Exploring Constructive Cascade Networks

N.K. Treadgold and T.D. Gedeon
Department of Information Engineering
Computer Science and Engineering
University of New South Wales

May 25, 1999

Abstract

Constructive algorithms have proved to be powerful methods for training feedforward neural networks. An important property of these algorithms is generalization. A series of empirical studies were performed to examine the effect of regularization on generalization in constructive cascade algorithms. It was found that the combination of early stopping and regularization resulted in better generalization than the use of early stopping alone. A cubic penalty term that greatly penalizes large weights was shown to be beneficial for generalization in cascade networks. An adaptive method of setting the regularization magnitude in constructive algorithms was introduced and shown to produce generalization results similar to those obtained with a fixed, user-optimized regularization setting. This adaptive method also resulted in the construction of smaller networks for more complex problems. The *acasper* algorithm, which incorporates the insights obtained from the empirical studies, was shown to have good generalization and network construction properties. This algorithm was compared to the Cascade Correlation algorithm on the Proben 1 and additional regression data sets.

Index Terms – Constructive algorithms, feedforward neural network, cascade, regularization, generalization.

1 Introduction

1.1 Model Selection and Generalization

A key difficulty faced in the field of Feedforward Neural Networks (FNNs) is model selection. Model selection involves matching the complexity of the function to be approximated with the complexity of the model. FNN model complexity is determined by factors such as weight number, magnitude and connection topology. If a model does not have the complexity to approximate the desired function, underfitting and poor generalization occur. If a model is too complex, then it may overfit the data and also give poor generalization.

It is possible to classify FNN model selection techniques into three groups: those that perform a search through models, those that begin with an overly complex model which is then simplified, and those that begin with a simple model whose complexity is increased. The first group is generally implemented by selecting various network architectures that are trained and compared. One well respected method of comparison is cross–validation [1]. This is often computationally expensive to perform due to the non-linear optimization process employed in training FNNs, although there have been some attempts to reduce this expense [1].

The second group of model selection techniques begin training with an oversize network. Pruning is one such technique [2]. The non-convergent technique of early stopping [3] also uses an oversize network, and works by stopping training at a point where the performance on a validation set begins to worsen. The performance on the validation set will typically start to worsen when an overly complex model starts to form. This occurs because the validation set is sampled from the underlying function of which a model is sought. Since the validation set is representative of the underlying function, the performance on this data set will worsen when a model more complex than the underlying function is formed.

There are situations, however, where it can be envisaged that early stop-

ping will fail to produce good generalization. One situation where an overly complex model may be produced using early stopping is through the formation of an overly complex model during the training process. In this case it is still possible that the validation error will fall, since *overall* the model moves closer to the validation data. As training continues the overly complex model still reduces the validation error as it learns the training data. As training continues, eventually the validation error will stop falling, and may even begin to rise. At this point, however, an overly complex model already exists and there is no way for early stopping to reduce the complexity.

One way of reducing the chance that an overly complex model is formed during the training process is to use regularization to constrain the network weights in some manner. Since regularization is present throughout the training process, the chance of producing over-complexity is reduced (assuming an appropriate level of regularization is used). A regularization function commonly used in FNNs is the sum of squares of the weight magnitudes. Additional forms of regularization that have been applied to FNNs include the Weigend–Rumelhart regularization term [4] and curvature driven regularizers [5].

The main disadvantage of regularization is the difficulty in selecting the appropriate magnitude for a given problem. One method used to set the regularization level is to train a given FNN a number of times, with each training run using a different regularization magnitude. A technique such as cross–validation can then be used to compare the trained FNNs [5]. Bayesian methods are another technique used to set the regularization magnitude [6, 7]. A simple method introduced in [4] dynamically sets the regularization magnitude during training based on the error performance of the FNN.

The third group of model selection techniques use constructive methods [8]. These methods consist of starting with a minimal size network (often with no hidden neurons), and sequentially adding weights according to some criterion.

1.2 Constructive Algorithms

There are a number of inherent advantages in using constructive algorithms over algorithms that begin training with an oversize network. First, it is often difficult to specify what size network can actually be considered an oversize network *a priori*. If the initial network selected is too small it will be unable to converge to a good solution and hence underfit the data. On the other hand, selecting an initial network that is much larger than required makes training computationally expensive. Constructive algorithms however, initially select a minimal size network and can increase the network complexity until an appropriate level is reached. In addition, constructive algorithms will spend the majority of their time training networks smaller than the final network, as compared to algorithms that start training with an oversize network.

Another advantage of constructive algorithms is that the problem of encountering poorly performing local minima is avoided. The majority of FNN training algorithms are based on gradient methods, and hence can become trapped in local minima. Constructive algorithms are able to escape from a local minimum when more weights are added to the network. This can occur because the addition of more weights increases the dimensionality of the error surface and may allow the network to continue reducing the error level.

1.3 Constructive Algorithm Properties

There are a number of properties that must be defined for constructive algorithms. These include the type of training algorithm, establishing how the new hidden neuron is connected to the current network, defining which weights are to be updated and in which order, deciding on criteria to determine when a new hidden neuron is added, deciding on criteria to halt network construction, and selecting how much regularization to use, if any. A good review of constructive algorithms is given in [8].

In general, constructive networks use gradient based training algorithms due to their convergence speed [9, 10, 11, 12]. A number of ways of connecting a new hidden neuron to the network exist. The two common methods are to construct a single layer of hidden neurons [9, 12, 13, 14] or to create a cascade of hidden neurons [10, 15]. In a cascade architecture each new hidden neuron receives inputs from all inputs and previously installed hidden neurons. Since the hidden neurons in a cascade architecture receive additional information from some non-linear combination of the inputs (implemented by previous hidden neurons), these neurons are termed higher-order neurons and are capable of performing a more complex function of the input variables. Cascade networks, while having more representational power, are more likely to overfit the data.

There are a variety of ways of training the new hidden neurons in constructive algorithms. These can be classified into two general methods. The first consists of training the whole network after the addition of a new hidden neuron [9, 12, 13, 14]. The second entails only training a subset of weights, with the remaining weights being 'frozen' [10, 16]. The advantage of this second greedy strategy is that there are far fewer weights to optimize than when all the weights are trained. The disadvantage of weight freezing is that each optimization phase is unlikely to be optimal, and this can result in larger networks than those in which all the weights are optimized [17].

The method for adding a new neuron is standard across many constructive algorithms and in general consists of either adding a new neuron when the error fails to fall by a set amount over a given period [9, 10, 13] or testing for some criterion such as a local minimum [12]. Halting network construction is equivalent to finding the best model for a given problem, and hence techniques such as early stopping are employed [18].

The ability to control the complexity of the new hidden neuron is an important issue for constructive networks in terms of convergence speed and generalization. In cascade networks the hidden neurons become more powerful higher-order neurons as the network grows. A similar effect occurs for Ridge Polynomial Networks [19] which installs ever more complex higher-

order neurons. In Projection Pursuit Learning [11] the complexity of the hidden neurons is set prior to training through the selection of the Hermite polynomial order. Having neurons of too little complexity may slow convergence, while having neurons of too high complexity can cause poor generalization. A number of approaches have been taken to control hidden neuron complexity, either applied directly to the selection of neuron complexity [20], or through the use of regularization. Bayesian methods have been used to automatically set regularization magnitudes in a constructive algorithm framework [21].

The constructive networks examined in this paper are variants of the Dynamic Network Creation (*DNC*) algorithm proposed by Ash [9]. *DNC* is a relatively simple algorithm that uses Back Propagation (*BP*) [22]. In *DNC* a sigmoid hidden neuron is added to the single layer of hidden neurons after a period of training when the error has stopped decreasing by a given amount. After the addition of the new hidden neuron, the whole network is again trained with *BP*. This algorithm has a number of advantages: its strong convergence follows directly from its universal approximation ability [8], and it has been shown to perform well on some simple classification tasks. One criticism that has been made of this type of constructive algorithm is that it does not scale up well [8] since all the network weights need to be trained after the addition of a new hidden neuron. This may be a problem for complex modelling requiring large networks.

1.4 Paper Outline

This paper explores empirically the effect of regularization on generalization in a constructive cascade network. First, the benchmarking methodology and the data sets employed are described. Next the constructive algorithm that is used as the basis for exploring constructive cascade networks is outlined. The results of different regularization terms on generalization for this algorithm are then presented. An adaptive method of selecting the regularization magnitude is introduced and compared to the use of a

fixed, user-optimized regularization setting. The algorithm incorporating this method is then compared to the Cascade Correlation (*cascor*) algorithm on the Proben 1 [23] and additional regression data sets.

2 Benchmarking Methodology

In the past there has been some criticism of FNN benchmarking methodology, and suggestions for improvement [23, 24, 25]. In order to satisfy these requirements the following methodology was used. Data sets were partitioned into training, validation, and test sets. Fifty training runs were performed for each algorithm using different initial random starting weights. Early stopping was used as the halting criterion. If parameter tuning was performed using a data set, this is mentioned. A non-parametric statistical test was used to compare results. Standard data sets were selected to allow for comparison to previously published research.

For classifications tasks the performance measure used was the percentage of patterns misclassified on the test set using the *winner-takes-all* strategy. In this strategy the output neuron with the largest activation was designated as the network classification. For regression tasks two measures were used. One was the Fraction of Variance Unexplained (FVU), which is proportional to the total sum of squares error [11]:

$$FVU = \frac{\sum_{i=1}^{N}(r(\mathbf{x_i}) - \mathbf{y}(\mathbf{x_i}))^{\mathbf{2}}}{\sum_{i=1}^{N}(r(\mathbf{x_i}) - \mathbf{\bar{r}})}, \tag{1}$$

where $\mathbf{x_i}$ is the *ith* input vector, N is the number of patterns, $r(\mathbf{x_i})$ is the target function, $y(\mathbf{x_i})$ is the FNN approximation, and $\bar{r} = \frac{1}{N}\sum_{i=1}^{N} r(\mathbf{x_i})$. The other measure was squared error percentage:

$$E_{sep} = 100.\frac{1}{N.c}E, \tag{2}$$

where N is the number of patterns and c the number of output neurons [23]. It is assumed the output patterns range in value from 0 to 1. E is the

standard sum of squares errors function, defined as

$$E = \sum_{n=1}^{N} \sum_{k=1}^{c} \{y_k(x^n; w) - r_k^n\}^2, \tag{3}$$

where r_k^n is the target output for the kth output neuron for the nth pattern, x^n is the nth input pattern, w is the current set of network weights, and y_k is the function computed by the FNN for the kth output. Two different generalization measures were employed for regression tasks since certain data sets have historically used particular measures. Employing different test measures with the appropriate data sets allows for an easy comparison of results to past research.

The non-parametric independent sample Mann–Whitney U test was used to compare results [26], as implemented in the SPSS statistical package (version 8.0 for Windows). This test is similar to the Wilcoxon test, except that it is used with unpaired data. A non-parametric test was used because a number of the results obtained were significantly non-normal in distribution, even after various transformations were attempted. Although the Mann–Whitney U test is not as powerful as the parametric t-test when the data is normally distributed, it is more powerful for non-normal data. The Mann–Whitney U test compares the median results for two data groups. The results reported for this test consist of the two tailed P value obtained and the name of the algorithm with the lower median at a 5% significance level. Results which are not significant at this level or have the same median are reported with a dash (–).

The method used for measuring the computational cost of training was connection crossings. Fahlman defines connection crossings as the number of multiply-accumulate steps required to propagate activation values forward through the network, and error values backward [27]. The computational cost of training was measured after the halting criterion was satisfied.

The method of data presentation is an important aspect of comparative benchmarking. One good method for displaying results is to use a box-and-whisker plot. The length of the box is the inter-quartile range (IQR). The

median is represented by a line within the box. The whiskers extend to 1.5 IQR from the box edges. Outliers are data points which lie 1.5 to 3 IQR away from the box edges. Extreme data points lie over 3 IQR away from the box edges. Outliers are represented by an o and extreme points by a $*$.

The data sets used in the benchmarking can be broken up into three groups: the Proben 1 [23], additional regression, and Two Spirals data sets. The Proben 1 data sets are a collection of 'real world' data sets and consist of 10 classification and 4 regression tasks. All these data sets have inputs and outputs scaled between 0 and 1. For classification tasks with multiple classes, the outputs are encoded via a *1-of-c* coding scheme, where the output for the class to be learnt is set to 1, and all other class outputs are set to 0. In Proben 1 there are three versions of each data set. These versions are obtained from variations in the partitioning between training, validation, and test set patterns. For all benchmarking done only the first version of each data set is used. The characteristics of these data sets are given in Table 1.

Table 1: Proben 1 Data Set Attributes

Data Set	Inputs	Outputs	Training	Validation	Test
Cancer	9	2	350	175	174
Card	51	2	345	173	172
Diabetes	8	2	384	192	192
Gene	120	3	1588	794	793
Glass	9	6	107	54	53
Heart	35	2	460	230	230
Heartc	35	2	152	76	75
Horse	58	3	182	91	91
Soybean	82	19	342	171	170
Thyroid	21	3	3600	1800	1800
Building	14	3	2104	1052	1052
Flare	24	3	533	267	266
Hearta	35	1	460	230	230
Heartac	35	1	152	76	75

The second group of data sets is a series of five regression data sets of varying complexity, as originally described in [11]. The five regression data sets were generated from the Simple Interaction, Radial, Complex Additive, Harmonic, and Complex Interaction functions respectively. These functions are described below:

- Simple interaction function (Sif)

$$f^{sif}(x_1, x_2) = 10.391((x_1 - 0.4)(x_2 - 0.6) + 0.36). \quad (4)$$

- Radial function (Rad)

$$\begin{aligned} f^{rad}(x_1, x_2) &= 24.234(r^2(0.75 - r^2)), \quad (5) \\ r &= (x_1 - 0.5)^2 + (x_2 - 0.5)^2. \end{aligned}$$

- Complex additive function (Cadd)

$$\begin{aligned} f^{cadd}(x_1, x_2) = 1.3356(&1.5(1 - x_1) \\ &+ e^{2x_1 - 1} sin(3\pi(x_1 - 0.6)^2 \\ &+ e^{3(x_2 - 0.5)} sin(4\pi(x_2 - 0.9)^2)). \end{aligned} \quad (6)$$

- Harmonic function (Harm)

$$f^{harm}(x_1, x_2) = 42.659(0.1 + \tilde{x}_1(0.05 + \tilde{x}_1^4 - 10\tilde{x}_1^2\tilde{x}_2^2 + 5\tilde{x}_2^4)), \quad (7)$$

  where $\tilde{x}_1 = x_1 - 0.5$ and $\tilde{x}_2 = x_2 - 0.5$.

- Complex interaction function (Cif)

$$\begin{aligned} f^{cif}(x_1, x_2) = 1.9(&1.35 + e^{x_1} sin(13(x_1 - 0.6)^2) \\ &\times e^{-x_2} sin(7x_2)). \end{aligned} \quad (8)$$

For each data set a training set of 225 randomly selected points over the input space $[0, 1]^2$ was generated. A validation set of size 110 was similarly created. A test set of size 10,000 was generated by uniformly sampling the

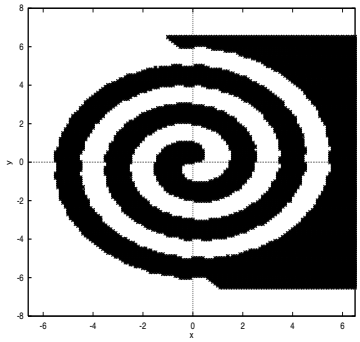

Figure 1: The Two Spirals test set as generated by nearest neighbor classification.

grid $[0,1]^2$. Gaussian noise with zero mean and 0.25 variance was added to both the training and validation sets.

The final data set was the Two Spirals data set originally proposed by Alexis Wieland of the MITRE Corporation. This data set consists of two interlocked spirals, each made up of 97 points, which the FNN must learn to distinguish. Each training pattern consists of two inputs (the x,y coordinates), and a single output (the spiral classification). For this benchmarking, a test set was generated by uniformly sampling 17,161 points over the input space $[-6.5, 6.5]^2$. These points were classified using nearest neighbor classification, with the training data as the templates, as done in [28]. A plot of the results obtained by the nearest neighbor classification is illustrated in Figure 1, in which an intuitively good separation of the spirals is obtained. The validation set was made up of two parts. The first consisted of two additional spirals of 96 points each, rotated slightly relative to the original spirals. The validation set also included an additional 200 points generated at random and classified using the same nearest neighbor classification used to create the test set. The second part of the validation set was added because the first part had little variation from the training data, a factor which can lead to poor generalization.

3 A Basic Constructive Cascade Network

The constructive algorithm described in this section forms the basis for the series of empirical studies performed, and is termed *casbias*. While this algorithm is based on the *DNC* algorithm, it has a number of significant differences.

The training algorithm used in *casbias* is the *RPROP* algorithm [29]. *RPROP* was selected for a number of reasons. First, it is a gradient based method which has fast convergence properties compared to many other gradient based algorithms, such as *BP* and its variants [29]. A further advantage of *RPROP* is that there are few parameters that are required to be set by the user. In addition, *RPROP's* performance is relatively insensitive to the values selected. The *RPROP* algorithm used the following standard constant settings [29]: $\eta^+ = 1.2, \eta^- = 0.5, \Delta_{max} = 50$, $\Delta_{min} = 10^{-6}$.

The *casbias* algorithm does not perform weight freezing, but trains all weights after the addition of a new hidden neuron. The network architecture constructed is a cascade. Training is begun in *casbias* with an initial network that has no hidden neurons, with the inputs connected directly to the outputs. These weights are initialized to random values in the range -0.7 to 0.7. Once a new hidden neuron is connected to the network, training is resumed using *RPROP*. The weights of the new hidden neuron are initialized to random values in the range -0.1 to 0.1.

The remaining *RPROP* parameter is the initial update value Δ_0, which sets the initial step size taken by the weights. For the initial network this value is set to 0.2. When a new hidden neuron is added to the network, the updates values are reset to values depending on their position in the network. This technique has been shown to increase convergence speed and is termed search direction biasing [30, 31]. In this constructive technique, the weights are separated into three different groups $L1$, $L2$ and $L3$, each with their own initial update value. The group $L1$ consists of all weights that are inputs to the new hidden neuron. The group $L2$ consists of all weights that are outputs from the new hidden neuron (and hence are connected to

the output neurons), and the group $L3$ consists of all the remaining weights, being all weights connected to, and coming from, the old hidden and output neurons.

In order to bias the search direction of the new weights in *casbias*, whenever a new hidden neuron is added to the network the values of $L1$, $L2$ and $L3$ are set such that $L1 >> L2 > L3$. The effect of these settings is to initially ensure that the weights connected to the new hidden neuron are able to rapidly learn the remaining error, without excessive interference from the old weights, which have a small Δ_0. During early training periods the old weights are thus effectively frozen. Since *RPROP* is an adaptive algorithm, the update values of the old weights can grow as training continues, and hence can have a greater participation in the reduction of network error as training proceeds. This technique has a similar effect to the use of weight freezing and the correlation measure in *cascor* [10], except that since all weights are still are to learn, smaller networks are often constructed [30]. The values selected for $L1$, $L2$, and $L3$ were 0.2, 0.005, and 0.001 respectively. These values were chosen after some initial tuning using the Two Spirals data set.

A simple criterion was selected to decide at which point to add a new neuron in *casbias*. When the RMS error on the validation set failed to decrease by a set amount ϵ, over a given period δ, a new neuron was added. This criterion was tested every δ epochs. A minimum training time of δ was specified for each hidden neuron to allow for a reasonable training time with the new network. This criterion is described below:

$$E_{RMS}(t+\delta) - E_{RMS}(t) < \epsilon, \qquad t = \delta, 2\delta, 3\delta \ldots, \tag{9}$$

where $E_{RMS}(t)$ is the RMS error at epoch t. The ϵ parameter was set to 0.01 and the δ parameter was set to 100 and 200 for classification and regression data sets respectively. These values were chosen after some initial tuning using the Two Spirals and Cif data sets.

Another simple criterion was chosen to halt network construction. The

halt criterion was met if either the validation error, measured after the installation of each hidden neuron, failed to decrease after the addition of η hidden neurons, or a maximum of κ hidden neurons were installed. Both the criteria for adding a new hidden neuron and halting network construction implement a form of early stopping that has proven to be a useful method for obtaining good generalization [3].

In *casbias* a *1-of-c* coding scheme for c classes is used for classification tasks. For a two class classification task, a single output is used with the values 1 and 0 representing the two classes. In this case an output value within 0.4 of the target value represents the designated class. For multiple classes a *winner-takes-all* strategy is used.

All hidden neurons use a symmetric sigmoid activation function ranging between -0.5 and 0.5. The output neuron activation function depends on the type of analysis performed. Regression problems use a linear activation function. Classification tasks use the sigmoid function for single output classification tasks. Classification tasks with multiple outputs use the softmax activation function [32]. The softmax function ensures that the total output neuron activation sums to one. This is useful in classification tasks when combined with *1-of-c* output encoding as it allows the output neurons to represent the probability that a particular class is represented by a particular neuron.

The error function used in *casbias* depends on the problem. Regression problems use the standard sum of squares error function. Classification problems use the cross–entropy error function [5]. For a single output classification task this is defined as:

$$E_{ces} = -\sum_{n=1}^{N} \left\{ r^n \ln y^n + (1 - r^n) \ln(1 - y^n) \right\}, \tag{10}$$

where r^n is the target output for the nth pattern and y_n is the network output for this pattern. The cross–entropy function for multiple classes c is

defined as:

$$E_{cem} = -\sum_{n=1}^{N}\sum_{k=1}^{c} r_k^n \ln\left(\frac{y_k^n}{r_k^n}\right). \tag{11}$$

The use of the sum of squares error function with a linear output neuron is standard for regression problems in FNNs and can be derived from the principle of maximum likelihood [5]. The use of the entropy error function with the sigmoid and softmax output activation functions is an appropriate choice to allow the network outputs to be used to model the posterior probability of class membership [5].

The *casbias* constructive algorithm has a number of important properties. First, it maintains the property of being a universal approximator in the case of regression analysis where linear output neurons are used. This property has been proved for an FNN with a single layer of sigmoid neurons [33]. It can be seen that the cascade network produced by the *casbias* algorithm satisfies this property since the additional weights that form the cascade and direct input to output connections can be set to zero to obtain a single layer network. Given this universal approximation ability, strong convergence follows directly for this type of constructive network, as noted in [8].

4 Regularization Variations

4.1 Incorporating *SARPROP*

In order to test the effect of regularization on a constructive algorithm, the *SARPROP* regularization term was added to the *casbias* algorithm. The *SARPROP* algorithm is an FNN training algorithm based on *RPROP* which has good convergence properties [34]. *SARPROP* incorporates regularization of the form:

$$\tilde{E} = E + \frac{\lambda}{2} S \sum_{ij} \ln(1 + w_{ij}^2), \tag{12}$$

where $S = 2^{\frac{-Epoch}{T}}$ is a Simulated Annealing (SA) term, T is the temperature constant, and *Epoch* is the number of epochs used by the training algorithm. This results in the following modification to the error gradient:

$$\frac{\partial \tilde{E}}{\partial w_{ij}} = \frac{\partial E}{\partial w_{ij}} + \lambda S \frac{w_{ij}}{1 + w_{ij}^2}. \tag{13}$$

The algorithm incorporating regularization of this form is termed *cassar*. In *cassar* the *Epoch* parameter within the SA term is reset to zero after the addition of each new hidden neuron. This reinitializes the amount of regularization to its original high level each time a new hidden neuron is added to the network.

The effect of the SA term is to reduce the amount of regularization as training proceeds. This was initially done in *SARPROP* to encourage exploration of the error surface. An additional effect of this term is that it provides a high initial regularization level to the network each time a new hidden neuron is added. This forces the network to fit the general characteristics of the underlying function. This fit can later be refined as the regularization level is reduced [34]. The use of early stopping should provide a good method to prevent the refinement process from overfitting the data.

An addition was also made to *cassar* to incorporate *SARPROP's* training algorithm. *SARPROP* uses a noise term that is added to the adapted update value of *RPROP* [34]. This was done to enhance *RPROP's* convergence speed and reliability. *SARPROP* was shown to generally converge faster and more reliability for a number of benchmark data sets.

The *SARPROP* training algorithm, however, was not used to train all the weights. This was done to stop *SARPROP* interfering with the search direction biasing performed in *cassar*. In *cassar* the *SARPROP* training method was used to train only the $L1$ weights, with the remaining weights in groups $L2$ and $L3$ continuing to be trained by *RPROP*. This was done to stop the $L2$ and $L3$ weights from being disturbed by the noise used in *SARPROP*. The $L2$ and $L3$ weights have relatively small initial update values, which results in these weights being virtually frozen in the initial training stages.

If *SARPROP* was used to train these weights, this effect would be disturbed through the addition of noise to the update values.

4.2 Additional Regularization Terms

The *SARPROP* regularization term was initially proposed to increase convergence speed. It may not, however, give the best generalization results when applied in the *cassar* algorithm. One reason for this may be that *cassar* constructs a cascade architecture. Cascade architectures result in the production of higher-order neurons, which while often allowing for faster convergence, may also result in overly complex models. The *cassar* algorithm may therefore benefit from a regularization term which penalizes complexity in a different manner.

In order to test this, two additional regularization terms were examined. The first was the standard sum of squared weights term which replaced the $ln(1 + w_{ij}^2)$ term in Equation 12. This gives:

$$\tilde{E} = E + \frac{\lambda}{2} S \sum_{ij} w_{ij}^2, \tag{14}$$

which results in the following modification to the gradient:

$$\frac{\partial \tilde{E}}{\partial w_{ij}} = \frac{\partial E}{\partial w_{ij}} + \lambda S w_{ij}^2. \tag{15}$$

The algorithm using this regularization term is referred to as *casdec2*. The second regularization function examined employed a cubic term:

$$\tilde{E} = E + \frac{\lambda}{3} S \sum_{ij} |w_{ij}^3|, \tag{16}$$

which results in the following modification to the gradient:

$$\frac{\partial \tilde{E}}{\partial w_{ij}} = \frac{\partial E}{\partial w_{ij}} + \lambda S w_{ij}^2 sign(w_{ij}), \tag{17}$$

where *sign* returns the sign (positive or negative) of its argument. The algorithm using this form of regularization is termed *casper* [30, 31]. The only difference between the *cassar*, *casdec2*, and *casper* algorithms is the regularization term used. All other parameters remain the same.

5 Comparative Simulations

In order to test the effects of early stopping, and early stopping combined with regularization of various forms, the *casbias*, *cassar*, *casdec2*, and *casper* algorithms were compared on the Two Spirals, Glass, Cadd, and Cif data sets. Importantly, the same halting criterion was used for all algorithms: training was continued until the validation error failed to decrease after the addition of a set number of hidden neurons η, or a maximum number of κ hidden neurons were installed. The values for η and κ were set to 6 and 30 respectively for all data sets, except for the Glass data set which used a κ of 8. A κ of 8 was used for all Proben 1 data sets henceforth because these data sets are generally simple and so do not require a larger κ value. After the halting criterion was reached, the results on the test set are reported at the point where the best validation results were obtained.

Since each different regularization term penalizes weights differently, the magnitude of the λ parameter may have a significant effect on the resulting network generalization. In order to give a fair comparison, a range of λ values were used for those algorithms employing regularization. Setting $\lambda = 10^{-\alpha}$, the range of α values used was 1 to 5. After some initial tuning using the Two Spirals data set, the temperature parameter T in the SA term was set to the constant value of 100 for all algorithms employing regularization.

The test set results of the best regularization level for each algorithm for each data set are illustrated in Figures 2 to 5. The algorithm with the lowest median value on the test set was designated as the best (if the medians were equal, the averages were used). The results for each algorithm over the range of regularization magnitudes examined are given in the appendix.

6 Discussion

The best results given by each regularization term are compared to the *casbias* results in Table 2, and to each other in Table 3. Both comparisons

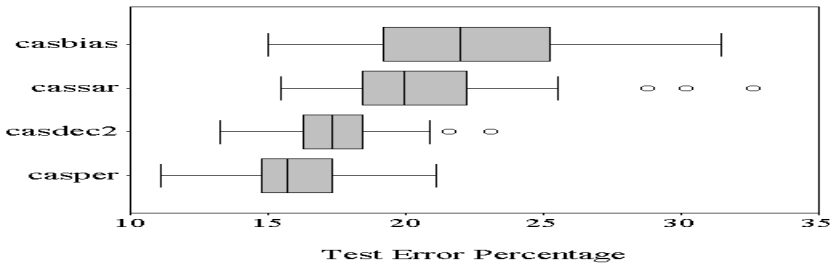

Figure 2: Two Spirals results for the best regularization levels.

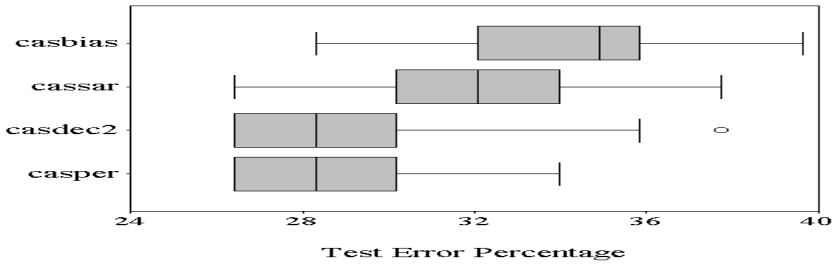

Figure 3: Glass results for the best regularization levels.

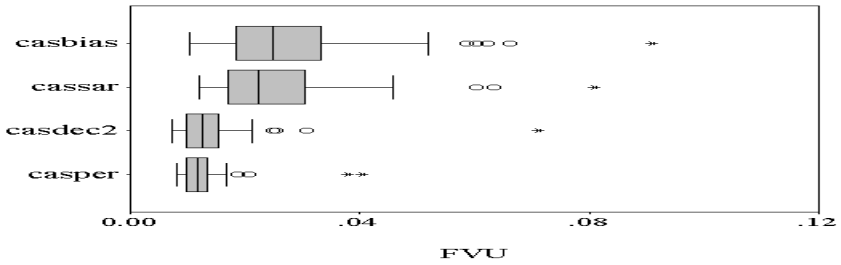

Figure 4: Cadd results for the best regularization levels.

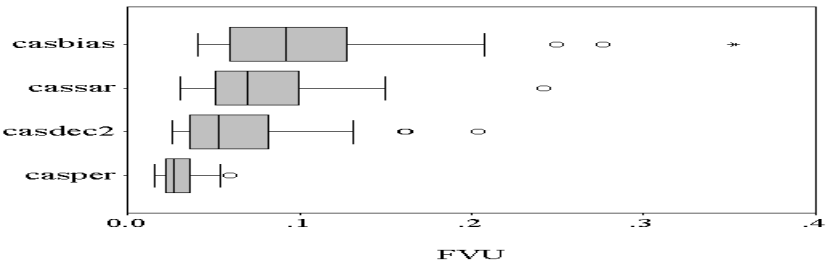

Figure 5: Cif results for the best regularization levels.

Table 2: U Test Comparison of *casbias* to *casper*, *casdec2*, and *cassar*

Data Set	*casbias* vs *casper*	*casbias* vs *casdec2*	*casbias* vs *cassar*
Two Spirals	*casper* (0.000)	*casdec2* (0.000)	*cassar* (0.050)
Glass	*casper* (0.000)	*casdec2* (0.000)	*cassar* (0.000)
Cadd	*casper* (0.000)	*casdec2* (0.000)	– (0.217)
Cif	*casper* (0.000)	*casdec2* (0.000)	*cassar* (0.011)

are made using the Mann–Whitney U test. The combination of regularization and early stopping consistently outperformed early stopping alone. The *casbias* algorithm gives worse generalization results than the algorithms employing regularization over a range of regularization terms and magnitudes. This is confirmed in Table 2 where the regularization terms were found to give significantly better test results for all but one of the data sets.

This result can be explained by noting that the formation of an overly complex model can still occur when early stopping is employed, as discussed in the introduction. This problem is likely to be exaggerated in cascade networks which make use of higher-order neurons. Regularization is able to address this problem by constraining weight magnitudes, thereby hindering the production of over-complexity.

In terms of comparison between regularization terms, both *casdec2* and *casper* give better generalization results than *cassar* over a number of reg-

Table 3: U Test Comparison of *casper*, *casdec2*, and *cassar*

Data Set	*casper* vs *casdec2*	*casper* vs *cassar*	*casdec2* vs *cassar*
Two Spirals	*casper* (0.000)	*casper* (0.000)	*casdec2* (0.000)
Glass	– (0.462)	*casper* (0.000)	*casdec2* (0.000)
Cadd	– (0.250)	*casper* (0.000)	*casdec2* (0.000)
Cif	*casper* (0.000)	*casper* (0.000)	*casdec2* (0.011)

ularization magnitudes. Similarly, *casper* often gives better generalization than *casdec2*. These observations are supported by the statistical comparisons given in Table 3. The effect of the regularization terms is to constrain weight magnitudes, and therefore to reduce the complexity of the higher-order neurons. The difference between the regularization terms is the amount of constraint they place on large weights.

The *cassar* term has the least effect on large magnitude weights because of its use of the *log* term. The *casdec2* term places greater constraint on large weights because of the square term, while the *casper* term places the most constraint through the use of a cubic term. Since cascade architectures are more likely to produce overly complex models, they benefit from regularization terms that greatly penalize large weights. Some preliminary simulations using a quartic regularization term confirmed this trend, however this form of regularization slowed convergence, often causing the construction of excessively large networks.

One difficulty introduced with regularization is the selection of the magnitude to be used. Varying this magnitude can affect generalization, with the optimal value depending on the function to be modelled, the amount of noise in the data set, and the final size of the network. This difficulty not withstanding, *casper* produces good generalization over a number of regularization magnitudes. An explanation for this is that since the training algorithm used is constructive, the network can compensate for a range of regularization magnitudes by increasing network complexity through the

addition of further hidden neurons. Since early stopping is also employed, network construction is halted when the complexity reaches the level necessary to produce a good model.

7 Adaptive Regularization

One method that would allow the amount of regularization to be automatically selected in constructive algorithms is to adapt this parameter as the network is constructed [35, 36]. The adaptation process works by noting the validation results on a range of regularization levels. When a new neuron is added to the network, the range of regularization levels used for the new network is modified based on the previous validation results.

This technique was applied to the *casper* algorithm as follows. The adaptation process relies on using three training stages for each new hidden neuron added to the network, instead of the usual single training stage. Each training stage uses a different regularization level. The validation results taken after the completion of each training stage are then used to adapt the regularization range for future network training. This process repeats as the network is constructed.

Each training stage is performed using the same method and is halted using the same criterion. The commencement of a new training stage results in all *RPROP* and SA parameters being reset to their initial values. Importantly, however, the final weights from the previous training stage are retained and act as the starting point for the next training stage. Since previously learnt information is retained, this technique is likely to increase convergence speed and thereby produce smaller networks.

The regularization level for the network when a new hidden neuron is added is set to the initial value λ_i, termed the initial decay value. It is this initial decay value that is adapted as the network is constructed. The first training stage uses the initial decay value. Each successive stage uses a regularization level that has been reduced by a factor of ten from the previous stage. After

each training stage the performance on the validation set is measured. The network weights are recorded at this point if the validation error is lower than previous stages. On completion of the third training stage, the initial decay value is adapted as follows: if the best-performing regularization level occurred during the either of the first two training stages, the initial decay value is increased by a factor of ten, otherwise it is decreased by a factor of ten. At this point the weights that produced the best validation results are restored to the network. This process repeats for the next hidden neuron and uses the newly adapted initial decay value.

The regularization level takes the form $\lambda = 10^{-\alpha}$. The initial network with no hidden neurons is trained using a single training stage with a regularization level of $\alpha = 0$. The adaptation scheme begins with the addition of the first hidden neuron, which is given an initial decay value of $\alpha = 2$. The limits placed on the initial decay value are $\alpha = 1$ to 4, which gives a total possible regularization range of $\alpha = 1$ to 6 (since there are three training stages). The lower initial decay limit ($\alpha = 4$) was selected to stop the regularization level falling too low, which can occur in early stages of training when the network is still learning the general features of the data set. The top initial decay limit ($\alpha = 1$) was selected because convergence becomes difficult with excessive regularization levels.

For reasons of efficiency, if the validation result of the second stage is worse than the first, the third training stage is not performed. In addition, if the validation results of the first training stage are worse than the best validation results of the previous network architecture, the weights are reset to their values before this training stage was commenced. The regularization level is then reduced as normal, and the second training stage is started. This was done to stop excessive regularization levels distorting past learning. The pseudo code for the adaptive regularization algorithm is given in Figure 6.

This regularization selection method allows the network to adapt the level of regularization as the network grows in size. The motivation for using this adaptation scheme is the relationship between good regularization levels in similar size networks. By finding a good regularization level in a given

network, it is likely that a slightly larger network will benefit from a similar regularization level. The adaptation process allows a good regularization level to be found by modifying the range of regularization magnitudes that are examined. The parameter values for this algorithm were selected after some initial tuning on the Two Spirals, Cancer, and Cif data sets.

8 Comparative Simulations

In order to test the performance of the adaptive regularization method, *acasper* was compared against *casper* on the four benchmark data sets used previously. Two additional data sets were introduced to extend the comparison: the Thyroid and Harmonic (Harm) data sets.

The same benchmarking set-up was used as previously described. In order to obtain good generalization results for *casper*, the regularization magnitude was user-optimized. This was achieved by training *casper* with a range of regularization values and picking the one with the best median performance on the validation set, using α settings in the range 1 to 5. The halting criterion was the same as described previously for both the *acasper* and *casper* algorithms. The results on the test sets at the point where the best validation result occurred after the halting criterion was satisfied are illustrated in Figures 7 to 12. Figures 13 to 18 illustrate the number of hidden neurons installed at the point where the best validation result occurred. The results of the Mann–Whitney U test comparing *casper* and *acasper* are given in Table 4.

Table 4: U Test Comparison of *casper* to *acasper*

Data Set	Test	Hidden Neuron
Two Spirals	– (0.114)	– (0.538)
Glass	*casper* (0.001)	*casper* (0.014)
Thyroid	– (0.758)	*acasper* (0.000)
Cadd	– (0.831)	*acasper* (0.000)
Harm	*acasper* (0.000)	*acasper* (0.000)
Cif	*acasper* (0.001)	*acasper* (0.001)

```
train initial network (λ = 1.0)
store weights
VE_oldnet ⇐ validation error
λ_i ⇐ 0.01
repeat
   add new hidden neuron; λ ⇐ λ_i
   VE_low ⇐ maximum error
   for 3 training stages do
      train network using λ
      VE ⇐ validation error
      if 1st training stage and VE > VE_oldnet then
         restore weights
      else if VE ≤ VE_low then
         store weights
         VE_low ⇐ VE
      else if 2nd training stage and VE > VE_low then
         do not do 3rd training stage
      end if
      λ ⇐ λ × 1/10
   end for
   if VE_low in 1st or 2nd training stage then
      λ_i ⇐ λ_i × 10
   else
      λ_i ⇐ λ_i × 1/10
   end if
   check λ_i limits
   restore weights
   VE_oldnet ⇐ VE_low
until halting criterion met
```

Figure 6: The adaptive regularization algorithm.

Figure 7: Two Spirals test results.

Figure 8: Glass test results.

Figure 9: Thyroid test results.

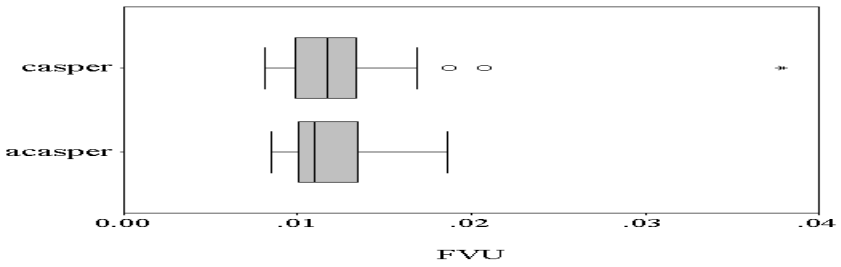

Figure 10: Cadd test results.

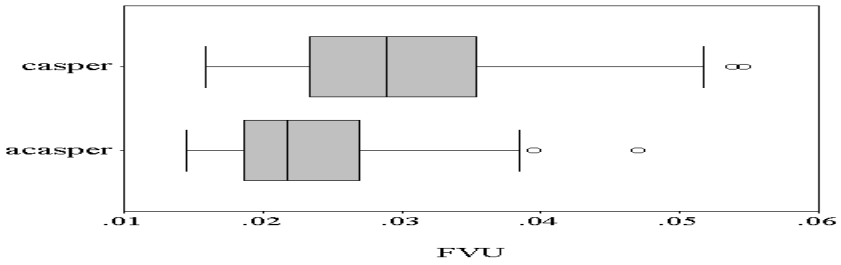

Figure 11: Harm test results.

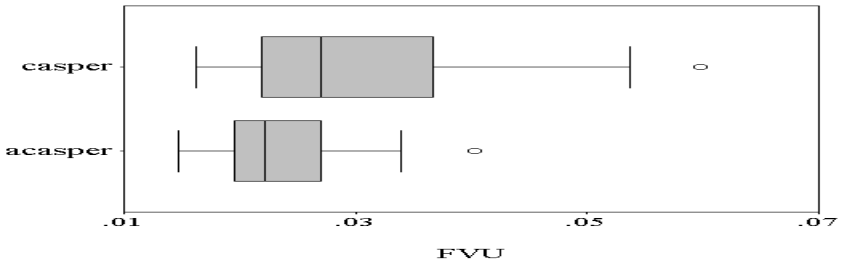

Figure 12: Cif test results.

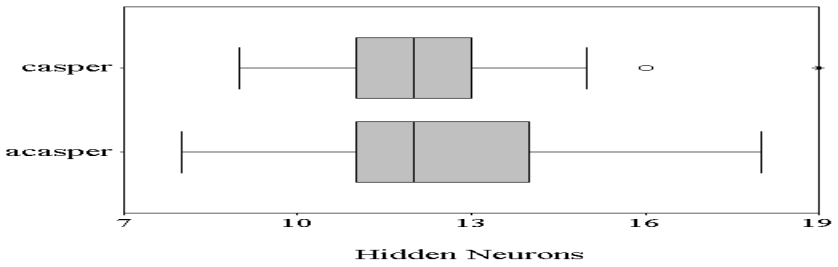

Figure 13: Two Spirals hidden neuron results.

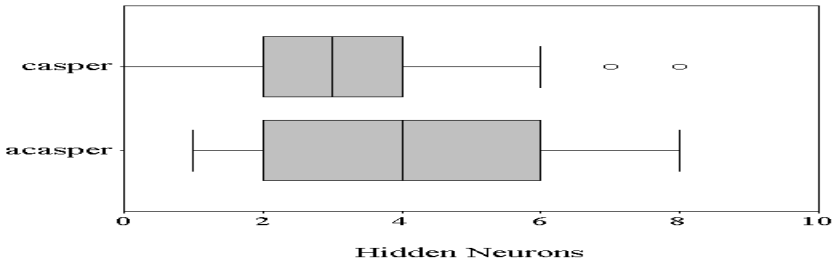

Figure 14: Glass hidden neuron results.

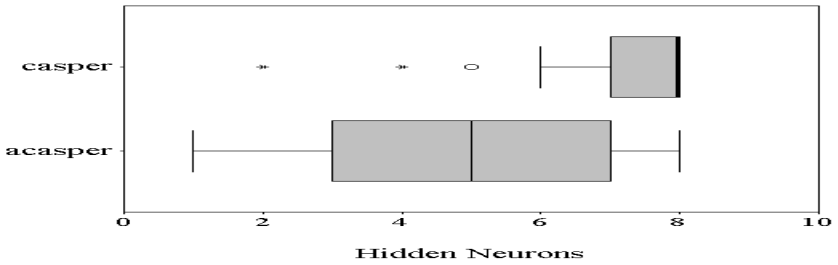

Figure 15: Thyroid hidden neuron results.

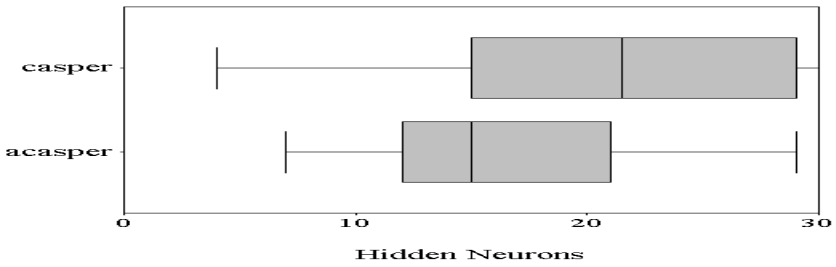

Figure 16: Cadd hidden neuron results.

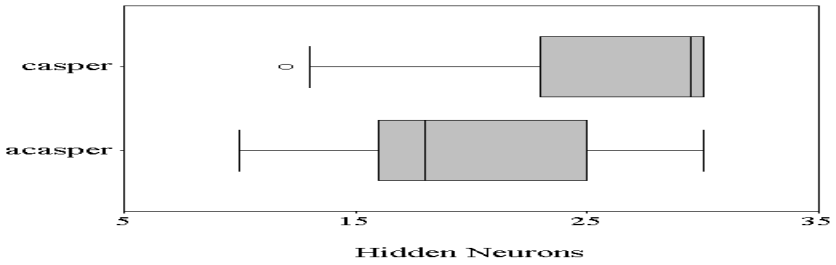

Figure 17: Harm hidden neuron results.

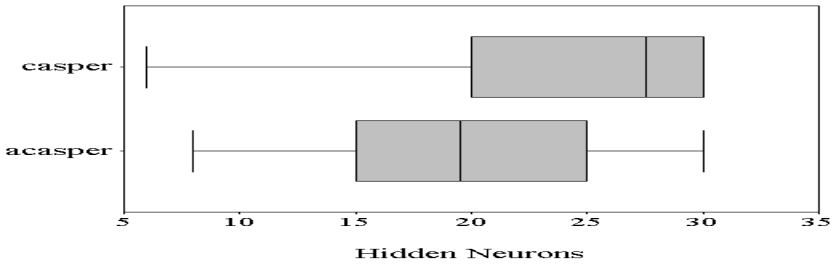

Figure 18: Cif hidden neuron results.

9 Discussion

The *acasper* algorithm obtains generalization results similar to a user-optimized *casper* algorithm. There are some data sets for which *acasper* obtains significantly better results, such as the Cif and Harm Data sets. The *acasper* algorithm performs significantly worse on the Glass data set. It is difficult to say from this benchmarking whether *acasper* performs better or worse on any particular kind of data set. Additional benchmarking was not performed because of the high computational cost involved in optimizing the regularization level for *casper*. An initial observation is that *acasper* seems to perform better on data sets which require the construction of larger networks, such as the Cif and Harm data sets. One reason for this may be that larger networks allow greater time for the adaptation process to refine the regularization level. Another interpretation is that *acasper* performs worse on data sets with high dimensional outputs, although this is not supported by the Thyroid data set in which there is no significant difference in test set results.

The good performance of *acasper* can be attributed to its ability to adapt the regularization level by taking into account such factors as current network size and the presence of noise. Figure 19 demonstrates *acasper's* ability to adapt regularization levels depending on the noise present in the data. This figure shows an example of the λ values selected by *acasper* on the Cif data set with and without added noise. The regularization magnitudes selected for the noisy data set become greater as training proceeds, and are successful in preventing the network overfitting the data.

In terms of the network size constructed, there are two trends which can be observed. First, for data sets that *casper* solves with relatively small networks, *acasper* produces similar size networks. For data sets that result in *casper* constructing larger networks, *acasper* is able to produce smaller networks. For example, with the Cadd, Cif, and Harm data sets *acasper* constructs significantly smaller networks than *casper*. The reason why *acasper* constructs smaller networks for more complex data sets is that the *acasper*

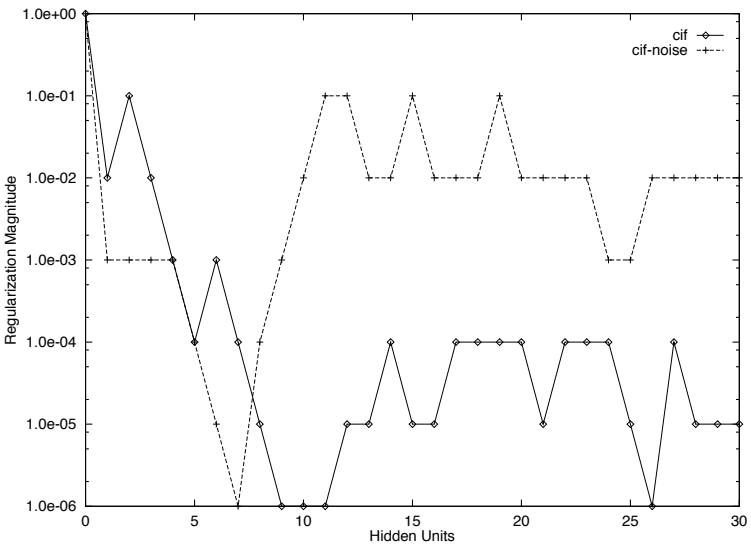

Figure 19: Regularization magnitudes selected by *acasper*.

algorithm performs more training at each period of network construction. This takes the form of restarting training with different regularization levels and with reset *RPROP* and SA parameters. This increases the chance of the network escaping from local minima or areas of slow convergence.

The adaptive regularization technique has the advantage of being applicable to the general class of constructive algorithms, and need not only be limited to *acasper*. The technique does not use any training algorithm specific information, but only requires that a number of training runs are performed at each stage of network construction. It would be an easy matter to incorporate adaptive regularization in such constructive FNNs as Ash's *DNC* and Fahlman's *cascor* algorithms.

The main disadvantage of the adaptive regularization method is the increase in computational cost. For the benchmark results obtained, the increase in connection crossings is in the range of two to four on average in comparison to *casper*. The increase in computational cost scales approximately linearly in comparison to corresponding size networks trained by *casper*. This is expected, since it is a result of at most three additional training stages at

each point of network construction. Part of the increased cost of training *acasper* may be balanced by its ability to construct smaller networks than *casper* for some cases. The use of adaptive regularization also removes the need to select a regularization level in *casper*. The computational cost of such optimization is significant.

10 Benchmarking *acasper*

In order to allow for comparison between *acasper* and other FNN algorithms, an additional series of benchmarking was performed on the remaining Proben 1 and regression data sets. The benchmarking set-up was the same as used previously.

The results for *cascor* on the Proben 1 data sets are also reported, having been taken directly from Prechelt's benchmarking [18]. The raw results were obtained via anonymous FTP from ftp.ira.uka.de in directory /pub/neuron as file nndata.tar.gz. Prechelt's version of *cascor* implements the *cascor* algorithm of [27], but also incorporates a sophisticated form of early stopping for both candidate neuron training and halting network construction [18]. A pool of eight candidate neurons was used. In these results, the performance on the test set is reported at the point where the best validation result was obtained when Prechelt's early stopping criterion was satisfied. The size of the constructed network at this point is also reported.

For the benchmarking of the Sif, Rad, Cadd, Harm, and Cif regression data sets, Crowder's C implementation (version 1.33: April 16, 1992) of Fahlman's original *cascor* algorithm was used (available via anonymous ftp from ftp.cs.cmu.edu in directory /afs/cs/project/connect/code). For this benchmarking *cascor* used the same criterion as *acasper* for halting network construction, and for measuring test error and network size. A pool of eight candidate neurons was used.

The *cascor* algorithm was tested on a range of parameter values for the Sif, Rad, Cadd, Harm, and Cif data sets in order to obtain the best possible

results. The parameters which were varied were the input and output *patience*. The *patience* parameters affect how long training is continued on the candidate neurons and output network. These parameters were given equal values, and training was repeated using five different settings: 25, 50, 100, 150, and 200. The results reported are for the parameter settings which gave the lowest average validation error after the halting criterion was satisfied. The best parameter settings were found to be 25, 25, 50, 100, and 50 for the Sif, Rad, Cadd, Harm, and Cif data sets respectively.

Table 5 gives the results of the Mann–Whitney U test comparison between *acasper* and *cascor* for test set performance and the number of hidden neurons installed for all data sets. The actual results obtained for each algorithm are given in the appendix. For the Proben 1 data sets, there are eight data sets where *acasper* obtains significantly better test results than *cascor*, compared to two where *cascor* outperforms *acasper* (four with no significant difference). For the remaining regression data sets, *acasper* always obtains significantly better results than *cascor*, with the magnitude of improvement often being large. For example, on the Harm data set *acasper* obtains an average FVU of 2.37, compared to 22.48 for *cascor*. There are a number of possible reasons for *acasper's* better test set performance in comparison to *cascor*. First, *acasper* employs the combination of early stopping and regularization, while *cascor* uses early stopping alone. The advantage of combining early stopping with regularization was shown previously.

A second reason for the poor generalization of *cascor* was pointed out in [37], which noted that the correlation measure in *cascor* forces the hidden neurons to saturate in order to obtain a large correlation value. This forces the hidden neurons to have large inputs weights. The effect of large weights in the network is to make the outputs *jagged* in appearance. This may not degrade results for some classification problems which require a definite boundary between classes. Regression problems, however, generally require the fitting of smooth functions, and so producing jagged outputs can result in poor generalization. This effect is reflected in the results obtained, with *acasper* obtaining its largest improvements over *cascor* on the Sif, Rad,

Table 5: U Test Comparison of *acasper* to *cascor*

Data Set	Test Results	Hidden Neuron
Cancer	– (0.692)	– (0.466)
Card	– (0.295)	*acasper* (0.000)
Diabetes	*acasper* (0.000)	*acasper* (0.000)
Gene	*acasper* (0.000)	*acasper* (0.000)
Glass	*acasper* (0.000)	*acasper* (0.000)
Heart	*acasper* (0.001)	*acasper* (0.000)
Heartc	– (0.000)	*acasper* (0.000)
Horse	*cascor* (0.000)	*acasper* (0.000)
Soybean	*acasper* (0.000)	*acasper* (0.000)
Thyroid	*acasper* (0.000)	*acasper* (0.000)
Building	*acasper* (0.000)	– (0.844)
Flare	*acasper* (0.017)	*acasper* (0.000)
Hearta	*cascor* (0.000)	*acasper* (0.000)
Heartac	– (0.937)	*acasper* (0.000)
Sif	*acasper* (0.000)	*acasper* (0.000)
Rad	*acasper* (0.000)	*acasper* (0.000)
Cadd	*acasper* (0.000)	*cascor* (0.002)
Harm	*acasper* (0.000)	– (0.070)
Cif	*acasper* (0.000)	– (0.917)

Cadd, Harm, and Cif regression data sets.

The one data set where *cascor* obtains a noticeably better result than *acasper* is the Horse data set. The average test error rate for the Horse data set is 26.37% and 32.46% for *cascor* and *acasper* respectively. This data set is solved with a very simple model, with both *acasper* and *cascor* often using no hidden neurons at all. It is also very susceptible to overfitting early in training, and this is the reason for *acasper's* poor performance. The *acasper* algorithm has a specified minimum training time for each training stage. It was found that overfitting was occurring in *acasper* before this minimum training time had expired and before the addition of any hidden neurons. Poor generalization results were thus obtained. The *cascor* algorithm on

the other hand, had no such minimum training time and so was able to halt training before overfitting occurred.

The *acasper* algorithm was able to produce smaller networks than *cascor* on average for all of the Proben 1 data sets, with significant results for twelve out of the fourteen data sets. There are some cases where the difference is surprisingly large, for example the Soybean and Thyroid data sets. One reason for this may be that unlike *cascor*, *acasper* does not use weight freezing. The *acasper* algorithm is therefore able to converge more quickly, and so produce smaller networks.

Another reason for *acasper* obtaining smaller networks may be that the halting criterion for *acasper* specifies a maximum network size of eight for the Proben 1 data sets. There is no maximum size specified for *cascor*, which only uses the validation results to halt training [18]. This limit in *acasper* does not affect the results significantly however, since it was rarely reached during the benchmarking. The limit was not reached at all for nine of the Proben 1 data sets, and only rarely for the remaining five.

The fact that *acasper* is able to solve the majority of the Proben 1 data sets using simple networks, often with no hidden neurons at all, illustrates a major advantage of using constructive networks: the simple solutions are tried first. It is often the case that many real world data sets, such as the ones in Proben 1, can be solved by relatively simple networks.

For the Sif, Rad, Cadd, Harm, and Cif regression data sets the trend in network size is less clear. The *acasper* algorithm constructed significantly smaller networks for the Sif and Rad data sets, while *cascor* constructed a significantly smaller network for the Cadd data set. There was no significant differences for the Harm and Cadd data sets. One reason why *acasper* performs relatively worse on these data sets is that the regression data sets are more difficult to solve. These data sets require much larger networks before the best results on the validation set are obtained. The *cascor* algorithm also finds these regression data sets more difficult to solve, except that because it performs poorly on regression problems it quickly begins to

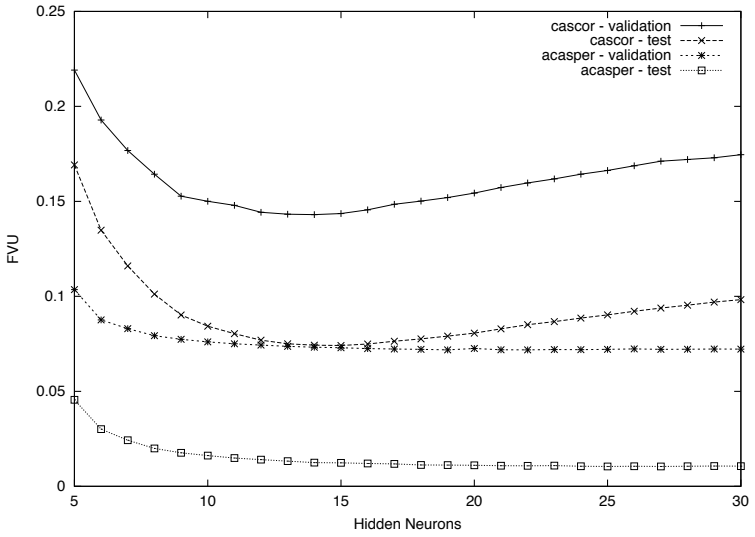

Figure 20: Average FVU for *acasper* and *cascor* for the Cadd data set.

get worse validation errors. This results in training being halted at an early stage, and the construction of small networks with poor generalization. The *acasper* algorithm, however, often continues to find better validation results, and so creates larger networks.

This effect can be seen in Figure 20, which plots the average validation and test FVU results for each hidden neuron added to the network for the Cadd data set. The Cadd data set was the only data set where *cascor* constructed significantly smaller networks than *acasper*. In this figure the overfitting of the validation and test sets can be seen for *cascor* by the rising FVU as more hidden neurons are added. The *acasper* algorithm, however, is able to continue to reduce the validation error as the networks increase in size.

In terms of computational cost, *cascor* is significantly more efficient than *acasper* on the Sif, Rad, Cadd, Harm, Cif data sets. For these data sets *cascor* uses approximately 5 to 14 times less connection crossings on average than *acasper*. The *cascor* algorithm is efficient to train because of its use of weight freezing and its ability to cache activations [27]. The computational difference between *cascor* and *acasper* can be expected to fall greatly for the

construction of smaller networks, such as those in Proben 1. This is because the difference in computational cost of training all weights, as in *acasper*, and only a small subset of weights, as in *cascor*, is greatly reduced for small networks. The difference in computational cost, however, rapidly grows as the networks increase in size.

11 Conclusion

Improving the generalization ability of FNNs is an important area of investigation. A series of empirical studies were performed to examine the effect of regularization on generalization in constructive cascade algorithms. It was found that the combination of early stopping and regularization resulted in better generalization than the use of early stopping alone. A cubic penalty term that greatly penalizes large weights was shown to be beneficial for generalization in cascade networks. An adaptive method of setting the regularization magnitude in constructive algorithms was introduced and shown to produce generalization results similar to those obtained with a fixed, user-optimized regularization setting. This adaptive method also resulted in the construction of smaller networks for more complex problems.

The *acasper* algorithm, which incorporates the insights obtained from the empirical studies, was shown to have good generalization and network construction properties in comparison to *cascor*. A major advantage of this algorithm is that it performs automatic model selection through automatic network construction and regularization. This removes the need for the user to select these parameters, and in the process makes the *acasper* algorithm free of parameters which must be optimized prior to the commencement of training.

Acknowledgment

The authors would like to thank the anonymous referees for thier constructive comments and suggestions.

References

[1] J. Moody, "Prediction risk and architecture selection for neural networks," in *From Statistics to Neural Networks: Theory and Pattern Recognition Applications*, V. Cherkassky, J. Friedman, and H. Wechsler, Eds. vol. 136 of *NATO ASI Series F*, New York: Springer-Verlag, 1994, pp. 147–165.

[2] R. Reed, "Pruning algorithms — a survey," *IEEE Transactions on Neural Networks*, vol. 4, pp. 740–747, Sept. 1993.

[3] W. Finnoff, F. Hergert, and H. G. Zimmermann, "Improving model selection by nonconvergent methods," *Neural Networks*, vol. 6, pp. 771–783, 1993.

[4] A. S. Weigend, D. E. Rumelhart, and B. A. Huberman, "Generalization by weight-elimination with application to forecasting," in *Advances in Neural Information Processing Systems 3*, R. P. Lippmann, J. E. Moody, and D. S. Touretzky, Eds. San Mateo, CA: Morgan Kaufmann, 1991, pp. 875–882.

[5] C. Bishop, *Neural Networks for Pattern Recognition*. Oxford: Oxford University Press, 1995.

[6] D. McKay, "A practical Bayesian framework for backpropagation networks," *Neural Computation*, vol. 4, pp. 448–472, 1992.

[7] W. Buntine and A. Weigend, "Bayesian back-propagation," *Complex Systems*, vol. 5, no. 6, pp. 603–643, 1991.

[8] T.-Y. Kwok and D.-Y. Yeung, "Constructive algorithms for structure learning in feedforward neural networks for regression problems," *IEEE Transactions on Neural Networks*, vol. 8, pp. 630–645, May 1997.

[9] T. Ash, "Dynamic node creation in backpropagation networks," *Connection Science*, vol. 1, no. 4, pp. 365–375, 1989.

[10] S. E. Fahlman and C. Lebiere, "The Cascade-Correlation learning architecture," in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. San Mateo, CA: Morgan Kaufmann, 1990, pp. 524–532.

[11] J.-N. Hwang, S.-R. Lay, M. Maechler, R. D. Martin, and J. Schimert, "Regression modeling in back-propagation and projection pursuit learning," *IEEE Transactions on Neural Networks*, vol. 5, pp. 342–353, May 1994.

[12] R. Setiono and L. Hui, "Use of a quasi-Newton method in a feedforward neural network construction algorithm," *IEEE Transactions on Neural Networks*, vol. 6, pp. 273–277, Jan. 1995.

[13] Y. Hirose, K. Yamashita, and S. Hijiya, "Back-propagation algorithm which varies the number of hidden units," *Neural Networks*, vol. 4, no. 1, pp. 61–66, 1991.

[14] E. B. Bartlett, "Dynamic node architecture learning: An information theoretic approach," *Neural Networks*, vol. 7, no. 1, pp. 129–140, 1994.

[15] P. Courrieu, "A convergent generator of neural networks," *Neural Networks*, vol. 6, pp. 835–844, 1993.

[16] T.-Y. Kwok and D.-Y. Yeung, "Objective functions for training new hidden units in constructive neural networks," *IEEE Transactions on Neural Networks*, vol. 8, pp. 1131–1148, Sept. 1997.

[17] T.-Y. Kwok and D.-Y. Yeung, "Experimental analysis of input weight freezing in constructive neural networks," in *Proc. of the IEEE Int. Conf. on Neural Networks*, San Francisco, Mar. 1993, pp. 511–516.

[18] L. Prechelt, "Investigation of the cascor family of learning algorithms," *Neural Networks*, vol. 10, no. 5, pp. 885–896, 1997.

[19] Y. Shin and J. Ghosh, "Ridge polynomial networks," *IEEE Transactions on Neural Networks*, vol. 6, pp. 610–622, May 1995.

[20] S. Lay, J. Hwang, and S. You, "Extensions to projection pursuit learning networks with parametric smoothers," in *Proc. of the IEEE Int. Conf. on Neural Networks*, vol. 3, Orlando, June 1994, pp. 1325–1330.

[21] T.-Y. Kwok and D.-Y. Yeung, "Bayesian regularization in constructive neural networks," in *Proc. of the Int. Conf. on Neural Networks*, Bochum, July 1996, pp. 557–562.

[22] D. Rumelhart and J. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, 2. Cambridge: MIT Press, 1986.

[23] L. Prechelt, "Proben1 - a set of neural network benchmark problems and benchmarking rules," Tech. Rep. 21/94, Fakultät für Informatik, Universität Kahlsruhe, 1994.

[24] A. Flexer, "Statistical evaluation of neural network experiments: Minimum requirements and current practice," Tech. Rep. oefai-tr-95-16, Austrian Research Institute for Artificial Intelligence, 1995.

[25] L. Prechelt, "A quantitative study of experimental evaluations of neural network learning algorithms," *Neural Networks*, vol. 9, pp. 457–462, 1996.

[26] R. Steel and J. Torrie, *Principles and Procedures of Statistics A Biomedical Approach*. Singapore: McGraw-Hill, 1980.

[27] S. E. Fahlman and C. Lebiere, "The Cascade-Correlation learning architecture," Tech. Rep. CMU-CS-90-100, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Feb. 1990.

[28] T.-Y. Kwok and D.-Y. Yeung, "Use of a bias term in projection pursuit learning improves approximation and convergence properties," *IEEE Transactions on Neural Networks*, vol. 7, pp. 1168–1183, Sept. 1996.

[29] M. Riedmiller, "Advanced supervised learning in multi-layer perceptrons - from backpropagation to adaptive learning algorithms," *International Journal of Computer Standards and Interfaces*, vol. 16, pp. 265–278, 1994.

[30] N. K. Treadgold and T. D. Gedeon, "A cascade network algorithm employing progressive rprop," in *Proc. of the Int. Work-Conf. on Artificial and Natural Neural Systems*, Lanzarote, June 1997, pp. 723–732.

[31] N. K. Treadgold and T. D. Gedeon, "Extending casper: A regression survey," in *Proc. of the Int. Conf. on Neural Information Processing*, Dunedin, Nov. 1997, pp. 310–313.

[32] J. S. Bridle, "Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition," in *Neuro-computing: Algorithms, Architectures and Applications*, F. Fogelman Soulié and J. Hérault, Eds. Berlin: Springer-Verlag, 1990, pp. 227–236.

[33] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, pp. 359–366, 1989.

[34] N. K. Treadgold and T. D. Gedeon, "Simulated annealing and weight decay in adaptive learning: The sarprop algorithm," *IEEE Transactions on Neural Networks*, vol. 9, pp. 662–668, July 1998.

[35] N. K. Treadgold and T. D. Gedeon, "Adaptive regularization in a constructive cascade network," in *Proc. Int. Conf. on Neural Information Processing*, Kitakyushu, Oct. 1998, pp. 805–808.

[36] N. K. Treadgold and T. D. Gedeon, "A constructive cascade network with adaptive regularization," in *Proc. of the Int. Work-Conf. on Arti-*

ficial and Natural Neural Systems, Alicante, (accepted for publication), June 1999.

[37] J.-N. Hwang, S.-S. You, S.-R. Lay, and I.-C. Jou, "The Cascade-Correlation learning: A projection pursuit learning perspective," *IEEE Transactions on Neural Networks*, vol. 7, pp. 278–289, Mar. 1996.

Appendix

The results given in Tables 6 to 9 are from the comparative simulations performed between the *casbias*, *cassar*, *casdec2*, and *casper* algorithms using different regularization levels. The results on the test sets at the point where the lowest validation error occurred after the halting criterion was satisfied are given for each different regularization term and α setting. The results for the *casbias* algorithm are given in Table 10.

The test set and hidden neuron results for the *acasper* and *cascor* algorithms are given in Tables 11 to 15 on the Proben 1 and Sif, Rad, Cadd, Harm, and Cif regression data sets.

Table 6: Two Spirals Test Error Percentage

Algorithm		Regularization Level $10^{-\alpha}$				
		1	2	3	4	5
cassar	Mean	30.10	21.62	20.78	20.88	20.59
	StDv	4.61	3.88	2.62	3.32	2.21
	Median	31.85	20.55	20.66	19.94	20.51
casdec2	Mean	20.72	17.36	18.68	21.61	21.44
	StDv	6.94	3.73	2.36	3.37	2.79
	Median	18.08	17.31	18.40	21.18	21.58
casper	Mean	17.43	15.67	18.56	19.80	20.64
	StDv	6.01	2.22	2.99	2.46	2.49
	Median	16.04	15.69	18.50	19.49	20.60

Table 7: Glass Test Error Percentage

Algorithm		Regularization Level $10^{-\alpha}$				
		1	2	3	4	5
cassar	Mean	33.51	32.45	33.32	33.17	32.98
	StDv	1.45	2.22	2.24	1.98	1.95
	Median	33.96	32.08	33.96	33.96	32.08
casdec2	Mean	28.68	31.28	31.47	33.17	33.55
	StDv	2.50	3.30	3.15	3.15	2.48
	Median	28.30	32.08	30.19	32.08	33.96
casper	Mean	30.34	28.94	31.05	31.92	32.00
	StDv	1.82	2.34	2.53	1.97	3.17
	Median	30.19	28.30	32.08	32.08	31.13

Table 8: Cadd Test FVU ($\times 10^{-2}$)

Algorithm		Regularization Level $10^{-\alpha}$				
		1	2	3	4	5
cassar	Mean	94.23	4.31	2.67	2.82	2.89
	StDv	0.00	9.20	1.42	1.17	1.11
	Median	94.23	1.74	2.22	2.60	2.59
casdec2	Mean	94.23	1.46	1.59	2.32	3.20
	StDv	0.00	0.95	0.54	0.91	1.65
	Median	94.23	1.25	1.49	2.15	3.05
casper	Mean	3.35	1.29	1.49	1.69	2.26
	StDv	5.94	0.60	0.44	0.48	0.93
	Median	2.43	1.17	1.39	1.67	1.99

Table 9: Cif Test FVU ($\times 10^{-2}$)

Algorithm		Regularization Level $10^{-\alpha}$				
		1	2	3	4	5
cassar	Mean	82.25	34.47	7.90	8.72	9.45
	StDv	0.00	36.42	3.79	3.65	4.48
	Median	82.25	8.58	6.98	8.55	7.86
casdec2	Mean	82.25	51.19	6.55	8.97	8.02
	StDv	0.00	38.45	3.90	5.08	3.64
	Median	82.25	82.25	5.33	7.39	7.60
casper	Mean	9.79	2.98	3.97	6.18	7.95
	StDv	12.73	0.98	1.34	2.17	3.97
	Median	5.23	2.69	3.77	5.87	6.59

Table 10: Test Results for *casbias*

Data Set	Mean	StDv	Median
Two Spirals (% error)	22.26	4.11	21.95
Glass (% error)	34.53	2.56	34.91
Cadd (FVU)	2.96	1.60	2.49
Cif (FVU)	10.73	6.41	9.17

Table 11: Proben 1 Test Error Percentage Results

Data Set	Algorithm	Mean	StDv	Median	Min	Max
Cancer	*acasper*	1.89	0.80	1.72	0.57	4.02
	cascor	1.95	0.38	1.72	1.15	2.87
Card	*acasper*	13.72	0.59	13.37	13.37	16.86
	cascor	13.58	0.43	13.37	12.79	14.54
Diabetes	*acasper*	23.14	1.26	22.92	20.31	27.08
	cascor	24.53	1.44	24.48	22.40	28.65
Gene	*acasper*	11.72	0.09	11.73	11.60	11.85
	cascor	13.38	0.47	13.49	11.98	14.38
Glass	*acasper*	30.68	2.61	30.19	26.42	35.85
	cascor	34.76	5.88	33.96	26.42	47.17
Heart	*acasper*	19.21	0.44	19.13	16.96	20.00
	cascor	19.89	1.58	20.44	16.09	22.17
Heartc	*acasper*	18.85	1.14	18.67	18.67	26.67
	cascor	19.47	1.28	18.67	18.67	24.00
Horse	*acasper*	32.46	0.71	32.97	29.67	34.07
	cascor	26.37	2.58	26.37	20.88	31.87
Soybean	*acasper*	7.89	1.03	7.65	5.29	10.00
	cascor	9.46	0.86	9.41	7.65	11.77
Thyroid	*acasper*	1.67	0.26	1.64	1.28	2.28
	cascor	3.03	1.15	2.67	2.11	6.56

Table 12: Proben 1 Squared Error Percentage Results

Data Set	Algorithm	Mean	StDv	Median	Min	Max
Building	*acasper*	0.64	0.02	0.64	0.61	0.71
	cascor	0.82	0.23	0.72	0.49	1.42
Flare	*acasper*	0.53	0.01	0.52	0.52	0.58
	cascor	0.53	0.01	0.53	0.51	0.55
Hearta	*acasper*	4.74	0.10	4.69	4.67	5.23
	cascor	4.62	0.15	4.60	4.43	5.02
Heartac	*acasper*	2.75	0.14	2.72	2.62	3.11
	cascor	2.87	0.44	2.70	2.48	4.25

Table 13: Regression Test FVU ($\times 10^{-2}$) Results

Data Set	Algorithm	Mean	StDv	Median	Min	Max
Sif	*acasper*	0.84	0.20	0.81	0.51	1.35
	cascor	3.29	0.89	3.14	2.01	5.58
Rad	*acasper*	0.99	0.20	0.98	0.69	1.73
	cascor	4.04	1.27	3.83	2.09	7.07
Cadd	*acasper*	1.18	0.24	1.09	0.84	1.86
	cascor	7.43	2.09	6.92	3.95	11.90
Harm	*acasper*	2.37	0.69	2.18	1.45	4.70
	cascor	22.48	4.05	21.81	14.43	34.36
Cif	*acasper*	2.38	0.61	2.21	1.48	4.03
	cascor	10.50	2.47	9.95	6.86	18.88

Table 14: Proben 1 Hidden Neuron Results

Data Set	Algorithm	Mean	StDv	Median	Min	Max
Cancer	*acasper*	4.86	2.08	4.50	1.00	8.00
	cascor	5.18	2.05	4.00	3.00	10.00
Card	*acasper*	0.12	0.59	0.00	0.00	4.00
	cascor	1.07	0.25	1.00	1.00	2.00
Diabetes	*acasper*	3.02	1.55	3.00	1.00	8.00
	cascor	9.78	5.32	9.00	0.00	25.00
Gene	*acasper*	0.00	0.00	0.00	0.00	0.00
	cascor	2.73	1.19	2.00	1.00	6.00
Glass	*acasper*	4.18	2.21	4.00	1.00	8.00
	cascor	8.07	5.19	7.00	1.00	24.00
Heart	*acasper*	0.10	0.36	0.00	0.00	2.00
	cascor	2.64	1.17	2.00	1.00	7.00
Heartc	*acasper*	0.10	0.36	0.00	0.00	2.00
	cascor	1.38	0.49	1.00	1.00	2.00
Horse	*acasper*	0.12	0.59	0.00	0.00	4.00
	cascor	0.82	0.39	1.00	0.00	1.00
Soybean	*acasper*	2.16	1.08	2.00	1.00	5.00
	cascor	16.04	5.17	16.00	6.00	24.00
Thyroid	*acasper*	4.64	2.34	5.00	1.00	8.00
	cascor	25.04	8.71	27.00	2.00	44.00
Building	*acasper*	6.36	2.15	7.00	1.00	8.00
	cascor	9.27	9.73	6.00	0.00	29.00
Flare	*acasper*	1.30	1.59	1.00	0.00	6.00
	cascor	2.63	0.67	3.00	2.00	4.00
Hearta	*acasper*	0.40	0.57	0.00	0.00	2.00
	cascor	2.77	1.72	2.00	0.00	7.00
Heartac	*acasper*	0.20	0.86	0.00	0.00	5.00
	cascor	1.47	0.73	1.00	0.00	3.00

Table 15: Regression Hidden Neuron Results

Data Set	Algorithm	Mean	StDv	Median	Min	Max
Sif	*acasper*	9.44	5.83	8.50	3.00	27.00
	cascor	17.36	4.53	17.50	10.00	27.00
Rad	*acasper*	13.30	6.57	12.00	4.00	30.00
	cascor	20.06	5.28	20.00	9.00	30.00
Cadd	*acasper*	16.16	5.74	15.00	7.00	29.00
	cascor	12.72	3.43	12.50	4.00	23.00
Harm	*acasper*	19.34	5.30	18.00	10.00	30.00
	cascor	17.34	4.84	17.00	10.00	30.00
Cif	*acasper*	20.16	6.24	19.50	8.00	30.00
	cascor	19.86	4.80	19.00	9.00	30.00