Knowledge Based Neural Networks
Testing and Improving the KBANN Algorithm

Tristan Grimmer

December 1995

Abstract

Artificial Neural Networks have a large learning time and fail to capitalize on any domain knowledge. The KBANN system is an attempt to reduce training time by creating a network that is initially consistent with the domain knowledge. This network is then trained using backpropagation. For a reasonably simple domain theory, the networks created by KBANN are relatively shallow (having few layers). When the domain theory is only relatively complex the depth of the created network becomes large, and the time required to train this network increases. This problem with KBANN is tested by implementing a system capable of recognizing simple geometric shapes in a 7x7 array. The KBANN network is compared with a single hidden layer feed-forward network. Finally, a new, independently invented, system is tested. This new system, ShallowKBANN, keeps the small number of layers while still using the domain knowledge. SKBANN is found to outperform both other systems in terms of number of training cycles necessary to reach maximal generalization of the network. Finally, a suggestion is made on how to incorporate more domain knowledge into an ANN. Often the rules in a domain theory can have a certainty associated with them. It may be possible to incorporate this certainty factor into an initial network.

1 Introduction

This report intends to take a critical look at the KBANN algorithm and consider ways of improving it. The KBANN (Knowledge-Based Artificial Neural Network) algorithm implements a hybrid system. That is, it is an attempt to merge explanation-based learning, which can incorporate prior knowledge, with an inductive learning method, in this case a neural network. This is desireable since explanation-based learning requires that the domain theory , in this case our prior knowledge, is sound and in some cases complete (see [Mit95, Chaper 6]). This knowledge base may not exist in the detail required and may have inconsistencies. Simulated Neural Networks, on the other hand, do not require this initial domain theory. They are capable of learning from noisy examples, and generalize well to unseen cases. The problem here, however, is the large learning time required to train the network.

KBANN is an attempt to reduce this training time and avoid the problems that may be present in the initial domain thoery. Usually ANN's have their initial weights and thresholds (unit biases) initialized to small random values. KBANN, however, creates a network that agrees with the domain theory initially. That is, some of the weights and thresholds are set to specific values so that the network reflects the domain theory. At this point the network is capable of classification in much the same way as a rule based system [1]. The KBANN system dictates the entire structure of the neural network: number of units in total, number of layers, and number of links. The problem is with the number of layers generated (see [TS92]). A relatively simple set of domain rules can lead to a network that is too deep to be trained effectively with, for example, the standard back-propagation algorithm ([RHW86]). This scenario is studied by comparing the training times of a KBANN net versus a simple 1-hidden-layer fully connected net. The number of hidden units of both networks was normalized to 11 in an attempt at a fair comparison.

[1] The Incompleteness and possible unsoundness of the domain theory may be handled differently by different ruled based systems.

As a potential solution to this problem, a third system (ShallowKBANN) is implemented and tested. This system, which was developed by the author, only has one hidden layer but still manages to use the domain theory to 'initialize' some of it's hidden units as feature detectors. Those features which are most likely to be of importance in determining the concept (in this case a shape) are given the largest thresholds and link weights. This means that the more important features are harder to 'unlearn' during the training process, while the less important ones, which may not even be necessary or correct, would be easily swamped by any conflicting training examples. That is, the units implementing these 'less important features' could easily converge to recognize a different concept if necessary. The development of SKBANN then leads us to the possibility of mapping the certainty of rules in the domain base (if these certainties are available) into the initial network. The validity of this mapping technique, however, has not been tested as it is beyond the scope of this experiment.

The rest of this report is organized as follows:

- **Section 2:** A general form of the KBANN algorithm is described.
- **Section 3:** The domain in which the experiments will be carried out is described. It consists of simple shapes that exist on a 7x7 grid. The initial domain theory is also introduced in this section.
- **Section 4:** An initial KBANN network is created using this domain theory.
- **Section 5:** A description of how to train and test a neural network using back propagation is given. The network simulator that was used in the experiments is introduced. The training data used in the shape domain is given as well as the validation data.
- **Section 6:** Results from training the KBANN network are given.
- **Section 7:** For comparison purposes, the results from training a standard fully connected neural network are presented.
- **Section 8:** The SKBANN algorithm for mapping the domain theory to an initial ANN is described.
- **Section 9:** Results for the SKBANN algorithm are given. These are compared to the KBANN results and the standard network results.
- **Section 10:** The possibility of incorporating the certainty factors of the domain theory rules into the initial network is described.
- **Section 11:** Conclusions and possible limitations of the algorithms are given.

2 KBANN with Disjunction

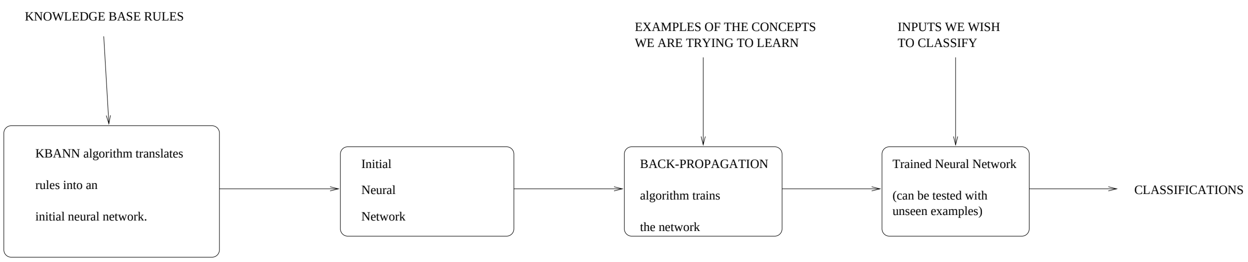


Figure 1: The KBANN System

In [Mit95] the KBANN algorithm is briefly described for the case when all the variable free rules in the knowledge base occur only once (i.e. no disjunction). However, there is a more general form (see [TSN90]) that can handle both disjunction and negation. Although the algorithm is always presented as learning a single concept, it is not too difficult to generalize it to multiple concepts. The most general formulation will be needed for this experiment. Figure 1 shows a schematic representation of how KBANN is supposed to work. The back-propagation algorithm for training the neural network is well documented (see for example [RHW86]) and will not be described here. The rules-to-network translation, however, is not so well known and is described in full below.

2.1 Step 1 Mapping the Rules

Basically each unit of the generated neural net corresponds to a predicate. For example, lets suppose we have the rule:

$$A \leftarrow B \wedge C \tag{1}$$

The network for this rule can be seen in figure 2a. The value 'w' beside the links represents the weight on that link. The value beside unit A is the threshold. If the sum of the inputs and the threshold is greater than zero then A 'fires'. That is, the value of A is described by,

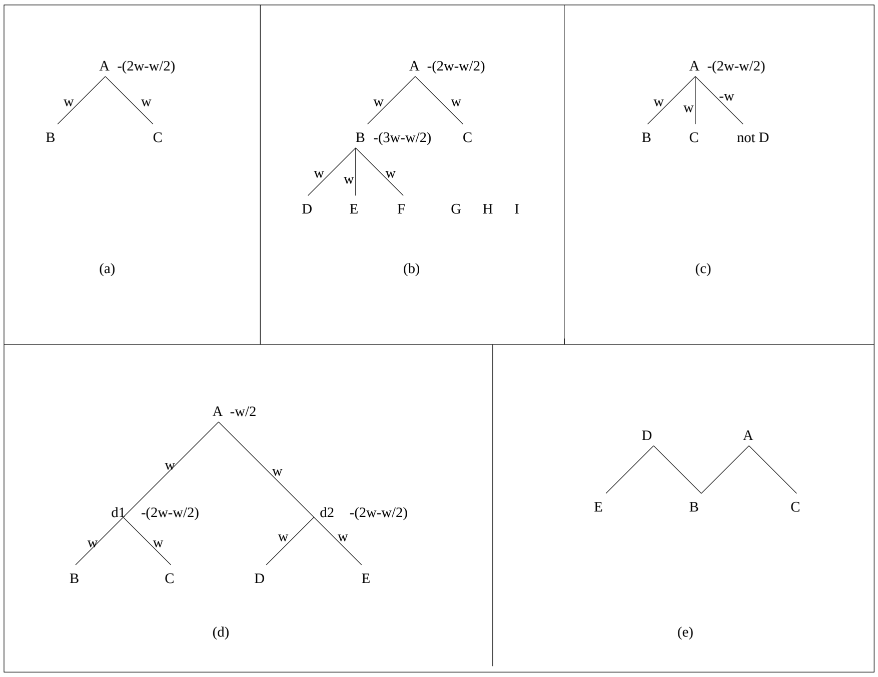


Figure 2: An Example of KBANN

$$A = \begin{cases} 1 & \text{if } Bw + Cw - (2w - w/2) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Note that this does not accurately depict how a unit works as implemented in a real neural network, but is used for illustrative purposes only. As we can see A will only 'fire' if both B and C have value 1. Assuming that the inputs, B and C, are either 0 or 1. This correctly models the rule. There seem to be 2 general forms for the equation that gives the bias value of unit A. In [TSN90], it is given as $-(nw - \phi)$, where n is the number of mandatory antecedents. Phi is a value between zero and w that was found empirically to work well. This does not make sense. For a value of phi either side of $w/2$ an assumption is being made about which way the weights are likely to be modified during back propagation. The direction taken will be a reflection of the accuracy of the rule. There seems to be no reason to expect, on the majority of the units, that a weight change will be one way or the other. For this reason, the bias function given in [Mit95] will be used. It is $-(nw - w/2)$, right in the middle. It should be noted that the value 3 for w was found, again empirically, to work well.

Now lets assume a new rule is added:

$$B \leftarrow D \wedge E \wedge F \tag{3}$$

The network will now look like figure 2b. As can be seen, it is very easy to add another layer to the network. Another potential problem is the fact that having many positive antecedents should not make that rule 'more important'. Given the assumption that the training method employed [2] on the network cannot make arbitrarily large weight adjustments during a training cycle it can be concluded that the units with these larger biases and link weights will be harder to modify. This is not desireable.

Now lets use a rule with negation (figure 2c).

$$A \leftarrow B \wedge C \wedge \neg D \tag{4}$$

Nothing really changes except that the weight on the 'not' link is set to $-w$. Since only the mandatory antecedents are taken into account during the bias calculation no changes are needed. That is, the prohibitory antecedents (the ones with the negation) are ignored. It can be easily verified that the network is still valid and that 'D' must be zero for A to be activated.

To add disjunction to the network some extra units (that do not correspond to a predicate in the rules) will have to be added. Suppose the following two rules are true:

$$A \leftarrow B \wedge C \tag{5}$$

$$A \leftarrow D \wedge E \tag{6}$$

Then figure 2d shows how this can be handled. The added units, d1 and d2, are what can activate A. By setting the bias of A to $-w/2$ we ensure that if any of d_1 or d_2 or d_3 or $\ldots$ or d_n fire that A will be above the threshold and will fire. The units d1, and d2, are then given weights like any other unit.

Finally, dealing with more than one concept that has common antecedents is not a problem. Suppose:

$$A \leftarrow B \wedge C \tag{7}$$

$$D \leftarrow B \wedge E \tag{8}$$

Then all that needs to be done is link A and D to their common antecedent B as in figure 2e. The weights and biases can be computed in the normal manner.

[2] It is not necessary to use backpropagation. It's just the norm.

2.2 Step 2 Adding More Units

Since it is possible, and even likely, that there are predicates missing from the knowledge base that are vital to the determination of a concept, new units must be added to the network. These take the form of additional input units. For example, in figure 2b the units G, H, and I have been added.

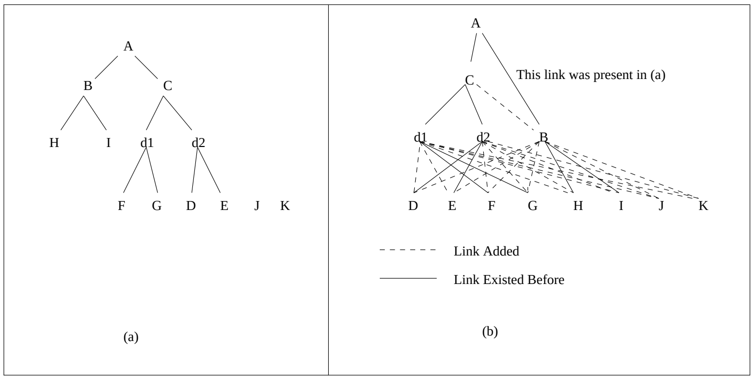


Figure 3: Adding Links

2.3 Step 3 Adding More Links and Perturbing the Network

These new units must be linked in. The most common way to do this is to order each unit according to it's maximum path length from an input unit (which corresponds to a predicate with no antecedents). All units that have the same 'depth' are put in a layer. For figure 3b we have:

Unit A	Max Path Length (level) 3
Unit C	Max Path Length (level) 2
Units d_1,d_2,B	Max Path Length (level) 1
Units D,E,F,G,H,I,J,K	Max Path Length (level) 0

Each layer is then fully connected and each added link is assigned a weight of zero. Of course if a link is present due to the previous rules-to-network mapping then it is left untouched. The 'bottom' layer of units (i.e. the inputs) will contain any added units. These are linked in at this time. Figure 3a shows a network with the added units but not the added links. Figure 3b shows the same network organized into layers and linked. The purpose of these added links is simply to facilitate the learning process when the back-propagation algorithm is applied. They represent extra degrees of freedom in the search for minimal error. It should be noted that there is now a link between C and B, basically allowing C's output to depend on B's. This makes sense intuitively since B is a more general concept. That is, it is directly influenced by the inputs. By dropping each unit as close to the inputs as possible we allow each unit to be as general as possible since it will be used by more units at higher levels.

For a more thorough introduction to KBANN see [TSN90]. The basic idea has been covered so it is now possible to move on and describe the domain in which the experiments will be carried out.

3 A Domain of Simple Pictures

All the concepts that will be learned in these experiments are simple shapes that live on a 7x7 grid of pixels that are either off or on. The noise free representations of these shapes are shown in figure 4.

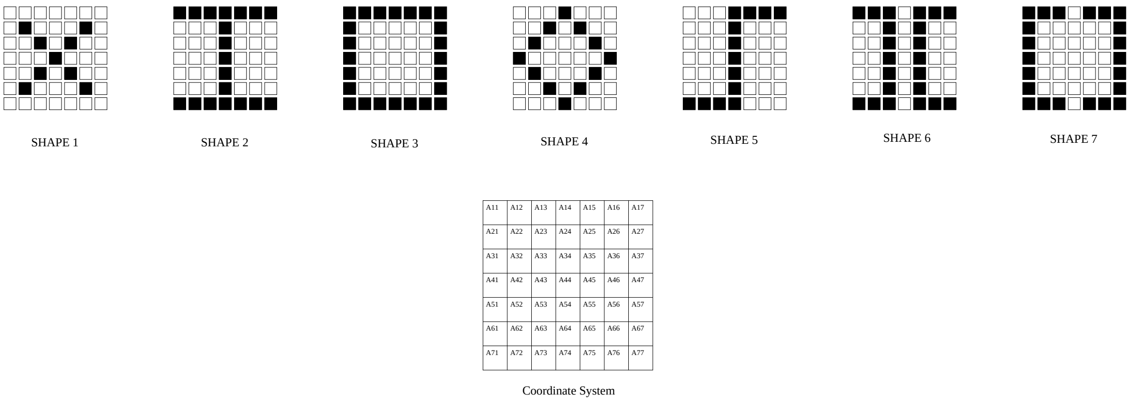


Figure 4: The Shape Concepts

The domain theory that will be assumed can be seen in figure 5.

This domain theory is neither complete nor correct. It takes the form of a number of clauses much like Prolog rules. Note the inclusion of negation of some of the predicates and the inclusion of disjunction in the form of multiple rules with the same head. As can be easily verified, the rules do capture some of the important features necessary to each shape. However, it is not a complete specification, as not all pixels in the array are referenced. Furthermore, it can be seen that even if supplied with noise free inputs, a unique determination of the shape is not possible with the given rules. For example, rule 7 and 8 both have the same antecedent, so if this antecedent is true then it will not be known whether we are dealing with shape 7 or shape 8.

Not only is it incomplete, but there are errors as well. Rule 5 allows a determination of shape 4 when it is entirely possible that A34 is on. This conclusion would therefore be incorrect. These problems with the knowledge base highlight the need to involve a connectionist approach. In the next section a KBANN network is created using the rules of figure 5, so it is worth while having a close look at the rules.

4 The Shape KBANN Net

The details of working out the KBANN net for the shape knowledge base are not worth repeating. In essence it was just a mechanical application of the rules given in section 3. Figure 6 shows step 1: mapping the rules. Note that the weights are set to either plus or minus 6. This larger weight size was necessary due to the network simulation tool being used. It is suspected that the errors in the floating point arithmetic used by the simulator are not kept bounded by a constant factor.

The network, after adding the extra units and links, can be seen pictorially in figure 7. This figure shows the activation level at the bottom of each unit and the unit number on top. As can be seen, unit 59 has a nonzero activation. This should not be the case. Since the network has not been trained or perturbed by small random numbers there is no satisfactory explanation for this discrepency. It is highly suspected that it is a manifestation of floating point error. Perhaps the simulation tool (SNNS is used in this experiment) divides two near equal numbers causing catastrophic cancellation. It is also, of course, possible that there is some rather more obvious bug in the simulation software. Regardless, the task of determining the origin of this error by debugging the source code has proven to be nontrivial and will be left to the creators of the

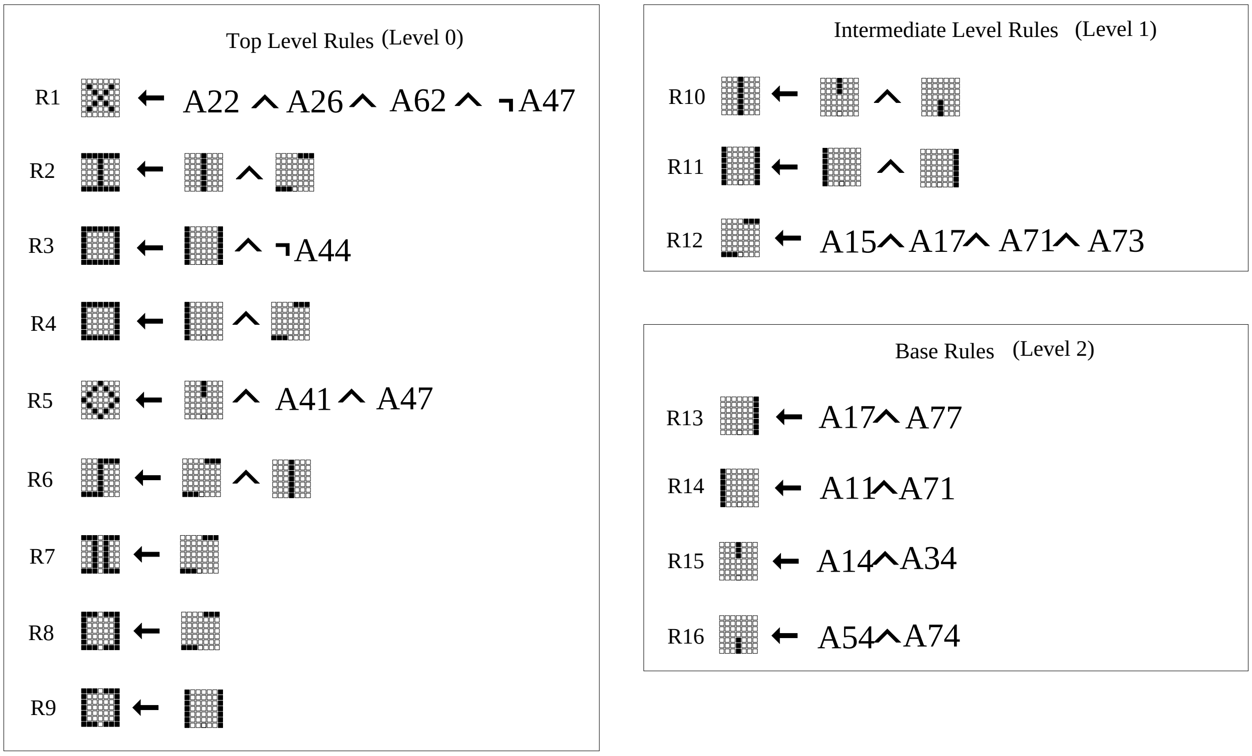


Figure 5: The Knowledge Base Rules

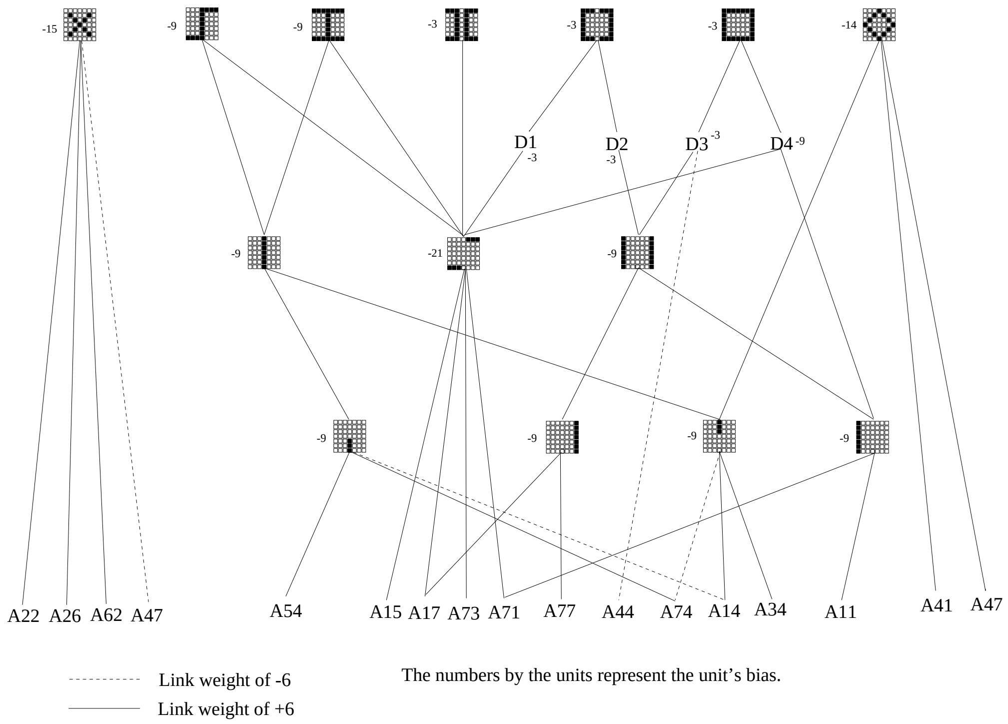
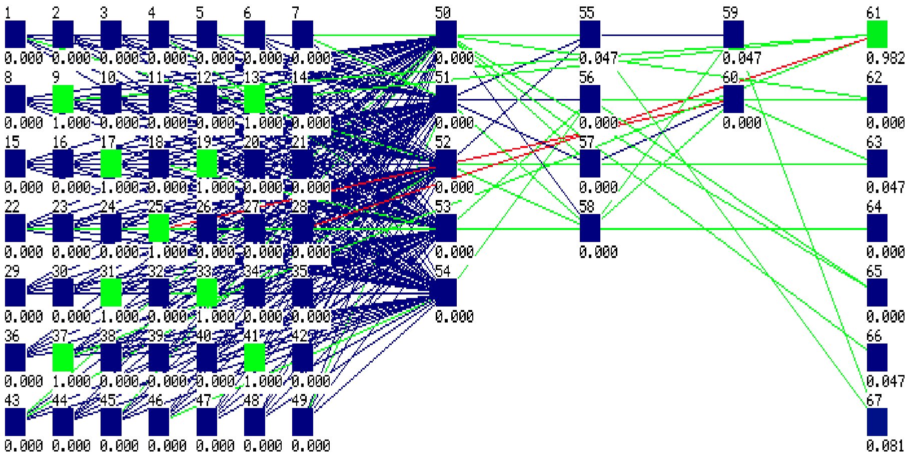


Figure 6: The Shape KBANN Net, Step 1

simulation tool (see [MV95]). Figure 7 shows the network recognizing shape 1, which it is capable of doing correctly given the rule base.



Green Links: Weight +6
Red Links: Weight -6
Blue Links: Weight 0

Figure 7: The Shape KBANN Net With Added Links And Units

5 Training a Network Using SNNS

SNNS is the Stuttgart Neural Network Simulator. It was downloaded and compiled to run on an Alpha DEC 3000. Although there are several bugs and missing features in the software, it is general enough to do everything needed. All simulations will be done using SNNS.

In order to train a network the weights and bias values of all links and units must be perturbed to allow for symmetry breaking during training ([RHW86]). In this case an ϵ of 0.001 is used. That is, a random value between 0 and 0.001 is chosen and then either added or subtracted from the weight/bias. A new random value is chosen for each weight/bias perturbation.

The training process involves supplying the back-propagation algorithm a number of input/output pairs from the training set. Let the training set have n training examples. Applying the back-propagation algorithm to each of these examples constitutes what is known as a training *cycle*[3]. Usually many cycles are necessary before the ANN starts to converge. It is the number of cycles that this report is concerned with reducing. In [MV95] it is also noted that supplying the network with the training pairs in the *same order* for each cycle results in a larger number of cycles before the network converges. For this reason the input/output examples will be randomly *shuffled* before each cycle. Since SNNS supports this operation it is trivial to implement.

[3] The term *epoch* is also common.

5.1 Stopping Criteria

Next, it is necessary to determine a stopping criteria. It is desireable to stop when the network generalizes best to unseen examples [4]. What can be done is to divide up the training set into an actual training set and a validation set. This approach is used here. The validation set can be considered the 'unseen' test cases. What may happen can be seen in figure 8.

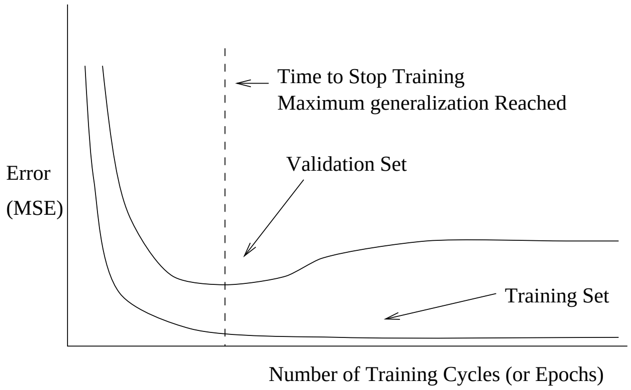


Figure 8: Stopping Criteria

If training proceeds past the dotted line then the network is being overtrained and is simply specializing to recognize the shapes it is being trained with. The other possibility is that the two sets are related so closely (even though they contain no input/output pair in common) that the performance just keeps on improving. If this is the case then the stopping criteria will be a Mean Squared Error (MSE) of the outputs of below 0.05 [5].

The error metric used, as stated above, will be the MSE. The MSE is defined as:

$$MSE = \frac{SSE}{N-k} \tag{9}$$

where N is the total number of samples, k is the number of parameters being estimated, and

$$SSE = \sum_{i=1}^{N} e^2 \tag{10}$$

The variable e is just the error (the difference between the estimated value and the true value). For example, suppose we want to know the MSE when 2 validation input/output pairs are supplied to the network. The data may look something like:

Desired Output	**Network Output**
Pair#1 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0	0.1 0.8 0.2 0.3 0.0 0.0 0.3 0.4
Pair#2 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0	0.1 0.5 0.9 0.1 0.1 0.0 0.2 0.4

The 'Desired Output' column is just the output for that training pair. The 'Network Output' contains the

[4] The SNNS user manual, [MV95], gives a more in depth explanation

[5] This value was pulled out of thin air and should be customized for the particular domain and task at hand. Some specifications may allow for larger values than others.

values that the network actually returned. For this hypothetical case $N = 14$ and the SSE will be:

$$SSE = \sum_{i=1}^{N} e^2 = 0.1^2 + 0.2^2 + 0.2^2 + 0.3^2 \ldots 0.2^2 + 0.4^2 = 0.92 \tag{11}$$

In this case $k = 7$ since there are seven separate parameters that the network must predict each time. The MSE is therefore given as:

$$MSE = \frac{SSE}{14 - 7} = 0.131 \tag{12}$$

For a thorough discussion of why the MSE is a good error metric for this experiment see [HL87, pages 188-192]. In general the MSE is a better indicator if the sample size, N is large. In our case the validation set will contain 28 input/output pairs yielding a sample size of 196 (since there are seven units per output). The value of k will remain at 7 since there are still only 7 parameters that the neural network is learning. That is, the seven output units.

5.2 Training and Validation Data

As explained above, there is a pool of input/output pairs that will be divided into a training set and a validation set. This initial 'pool' of values can be seen in figure 9

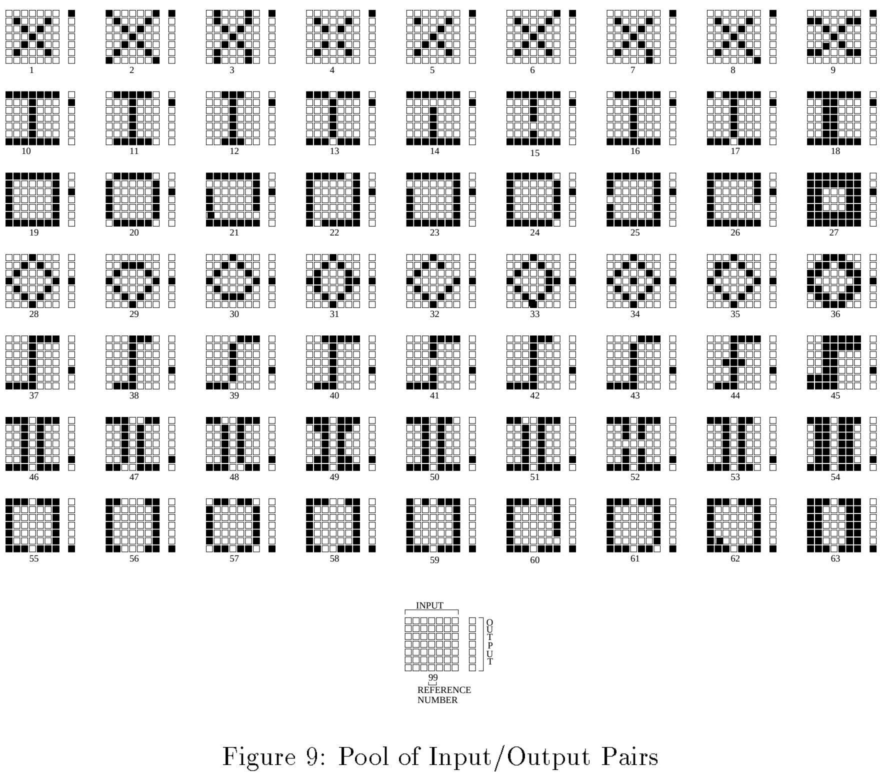


Figure 9: Pool of Input/Output Pairs

The pool was divided into a set of 28 validation pairs and 35 training pairs. The reason for more training pairs is because the noise free descriptions of each shape concept were forced into the training set. This makes sense as we can assume that these are the **given** concepts that require learning. The remaining 56 pairs were divided randomly between the training and data sets with the constraint that each set of pairs describing the same shape would be divided evenly between the data and training set[6]. That is, it is undesireable to, by chance, put all the examples of a particular shape in only one of the sets. The following table shows the division. V stands for validation set while T for training set.

Reference Number	1	2	3	4	5	6	7	8	9
Set	T	T	V	V	T	V	T	V	T
Reference Number	10	11	12	13	14	15	16	17	18
Set	T	T	T	V	T	V	V	T	V
Reference Number	19	20	21	22	23	24	25	26	27
Set	T	V	T	T	V	V	T	V	T
Reference Number	28	29	30	31	32	33	34	35	36
Set	T	V	T	T	T	V	V	V	T
Reference Number	37	38	39	40	41	42	43	44	45
Set	T	T	T	V	T	V	T	V	V
Reference Number	46	47	48	49	50	51	52	53	54
Set	T	T	T	V	T	T	V	V	V
Reference Number	55	56	57	58	59	60	61	62	63
Set	T	T	V	T	T	T	V	V	V

6 Training Results: The KBANN Network

Due to the large depth of the network the training did not go as well as expected. The standard backpropagation algorithm was used with a learning rate of 0.2. This rate was found by trial and error to work the best. Figure 10 shows the generated error curve. Figure 11 shows an example of the network failing to recognize one of the training set examples! It can't tell whether it is shape 2 or shape 5.

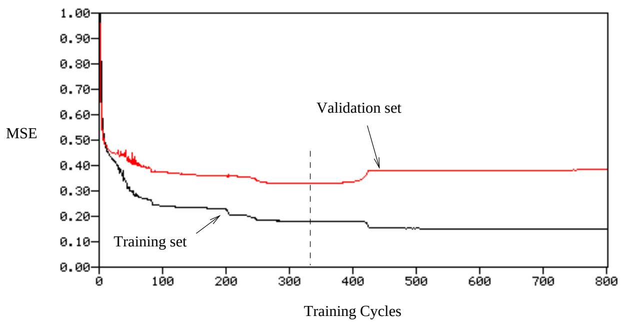


Figure 10: Learning Graph KBANN

In order to remedy this situation a 'new' KBANN net was created with a few more hidden units, in order to alleviate the apparent bottle-neck before the outputs. Figure 12 shows it's structure and figure 13 it's learning curve.

The improvement is negligible. The over-all performance of these deep KBANN networks leaves much to be desired. As will be seen, a one hidden layer vanilla network outperforms readily.

[6] The process involved the flipping of a **real** coin, and resulted in an equal probability of T or V in each position. It was ensured that the constraint did not skew the distribution.

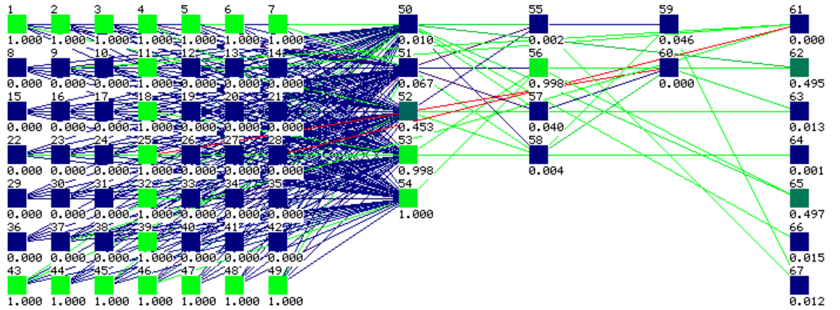


Figure 11: Failure of KBANN

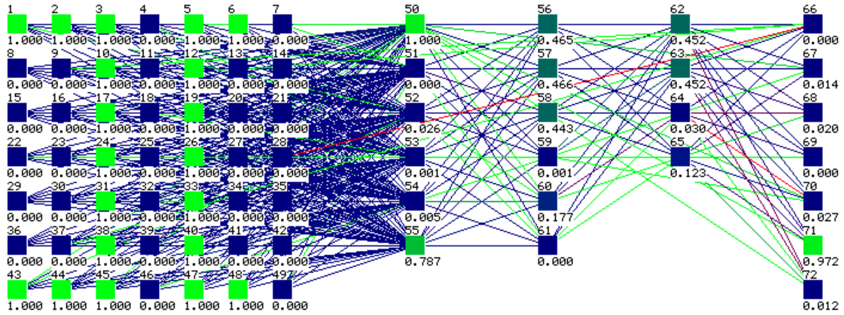


Figure 12: New KBANN Net

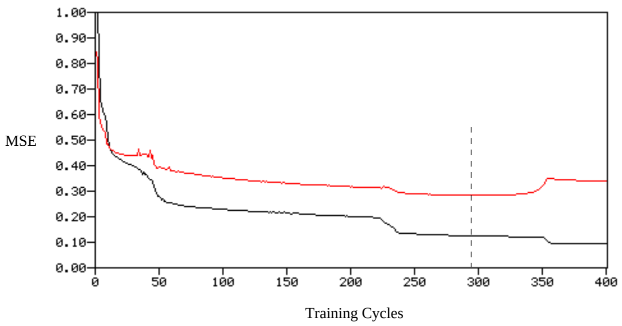


Figure 13: New KBANN Net Learning Curve

7 Training Results: The Vanilla Network

In order to be fair about the comparison between the standard KBANN network and a simple one layer network the same number of units will be used. There were 11 hidden units in the KBANN network, so 11 will be used in the vanilla network (a fully connected network). It should be noted, however, that there is the real possibility that the number of units as specified by the domain theory may be too few to be able to generalize to the concepts that need learning [7]. It may also be the case that the domain theory divided the world up into too many small, and perhaps irrelevant, pieces; which would allow the vanilla net to perform even better. Figures 14 and 15 show the structure and learning curve of the vanilla network. Initially all weights and thresholds were set to zero (before the mandatory perturbation). The learning rate was found (empirically) to be most effective at 1.5.

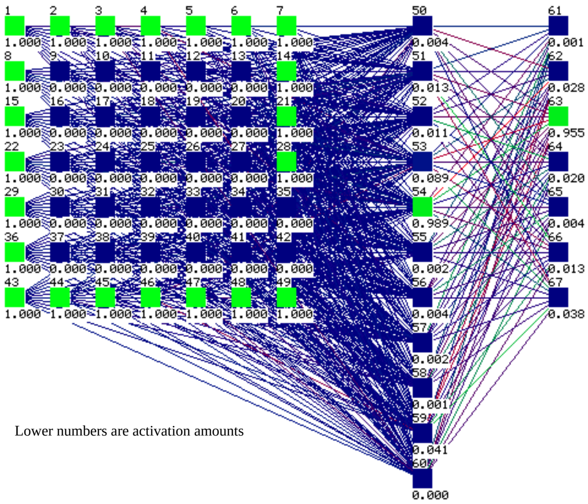


Figure 14: Trained Vanilla Network

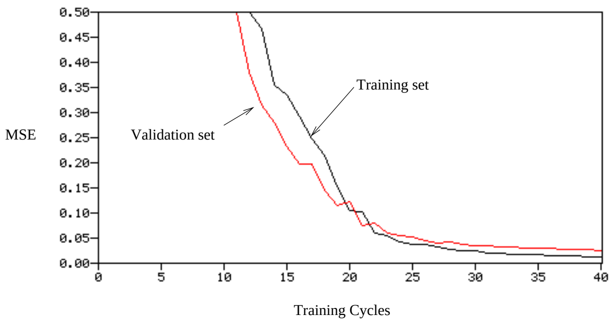


Figure 15: Vanilla Network Training Curve

As can be seen the network is able to correctly identify any shape from the validation set. And it

[7] This is likely what happened in our case

managed an MSE of under 0.05 for *both* data sets in under 40 training cycles. For many tasks, however, a fully connected network using Back-Propagation can be improved upon readily[8]. The question becomes can our rule based knowledge be encoded into a shallow neural network?

8 The SKBANN Algorithm

8.1 Main Assumptions

The SKBANN Algorithm is a method for translating the domain knowledge into an initial neural network in much the same way as KBANN. The domain knowledge must take the form of 'Prolog-like' rules (variable free) , just as with KBANN. The structure of a SKBANN network is the same as a single hidden layer **fully connected** network, as with our 'vanilla' network. The difference is that some of the links from the inputs to the hidden layer will have specific weights and the some of the hidden units will have specific thresholds. That is, some of the hidden units will be initialized to recognize certain *important* features of the input. Central to the developement of SKBANN are the following assumptions:

1. Rules in the database that are *used*[9] often have a high degree of *confidence*[10] associated with them.
2. Rules that are used *closer* to the concepts being learned are more *important*.

To explain these assumptions assume the following domain knowledge exists:

$$r_1 : A \leftarrow D \wedge C$$

$$r_2 : B \leftarrow C \wedge E$$
$$r_3 : D \leftarrow F \wedge G$$
$$r_4 : F \leftarrow H \wedge I$$
$$r_5 : G \leftarrow J \wedge K$$
$$r_6 : C \leftarrow L \wedge M$$
$$r_7 : E \leftarrow N \wedge O$$

Also presume that the concepts being learned are A and B. These rules can been seen in a tree in figure 16.

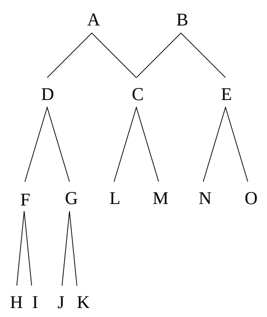


Figure 16: A tree of rules

As an example of assumption 1, There is more confidence in rule r_6 than in, say, rule r_7. This is because, as can be seen in the tree, C is *used* twice while while E only once. It should be noted that even if rule r_1

[8] Consider, for example, feature detectors and time-delay neural nets.
[9] That is, the heads of these rules occur as either mandatory or prohibatory antecedants of other rules.
[10] A measure of how likely it is that the rule is actually valid for the domain.

was $A \leftarrow D \wedge \neg C$ it would not affect the *confidence* in r_6. It is the fact that the rule was used at all that is important, not whether the negation was used or not.

As an example of the second assumption, rule r_7 is more *important* than both rules r_4 and r_5. This is because it occurs at a *lower* level. That is, it has much more direct effect on the concepts we are trying to learn.

These assumptions are just that, assumptions. For certain domains they may hold while for others they may not. For example, the assumptions probably would hold for the rules that doctors use in diagnosing patients. A rule that is relied on heavily (i.e. is used a lot) will have a high degree of confidence associated with it. The rule simply would not have survived as long as it did if this were not the case. For example, the rule:

Conscious $\leftarrow$ **Talking** $\wedge$ **Hand-is-Moving**

is probably a pretty safe bet, and in diagnosing a patient this sort of rule would be used all the time. The second assumption would also hold in this domain. If we are trying to tell if the patient is conscious then the rule:

Talking $\leftarrow$ **Mouth-moving** $\wedge$ **Word-sounds-present**

is more important than some obscure rule many layers deep. These assumptions will be presumed to hold in the shape domain of this experiment. The SKBANN method uses the notion of *strength* to initialize the weights and thresholds of some of the hidden units. A unit with more *strength* is one in which it is harder for the backpropagation algorithm to 'untrain'. That is, units that require large changes to their link weights and thresholds before giving substantially different responses are stronger. In SKBANN, units with high confidence and importance values will be stronger. With the assumptions and terms now defined, the SKBANN algorithm will now be introduced and applied to the shape rules given in figure 5.

8.2 Applying Shallow kBANN

It is assumed that the antecedents of each shape rule (rules 1 to 9) are important for the neural net to learn because if it can learn these it will be well on it's way to recognizing the shape (or concept, if this algorithm will be applied more generally). The importance of each rule can now be determined. Imagine a layering system where the most important concepts are placed at the bottom (call this layer zero). Of all 14 concepts given by the rules (figure 5) the most important are the ones we are trying to learn (i.e. the 7 concepts labeled shape 1 to shape 7). Clearly any concept that directly affects rule 1 to rule 9 will also be of fairly high importance. These shapes are the heads of rules 10 to 12 (The intermediate level rules in figure 5). Basically, it is desireable to let each rule 'fall' as much as possible towards ground zero. It may be asked why rule 13 is at layer 2 while rule 12 is at layer 1. The answer is that the head of 13 is one of the antecedents of a rule in layer 1. That is, rule 11 incorporates the importance of 13. Rule 12, however, is capable of 'falling' to one above zero since none of it's antecedents are heads of a rule in the same or lower level. In summary, level zero depends on levels 1, 2, and 3 (letting 3 be the inputs). Level 1 depends on levels 2 and 3. And level 2 depends only on level 3. Just put each rule as low as possible while maintaining these dependencies [11]. It may also be helpful to draw the tree of rules as shown in the previous section.

Now that we have some notion of the importance, the confidence can be worked out: The more times a concept is used the more vital it is (and therefore the more vital the rule that describes it). Note that it does not matter if the negation is used. It's the fact that it was referenced at all that adds to it's confidence. So, for an overall *strength* level we get:

$$I = \alpha(c/l) \tag{13}$$

[11] The method described here generalizes easily to more rules and more layers

where I is the strength, c is number of times it's used, l is the level, and α is a constant of proportionality. For this experiment α will be set to 1. It may also be desireable to add a constant to c, giving the ability to control how much 'default weight' is given to the confidence rating.

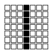
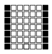
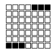
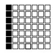
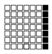
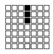
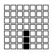
CONCEPT	LEVEL	NUMBER OCCUR	Negative of Bias (I - w/2)	Atomic Predicates	Weight on each link to that predicate
	1	2	2/1 - 2/8 = 14/8	A14 A34 A54 A74	2/4
	1	2	2/1 - 2/8 = 14/8	A17 A77 A11 A71	2/4
	1	5	5/1 - 5/8 = 35/8	A15 A17 A71 A73	5/4
	2	2	2/2 - 1/4 = 3/4	A11 A71	1/2
	2	1	1/2 - 1/8 = 3/8	A17 A77	1/4
	2	2	2/2 - 1/4 = 3/4	A14 A34	1/2
	2	1	1/2 - 1/8 = 3/8	A54 A74	1/4

Figure 17: SKBANN (handi-capped) Weight and Bias Values

Now that the strength of each rule is known, we can map each rule to a hidden unit of the network. Each hidden unit will basically implement the corresponding rule. Since, in this case, there are more hidden units than shapes being learned we could simply make each hidden unit of the network 'recognize' a particular shape. This approach, although it works for this problem, may not be applicable if there are more concepts to learn than hidden units permitted. It has been decided to handi-capSKBANN [12] in this experiment so that it is not allowed to map any of the 7 (level 0) rules directly into 7 hidden units. In fact, if there are more hidden units than concepts to learn then there is no need for the importance or confidence ratings. All that would need to be done is find the atomic predicates (inputs), assign weights of w to those links, and assign a threshold of $-(nw - w/2)$ to that unit (where n is the number of atomic predicates).

In general, if there are more concepts than hidden units then the rules would be ordered by strength and the highest m rules would be mapped to the network; where m is the desired number of hidden units. If a network with more hidden units than concepts is desired (like in our case of 7 concepts because of the handi-cap) then additional weight zero /bias zero units can be added. For this experiment 11 hidden units are desired so 4 units will have threshold zero.

Now, what makes a rule stronger in terms of the actual network? Well, a unit with higher strength will not be affected as easily by changes in the weights of it's input links or it's threshold. This reminds us of having larger thresholds and input link weights (a variation in weight of threshold will affect such a unit less).

Since there will only be one hidden layer of units, the rule that the hidden unit is supposed to implement will have to be expressed in terms of the 49 inputs. This is just a case of finding out what all the atomic

[12] SffBANN is suspected to work extremely well for this domain if we didn't.

predicates of each concept are. For example, the concept that is the head of rule 11 has atomic predicates A17, A77, A11, and A71.

Finally a formula can be given for the weights of the input links (from the base dependencies, or inputs) to a hidden unit:

$$w = I/n \tag{14}$$

where I is the strength of the unit being connected to and n is the number of base dependencies. This yields the same 'midpoint' equation for the unit's threshold:

$$t = -(I - w/2) \tag{15}$$

Figure 17 shows a table of what is being done.

9 Training Results: The SKBANN Network

Figure 18 shows the network that was created after it was trained with learning parameter 1.5 (same as vanilla). The first 7 hidden units correspond , in order, to the 7 concepts in table 17. Figure 19 shows the learning curve. As hoped, the network converges faster! Note that the SKBANN plot was placed directly on top of the old Vanilla training curve so an easy comparison can be made. Although no research has been done into expanding the SKBANN algorithm to multiple hidden layers, I believe it is not difficult to accomplish. The hard question would be 'when is an expansion to more layers required?' not 'How do we do it?' Another possible variation to SKBANN would be to weight occurances of a concept more if they occur in lower levels. That is, if one counts 5 occurances in level 1 it should be more significant than 5 occurances in level 8.

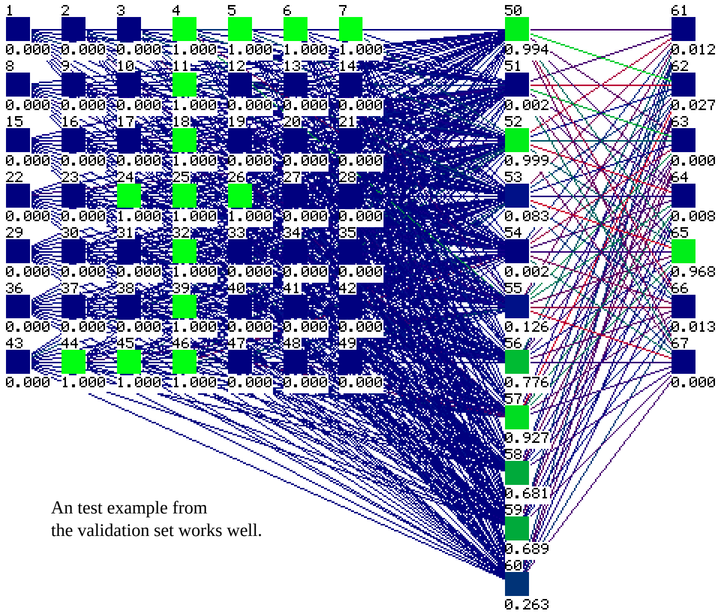


Figure 18: Trained SKBANN Net in Operation

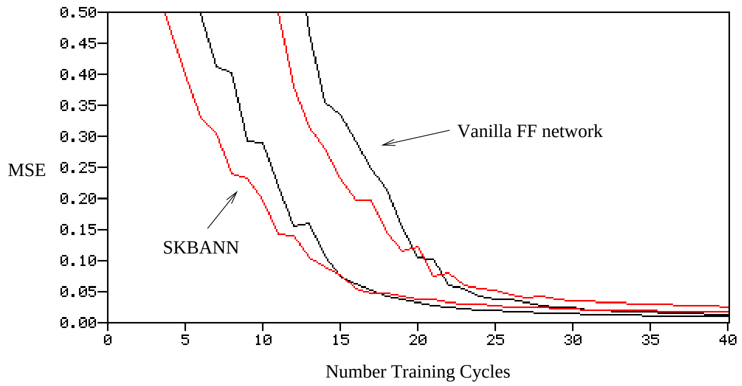


Figure 19: SKBANN Learning Curve

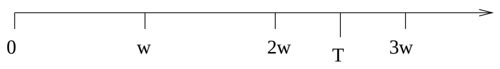
9.1 A Quick Look at the Hidden Units

Figure 20: Two Separately Trained SKBANN Networks

In order to confirm suspicions that the weaker (less strong) units may converge to different concepts, a number of SKBANN networks were trained. In figure 20 we see two of them recognizing the same 'I' shape. As can be seen, the third hidden unit (which is activated) can not get out of recognizing it's respective concept. This is because it is relatively strong (a weight of 5/4). However, the fourth hidden unit (which recognizes the two left most corner inputs A11 and A71), and similarly the fifth, have opposite activations in each network. This is because, during training, they specialized to somthing other than what they were initialized to. This makes sense as they were weaker, having weights of 1/2 and 1/4.

10 Incorporating More Domain Knowledge: Rule Certainty

This sectoin, very briefly introduces the possibility of mapping the certainty rating of a rule into an initial neural network. In some cases a measure of the confidence that a certain rule is correct (or incorrect) may



T: threshold value with highest strength.

Figure 21: Threshold position with respect to link weights

be available. In [PMG95] the confidence is defined as a value ranging from -1 (known false) to $+1$ (known true) with 0 being no idea of the correctness of the rule. If a rule has a confidence level of $+1$ then it's corresponding unit should have it's threshold in a position yielding the highest *strength*.

Figure 21 shows the case when there are 3 antecedants in the rule. Each link gets weight w and the threshold will be set to $(2w + w/2)$ if there is a certainty factor of 1 associated with it. As the certainty factor gets closer to -1 the threshold will have to be placed further away from the midpoint. A certainty of 0 should coincide with one of the interfaces (that is, threshold values of either $2w$ or $3w$ in the diagram). If the rule is known to be false, then the threshold could be assigned a value larger than $3w$ or less than $2w$. Some sort of curve would have to be fit to map the certainty to the distance away from the midpoint.. If the curve was linear then a certainty of -1 would map to $3w/2$ or $7w/2$. The question for further discussion is on which side of the midpoint should it be placed. The author wonders how a network that randomly chooses which side would perform.

11 Conclusions

This report communicated a set of experiments with Artificial Neural Networks. The applicability of the KBANN algorithm was studied and a new system which avoids producing large numbers of hidden layers was tested. The new system outperformed both a vanilla back-propagation feed forward network and the KBANN network. The domain, although artificial, serves to demonstrate that applications do exist where the SKBANN algorithm outperforms the others. However, the general applicability of the SKBANN system has not been explored fully. One rather significant failing in SKBANN is that it fails to specify unambiguously how many hidden units should be used. It is this determination that can affect drastically the performance of a neural network. That is, too many units cause too many degrees of freedom and the network will not generalize well, while too few units and the network may be incapable of representing the necessary concepts and so will be impossible to train (see [Win92, pages 464-469] for a full discussion of this phenomanon). The KBANN system, however, fully specifies the structure of the network, and many researchers [13] claim that this is why KBANN networks are capable of generalizing so well. The number of units is a direct function of the number of rules. KBANN nets have also been shown to be capable of producing lower errors than 'vanilla' nets regardless of the number of cycles the vanilla net was trained with. There is also the question of the structure of the SKBANN network. It only has a single hidden layer and so cannot represent all arbitrary boolean functions. The author believes, however, that extending SKBANN to a multiple hidden layered network would be a relatively straingworward task. As a final note, this is by no means a completely thourough examination of either KBANN or SKBANN. Whether SKBANN will outperform KBANN or 'vanilla net' when there are fewer training examples to work from is still an open question. It is suspected that KBANN did not work too well in this experiment because there were (a) not enough rules given, and (b) the rules produced a network that backpropagation had difficulty training (too deep). Since this report was mostly an exercise in the implementation of ANN's there remains the work of formally linking the behaviour of knowledge-based systems and connectionist systems. That is, most of the ideas presented in this paper were arrived at by an intuition of the behaviour of simulated neural networks, but a more formal analysis is still required.

[13] for example [TS92] and [TSN90]

References

[HL87] Robert V. Hogg and Johannes Ledolter. *Engineering Statistics.* 1987, Macmillan Publishing Company.

[Mit95] Tom M. Mitchell. *Machine Learning.* 1995, McGraw-Hill.

[MV95] Andreas Zell, Gunter Mamier, et al. *Stuttgart Neural Network Simulator User Manual V4.0.*, 1995, Unpublished.

[PMG95] David L. Poole, Alan K. Macworth, Randy G. Goebel. *Computational Intelligence.* 1995, Unpublished.

[RHW86] D. E. Rumelhart, G.E. Hinton, R.J. Williams. Learning Internal Representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* pages 318-363, 1986, MIT Press.

[TS92] Geoffery G. Towell and Jude W. Shavlik. Using Symbolic Learning to Improve Knowledge-Based Neural Networks. In *Proceedings of the Tenth National Conference on Artificial Intelligence,* pages 177-182, San Jose, California, July 1992. AAAI Press.

[TSN90] Geoffery G. Towell, Jude W. Shavlik, and Michiel O. Noordewier. Refinement of Approximate Domain Theories by Knowledge-Based Neural Networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence,* pages 861-866, July 1990, AAAI Press.

[Win92] Patrick H. Winston. *Artificial Intelligence.* 1992, Addison-Wesley.