Self-Splitting Modular Neural Network – Domain Partitioning at Boundaries of Trained Regions

V. Scott Gordon and Jeb Crouson

Abstract—**A modular neural network works by dividing the input domain into segments, assigning a separate neural network to each sub-domain. This paper introduces the self-splitting modular neural network, in which the partitioning of the input domain occurs during training. It works by first attempting to solve a problem with a single network. If that fails, it finds the largest chunk of the input domain that was successfully solved, and sets that aside. The remaining unsolved portion(s) of the input domain are then recursively solved according to the same strategy. Using standard backpropagation, several large problems are shown to be solved quickly and with excellent generalization, with very little tuning, using this divide-and-conquer approach.**

I. Introduction

NEURAL networks traditionally attempt to solve problems by replicating the behavior codified in a training set. A variety of training methods for supervised learning exist. Backpropagation, for example, adjusts the network weights until the desired outputs are produced for each example in the training set, within given criteria. However, there are times when a training set is so large, or the problem so complex, that the network never learns it completely, or learning is too slow to be practical.

Any time that a problem is too large to be solved by any one method or model, it is natural to attempt to use a divide-and-conquer approach, by breaking the problem into smaller pieces and solving the pieces separately. In the case described above, one way of doing this is by applying more than one neural network to the problem at hand. Such systems are often called *multi-nets*. When a multi-net is configured so that the computations of multiple networks are combined into a single answer, it is called an *ensemble*. When a multi-net is configured so that each case is handled by just one of the networks, it is called a *modular* network.

Manuscript received November 30, 2007.

V. Scott Gordon is with the Computer Science Department at California State University, Sacramento, CA 95819 USA (phone: 916-278-7946, email: gordonvs@ecs.csus.edu).

Jeb Crouson was also with the Computer Science Department at California State University, Sacramento (crousonj@ecs.csus.edu).

II. Modular Neural Networks

In a modular neural network, each network is assigned a portion of the input domain. The input domain is divided into partitions, and a separate network is assigned to each partition. After each network is trained, the modular system can be used on other, non-trained cases by directing the input to whichever network was trained on the portion of the domain in which the input belongs.

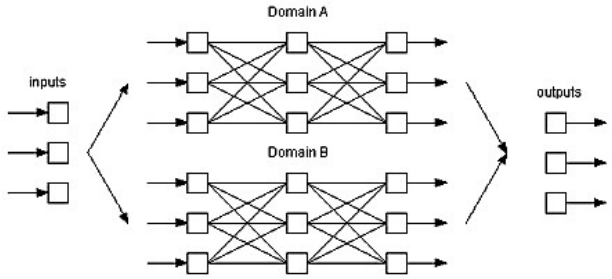

Fig. 1. Modular Network Architecture

In Figure 1, the inputs are routed to *one* of the networks (here shown as either Domain A or Domain B), depending in which portion of the input space those input values are contained. Whichever one network is assigned to those inputs, it (and it alone) produces the corresponding output values.

Modular neural networks were first described in the early 1990s by various researchers [1], [2] and explored further by Sharkey [3], [4]. Chen et al. used a divide-and-conquer method to assign one or more networks to each case, in a sort of hybrid modular/ensemble approach [5]. Olsen-Darter and Gordon used a modular configuration of 120 neural networks for a vehicle control application, but the division of the input domain was determined by experimentation [6].

III. Input Domain Partitioning

Dividing the input domain into segments must take into account the multi-dimensionality of most problems. For example, a problem with 5 inputs has an input domain which can be divided along any combination of its 5 dimensions. Each input variable has its own range, and could be split independently. *Splitting one variable at one value effectively divides the entire input domain into two pieces.* Similarly, splitting all five variables in half simultaneously would create 32 separate partitions. Which dimension(s) to divide, and where, are the focus of this paper.

A. Partitioning by Hand

In a previous study, Olsen-Darter and Gordon [6] described partitioning an input domain of five variables by hand-selecting a number of divisions for each variable. Some variables were divided into 2 equal partitions, others into 3 or 5 equal partitions. The input domain was thus segmented into 120 equal-sized sub-domains, and each was assigned its own separate neural network. Each training case was then examined to see in which sub-domain it belonged, and was then routed to the appropriate network. The multi-net successfully learned a large training set (30,720 cases). The splitting of input variables was arrived at by trial and error.

Partitioning by hand proved to be tedious, and it is not clear if a significant improvement could be obtained by choosing a different set of partitions. It also is not clear whether equal partitions is necessarily best, or whether there are more effective values on which to partition depending on the problem. A more effective partitioning might result in faster training, or result in fewer networks being required to learn the training set.

B. Automatic Partitioning, or Self Splitting

Rather than partitioning the input domain ahead of time, our approach is to try to learn the training set using as few networks as possible. The *self-splitting* approach starts with a single network, and proceeds as follows:

Algorithm 1 – Self Splitting Framework

1. attempt to solve the training set
2. if success, the input domain is solved ... stop.
3. if failure:
   (a) select a variable and value(s) for splitting
   (b) assign new network(s) to each sub-domain
   (c) distribute training cases by sub-domain
   (d) recursively apply algorithm 1 to each network

Apply recursively until all networks learn their subset of the training cases.

Note that a modular network trained according to the framework given in algorithm 1 will invariably result in a set of networks with varying sizes of input sub-domains. That is, some domains will solve more quickly and require less splitting than others. This is unlike the hand-partitioning described earlier [6], where partitions were of equal size regardless of the nature and subtlety of the data in the various partitions.

The framework still leaves undefined how a variable is chosen, and where the split occurs. We examined two splitting methods: by *centroid*, and by *trained region:*

1) Splitting by Centroid

One simple and natural approach to selecting a split point is to split variables in a round-robin fashion, thus splitting on the next variable who's turn it is to split. To do this, each variable is given a sequence number, and each network needs to keep track of which variable was split when its sub-domain was defined. Then, if the network fails to train, it is split on the next variable in turn, dividing the sub-domain into two sub-sub-domains. When splitting by centroid is used, a variable is split in exactly one point, producing two unsolved partitions.

2) Splitting by Trained Region

An alternative method for selecting split point(s) is to first examine the results of training. Even though the network failed to learn its entire training set, that does not mean that it learned nothing. In fact, it may have learned a substantial portion of the training set. It is possible, in most cases, to extract the trained portion, and split along the solved partition boundaries.

For example, consider the case where a network is training on a problem of 5 variables (A, B, C, D, and E), and it manages to learn every training case where the value of C was in the range [4.0 ... 6.0]. That network could then be permanently assigned the following partition:

A: all values in original domain
B: all values in original domain
C: [4.0 ... 6.0]
D: all values in original domain
E: all values in original domain

When splitting by trained region is used, a variable's domain is split at two points *x* and *y*, producing one solved partition and two unsolved partitions:

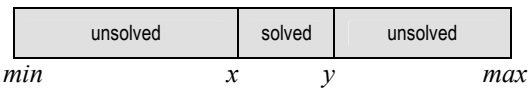

Fig. 2. Splitting by trained region

Continuing the example described above, the two unsolved sub-domains (regions) would be:

A: all values in original domain
B: all values in original domain
C: [*min* ... 4.0]
D: all values in original domain
E: all values in original domain

and

A: all values in original domain
B: all values in original domain
C: [6.0 ... *max*]
D: all values in original domain
E: all values in original domain,

where *min* and *max* are the extrema of the values for C in the original domain (or sub-domain if this is already a recursively-defined sub-domain).

It is possible, even likely, that trained regions can be identified for more than one variable. As well, there may be more than one solved region for a particular variable. Our approach is to find the *largest solved region*. In that way, we try to favor networks that generalize rather than networks that specialize.

Here, for completeness, it should be noted that it is possible for a region to occur on one end of a variable's domain, for example if all training cases are solved when a particular variable is greater than some value *x*. In such cases, the domain is partitioned at one point, producing two partitions, one solved and one unsolved:

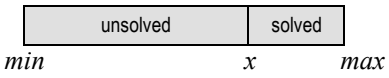

Fig. 3. Splitting by trained region, special case

IV. Finding Trained Regions

Assuming that training has failed to converge after the specified maximum number of training epochs, the next step in a self-splitting network is to search for the largest successfully trained region. Each variable needs to be checked for trained regions, and in case there exist multiple trained regions, the largest is selected.

The process begins by sorting the training data on the first input variable. The training set is then scanned for the largest contiguous number of pairs for which the unsuccessfully trained network had produced an output that was within the training criterion. Note that if two or more training cases have the same value for the variable in question, that all cases with that value must output correctly for that value to be considered successfully contained within a trained region. This is because if there is even one occurrence of a variable's value for which it did not train successfully, then that value was not completely learned.

Algorithm 2 illustrates the process of identifying the largest trained region:

ALGORITHM 2 – IDENTIFYING REGION BOUNDARIES

1. For each input *s* of the current network:
   (a) sort the training pairs on *s*
   (b) scan the pairs from lowest to highest value
   (c) find the subrange $min_s...max_s$ for which:
   (i) all pairs within $min_s...max_s$ are solved, and
   (ii) no other subrange contains more such pairs
   (d) if the size of subrange is greater than any found so far (for other inputs), copy *s* and $min_s...max_s$ into *i* and $min_i...max_i$.
2. If an input (*i*) with a trained region was found,
   split on variable *i* and subrange $min_i...max_i$
   else
   split by centroid on the variable with the largest domain.

Consider the following very small illustrative example, showing training data and associated training results. An "X" indicates that the network output was sufficiently within the training criteria. Absence of an "X" indicates that the network failed to learn that case:

case #	input A	input B	success
1	15.5	2	X
2	8.5	4	
3	12.0	1	X
4	19.0	0	X
5	19.0	1	
6	6.0	3	X
7	12.0	6	X

Fig. 4. Training data (example), with results

The example data above includes 7 training cases. The problem has two inputs, A and B. Expected and actual output values are not shown. However, an X indicates that the actual output was correct to within the training criteria. We start by sorting the training data on input *A*, as shown in Figure 5:

case #	input A	input B	success
6	6.0	3	X
2	8.5	4	
3	12.0	1	X
7	12.0	6	X
1	15.5	2	X
4	19.0	0	X
5	19.0	1	

Fig. 5. Training data sorted on input *A*

At first glance, it would appear that there is a contiguous trained region of size 4, for input *A* over the range *12.0...19.0*. However, upon closer inspection, the value 19.0 cannot be included in this range, because there is a training case for which the value of *A=19.0* was not successful (case #5). Therefore, the largest contiguous training region found for variable *A* is actually of size 3, and is for the subrange *12.0...15.5*.

We next sort the training data on input *B*, as shown in Figure 6:

case #	input A	input B	success
4	19.0	0	X
3	12.0	1	X
5	19.0	1	
1	15.5	2	X
6	6.0	3	X
2	8.5	4	
7	12.0	6	X

Fig. 6. Training data sorted on input B

Clearly, for input *B*, the largest trained region is of size 2 and is for the subrange *2...3*. Since this region is smaller than the largest region found for input *A*, we would choose to split on input *A*. This gives rise to the following definition of trained and untrained regions:

untrained region:
input A *6.0...12.0*
input B *0...6* (all values in original domain)
cases: *6, 2, 3, 7*

trained region:
input A *12.0...15.5*
input B *0...6* (all values in original domain)
cases: *3, 7, 1*

untrained region:
input A *15.5...19.0*
input B *0...6* (all values in original domain)
cases: *1, 4, 5*

For the trained region, we store both the ranges of input values, and the network weights. Note that these weights represent a neural network configuration that was successfully used to solve the cases in that range. The untrained regions are assigned to new neural networks which are trained just on those cases. Since there are fewer training cases for each of those sub-domains, they represent smaller problems than the original, and thus a divide-and-conquer strategy.

Note that we have also decided to *overlap* the regions. This is in an effort to ensure that the entire range of possible values is sufficiently modeled. Overlapping does give rise to certain details that must be considered to avoid a *runaway* splitting, which can happen when a sub-domain is the same as the original domain. These problems are solved by verifying the cardinality of training cases before and after splitting.

It is possible that no trained regions are found. This can happen if for all solved cases there happen to exist other unsolved cases that have duplicate input values. When no trained regions are found, or if all trained regions would give rise to a runaway split (one that duplicates an untrained region), we perform splitting by centroid, so that the divide-and-conquer strategy can continue anyway.

It is then also possible for splitting by centroid to fail, for the same reasons as described above for splits by trained region. When this happens, training fails. This situation is most likely to occur when the training data contains large numbers of duplicate input values, such as when training cases are made up of many permutations of a few input values on several variables. Sometimes this situation can be mitigated by adding a small amount of noise to the input values beforehand.

Even if training fails, it is possible for training to proceed by eliminating one or more of the overlap points, although when this is done there is of course a risk of poorer modeling of the application problem.

V. Running the Fully Trained Multi-Net

After all neural networks have been successfully trained on their respective sub-domains, they can be tested or deployed on other cases:

Algorithm 3 – Running the Multi Net

1. retrieve a set of new (untrained) inputs
2. for each network, if the inputs are in its sub-domain, compute the output of the network (there is usually *only one* such network).
3. if only one network responded to step 2, the output of that network is the output of the multi-net.
   if more than one network responded, the final output is averaged from the outputs of those networks.

In Algorithm 3, the trained multi-net acts as a modular network, described in section 2. The only exception is in those cases where the inputs fall exactly on partition boundaries. In those cases, splitting had generated overlapping partitions, and more than one network may have been assigned to that situation. In those less frequent cases, the input falls within the domains of two different networks. Processing in those cases is similar to a small ensemble, where the outputs of the networks involved are averaged. An acceptable alternative would be to simply select one of the networks at random, since both networks were trained successfully on the boundary case and both should therefore output very similar acceptable values.

VI. Experiments

We implemented the self-splitting modular network in C++, using the vector container class from the Standard Template Library (STL). It has a simple console-based user interface, with a configuration file given as a parameter on the command line. The configuration file includes problem specific settings, desired network topology (number of layers, nodes per layer, etc.), and the location of training and testing data. It currently runs on Windows and Linux workstations.

For each experiment, we examined both the speed of learning and the ability of the final multi-net to generalize. In all comparisons, networks were given a maximum of one billion iterations (totaled amongst all networks in the multi-net) to converge, or the learning is considered to have failed.

Standard backpropagation with momentum and scaling was used in each case.

For each test problem, we tested: (a) a single neural network, (b) a multi-net using self-splitting by centroid, and (c) a multi-net using self-splitting by trained region. We always utilized the following parameters:

Learning Rate	0.3
Momentum	0.85
Scaled input range	0.1 - 0.9
Maximum iterations before splitting	1,000,000

Fig. 7. Experimental parameter settings

A. Test Problems

We used five problems which were obtained from the webpage of the Image Processing and Neural Networks Lab at the University of Texas at Arlington [7], and one problem obtained from the Carnegie Mellon University School of Computer Science AI Repository [8]. Details for each problem are shown in Figure 8:

Problem	Source	# inputs / # outputs	# training cases	# testing cases
FM demodulator	UTA	5 / 1	1024	1024
Arabic Numerals	UTA	16 / 4	3000	3000
Matrix Inversion	UTA	4 / 4	2000	2000
Power Load Forecasting	UTA	12 / 1	1415	1413
Surface Parameter Inversion	UTA	8 / 7	1768	1000
Two-Spiral	CMU	2 / 1	1058	1056

Fig. 8. Test Problem Details

The *Arabic Numerals* and *Two-Spiral* problems are classification tasks. The others are problems that require the network to approximate various functions. The Two-Spiral problem was generated using CMU's benchmark data set generation software [8].

For each test problem, we chose training criteria (permissible errors for outputs during training), testing criteria (permissible errors for outputs during testing), and network topology (number of nodes in each network layer, including bias nodes). These values are shown below in Figure 9:

Problem	Topology	Training Criteria	Testing Criteria
FM demodulator	6-6-1	0.02	0.05
Arabic Numerals	17-13-11-4	0.4	0.4
Matrix Inversion	5-5-4	0.05	0.1
Power Load Forecasting	13-13-1	2.0	2.5
Surface Parameter Inversion	9-9-7	0.5	0.6
Two-Spiral	3-4-1	0.4	0.4

Fig. 9. Test Problems and Network Details

Descriptions of each test problem are shown in Figure 10. They are each taken (or paraphrased) from descriptions given in their source material.

FM Demodulator – Trains the network to demodulate an FM signal that contains a sinusoidal message [7].

Arabic Numerals – Trains the network to recognize Arabic numerals written by different people [7].

Matrix Inversion – Trains the network to invert random 2x2 matrices [7].

Power Load Forecasting – Trains the network to forecast the power load for fifteen minutes from the current time based on the last ten minutes of power load and two control error values [7].

Surface Parameter Inversion – Trains the network to invert the surface scattering parameters from an inhomogeneous layer above a homogeneous half space, where both interfaces are randomly rough [7].

Two Spirals – Trains the network to distinguish on which of two intertwined spirals a point lays [8].

Fig. 10. Test Problem Descriptions

B. Results

For each test problem, a single neural network was unable to converge successfully on the training data after one billion iterations, with the parameters and criteria as given earlier.

Self-splitting modular networks fared considerably better. When splitting by centroid was used, 3 of the 6 problems were solved. When splitting by trained region was used, 5 of the 6 problems were solved. The problems were solved with varying degrees of generalization, with the results summarized in Figure 11. *Training Epochs* is the total number of backpropagation iterations it took to completely solve the training data within the required training criteria. *#Networks* is the number of networks produced in the multi-net (due to splitting). *Generalization* is the percentage of (untrained) testing cases that produced outputs within the testing criteria, after training had completed.

C. Discussion

Examining the data, it is clear that splitting by trained region performs consistently better than splitting by centroid. This is not surprising, since it is an intuitively more intelligent approach, and one that leverages the work done during training. Splitting by centroid did perform slightly better in one case (Matrix Inversion), although the training was slower.

Problem	Training Epochs centroid vs. trained region	# Networks centroid vs. trained region	Generalization centroid vs. trained region
FM demodulator	284,884 62,110	250 95	89 % 97 %
Arabic Numerals	unsolved 11,074	n/a 58	n/a 84 %
Matrix Inversion	82,066 61,223	133 204	90 % 86 %
Power Load Forecasting	unsolved unsolved	n/a n/a	n/a n/a
Surface Parameter Inversion	unsolved 97,242	n/a 279	n/a 49 %
Two-Spiral	45,350 34,146	45 63	94 % 98 %

Fig. 11. Experimental Results

VII. CONCLUSION

Adding self-splitting to a standard neural network with backpropagation results in considerable improvement on the difficult problems that we tested. Only one of the problems was unable to train within the total time we allotted.

When we first considered the self-splitting approach, we were concerned that subdividing the problems would adversely affect generalization by missing the "big picture." Yet, despite the carving of the problems into dozens of sub-problems, generalization was generally very good, sometimes excellent. Only in one case was generalization after training poor.

We contend that difficult problems, particularly those that represent real-world situations, are probably complex and multimodal in nature. Most pragmatic solution approaches strive to break such problems into manageable components, and our approach reflects this. In fact, in the one task that could be considered contrived, or requiring a unified model – the two-spiral problem – generalization is extremely good (98%) despite the problem being divided into 63 pieces.

Unlike previous studies, automatic splitting requires no analysis of the problem to help decide how to subdivide it.

Our experiments involved virtually no special tuning for each of the various problems. The only settings made were topology of the network, which we generally set such that the number of nodes in each hidden layer was equal to the number of inputs +1 (to account for the bias node). It is possible that, had we taken more effort to hand-tune the topology for each problem, performance might have been even better.

VIII. Future Work

The self-splitting modular neural network raises a number of questions. Obviously, we should re-examine our parameter choices and see if other selections would work better with this algorithm. Similarly it is possible that some long-held assumptions could be re-examined.

For instance, it is commonly assumed that training cases should be randomly shuffled prior to training, rather than sorted. Perhaps, since the splitting algorithm is attempting to find useful chunks within the sub-domain, it would be useful to do *some* sorting before training, such as on one of the input variables (while leaving the others unsorted). This might encourage the formation of larger sub-domains on which to split, reducing the number of networks produced.

An even more logical change would be to replace backpropagation with a training method that can utilize a fitness function. This would enable us to drive the training by rewarding solutions that find larger chunks, rather than solutions that reduce sum squared error. In a related work, a simple random annealing method is shown to be suitable and effective for this task [9].

The splitting algorithm itself, as described in this article, is a simple one utilizing only one dimension at a time. More sophisticated methods which look for solved chunks across multiple dimensions, or via clustering techniques, need to be explored.

Finally, there is nothing about the splitting algorithm that inherently requires the use of supervised learning – or even the use of neural networks. We have not yet tried using the splitting algorithm with other training methods, nor with even simpler approximation techniques.

References

[1] R. A. Jacobs and M. I. Jordan, *A Modular Connectionist Architecture for Learning Piecewise Control Strategies*. Proc. of the American Control Conference, pp 343-351, 1991.

[2] S. D. Whitehead, J. Karlsson, and J. Tenenberg., *Learning Multiple Goal Behavior via Task Decomposition and Dynamic Policy Merging*. Robot Learning, MIT Press, 1992.

[3] A. Sharkey, *On Combining Neural Networks*, Connection Science, 8(3/4): 299-314, 1996.

[4] *Combining Artificial Neural Networks – Ensemble and Modular Multi-Net Systems*. ed. A. Sharkey, Springer-Verlag, 1999.

[5] K. Chen, L. Yang, X. Yu, and H. Chi, *A Self-Generating Modular Neural Network Architecture for Supervised Learning*, Neurocomputing 16: 33-48, 1997.

[6] M. Olsen-Darter and V. Gordon, *Vehicle Steering Control Using Modular Neural Networks*, IEEE International Conference on Information Reuse and Integration (IRI), pp 374-379, 2005.

[7] L. Pramod, *Training Data Files*, University of Texas at Arlington, Image Processing and Neural Networks Lab, 26 April 2006: http://www-ee.uta.edu/eeweb/IP/training_data_files.htm

[8] M. White, *Two-Spirals Benchmark Data Set Generator*, 26 April 2006:http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/bench/cmu/bench.tgz

[9] V. Gordon, *Neighbor Annealing for Neural Network Training*, to appear in the International Joint Conference on Neural Networks (IJCNN 2008).