Investigation of the CasCor Family of Learning Algorithms

Lutz Prechelt (prechelt@ira.uka.de)
Fakultät für Informatik
Universität Karlsruhe
D-76128 Karlsruhe, Germany
+49/721/608-4068, Fax: +49/721/694092

May 30, 1996

To appear in "Neural Networks"

Abstract

Six learning algorithms are investigated and compared empirically. All of them are based on variants of the candidate training idea of the Cascade Correlation method. The comparison was performed using 42 different datasets from the PROBEN1 benchmark collection. The results indicate: (1) for these problems it is slightly better *not* to cascade the hidden units, (2) error minimization candidate training is better that covariance maximization for regression problems but may be a little worse for classification problems, (3) for most learning tasks, considering validation set errors during the selection of the best candidate will not lead to improved networks, but for a few tasks it will.

Section — Computational Analysis.
Keywords — constructive learning, additive learning, cross validation, cascade correlation, empirical study.

1 Introduction

In order to obtain good generalization performance when training a neural network, it must have the right size. Networks that are too small cannot represent the required function, while networks that are too large are prone to overfitting (Geman et al., 1992). To limit the effective size of a network in order to avoid overfitting one can either use additive or subtractive methods or regularization. Additive (often called constructive) methods start out from a small network and then insert additional units (also known as nodes or neurons) and connections (also known as weights or links) until the network can represent the required function (Ash, 1989, Fahlman and Lebiere, 1990, Frean, 1990, Gallant, 1986, Hanson, 1989, Mèzar and Nadal, 1989, Simon, 1993, Wang et al., 1994, Wynne-Jones, 1991, Zollner et al., 1992). Subtractive (often called pruning) methods start out from a large network and remove superfluous parts until the network can just still represent the required function (Le Cun et al., 1989, Finnoff et al., 1993, Hassibi and Stork, 1993, Levin et al., 1994). Regularization methods use a network with a large number of parameters, but limit the size of each parameter dimension by imposing additional constraints on each weight besides error minimization; examples are weight decay (Krogh and Hertz, 1991), soft weight sharing (Nowlan and Hinton, 1992), and others; see (Finnoff et al., 1993, Reed, 1993) for an overview.

Although additive methods seemed to be quite interesting in terms of computation time and ease of use, only few of them have actually been used in real applications. The notable exception is the Cascade Correlation (CasCor) algorithm proposed by (Fahlman and Lebiere, 1990), which has been used in many applications and is implemented in most larger NN simulator programs. The idea of the CasCor algorithm is to add hidden units one by one, each on a separate hidden layer, to form a multi layer perceptron capable of solving a learning problem without prior assumptions about its size and structure. Each new unit is selected from a pool of several which are trained concurrently before one is being inserted into the network permanently; this reduces the sensitivity of the algorithm with respect to the random initialization of the weights.

There are a number of possible problems with and improvements for CasCor. Therefore, the present study investigates the behavior of CasCor and five variants of it on a variety of classification and approximation (regression) problems. Three of the variants have not been described in the literature before.

The next sections describe the original CasCor algorithm, point out possible problems with it and proposed solutions, and describes the members of the CasCor algorithm family considered. Then the empirical study is described and its results are interpreted.

2 Cascade Correlation

CasCor works as follows: Initially, we train a network without hidden units by gradient descent. Then, in subsequent training phases, one hidden unit each is inserted into the network. This is done in two partial phases: First, we generate a pool of candidate units. Each of them is initialized differently and trained independently of the others. Afterwards, the best candidate is selected and inserted into the network permanently. Second, the rest of the network is adjusted for best cooperation with the new unit. Here is a more precise description in pseudo code:

Cascade Correlation:
  Generate network without hidden units;
  REPEAT

```
    IF NOT Is first iteration
    THEN Generate and train candidates;
         Insert best candidate into the network;
    END;
    Train output connections;
  UNTIL End;
```

The *output connections* are those that lead to output units. CasCor trains those connections that lead to a hidden unit only before the unit is inserted into the network and leaves them constant later on. The structure of the initial network used in CasCor is shown in Figure 1.

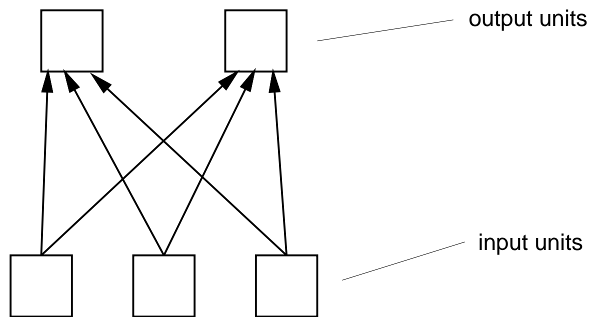

Figure 1: Initial CasCor net.

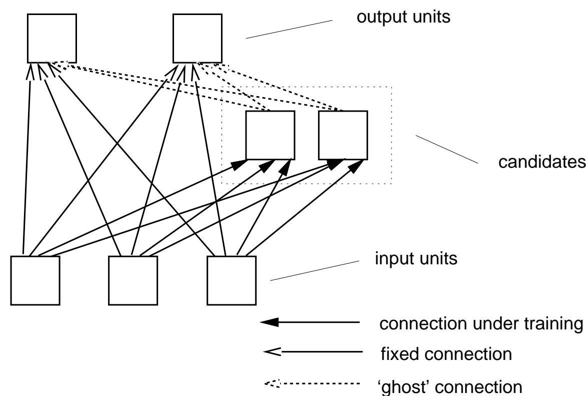

Figure 2: CasCor net during first candidate training phase.

```
Generate and train candidates:
  Generate m new candidate units C_1 to C_m;
  Connect their inputs with the outputs of the input units
  and previously existing hidden units;
  REPEAT
    Perform a gradient ascent step for covariance of
    candidates' activation with output deviation;
  UNTIL End of candidate training;
```

The structure of the network during the first candidate training phase is shown in Figure 2; a network during the third candidate training phase (i.e., with two installed hidden units) in Figure 4.

Using gradient ascent on the covariance is based on the following idea: train the candidates to have a large activation whenever the rest of the network produces a different deviation from the target output than it does on average (which is close to zero usually). Such a unit can later be used to reduce that component of the network error with which it is correlated. The covariance S of a candidate with activation C_p and output deviation E_o at output unit o over the set of all training examples p in the formulation of Fahlman and Lebiere is

$$S = \sum_o \left| \sum_p (C_p - \overline{C})(E_{p,o} - \overline{E_o}) \right|$$

where $\overline{C}$ is the average activation of the candidate and $\overline{E_o}$ is the average linear deviation at output unit o, i.e., the difference between the actual and target output value. The partial

derivative of S by the weights w_i from unit i into the candidate is

$$\frac{\partial S}{\partial w_i} = \sum_{p,o} \sigma_o (E_{p,o} - \overline{E_o}) f'_p \, out_{i,p}$$

where f'_p is the derivative of the candidate's activation function with respect to the sum of its inputs and $out_{i,p}$ is the output of unit i for example p. σ_o is the sign of the covariance for output unit o, i.e., the derivative of the absolute value function in the covariance expression. During covariance training there are no real connections from the candidates to the output units, because the candidates must not all influence the output at the same time. Instead, there are "ghost" connections that are used only to back-propagate error information but not to forward-propagate output activation. These ghost connections become normal trainable connections when a candidate is selected.

In original CasCor, the termination criterion *End candidate training* has three user selectable Parameters. It is *Permitted number e of candidate training epochs trained* `OR` *candidate training stagnates*, where stagnation means that the highest covariance produced by any of the candidates has increased by less than a certain amount a during the last k epochs. This criterion, however, is too simple; therefore, we will derive a better one below.

The covariance developed by a candidate during training depends on the random initialization of its input weights. For this reason, not only a single candidate is being trained, but a whole pool of them (typically about 8 to 16). This is possible because during covariance training, the candidates do not influence the outputs of the network; thus, they are independent of each other. The capability to train such a candidate pool is one of the big advantages of CasCor. After the end of the candidate training, the best candidate is inserted into the network, the others are deleted:

Insert best candidate into the network:
  Select candidate with highest covariance;
  Delete all other candidates;
  Connect remaining candidate to the output units;

This turns the best candidate into a new hidden unit. The resulting configuration is shown in Figure 3. The weights of the new output connections are initialized with small values, the sign of which is the inverse of the covariance with the respective output unit. The weights of the new unit's input connections are held constant for the rest of the training process. Such a hidden unit could be called a feature detector for a component of the input data that was responsible for a part of the remaining network error. This component of the remaining error can now be reduced by proper adjustment of the output connection weights: After the insertion of a hidden unit (and before the insertion of the first), the output connections are trained. Output connections are exactly all those that lead to output units.

Train output connections:
  `REPEAT`
    Perform gradient descent step for output error on the output weights;
  `UNTIL` *End of output training*;

This training phase works exactly like training a network without hidden units; the hidden units are treated just like additional input units. A backward propagation of errors through hidden units is not necessary.

In original CasCor, the termination criterion *End output training* is: *Permitted number of output training epochs done* `OR` *output training stagnates*, similar to the criterion for candidate training

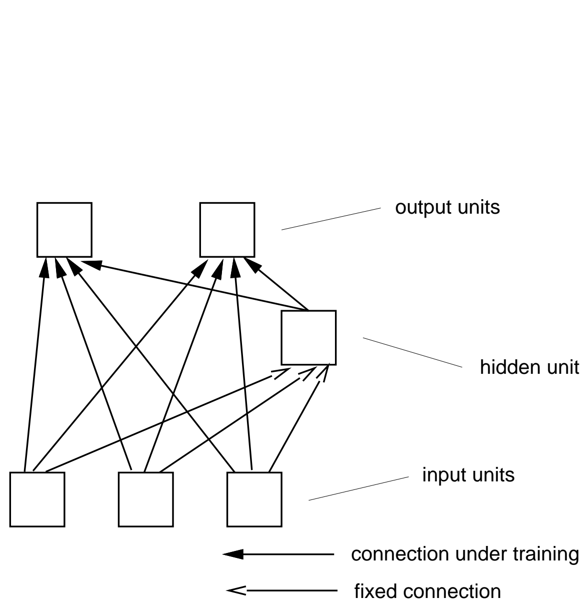

Figure 3: CasCor net after end of first candidate training phase.

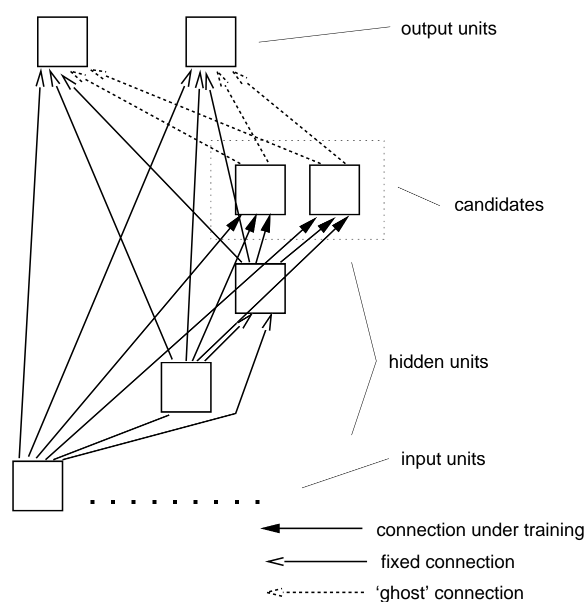

Figure 4: CasCor net during third candidate training phase. Two of the input units are not shown.

shown above. The termination criterion *End* for the overall training is: *Permitted number of hidden units inserted* OR *last hidden unit not resulted in sufficient decrease of error* OR *error small enough*. In the original formulation of CasCor, the parameters for these criteria must explicitly be set by the user, which is not acceptable for an easily usable training algorithm. Therefore, we discuss fixed criteria in Section 3.1 below.

2.1 Possible problems

Cascade Correlation has two main problems:

1. In principle, the covariance is an ill-suited target function for training the candidates. Maximizing covariance trains candidates to have a large activation (exact: large deviation from average activation) whenever the error at their output is not equal to the average error. That is, even when the error deviation is only small, a large activation leads to higher payoff (higher covariance) than a small one, although the latter would be more appropriate. Therefore, CasCor has a tendency to overcompensate errors. This makes the algorithm not well suited for regression tasks, it works better for classification tasks.

2. Cascading the hidden units results in a network that can represent very strong nonlinearities. Although this power is in principle useful, it can be a disadvantage if such strong nonlinearity is not required to solve the problem and no sufficient number of training examples is available to control the power (Sjøgaard, 1991).

2.2 Possible solutions

1. To remedy the first problem, one can change the learning rule and train directly for minimization of the output errors instead of for maximization of covariance. This error minimization approach for candidate training is used in the as yet unpublished Cascade2 algorithm of

Fahlman[1] and, in a somewhat different form, the CasEr algorithm (Littmann and Ritter, 1992). The form given below is similar to that of Cascade2 (except where noted).

For direct error minimization we must create virtual output connections for the candidate units. These connections do not propagate an activation to the output units, but nevertheless receive an error signal during the backward pass. This signal is corrected by the would-be contribution of the candidate unit and is then handled like in normal backpropagation; but only the candidate units are being trained while the rest of the network is fixed. That is, a virtual connection from a candidate i to an output unit j does not compute its gradient contribution Δw_{ij} like a normal connection as

$$\Delta w_{ij} = \underbrace{\frac{\partial E(y_j - f(in_j))}{\partial f(in_j)} \frac{\partial f(in_j)}{\partial in_j}}_{=:\delta_j} out_i$$

(where in_j is the summed input to the unit, E the error function, y_j the target output value, and f the activation function of j), but instead as

$$\Delta w_{ij} = \frac{\partial E(y_j - f(in_j + w_{ij} out_i))}{\partial f(in_j + w_{ij} out_i)} \frac{\partial f(in_j + w_{ij} out_i)}{\partial (in_j + w_{ij} out_i)} out_i$$

This computation becomes simpler, if f is the identity function (i.e., we use linear output units) and E is the squared error function. In this case the correction can be computed locally in the connection, because its contribution adds linearly to the delta value to be backpropagated, which is routinely available as δ_j. The resulting computation becomes

$$\Delta w_{ij} = \underbrace{(\delta_j - w_{ij}\, out_i)}_{=:\hat{\delta}_j} out_i$$

In the following we always assume the squared error function, although others would be possible as well.

The *goodness G of a candidate* can be expressed using the quotient of total error of the network with and without the candidate. We normalize this value to zero for candidates without any effect and otherwise measure in percent, that is

$$G := 100 \left(\frac{E_{net}}{E_{cand}} - 1 \right) = 100 \left(\frac{\sum_{j,p} (y_j(p) - o_j(p))^2}{\sum_{j,p} \hat{\delta}_j(p)} - 1 \right)$$

This value can be negative, namely if the candidate increases the total error. This is common at the beginning of the candidate training or for the errors in a validation set. A unit with negative goodness may nevertheless improve the network, because the subsequent training of the output connections can turn it into a positive overall effect.

In contrast to the above, Cascade2 uses nonlinear activation functions in the output units and nevertheless trains using the simplified error correction term. The same is true for CasEr (tanh activation function), with the additional restriction that the output weights are assumed to have constant value 1, i.e., are not trained.

The CasEr article (Littmann and Ritter, 1992) evaluated the algorithm for classification problems only (plus a failed attempt to learn the Mackey-Glass time series) and concludes that CasCor works better than CasEr. However, for regression tasks the covariance learning rule is obviously ill-suited and should be replaced by direct error minimization.

[1] All information about Cascade2 is from personal communication with Scott Fahlman, April 1994 and later.

2. For the second problem (Sjøgaard, 1991) shows that networks generated with CasCor can systematically have worse generalization than networks trained with the same method without cascading of the hidden units. He places all hidden units in one hidden layer. Unfortunately, Sjøgaard uses but a single artificial learning problem with only two inputs and one output. (Yeung, 1991) had the same algorithm idea and concludes that there is almost no difference to CasCor for a number of learning problems. Fahlman says that the cascading is important for some problems and will not hurt for the others. Obviously additional empirical data is required to find out who is right. Such data is presented in the study at hand.

2.3 Related work

There are many other suggestions for additive learning besides the CasCor family. Among the earliest are Gallant's *tower* and *inverted pyramid* proposals based on simple perceptrons (Gallant, 1986) followed by several refinements, extensions, and re-inventions, e.g. (Baffes and Zelle, 1992, Frean, 1990, Mèzar and Nadal, 1989, Simon, 1993).

Several methods, e.g. (Ash, 1989, Wang et al., 1994), propose addition of units in the hidden layers of standard MLPs during normal backpropagation training, but this approach severely disturbs the training process because of the interaction of hidden units. Not even constructive unit splitting with reasonable initializations for the new units works well (Hanson, 1989, Wynne-Jones, 1991). (Littmann and Ritter, 1993) propose *direct cascading* where local linear maps or different neural modules are cascaded and produce the output from the union of the original network inputs and the outputs of previous modules. This approach works well e.g. for predicting the Mackey-Glass time series.

A number of completely different approaches to additive learning are based on radial basis functions (RBFs) that cover the input space with regions instead of segmenting it with borders as MLPs do. One of the most sophisticated RBF methods is the supervised version of Growing Cell Structures (Fritzke, 1994) . It uses a front-end network based on an approximated k-dimensional Voronoi tessellation of the input space using k-dimensional simplices of units and unsupervised local learning and a back-end to re-interpret the front-end network units as RBFs and to compute outputs from them. The learning rule inserts new units in regions where large errors occur and thus allows to learn both the number and the position of the RBFs; a suitable value of k has to be selected by hand, though.

3 The algorithm family

In this section I will present those six members of the CasCor family that have been investigated in the study. All these algorithms are specified with fixed values for all control parameters, so that no user tuning is required and the algorithms could be called "automatic".

3.1 Termination criteria

All members of the CasCor family are very sensitive to changes in the stopping criteria of the candidate and output training phases. If training is too short, the components of the network will not work together well enough for good results. If training is too long, it costs much computation time and may result in overfitting and bad generalization. In the original version of CasCor and the other published candidate training algorithms, no measures against overfitting

are described. Therefore, we extend the criteria for changing training phases accordingly by applying the method of early stopping using a validation set.

To formally describe the criteria, we need some definitions. Let E be the objective function (error function) of the training algorithm, in our case the squared error. Then $E_{tr}(t)$ is the average error per example of the network over the training set, measured after epoch t. $E_{va}(t)$ is the corresponding error on the validation set and is used by the stopping criterion. $E_{te}(t)$ is the corresponding error on the test set; it is not known to the training algorithm but characterizes the quality of the network resulting from training.

Let $E_{opt}(t)$ be the lowest validation set error obtained in epochs up to t:

$$E_{opt}(t) = \min_{t' \leq t} E_{va}(t')$$

Now we define the *generalization loss* at epoch t to be the relative increase of the validation error over the minimum-so-far (in percent):

$$GL(t) = 100 \cdot \left(\frac{E_{va}(t)}{E_{opt}(t)} - 1 \right)$$

High generalization loss will be one of the criteria used to stop training or training phases, because it directly indicates overfitting.

However, we might want to suppress stopping if the training is still "progressing rapidly". To formalize this notion we define a *training strip of length* k to be a sequence of k epochs numbered $n+1 \ldots n+k$ where n is divisible by k. The training *progress* (in per thousand) measured after such a training strip is then

$$P_k(t) = 1000 \cdot \left(\frac{\sum_{t' \in t-k+1 \ldots t} E_{tr}(t')}{k \cdot \min_{t' \in t-k+1 \ldots t} E_{tr}(t')} - 1 \right)$$

that is, "how much was the average training error during the strip larger than the minimum training error during the strip?" Note that this progress measure is high for instable phases of training, where the training set error goes up instead of down.

For the candidate training termination criteria, we always consider the candidate with the best goodness $\hat{G}(t)$ in some epoch t and define the *progress of candidate training* for a sequence of k epochs, measured in per thousand, to be

$$\ddot{P}_k(t) = 10 \left(\max_{t' \in t-k+1 \ldots t} (\hat{G}(t)) - \frac{1}{k} \sum_{t' \in t-k+1 \ldots t} \hat{G}(t) \right)$$

The definition has to use a difference measure instead of a proportion measure, because $\hat{G}(t)$ may become zero. This goodness is always goodness on the training set. There is also the goodness $\hat{G}_{va}(t)$ on the validation set and we sometimes write $\hat{G}_{tr}(t)$ to mean $\hat{G}(t)$.

In analogy to the generalization loss GL we define the *goodness loss* VL_{va} on the validation set

$$VL_{va}(t) = 100 \, \frac{\max_{t' \leq t}(\hat{G}_{va}(t')) - \hat{G}_{va}(t)}{\max\left(\left| \max_{t' \leq t} \hat{G}_{va}(t') \right|, 1 \right)}$$

This measure normalizes the difference of goodness values with their absolute value, except when they are in the range $-1 \ldots 1$ because otherwise there would be a singularity at zero. We want

to know the goodness loss for the training set as well; we write VL_{tr} and define it in analogy to VL_{va} above.

With these definitions we can describe the termination criteria. We start with the more critical one, namely, candidate training:

1. Stagnation conditions for candidate training are known to be difficult to find. Even with CasCor, a candidate can show almost no improvement in covariance for, say, 10 epochs and then suddenly rise further. Such effects are still more pronounced in error minimization candidate training; the error curves over time are very turbulent, intermittent increase of the error happens frequently. Therefore, termination criteria must not react to low progress too fast. We define the *last improvement epoch* as the last epoch t, for which $\ddot{P}_5(t) > 0.5$ and terminate candidate training because of stagnation only when t is 40 epochs ago. Fahlman uses a similar criterion with an interval of only 12 epochs; my experiments indicated that this value is too small for many real datasets.

2. Candidate training exhibits overfitting as well, as is shown by the severe example in Figure 5. Cascade2 is much more sensitive to candidate overfitting than CasCor. However, due to the

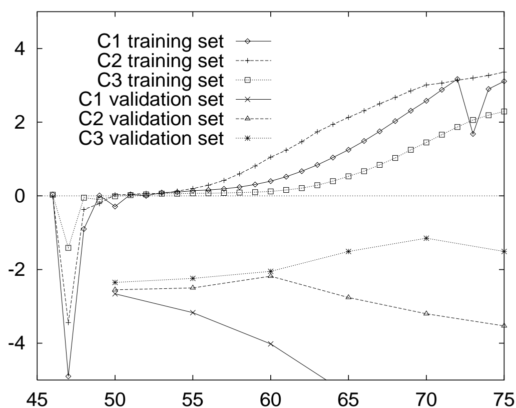

Figure 5: Goodness values over time for three candidates C1, C2, C3 on the training and the validation set during first candidate training of glass1 with cascade algorithm. There is severe overfitting in all candidates and a turbulence for C1.

turbulent development of the candidates, the termination criterion must not be too responsive to overfitting. We terminate when $VL_{va}(t) > 25$, but not before epoch 25 and only when $VL_{tr}(t) = 0$ at the same time, i.e., when goodness on the training set has not decreased. After termination of candidate training, the weights of each candidate C are reset to their values in epoch $\hat{t}_C$, where C had its highest goodness $G_{va}(\hat{t}_C)$ on the validation set. Accordingly, the goodness used to select the "best" candidate is the goodness on the training set at the same point: $G_{tr}(\hat{t}_C)$.

3. Even with both of the above termination criteria, candidate training may take very long. Since multiple long candidate training phases would result in unacceptably long overall training times, we also limit the absolute number of epochs per candidate training phase. This value must not be too small, because otherwise the network will contain insufficiently trained units. 150 epochs seems to be a good compromise; however, a higher value would probably lead to slightly better results in some cases.

These three parts together result in the following termination criterion for candidate training:

End of candidate training:
Last improvement epoch is 40 epochs ago `OR`
($VL_{va}(t) > 25$ `AND` At least 25 epochs trained `AND` $VL_{tr}(t) = 0$) `OR`
Candidate training performed for 150 epochs.

An interesting question for the the termination of the output training is how to discriminate it from the overall termination of the training. Output training should have a minimum length, just like candidate training, and I choose 25 epochs again. Apart from that, the same criteria apply for output and overall training: high generalization loss, low progress, and exceeding a maximum number of epochs. Experiments resulted in the following as reasonable choices:

End of output training:
At least 25 epochs trained in this output training `AND`
(Altogether more than 5000 epochs trained `OR`
Generalization loss $GL(t) > 2$ `OR` Progress $P_5(t) < 0.4$)

Overall training is terminated when either the generalization loss is substantial or the last insertion of a hidden unit resulted in hardly any progress:

End:
Altogether more than 5000 epochs trained `OR`
Generalization loss $GL(t) > 5$ `OR`
(Progress $P_5(t) < 0.1$ `AND` *Little improvement from last new hidden unit*)

Little improvement from last new hidden unit:
Reduction of training set error due to last new hidden unit less than 0.1% `AND`
Validation set error increased due to last new hidden unit.

To derive the above criteria, I always used training set errors and validation set errors only; never any test set errors. It is a little unsatisfactory that all of the criteria are heuristic. However, I do not know of any theory that would allow the derivation of criteria that are both efficient and effective.

3.2 Six algorithms

This subsection presents the derivation of the six algorithms investigated in this study. They have several things in common (except where noted): (1) output units use the identity activation function, (2) hidden units use the $x/(1+|x|)$ activation function, (3) squared error is used as cost function, (4) termination criteria are as described above, (5) the RPROP learning rule (Riedmiller and Braun, 1993) is used for weight update, with the parameters $\eta^+ = 1.2$, $\eta^- = 0.5$, $\Delta_0 \in [0.05 \ldots 0.2]$ randomly per weight, $\Delta_{max} = 50$, $\Delta_{min} = 0$, initial weights from [-0.1...0.1] randomly, and (6) nine units are used in each candidate pool.

The first three algorithms are already known from above:

cascor is the mother of the algorithm family;

cascade is CasCor with error minimization instead of covariance maximization for the candidates;

cand is cascade with all hidden units in just one layer as suggested for CasCor by Sjøgaard and Yeung. Hidden units receive input connections from the input units only.

The other three algorithms are based on the following idea: If we have already produced a number of candidates with different goodness on training set and validation set, why select one of them solely based on training set results? If we used the goodness on the validation set as well, we could often select a candidate with better generalization. So let us redefine the notion of "best candidate" using a combination of G_{tr} and G_{va} as the measure.

This idea has two possible drawbacks: First, if G_{tr} is not weighed high enough, the algorithm will confuse itself, because the chosen candidate may be insufficiently adapted to the training set and thus disturb subsequent training progress. Second, using the validation set for the selection strikes a "leak" through which small amounts of information about the validation set flow into the training process. This may confuse the termination criteria, because they are built on the assumption that the validation set results are not optimized by the algorithm directly.

Due to these two problems it is impossible to say in advance whether the advantage of using G_{va} will indeed lead to better networks. To answer this question, I used three algorithms from this class, which I call *kogi*. The chosen representatives are called kogi2, kogi3, and kogi9 for historical reasons.

kogi2 is equivalent to cascade, except that it uses G_{kogi} instead of G_{tr} to select the best candidate, where $G_{kogi} := 1/3(G_{va} + 2G_{tr})$. The same criterion is used in the other kogi algorithms. This weighting was found in pretests as a good compromise between robustness (due to preference for G_{tr}) and possible improvements (due to use of G_{va}).

kogi3 is equivalent to cand, with two changes: First, the use of G_{kogi} and second the use of several different activation functions in the candidates. In each candidate training phase, the same number of candidates use the activation functions $x/(1+|x|)$ (symmetric soft sigmoid), $1/(1+e^{-x})$ (standard sigmoid), and $e^{-x^2/2}$ (Gaussian), respectively. This idea was already suggested (but not tried) by Fahlman for CasCor; Sjøgaard finds for his learning problem that this approach makes training faster (with equal quality of the solutions) for the non-cascading network, but makes generalization worse with CasCor. For that reason I do not use different activation functions for the cascading networks.

kogi9 tries to combine the advantages of kogi2 (powerful network due to cascading) and kogi3 (less inclination to overfitting). It is based on the observation that only rarely a result using a static network is published that uses more than two hidden layers. A two hidden layer network is generated by kogi9 by using two pools of candidates; pool 1 supplies the first hidden layer, pool 2 the second hidden layer, receiving connections from the input units and all previously inserted hidden units in the first hidden layer. All candidates compete for selection. If only G_{tr} was used for the selection, pool 1 units would hardly have a chance to be selected in later training phases, because pool 2 units have more input information available. But this information may also lead to overfitting so that the G_{kogi} selection measure picks the best candidate from either pool 1 or pool 2, whichever is more appropriate depending on the learning problem and training stage. This approach makes use of at least some of the power of a cascading algorithm, yet avoids most of its dangers. Depending on the size of the pools, kogi9 training has a higher computational cost than the other algorithms, because it uses more candidates. Just like kogi3, kogi9 also uses different activation functions.

4 Experimental results

To compare the six algorithms described above, a large set of experiments was performed using a MasPar MP-1 16384 processor SIMD machine, a KSR-1 32 processor MIMD machine, and six Sun workstations over several weeks. 8524 completed runs were made overall.

4.1 Setup

All experiments used the datasets of the PROBEN1 benchmark collection. 42 different datasets were used, representing 14 different tasks, i.e., there are three versions of each task, each using

a different partitioning of the data into training set, validation set, and test set. The problems have between 8 and 120 inputs, between 1 and 19 outputs, and between 214 and 7200 examples. All inputs and outputs are normalized to range 0...1. 10 of the problems are classification tasks using 1-of-n output encoding (*cancer, card, diabetes, gene, glass, heart, heartc, horse, soybean*, and *thyroid*), 4 are regression tasks (*building, flare, hearta*, and *heartac*). All problems are real datasets from realistic application domains; for a detailed description of the domains and the datasets refer to (Prechelt, 1994). Due to technical problems that are hardly avoidable in such a long series of experiments, the samples for each algorithm/dataset pair are not completely balanced. On the average, 34 runs were made for each pair.

All runs used the parameters and termination criteria shown in the previous sections. The results of the runs were tested for statistically significant differences in the mean of the resulting squared error on the test set between the samples for the same dataset but different algorithms. The test used was a t-test with Cochran/Cox-approximation for the case of unequal variances as implemented in the SAS system. The values tested were the logarithms of the mean squared error on the test set using that state of the network that exhibited the lowest error on the validation set. Logarithms are used, because the test set errors are usually log-normally distributed. There are some samples (16% overall), with substantial deviations from a log-normal distribution; these make the t-test underestimate the significance of differences. The corresponding results are marked accordingly in the tables below. Furthermore, many samples had a small number of outliers, which where removed from the samples before the tests (3% of the runs overall).

Using the results of these tests, I tried to answer the following questions (for the set of domains covered by PROBEN1):

1. Is cand better than cascade, as Sjøgaard's results would suggest, or is it the other way round, as Fahlman assumes?
2. Is cascade better than cascor, as we would expect from theoretical considerations?
3. Is kogi3 better than cand?
4. Is kogi2 better than cascade?
5. How good is kogi9 compared to kogi2 and kogi3?

4.2 Results and discussion

The detailed results of the individual runs with error, classification error, number of epochs, number of units etc. are available for anonymous FTP from `ftp.ira.uka.de` in directory `/pub/neuron` as file `nndata.tar.gz` (774 kB). This file also includes result data from several other experiment series with the PROBEN1 collection.

4.2.1 cand vs. cascade

The first question can be answered by looking at the left part of Table 1: For the domains considered here, cand is superior to cascade more often (building, glass, thyroid), than vice versa (diabetes). In most cases, however, the differences, if any, are rather small.

It is interesting to examine the size of the networks built by cand and cascade, as shown in Table 2. Two observations can be made: First, cascade does *not* always use less units than cand, although this would be expected due to the higher representational power of cascaded units and the higher number of connections for fixed number of units. This contradiction suggests that cascade is not able to use its resources in a very efficient way. Second, we observe a high variance

Table 1: Comparison of cand, cascade, and cascor using t-test

Problem	cand (c) vs. cascade (C) 1	2	3	cascade (C) vs. cascor (R) 1	2	3	cand (c) vs. cascor (R) 1	2	3
building	—	c 0.0	(c 0.0)	—	R 1.9	C 9.7	R 0.7	c 0.0	c 0.0
cancer	—	(c 0.2)	—	R 0.0	(R 3.5)	R 0.0	R 0.0	c 7.5	R 0.0
card	—	—	(c 9.8)	—	—	—	c 4.8	—	—
diabetes	—	C 2.6	C 2.2	C 0.0	(C 0.0)	—	c 0.0	(c 0.2)	R 2.0
flare	—	—	—	—	C 0.0	C 0.7	—	c 0.0	c 9.9
gene	—	—	(C 2.5)	—	—	—	—	—	(R 0.3)
glass	c 0.1	—	c 0.0	R 0.0	—	R 0.0	R 3.6	—	—
heart	—	C 2.1	—	—	—	—	R 9.5	(R 0.1)	c 1.1
hearta	—	—	—	R 4.2	C 1.4	—	R 7.0	c 6.3	—
heartac	—	—	—	—	—	—	—	—	—
heartc	—	—	—	—	—	—	—	R 1.8	—
horse	—	—	C 1.3	—	—	—	—	—	—
soybean	c 0.0	(C 8.2)	—	R 0.9	R 0.0	—	—	(R 2.1)	—
thyroid	c 0.0	(c 8.3)	(c 0.0)	R 0.9	(R 0.0)	(R 0.7)	c 0.0	—	c 0.0

Each entry represents the results of a t-test comparing the means of logarithmic test set errors for two problem/algorithm pair samples on versions 1, 2, 3 of each problem. Dashes mean differences that are not significant on a 10% level, other entries indicate the superior algorithm and the p-value of the test in percent, i.e., the probability that the observed differences are purely accidental. Results in parentheses are imprecise, because at least one of the samples was not log-normally distributed; the given probabilities in these cases are over-estimations.
cand vs. cascade: 26 times no significant difference, 10 times cand better ("c"), 6 times cascade better ("C"). For regression tasks: 10 times no significant difference, 2 times cand better.
cascade vs. cascor: 24 times no significant difference, 12 times cascor better ("R"), 6 times cascade better ("C"). For regression tasks: 6 times no significant difference, 4 times cascade better, 2 times cascor better.
cand vs. cascor: 19 times no significant difference, 12 times cand better ("c"), 11 times cascor better ("R"). For regression tasks: 5 times no significant difference, 5 times cand better, 2 times cascor better.

of unit numbers for some of the problems. This is a sign for the inability of both algorithms to reliably produce the "right" network.

4.2.2 cascade vs. cascor

For the second question (see middle part of Table 1), we must differentiate. As expected, cascade is slightly better better than cascor for the regression tasks (building, flare, hearta, heartac). For classification tasks, on the other hand, cascor seems to be a bit better despite the fact that we do not compare the classification errors but the squared errors (i.e., we interpret the task to be the approximation of posterior probabilities). A possible explanation of this phenomenon is the faster convergence of candidates with covariance training compared to error minimization training. The limit of 150 epochs for candidate training is a harder restriction for cascade candidates than for the faster developing cascor candidates. Furthermore, due to their stronger tendency to overfit, cascade candidates can often not even use all of these 150 epochs. As we can see from the right part of Table 1, cascor is roughly as good as cand.

It is an interesting observation that the not theoretically justified learning rule of cascor has (small) advantages over the "correct" rule for many learning problems.

Table 2: Numbers of hidden units generated by cand and cascade

	Number of units generated by cand						Number of units generated by cascade					
	dataset 1			dataset 2			dataset 1			dataset 2		
Problem	μ	σ	min-max	μ	σ	min-max	μ	σ	min-max	μ	σ	min-max
building	19	32	0–92	58	15	0–65	4	5	0–16	9	6	0–25
cancer	16	9	3–43	24	13	3–59	15	14	3–54	15	15	0–60
card	2	1	1–6	2	1	0–7	1	1	1–3	2	1	1–6
diabetes	12	8	3–34	8	6	0–32	8	8	0–33	9	7	0–34
flare	3	2	1–8	3	1	2–5	3	2	1–8	3	1	2–7
gene	5	2	2–13	5	2	1–12	7	4	2–19	6	3	2–13
glass	9	3	4–21	2	3	0–20	12	9	2–38	3	5	0–29
heart	5	3	2–14	5	3	2–15	5	4	2–24	6	7	2–42
hearta	5	3	0–13	4	2	2–11	6	3	0–14	5	4	2–18
heartac	3	2	2–8	1	1	0–4	2	1	1–4	0	0	0–1
heartc	3	1	2–6	4	4	0–19	3	1	1–5	5	3	1–15
horse	1	1	0–2	2	1	0–5	1	1	0–2	1	1	0–3
soybean	25	6	11–39	25	10	1–44	32	9	13–43	25	12	2–43
thyroid	48	12	15–59	37	16	2–59	23	19	2–56	15	17	2–55

For two versions (1 and 2) of each problem: number of hidden units generated by cand and cascade algorithm, respectively, described by the mean μ, standard deviation σ, minimum, and maximum over all runs.

Table 3: Comparison of cand/cascade with kogi2/kogi3 using t-test

	cand (c) vs. kogi3 (k)			cascade (C) vs. kogi2 (K)		
Problem	1	2	3	1	2	3
building	—	—	(c 1.1)	—	—	—
cancer	(c 0.0)	c 0.0	c 0.5	C 7.8	—	—
card	—	—	—	C 2.3	C 9.4	—
diabetes	c 0.0	—	c 0.0	—	—	—
flare	—	(c 0.6)	c 0.1	—	K 0.1	—
gene	k 0.0	k 0.1	k 0.0	—	—	(C 4.0)
glass	c 0.0	k 0.3	—	—	K 5.1	—
heart	c 0.0	—	c 0.0	—	—	—
hearta	c 0.3	(c 0.1)	(c 1.7)	—	(C 2.2)	—
heartac	c 1.8	—	—	—	—	—
heartc	c 2.4	—	—	—	—	—
horse	—	—	—	—	—	—
soybean	c 0.0	(c 3.1)	c 0.0	—	(C 0.9)	—
thyroid	c 0.0	—	—	—	—	—

The structure of the table is analog to Table 1.
cand vs. kogi3: 18 times no significant difference, 20 times cand better ("c"), 4 times kogi3 better ("k").
cascade vs. kogi2: 34 times no significant difference, 6 times cascade better ("C"), 2 times kogi2 better ("K").

4.2.3 cand/cascade vs. kogi3/kogi2

The comparison of cand and cascade with their counterparts kogi3 and kogi2, respectively, is shown in Table 3. This data allows to answer questions 3 and 4: kogi3 and kogi2 are not generally better than their conventional counterparts. For cand and kogi3 we find a clear inferiority of kogi3 with one notable exception: for the gene problem, the kogi3 results are much better than those of cand. An explanation for this discrepancy may be the high number of 120 input units present in the gene problem, which is far more than in any other problem. The large number of inputs may make the candidates particularly susceptible to overfitting and kogi3 can avoid such overfitting better than cand can.

A corresponding inferiority of kogi2 compared to cascade is hardly present. Probably the relative usefulness of using the validation set error is higher in this case due to the higher overfitting capabilities of cascaded units. This hypothesis is supported by the direct comparison of kogi2 and kogi3 as shown in Table 4: In direct comparison, the cascading kogi2 is better than kogi3,

Table 4: Comparison of kogi2, kogi3, and kogi9 using t-test

	kogi3 (k) vs. kogi2 (K)			kogi3 (k) vs. kogi9 (Z)			kogi2 (k) vs. kogi9 (Z)		
Problem	1	2	3	1	2	3	1	2	3
building	—	k 0.0	—	—	k 0.0	—	—	Z 0.0	—
cancer	(K 0.4)	—	—	—	—	—	—	—	—
card	—	—	—	—	(Z 1.0)	k 3.6	—	Z 0.1	—
diabetes	K 0.0	—	K 0.0	—	—	—	K 0.0	—	K 0.0
flare	—	(K 0.0)	K 0.1	—	—	—	—	K 0.0	K 0.1
gene	k 0.0	k 1.9	k 0.0	—	—	k 9.0	Z 0.0	Z 3.7	Z 0.0
glass	K 0.8	k 4.1	k 4.2	—	k 1.5	—	K 0.8	—	Z 2.0
heart	K 0.0	K 0.3	K 0.1	—	—	—	K 0.0	K 0.0	K 0.0
hearta	K 8.6	(K 2.7)	(K 5.5)	—	—	—	—	(K 1.1)	K 9.2
heartac	K 3.3	—	k 7.1	—	—	—	K 4.9	—	—
heartc	K 1.0	—	—	—	Z 8.8	—	—	—	—
horse	K 8.7	K 4.9	—	—	—	—	—	—	—
soybean	K 0.0	(K 4.7)	K 0.0	—	—	—	K 0.0	(K 2.9)	K 0.0
thyroid	k 1.6	—	(k 0.0)	—	—	k 0.0	—	(Z 1.4)	(Z 0.0)

The structure of the table is analog to Table 1.
kogi3 vs. kogi2: 14 times no significant difference, 19 times kogi2 better ("K"), 9 times kogi3 better ("k").
kogi3 vs. kogi9: 35 times no significant difference, 5 times kogi3 better ("k"), 2 times kogi9 better ("Z").
kogi2 vs. kogi9: 20 times no significant difference, 14 times kogi2 better ("K"), 8 times kogi9 better ("Z").

which builds single hidden layer networks — in contrast to the comparison of the conventional cascade and cand algorithms. This suggests that the kogi idea of using validation set goodness for selecting candidates seems to be particularly useful for cascading networks. However, the kogi variants used in this study are probably not yet optimally tuned.

4.2.4 kogi2/kogi3 vs. kogi9

Finally, let us have a look at the hybrid kogi9, which builds a network with at most two hidden layers. Comparisons of this algorithm to kogi3 and kogi2 are shown in Table 4.

Obviously the crossing of kogi2 and kogi3 was successful: kogi9 sometimes is better than kogi2 and sometimes is better than kogi3. In general, its behavior seems to be quite close to that of kogi3, to which only a few significant differences were found. However, altogether kogi9 is not better than either kogi3 or kogi2, except in cases where a network with exactly two hidden layers seems to be a particularly good choice — the gene problem is such a case.

5 Conclusion

We have derived six members of the CasCor family by varying the following aspects:

1. Either cascade or not cascade the hidden units.
2. Either use covariance training or error minimization training for candidates.
3. Either use goodness on training set alone or on training set and validation set for selecting best candidate.

The six algorithms were compared using the results of 8524 runs performed for 42 different datasets from the PROBEN1 benchmark collection. The comparisons indicate the following with respect to the kind of domains represented by the datasets:

1. Not cascading hidden units is superior to cascading them for some problems and inferior for others; the former case occurred more often. In most cases the decision does not make a significant difference at all.
2. Covariance candidate training is usually inferior to error minimization training for regression problems. For classification problems it is often superior because it converges faster.
3. Utilizing the goodness on the validation set to select a candidate can sometimes improve the results, in particular for cascading networks. In most cases, however, no improvement occurs and often the results will even become worse, in particular for non-cascading networks.

References

Ash, T. (1989). Dynamic node creation in backpropagation networks. *Connection Science*, 1(4):365–375.

Baffes, P. T. and Zelle, J. M. (1992). Growing layers of perceptrons: Introducing the Extentron algorithm. In *Proc. Int. Joint Conf. on Neural Networks 1992, vol. 2*, Baltimore, pp. 392–397.

Cun, Y. L., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In Touretzky, D. S. (ed.), *Advances in Neural Information Processing Systems 2*, San Mateo, CA. Morgan Kaufman Publishers, pp. 598–605.

Fahlman, S. E. and Lebiere, C. (1990). The Cascade-Correlation learning architecture. Technical Report CMU-CS-90-100, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Finnoff, W., Hergert, F., and Zimmermann, H. G. (1993). Improving model selection by non-convergent methods. *Neural Networks*, 6:771–783.

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

Fritzke, B. (1994). Growing cell structures — a self-organizing network for unsupervised and supervised learning. *Neural Networks*, 7(9):1441–1460.

Gallant, S. I. (1986). Three constructive algorithms for network learning. In *Proc. of the 8th Annual Conf. of the Cognitive Science Society*, pp. 652–660.

Geman, S., Bienenstock, E., and Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural Computation*, 4:1–58.

Hanson, S. J. (1990). Meiosis networks. In Touretzky, D. S. (ed.), *Advances in Neural Information Processing Systems 2*, San Mateo, CA. Morgan Kaufman Publishers, pp. 533–541.

Hassibi, B. and Stork, D. G. (1993). Second order derivatives for network pruning: Optimal brain surgeon. In Hanson, S. J., Cowan, J. D., Giles, C. L. (eds.) *Advances in Neural Information Processing Systems 5*, San Mateo, CA. Morgan Kaufman Publishers, pp. 164–171.

Krogh, A. and Hertz, J. A. (1992). A simple weight decay can improve generalization. In Moody, J. E., Hanson, S. J., Lippmann, R. P. (eds.) *Advances in Neural Information Processing Systems 4*, San Mateo, CA. Morgan Kaufman Publishers, pp. 950–957.

Levin, A. U., Leen, T. K., and Moody, J. E. (1994). Fast pruning using principal components. In Cowan, J. D., Tesauro, G., Alspector, J. (eds.) *Advances in Neural Information Processing Systems 6*, San Mateo, CA. Morgan Kaufman Publishers, pp. 35-42.

Littmann, E. and Ritter, H. (1992). Cascade network architectures. In *Proc. Int. Joint Conf. on Neural Networks, vol. 2*, Baltimore, pp. 398–404.

Littmann, E. and Ritter, H. (1993). Generalization abilities of cascade network architectures. In Hanson, S. J., Cowan, J. D., Giles, C. L. (eds.) *Advances in Neural Information Processing Systems 5*, San Mateo, CA. Morgan Kaufman Publishers, pp. 188–195.

Mézard, M. and Nadal, J.-P. (1989). Learning in feedforward layered networks: The Tiling algorithm. *Journal of Physics A: Math. Gen.*, 22(12):2191–2203.

Nowlan, S. J. and Hinton, G. E. (1992). Simplifying neural networks by soft weight-sharing. *Neural Computation*, 4(4):473–493.

Prechelt, L. (1994). PROBEN1 — A set of benchmarks and benchmarking rules for neural network training algorithms. Technical Report 21/94, Fakultät für Informatik, Universität Karlsruhe, Germany. Anonymous FTP: /pub/papers/techreports/1994/1994-21.ps.gz on ftp.ira.uka.de.

Reed, R. (1993). Pruning algorithms — a survey. *IEEE Transactions on Neural Networks*, 4(5):740–746.

Riedmiller, M. and Braun, H. (1993). A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proc. of the IEEE Intl. Conf. on Neural Networks*, San Francisco, CA, pp. 586–591.

Simon, N. (1993). Constructive supervised learning algorithms for artificial neural networks. Master's thesis, Delft University of Technology, Department of Electrical Engineering, Delft, Netherlands.

Sjøgaard, S. (1991). *A Conceptual Approach to Generalisation in Dynamic Neural Networks.* PhD thesis, Aarhus University, Aarhus, Danmark.

Wang, Z., Massimo, C. D., Tham, M. T., and Morris, A. J. (1994). A procedure for determining the topology of multilayer feedforward neural networks. *Neural Networks*, 7(2):291–300.

Wynne-Jones, M. (1991). Node splitting: A constructive algorithm for feed-forward neural networks. In Moody, J. E., Hanson, S. J., Lippmann, R. P. (eds.) *Advances in Neural Information Processing Systems 4*, San Mateo, CA. Morgan Kaufman Publishers, pp. 1072–1079.

Yeung, D.-Y. (1991). A neural network approach to constructive induction. In Lawrence A. Birnbaum, G. C. C., editor, *Machine Learning - Proc. of the 8th Int. Workshop*, San Mateo, CA. Morgan Kaufman Publishers, pp. 228–232.

Zollner, R., Schmitz, H. J., Wünsch, F., and Krey, U. (1992). Fast generating algorithm for a general three-layer perceptron. *Neural Networks*, 5(5):771-777.