Let this neuron be denoted by L_1. If the accuracy on the training set is less than 50%, discard the neuron and repeat.
2. Choose another set of m examples, selected in a manner that guarantees that the previous neuron, L_1, classifies them with 50% accuracy. From these examples, the learner induces neuron L_2.
3. Choose m examples on which the neurons L_1 and L_2 disagree. The learner induces from them neuron L_3.
4. Create a multilayer perceptron where L_1, L_2 and L_3 are hidden neurons, and where all weights from the hidden neurons to the output neuron are equal to 1.

neurons (in a single hidden layer) and one output neuron. Only the hidden neurons have weighted inputs; links from the hidden layer to the output layer are all equal (e.g., $w_i = 1$). As each hidden neuron has the same weight, the final decision is reached by voting. For instance, if two hidden neurons output 1 and only one outputs -1, the given example is deemed positive because the sum of all "votes" is $1 + 1 + (-1) = 1$ which is a positive number.

Errors of individual neurons can often be corrected (outvoted) by the remaining neurons, simply because each of them will tend to err in a different part of the instance space.[5] Formal analysis has shown that this is indeed the case in error-free domains. Suppose that each neuron has its error rate upper bounded by some $\varepsilon < 0.5$. The error rate of the network induced by the mechanism from Table 3 is upper bounded by $3\varepsilon^2 - 2\varepsilon^3$. This expression is smaller than ε for any $\varepsilon < 0.5$. For instance, if the error rates of the neurons are all less than $\varepsilon = 0.3$, then the error of the resulting network will be upper bounded by $3 \times 0.3^2 - 2 \times 0.3^3 = 0.216$. If error rates of the individual neurons are all less than $\varepsilon = 0.2$, then the error or the resulting network will be upper bounded by 0.104.

The upper bound holds for any subclassifiers induced from training subsets selected as indicated in Table 3. This means that the error can further be reduced by recursion when a triplet of neurons, (L_1, L_2, L_3), is understood as a single subclassifier, whereas the second subclassifier will consist of another triplet, (L_4, L_5, L_6), and the third subclassifier will be yet another triplet, (L_7, L_8, L_9). The network will then have nine neurons organized as shown in Figure 8. Note that adjacent layers are only partially interconnected and that this network has only one layer of trainable weights (all other weights are set to 1). The formula for error rates still holds. For instance, if each individual neuron in the lowest layer has error rate below 0.3, then the error of each triplet will fall below 0.216, and the overall error of the network from Figure 8 will not exceed 0.112. Note that, with recursion of depth N, the number of neurons with trainable weights is 3^N.

The recursion can of course be continued *ad infinitum*, creating larger and larger networks with smaller and smaller error rates. Theoretically speaking, arbitrarily low error rates on the training set can be achieved be these recursive calls. To make the idea even more attractive, practical experiments show that this algorithm only rarely overfits training data.

The main difficulty with this technique is that it asks the system to choose for each subclassifier m training examples that satisfy specific conditions. In realistic domains, such examples are difficult to find for recursion depths beyond 2. The reason is that the original training set would have to be very large and *dense* to satisfy the conditions. The search for the subsets in such vast collections of data can become prohibitively expensive. Therefore, higher levels of recursion are rarely used. Schapire's boosting by filtering is only employed for networks of moderate sizes. This does not in the least diminish its importance.

[5]The observation that a simple learner is "boosted" by being repeatedly run on different examples gave the paradigm its name.

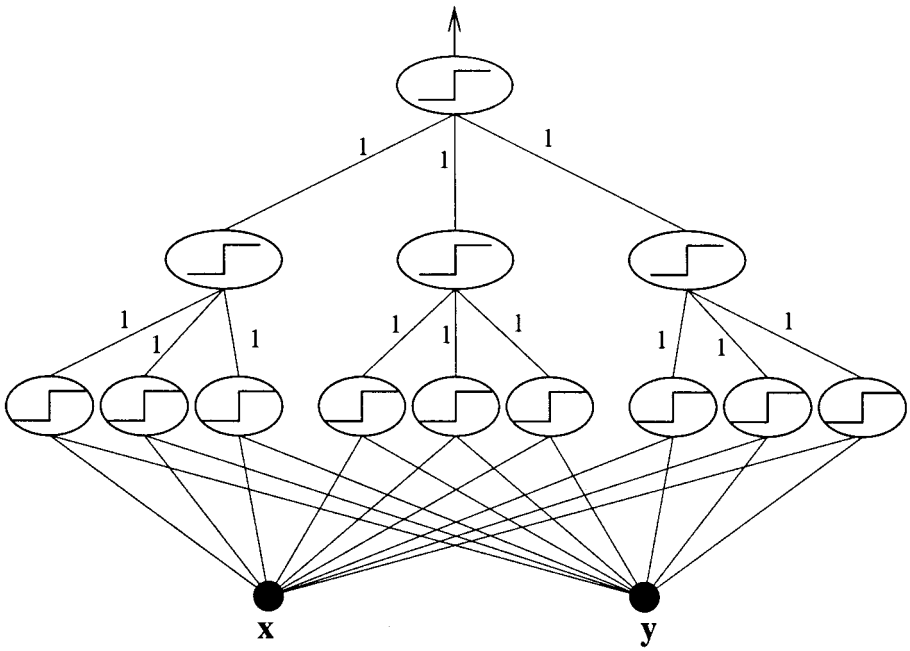

Figure 8 In neural networks created by recursive calls of Schapire's boosting, only the lowest layer of weights is trained. All the other weights are set to 1. This picture shows only two levels of recursion. Each neuron outputs 1 if the weighted sum of its inputs is positive and − 1 otherwise

6.3 Boosting by sampling

Although Schapire's boosting aroused much interest, it was also clear from the beginning that modifications were needed. For instance, Freund (1990, 1995) analyzed an algorithm that induces *many* subclassifiers. Whereas in Schapire's mechanism, each neuron addresses a different aspect of the classification task, in Freund's approach, each of the subsequently induced neurons is just "somewhat" different from its predecessors. To classify an example, the neurons undergo *weighted majority voting*. To construct the training subsets, the filtering principle is replaced with probabilistic *sampling*. A few years later, the two scientists joined forces and developed an even better version (Freund & Schapire, 1996, 1997) and gave it the name AdaBoost (*ada*ptive *boost*ing).

This last technique will, again, be discussed only from the perspective of its potential for the design of a multilayer perceptron (this time with a single hidden layer). The neurons will have the `hardlim` transfer functions that output 1 or − 1. The explanation will be limited to two-class problems with evenly distributed positive and negative examples. Only the basic principles will be presented. Readers interested in a more detailed exposition of AdaBoost, and its advanced versions, are referred to the original papers by Freund and Schapire.

Figure 9 shows the layout of the induced network. At step i, the algorithm induces neuron L_i from a sample subset T_i. The samples are drawn randomly, with replacement, from the original training set, according to a probabilistic distribution that is modified after each step (see below). The size of each T_i is equal to the size of the original training set—some training examples can appear in T_i more than once, whereas other examples many not be drawn at all.

The probabilistic distribution changes after each step so that previously misclassifed examples get a higher chance of being drawn in the future. For the induction of the first neuron, L_1, each training example has the same chance of being included in T_1: the probability of the jth example will be $p_j = 1/m$, where m is the size of the training set. The next step will focus rather on those examples that have been misclassified by L_1 because misclassified examples are likely to represent aspects that were poorly represented in T_1. The probability of choosing a correctly classified example should therefore be decreased, for instance by setting $p_j := p_j \cdot \beta$, where $\beta \in (0, 1)$, for all examples correctly classified by L_1 (and leaving $p_j := p_j$ for those examples that were misclassified by L_1). The

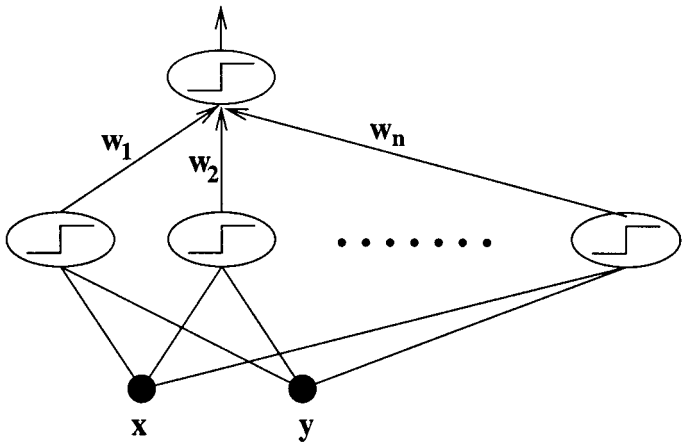

Figure 9 Neural networks created with AdaBoost have one hidden layer of neurons. All links between adjacent layers are weighted

probabilities are then re-normalized so that they sum up to 1. This new probability distribution, T_2, is used for the random choice of examples to be included in the second subset, T_2. From T_2, the second neuron (L_2) is induced and the distribution is again modified so as to decrease the probability of examples correctly recognized by L_2. The process continues until some reasonable termination condition has been met. The principle is summarized in Table 4.

The weights of the output neuron can in principle be induced using a transformed training set where each example is described by the binary outputs of the hidden neurons. For the needs of mathematical analysis, Freund & Schapire (1997) calculate these weights by a formula that guarantees favorable mathematical properties (the details are unimportant for this survey). They also explain how precisely to modify the probability distributions, and how to generalize the algorithm for domains with multiple classes.

In AdaBoost, the number of neurons is usually much higher than in filtering-based boosting. Still, the training costs are reasonable, because only a single set of weights is being trained at each step. The resulting networks do not tend to overfit the training data, so there is no need for periodic checks of the error rate on withheld examples. An important issue is how many subclassifiers are needed for some pre-specified level of accuracy (how to terminate the algorithm). To answer this question, Freund and Schapire show how the error rate depends upon the number of classifiers.

The main feature that distinguishes the resulting network from a network obtained using Schapire's original idea is that AdaBoost induces *many* neurons in a *single* hidden layer. Each of them is supposed to be just slightly different from the others. With this in mind, one can ask whether it is all that necessary to modify the probabilistic distributions. What if the examples are drawn uniformly for each T_i (here t_i is a subset of the original training set)?

To answer this question, Breiman (1996) explored a technique that he gave the name *bagging* (*b*ootstrap *agg*regat*ing*). Bagging creates many random subset using the same uniform distribution. To simplify things even further, he sets all weights to 1 (plain voting instead of the weighted-majority voting in AdaBoost). Breiman was primarily interested in applying his bagging to more

Table 4 The basic principle underlying the algorithm of AdaBoost

1. Let $i := 1$. Define the initial probabilistic distribution, D_i, by setting for each training example, $\mathbf{x}$, the same probability, $p(\mathbf{x}) = 1/m$, where m is the size of the training set.
2. Select the set T_i of m training examples randomly drawn (with replacement) according to distribution D_i. Induce from them neuron L_i.
3. Create a new distribution D_{i+1} as a modification of D_i by decreasing the probability of those examples from T_i that were correctly classified by L_i.
4. Unless a termination criterion is satisfied, set $i := i + 1$ and go to 2.

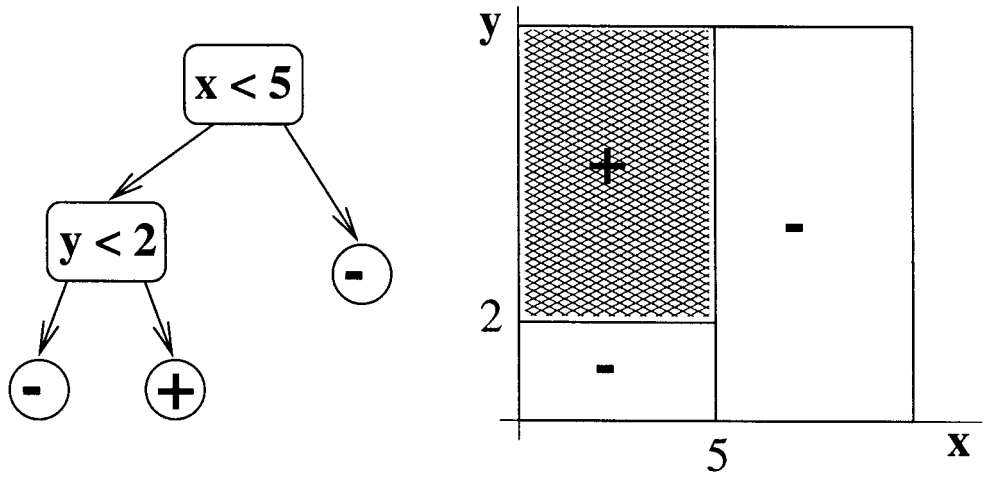

Figure 10 A decision tree is a partially ordered set of tests that partition the instance space into rectangular (or hyperrectangular regions)

sophisticated classifiers (rather than just neurons), and he was able to demonstrate some useful properties. The principle can still be used for the construction of multilayer perceptrons, although perhaps slightly less successfully than AdaBoost.

7 Decision-tree based strategies

In the techniques from section 5, the main problem was how to determine the initial size of a neural network from which further growth or trimming can start. This section shows a simple technique that determines such size using induction of decision trees. The technique is not guaranteed to return the best possible network; however, it arguably creates a network with low error rate and with the potential for further improvement at reasonable computational costs.

A decision tree is a partially ordered set of tests, usually of the form $x_i < \theta_j$, where x_i is an attribute and θ_j is a threshold.[6] Each branch of the tree is terminated by a *leaf* that contains a class label. Two or more leaves can contain the same label. To classify a numeric vector **x**, the system subjects **x** to the test in the *root*. A positive outcome decides that **x** should be propagated along the left branch, and a negative outcome sends it down the right branch. This continues until a leaf is reached. The class label associated with this leaf is assigned to **x**.

Note that the tests along a single branch are conjuncted: to reach a leaf, all tests along a branch have to be satisfied. For instance, if **x** is to end up in the positively labeled leaf of the tree in Figure 10, then the attribute x must be less than 5 and the attribute y must *not* be less than 2. The tree divides the n-dimensional space in hyperrectangular regions. For instance, the root test in Figure 10 splits the original instance space in two parts, separated by the line $x = 5$. The next test further divides one of these parts by the line $y = 2$. The decision surface is piecewise linear, each of its segments running parallel to one of the coordinates.

Induction of decision trees received significant attention in the 1980s, and the research continues to this day. Powerful software packages are in common use (e.g., Quinlan, 1993) and the behavior of the attendant learning techniques is well understood. Classification performance of decision trees is favorable and their induction is computationally effective—usually by orders of magnitude faster than the gradient-descent algorithm. To induce a decision tree in a difficult domain that has hundreds of thousands of examples can take just a few minutes. Training a neural network to achieve comparable error rate can take several days.

How do decision trees compare with neural networks in terms of classification accuracy? The

[6]Recall that this survey concentrates on applications with numeric attributes.

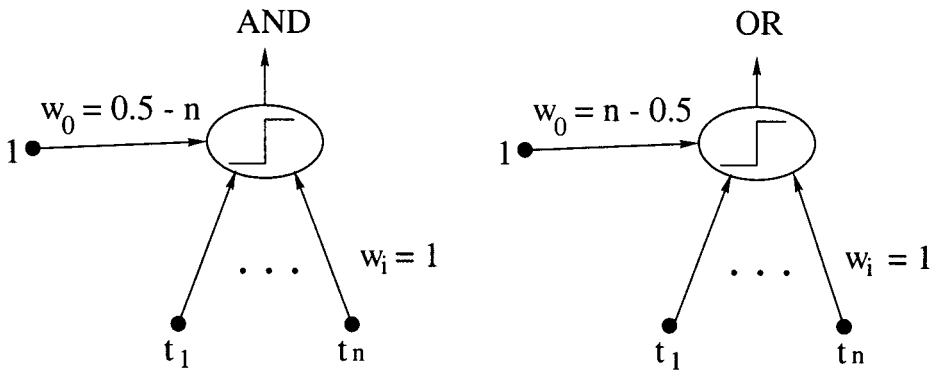

Figure 11 With a careful choice of w_0, a neuron can be made to carry out logical conjuctions or logical disjunctions. Here, the inputs have the value 1 for *true* and -1 for *false*; all weights except for w_0 are set to 1

large numbers of trainable parameters, and the nonlinearity of the `sigmoid` transfer function, guarantee high representational power of neural networks. Indeed, ample experimental evidence attests that large networks can implement more precise classifiers than decision trees. Whereas Weiss & Kapouleas (1989) observed that decision trees outperformed neural networks in simple applications, Fisher & McKusick (1989) and Fisher et al. (1989), who experimented with more difficult domains, were able to show that neural networks outstripped decision trees whenever provided with large training sets and sufficient training time. Atlas et al. (1990) reported that in three realistic domains, neural networks clearly outperformed decision trees, and similar observations were made also by Ivanova & Kubat (1995).

While decision trees are cheaper to induce, neural networks are able to reach lower error rates in hard tasks. Can these two advantages somehow be combined in a single system? To answer this question in the affirmative, we need to show that a decision tree can be mapped to a neural network with the same classification behavior. Then, one can expect that the performance of this network can further be "fine-tuned" by training with the backpropagation algorithm, and perhaps by some trimming of the resulting architecture.

The first step is to demonstrate that neurons can implement simple boolean functions. The left part of Figure 11 shows how conjunctions can be implemented. Here, a neuron with the `hardlim` transfer function accepts the signals from the lower layer and assigns to each of them the same weight, $w_i = 1$. The signals are -1 for *false* and 1 for *true*. The picture also contains one additional link, w_0, connected to a fixed input whose value is always 1. If this weight is set to $w_0 = 0.5 - n$, then the weighted sum of all inputs can be positive only if all inputs are 1. Only then will their combined contribution outbalance the signal arriving along the 0th link: $n + (0.5 - n) = 0.5 > 0$.

The right part of Figure 11 shows how to implement disjunction by setting the weight of the additional link to $w_0 = n - 0.5$. The output of the OR-neuron is 1 as long as at least one of its inputs is 1. If all other inputs are -1, then their combined contribution is $-(n-1) = (1-n)$. The contribution of the additional link is 1, and the total input of the neuron will therefore amount to $1 + (1-n) + (n-0.5) = 1.5 > 0$. Only if all inputs are -1 will the total input be $-n + (n-0.5) = -0.5 < 0$.

When mapping a decision tree to a neural network, the tests in the internal nodes can be carried out by neurons with `hardlim` transfer functions. Each of the tests will be represented by two neurons: one for the positive output (leading to the left branch); and one for the negative output (leading to the right branch). For instance, the root test of the tree from Figure 10 will require one neuron to carry out the test $x < 5$ and another neuron to test for $x \geq 5$. If the neuron's test is satisfied, then the output is 1, otherwise it is -1. For the attribute-value test $x < \theta$, the neuron needs only one input link (with weight 1) and one link from the fixed input signal of 1 (with weight $-\theta$).

A decision tree can therefore be mapped to a multilayer perceptron with two hidden layers. The lower hidden layer carries out the tests from the internal nodes; the higher hidden layer conjuncts the tests that are on the same tree branch; and the output layer carries out the disjunctions of those

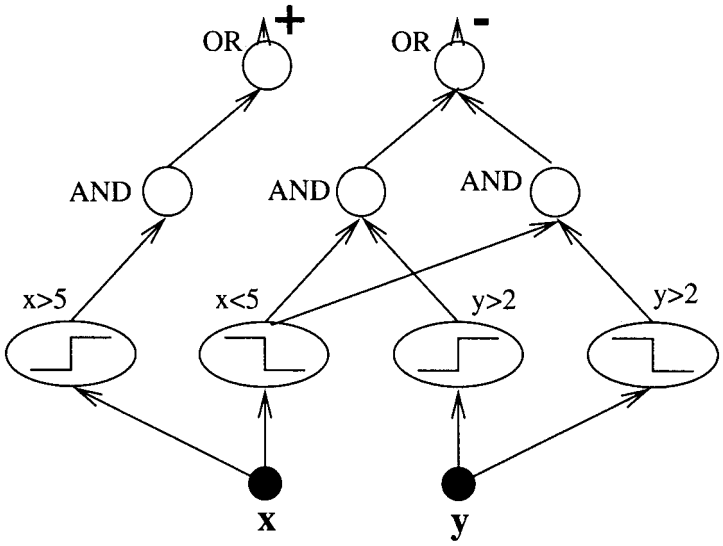

Figure 12 A decision tree can be mapped to a mutlilayer perceptron with two hidden layers. The lowest layer carries out the test from the decision tree. The next layer carries out conjunctions and the output layer carries out disjunctions

branches whose leaves have the same class label. Figure 12 shows a neural network that emulates the behavior of the decision tree from Figure 10. For simplicity, the additional links (from a fixed input of 1) whose weights help to implement conjunctions and disjunctions have been omitted.

Once the neural network has been created, further fine-tuning of its behavior by gradient-descent training and trimming can begin. An important prerequisite is that the `hardlim` transfer functions be changed to `sigmoid` functions (this can incur certain increase in the error rate). The flexibility of this partially interconnected network can further be increased by additional links that will make the network fully interconnected.

The idea to map a decision tree to a neural network with subsequent tuning was first suggested by Sethi (1990).[7] Soon thereafter, variations on the basic principle were published by many authors, among them by Brent (1991), Park (1994), Ivanova & Kubat (1995), Sahami (1995) and Bioch, Carsouw & Potharst (1995). These systems differed in certain topological details, in the choice of the transfer functions (`hardlim` versus `sigmoid`), and in the interconnection of adjacent layers. For instance, Ivanova & Kubat (1995) show how to map a decision tree to a network with fully interconnected adjacent layers of neurons with `sigmoid` transfer functions. Readers interested in details are referred to the papers cited above.

At this point, it has to be mentioned that several researchers have studied methods to intialize neural networks using prior knowledge encoded as *production rules*. A production rule (an *if-then* rule) has the form "`if` $a_1 \wedge a_2 \wedge \ldots \wedge a_k$ `then` c_i", where a_j are tests and c_i is a class label. Initial knowledge that can be expressed in these rules is available in many realistic domains. For example, one such rule can say that "`if` tall_athlete `then` plays_basketball" (note that the rule does not necessarily be true for each single example). The tests a_j can be carried out by neurons as shown above.

The background knowledge can be incomplete in the sense that only some aspects of the given class are known. Also, some rules can be incorrect, and thus misleading, which makes the rule-based initialization of neural nets a somewhat different task. A decision tree that has been induced from training data usually has a high classification accuracy, whereas the accuracy of hand-coded rules is often questionable. If the rules are themselves induced from training data, then the rule-to-network mapping can follow the same principles as the decision-tree based initialization. Readers that would

[7]His algorithm was somewhat different than that described here.

like to learn more about the related issues are referred to the papers by Goodman (1992), Goodman et al. (1992), Towell, Shavlik & Noordevier (1990), Towell & Shavlik (1994) and Bala, Michalski & Pachowicz (1994), and to additional references therein.

8 Conclusion

In multilayer perceptrons, mere induction of weights is not enough. The success or failure of a pattern-recognition undertaking will also depend upon the architecture of the given network. Over the past decade, several families of algorithms addressing this particular issue have been suggested. This paper has discussed the reasons why the network's layout can have such a great impact on accuracy, and then presented representative examples of three major approaches: search-based techniques; piecemeal induction of specialized neurons; and the use of decision trees. Each of them leads to somewhat different networks.

Search-based techniques are inevitably expensive because they require that the costly back-propagation algorithm be run on several different architectures. The piecemeal approach (Upstart, boosting and AdaBoost) can be understood as an advanced version of network growing. Thanks to the specialization of the individual neurons, the methods usually generate very compact networks. However, some of them (especially Upstart) have problems with proper definition of termination criteria. Decision-tree based initialization does not find the best possible architecture (the created networks can sometimes be rather large) but it enables the backpropagation training to start from a point that is well beyond most of the local minima. Whether the technique prevents overfitting is not certain. However, the author believes that the resulting network can serve as a good starting point from which a more traditional search technique can begin.

To keep the text focused, many other useful techniques, and even some vital details, had to be passed over. For instance, the paper considered only networks with neurons that have `hardlim` and `sigmoid` transfer functions, leaving aside—among others—the popular Radial-Basis Function (RBF) networks where the transfer function is represented by the gaussian "bell" function (Broomhead & Lowe, 1988). Even here, initialization with decision trees is possible, as shown by Kubat (1998). On the other hand, methods that would employ boosting algorithms for induction of RBF networks have yet to be analyzed and tested. Preliminary work in this direction was reported by Kubat & Cooperson (1999).

More research of the effects of various *combinations* of the described methods is needed. For instance, the boosting algorithm can generate small decision trees that may then be mapped to one large network. Likewise, the principle of the Upstart algorithm can be used as a basis of a much more general system, where advanced subclassifiers are used instead of the `hardlim` neurons **x** and **y**, and where multi-class problems are addressed. The potential for mutual combination of techniques, and cross-fertilization of ideas, was actually a major criterion for the choice of the techniques to be included in this survey.

Acknowledgements

The research was partly supported by the grant (NSF/LEQSF-SI-JFAP-06) from the Louisiana Board of Regions and NSF and by the grant LEQSF (1999-02)-RD-A-53 from the Louisiana Board of Regions.

References

Atlas, L, Cole, R, Muthusamy, Y, Lippman, A, Connor, J, Park, D, El-Sharkawi, M and Marks, RJ, 1990. "A performance comparison of trained multilayer perceptrons and trained classification trees" *Proceedings of the IEEE* **70** 1614–1619.

Bala, JW, Michalski, RS and Pachowicz, PW, 1994. "Progress on vision through learning at George Mason University" *Proceedings of the ARPA Image Understanding Workshop* pp 191–207.

Bioch, JC, Carsouw, R and Potharst, R, 1995. "On the use of simple classifiers for the initialization of one-hidden-layer neural nets" *Proceedings of the IEEE International Conference on Neural Networks (ICNN95)* **4** 1739–1743.

Breiman, L, 1996. "Bagging predictors" *Machine Learning* **24** 123–140.

Brent RP, 1991. "Fast training algorithms for multilayer neural nets" *IEEE Transactions on Neural Networks* **2** 346–354.

Broomhead, DS and Lowe, D, 1988. "Multivariable functional interpolation and adaptive networks" *Complex Systems* **2** 321–355.

Duda, RO and Hart, PE, 1973. *Pattern Classification and Scene Analysis*. John Wiley & Sons.

Fahlman, SE and Lebiere, C, 1900. "The cascade-correlation learning architecture" In: Touretzky, DS (ed.), *Advances in Neural Information Processing Systems* **2**. Morgan Kaufmann, pp 524–532.

Fisher, DH and McKusick, KB, 1989. "An empirical comparison of ID3 and backpropagation" *Proceedings of the 11th International Joint Conference on AI, IJCAI'89* Detroit, MI, pp 788–793.

Fisher, DH, McKusick, KB, Mooney, R, Shavlik, JW and Towell, G, 1989. "Processing issues in comparisons of symbolic and connectionist learning systems" *Proceedings of the 6th International Machine-Learning Workshop* Ithaca, NY, pp 169–173.

Frean, M, 1990. "The Upstart algorithm: A method for constructing and training feed-forward neural networks" *Neural Computation* **2** 198–209.

Freund, Y, 1990. "Boosting a weak learning algorithm by majority" *Proceedings of the 1990 Workshop on Computational Learning Theory*. Morgan Kaufmann, pp 202–216.

Freund, Y, 1995. "Boosting a weak learning algorithm by majority" *Information and Computation* **121** (2) 256–285.

Freund, Y and Schapire, RE, 1996. "Experiments with a new boosting algorithm" *Proceedings of the 13th International Conference on Machine Learning, ICML'96*. Morgan-Kaufmann, pp 148–156.

Freund, Y and Schapire, RE, 1997. "A decision-theoretic generalization of on-line learning and an application to boosting" *Journal of Computer and System Sciences* **55** 119–139.

Goodman, RM, Higgins, CM, Miller, JW and Smyth, P, 1992. *Neural Computation* **4** 781–804.

Haykin, S, 1994. *Neural Networks, A Comprehensive Foundation*. Maxmillan College Publishing Co., New York.

Hinton, GE, 1990. "Connectionist learning procedures" In: Carbonell, J (ed.), *Machine Learning: Paradigms and Methods*. MIT Press, pp 185–234.

Ivanova, I and Kubat, M, 1995. "Initialization of neural networks by means of decision trees" *Knowledge-Based Systems* **8** 333–344.

Kubat, M, 1998. "Decision trees can initialize radial-basis function networks" *IEEE Transactions on Neural Networks* **9** 813–821.

Kubat, M and Cooperson, M, Jr., 1999. "Initializing RBF-networks with small subsets of training examples" *Proceedings of the 16th National Conference on Artificial Intelligence, AAAI-99* Orlando.

Le Cun, Y, Denker JS and Solla, SA, 1990. "Optimal brain damage" In: Touretzky, DS (ed.), *Advances in Neural Information Procession Systems* **2** 598–605. Morgan Kaufmann.

Lee, TC, Peterson, AM and Tsai JC, 1990. "A multi-layer feed-forward neural network with dynamically adjustable structures" *IEEE International Conference on Systems, Man, and Cybernetics* Los Angeles, CA, pp 367–369.

Mozer, MC and Smolensky, P, 1989. "Skeletonization: A technique for trimming the fat from a network via relevance assessment" In: Touretzky, DS (ed.), *Advances in Neural Information Procession Systems* **1** 107–115. Morgan Kaufmann.

Nguyen, D and Widrow, B, 1990. "Improving the learning speed of two-layer neural networks by choosing initial values of the adaptive weights" *International Joint Conference on Neural Networks* **2** San Diego, CA, pp 21–26.

Park, Y, 1994. "A mapping from linear tree classifiers to neural net classifiers" *Proceedings of IEEE International Conference on Neural Networks* Orlando, FL, **I** 94–100.

Quinlan, JR, 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

Rosenblatt, M, 1958. "The Perceptron: A probabilistic model for information storage and organization in the brain" *Psychological Review* **65** 386–408.

Rumelhart, DE and McClelland, JL, 1986. *Parallel Distributed Processing*. MIT Bradford Press.

Rumelhart, DE, Hinton, GE and Williams, RJ, 1986. "Learning internal representation by error back-propagation" In: Rumelhart, DE and McClelland, JL (eds.), *Parallel Distribution Processing*. MIT Bradford Press.

Russo, AP, 1991. "Neural networks for sonar signal processing" Tutorial No. 8, *IEEE Conference on Neural Networks for Ocean Engineering* Washington, DC.

Sahami, M, 1995. "Generating neural networks through the induction of threshold logic unit trees" *Proceedings of the 8th European Conference on Machine Learning* pp 339–342.

Sarkar, D, 1995. ''Methods to speed up error back-propagation learning algorithms'' *ACM Computing Surveys* **27** 519–542.

Schapire, RE, 1990. ''The strength of weak learnability'' *Machine Learning* **5** 197–227.

Sethi, IK, 1990. ''Entropy nets: From decision trees to neural networks'' *Proceedings of the IEEE* **78** 1605–1613.

Towell, GC and Shavlik, JW, 1994. ''Knowledge-based artificial neural networks'' *Artificial Intelligence* (to appear).

Towell, GC, Shavlik, JW and Noordewier, MO, 1990. ''Refinement of approximate domain theories by knowledge-based neural networks'' *Proceedings of the 8th National Conference on Artificial Intelligence, AAAI'90* pp 861–866.

Weigend, AS, Rumelhart, DE and Huberman, BA, 1991. ''Generalization by weight-elimination with application to forcasting'' In: Lippmann, RP, Moody, JE and Touretzky, DS (eds.), *Advances in Neural Information Procession Systems 3*. Morgan Kaufmann, pp 875–882.

Weiss, SM and Kapouleas, I, 1989. ''An empirical comparison of pattern recognition, neural nets, and machine learning classification methods'' *Proceedings of the 11th International Joint Conference on Artificial Intelligence, IJCAI'89* Detroit, MI, pp 781–787.

Whitley and Schaffer (eds.), 1992. *COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*. IEEE Press.

Widrow, B and Hoff, ME, 1960. ''Adaptive switching circuits'' *IRE WESCON Convention Record* pp 96–104.