Neural Networks for Predicting the Behavior of Preconditioned Iterative Solvers

America Holloway and Tzu-Yi Chen

Computer Science Department, Pomona College, Claremont CA 91711, USA
{america,tzuyi}@cs.pomona.edu

Abstract. We evaluate the effectiveness of neural networks as a tool for predicting whether a particular combination of preconditioner and iterative method will correctly solve a given sparse linear system $Ax = b$. We consider several scenarios corresponding to different assumptions about the relationship between the systems used to train the neural network and those for which the neural network is expected to predict behavior. Greater similarity between those two sets leads to better accuracy, but even when the two sets are very different prediction accuracy can be improved by using additional computation.

Keywords: iterative methods, preconditioners, neural networks.

1 Introduction

Preconditioned methods are generally used to solve large, sparse linear systems $Ax = b$ when the time or memory requirements of a direct solver are unacceptable. Ideally the preconditioner is inexpensive to compute and apply, and the subsequently applied iterative solver works more effectively on the preconditioned system. Unfortunately, because of the number and variety of existing preconditioners, as well as the lack of a good understanding of how well any preconditioned solver is likely to work for a given system, choosing a preconditioner is rarely straightforward. As a result, there is interest in helping users choose a preconditioner, with guidelines for setting parameter values, for their particular system. One option is to use the results of extensive experiments to suggest rules-of-thumb (eg, [1–3]) — this technique can suggest default settings but not when or how to adjust those defaults. Another, more recently explored option, uses machine learning techniques to extract meaningful features for predicting the behavior of preconditioned solvers (eg, [4, 5]).

This paper contributes to the growing body of knowledge on using machine learning techniques for this problem by asking whether neural networks can be used to predict the behavior of preconditioned iterative solvers. In addition, this work explores a broad range of scenarios in which a user might wish to predict the behavior of a preconditioned solver. The first aspect considers the context for the problem and the amount of highly relevant data that is likely to be available. Is the system completely novel and therefore likely to be quite different from other systems that have already been solved? Does the system closely resemble others that have already been solved and analyzed? Or has the user already tried to solve the exact system using other preconditioners in the past? The second aspect takes into consideration the relative cost of computing the preconditioner and performing the subsequent iterative solve. If the first is believed to be inexpensive, how much of an improvement can be gained by first computing the preconditioner and using some data from that computation in predicting the behavior of the subsequent solve?

After first describing how this work fits into the larger body of work on predicting the behavior of preconditioned iterative solvers, we describe the range of parameters and scenarios that was explored, and finally discuss the results. Overall, we find that neural networks are promising, even when the inputs consist of only easily computable matrix values. We explore the importance of training a neural network on matrices that are similar to those for which it is expected to predict results. In the case where the two sets of matrices are different and the accuracy suffers, we suggest a method for increasing the accuracy of the system using some additional computation.

2 Background

Predicting the behavior of preconditioned iterative solvers has been studied in papers including [3–5]. Throughout, assorted structural and numerical features are extracted from a matrix, and attempts are made to use those features to create a classifier that can take a matrix and recommend a particular preconditioned solver by determining whether it is likely to converge to a solution.

In [3] they solve each matrix using different versions of preconditioned GMRES. Each matrix is labeled according to the behavior of the preconditioned solver (e.g. zero pivot encountered or solve successful). For each preconditioner, they describe the features of the matrices that correspond to a successful solve in order to give helpful guidelines to the user. In [5] a combination of support vector machines and clustering is used. After extracting assorted structural and numerical features from each matrix, they cluster the matrices based on these features and attempt to describe each cluster by its performance when different solvers are applied. Depending on the consistency of behavior within each cluster, they may also do further classification using support vector machines. In [4] boosting and alternating decision trees are used to create a classifier. In all cases, predictions and recommendations are generated for new matrices by first extracting the same set of features. Our work follows the same general framework, but we use a different machine learning technique, and we explore a broader range of scenarios that might be of interest to a user. Furthermore, our primary goal is not to describe a large software system, but simply to evaluate a single machine learning technique.

In particular, we consider neural networks, a supervised machine learning technique which has been used successfully in a variety of applications where the output is believed to be a complex function of possibly noisy input data.

Informally, a neural network is a web of interconnected nodes, each of which has a set of weighted input edges and output edges. Each node computes a linear function of the input signals and the edge weights, then passes the result to an activation function which determines the output value for the node. Training a neural network refers to learning the correct edge weights so that given an input signal, the correct output signal is achieved. For a more thorough introduction, see [6]. One of the disadvantages of neural networks is that, in general, little insight can be gained into why a network makes particular predictions. Thus in this initial study we focus primarily on making accurate predictions, rather than on also extracting rules for explaining the preconditioner behavior on a finer scale.

3 Methodology

Our primary goal is to train a neural network to distinguish between preconditioned systems that can be solved by a given iterative solver, and those that cannot. The network is first trained on examples, or instances, that each correspond to a specific preconditioned linear system. We then use the network to classify instances that were not a part of the training set.

In this section we describe the creation of the sample space, and the training of the network. We also describe the explicit construction of training and testing sets to model three scenarios users might encounter, as well as how information from the preconditioned system can be used to increase prediction accuracy.

3.1 Construction of the Sample Space

Our test suite consists of 260 matrices from the University of Florida Sparse Matrix Collection [7], on each of which we ran ILUTP_Mem [8] preconditioned GMRES(50) [9]. ILUTP_Mem allows the user to set the values of 3 parameters that together control the memory required, and the accuracy provided, by the preconditioner; we used a range of values for `lfil` (0,1,2,3,4,5), `droptol` (0,.001,.01,.1), and `pivtol` (0,.1,1). In addition, before computing the ILU factorization, matrices were first permuted and scaled using MC64 [10, 11] for stability, and then using COLAMD [12, 13] for sparsity. Note that by using value-based ILU preconditioners, instead of the level-based ILU preconditioners used in [5], we are working with a much larger number of parameter values. Each matrix was solved 72 different times, each time using a different combination of parameter values. This generated a sample space of over $18,000$ instances. Of these instances 49.7% of them were successfully solved using preconditioned GMRES(50).

For input to the neural network we computed an assortment of statistics about each matrix. We used 35 input values including 32 structural and numerical statistics used in [5], as well as the values of the lfil, droptol, and pivtol input parameters to the ILUTP_Mem preconditioner. Thus an instance in the sample space is a 35-tuple: 32 extracted matrix features, and the values of the 3 parameters to the preconditioner. In some situations described below we used 3 additional input values that were computed by running the ILU preconditioner.

3.2 Neural Network Parameters

The neural network used consisted of an input layer, a hidden layer, and an output layer. The output layer contains a single node whose output was either a 0, indicating that the preconditioned solver should be successful, or a 1, indicating otherwise. The number of hidden nodes was initially chosen by the Baum-Haussler heuristic, although some experiments were done with modified values. The weights were initialized with random values from the range $[-.3, .3]$ and the learning rate was set at .3. (For more discussion of these terms, see [6].)

We use the Backpropagation algorithm to train the neural network because, despite its simplicity, it can yield powerful results for multilayer neural networks. After each training instance is propagated through the neural network, the actual output is compared to the desired output to obtain an error value. The error is propagated backward through the network and used to update the weight values maintained at each node. Once the entire training set has been propagated through the network, the network is tested on the validity set. If the error over the entire validity set has increased since the last iteration, the network patience is decreased. Training stops when the network patience reaches zero, or when the error on the validity set falls below some threshold value. Once the training stops, the weight values that correspond to the lowest error over the validity set are loaded into the neural network, and the accuracy of the neural network is evaluated on a set of test matrices.

Since a sample space size of $18,000$ instances is smaller than the size recommended by the Baum-Haussler Rule, we used 17-fold cross validation. After choosing 1000 instances for the test set, we divide the remaining instances into 17 potential validity sets. We then train and test our neural networks 17 times, each time using the same test set, but a different validity and training set.

3.3 User Scenarios

When assigning instances to the testing, training, and validity sets, we considered three different scenarios that a user might be interested in:

Previously Solved: This corresponds to the situation where the system of interest has been studied before, but perhaps the user is curious about the likely effectiveness of a different preconditioner or of changes to user-set parameters. To simulate this scenario, we randomly choose 1000 instances from the entire sample space to create the test set. The remaining instances are used for the validity and training set. Thus for any single matrix, the instance generated from trying to solve it using a particular setting of the parameters might be in the testing set, while another instance with different parameter settings might end up in the validity or training sets.

Novel: This corresponds to the situation where the system comes from a novel application and is unlike any system seen before. In this scenario, we randomly choose 15 matrices to create the test set. If one of these matrices belongs to a family of matrices, we throw out these siblings so that they are not used in the training or validity sets.

Family: This corresponds to the situation where the system is similar to others seen before (e.g. resulting from a finer meshing of an existing problem). In this scenario, we randomly choose 15 matrices to create the test set. We allow other matrices in the same family to remain in the training and validity set. However for a given matrix in the testing set, any of the 72 instances generated by this matrix will only occur in the testing set and not in either the training or validity sets.

3.4 Using Information from the Preconditioner

Finally, we consider both the case where a user hopes to predict success based solely on information about the system, and the case where a user is willing to compute the preconditioner first and use information from this to predict the success of a subsequently applied iterative solver. In the latter case, we include information about the number of nonzeros in the computed preconditioner as well as a value indicating whether the preconditioner was successful. This creates an extended set of 38 input values to the neural network: 32 extracted matrix features, 3 parameters to the preconditioner, and 3 additional values from the preconditioned system. In both cases, whether information about the computed preconditioner is used or not, we consider all three scenarios described in the above section.

4 Results

In this section we first make some general observations, then present the results when only information from the system is used for prediction, and finally the results when additional information from computing the preconditioner is used. Since a user could be concerned about various forms of accuracy, we give the overall accuracy for all three user scenarios as well as the breakdown of incorrect responses into false positives (type 1 errors in which the network mistakenly predicts that a system is solvable) and false negatives (type 2 errors in which the network mistakenly predicts that a system is unsolvable).

4.1 General Observations

In each scenario we varied the number of hidden nodes, testing a range of values starting from the number recommended by the Baum-Haussler heuristic and increasing until the network performance stopped improving. We found that the number of hidden nodes affected the accuracy of the neural network, although more hidden nodes was not always better. Since the difference in accuracy was typically around 2%, and never more than 5%, in the rest of this section we always use the best result obtained over all the number of hidden nodes tried. In addition, since we used 17-fold cross validation we report on combined results across all 17 runs.

4.2 Using Information from Only the System

Initially we assume that the neural network is given only information about the matrix and the values of the `lfil`, `droptol`, and `pivtol` parameters given to the preconditioner. Table 1 summarizes the results under each of the three scenarios described in the previous section.

Table 1. Accuracy of classification with 35 input parameters

	Previously Solved	Novel	Family
Correctly classified	92.5%	67.9%	79.1%
False positives	3.9%	14.6%	11.4%
False negatives	3.7%	17.5%	9.6%

Looking at the first row of Table 1, the best performance is in the Previously Solved case, where 92.5% of instances in the test set were correctly classified. In this case, the number of hidden nodes had little effect. However in the other two situations, we found that the performance of the neural network increased to a certain point as the number of hidden nodes increased, and then began to decrease. For the Novel case the optimal number of hidden nodes was 36 and with the Family scenario it was 38.

The other rows of Table 1 shows the type of errors made in each scenario. Overall, the neural networks seem equally likely to mistakenly predict that the system is solvable versus unsolvable.

4.3 Using Information from the Preconditioner

We then repeated the experiments for all three scenarios above using three additional input parameters: whether the incomplete factorization was successful, and the number of nonzeros in the upper and lower triangular incomplete factors. Not surprisingly, as shown in the first row of Table 2, the accuracy increased in all three cases.

Table 2. Accuracy of classification with 38 input parameters

	Previously Solved	Novel	Family
Correctly classified	96.6%	73.5%	80.4%
False positives	1.1%	13.1%	11.2%
False negatives	2.3%	13.3%	8.3%

In the Previously Solved case, we correctly predicted the behavior of the preconditioned solver for 96.6% of the testing instances. This number was very consistent. Even though we report on the best performance (which was achieved using 38 hidden nodes), we note that the variance over all the different number of hidden nodes tried was only .006.

At the other extreme, in the Novel case we were only able to predict the behavior of the solver for 73.5% of the testing instances. The performance also varied more (with a variance of 2.39) as a function of the number of hidden nodes. The best performance here used 28 hidden nodes.

If we now look at the type of errors, we see that type 2 errors (false negatives) are more common than type 1 errors in the Previously Solved scenario. This means that the neural network is more likely to predict that a system cannot be solved when, in fact, it can — however, since examples in the first scenario are overwhelmingly correctly classified, this represents a very small number of incorrect predictions. In the Family case the neural network is slightly more likely to incorrectly predict that a system can be solved when it cannot, and in the Novel scenario there is little bias in either direction.

5 Discussion

In this paper we evaluate the effectiveness of neural networks as a tool for predicting the behavior of preconditioned iterative solvers under a range of scenarios, using matrix statistics that are relatively cheap to compute. The largest factor influencing prediction accuracy is the degree to which the training set is representative of the testing set. When the two sets are very similar, the neural network is highly accurate even when inexpensive matrix features, versus more expensive information about the preconditioner, are used as input. Not surprisingly, as the two sets become less similar, the accuracy drops. However, even when the two sets bear minimal resemblance to each other as in the Novel case discussed previously, the accuracy can be improved by adding information from the computed preconditioner. It is unfortunate that these neural networks do not seem strongly biased towards either type 1 or type 2 errors, since it means a user cannot make any assumptions about whether the prediction is likely to be conservative.

We are currently experimenting with using principal component analysis both to reduce the training time for the neural network, and to explain the behavior of the neural network. Preliminary results show that applying PCA to the sample space before training the neural network allows us to get the same accuracy (96% in the Previously Solved case), using only 15 inputs instead of 38. This points to a high level of redundancy and noise in the data. In addition, we plan to apply PCA to the weight values in the neural network. The hope is that this will determine which features are the most significant for prediction. In the situation where a user is interested in a completely novel system, this may enable us to create training sets of matrices that are similar to that system in a meaningful way.

Overall, we find these initial results encouraging, although the fact that we only considered ILU-preconditioned GMRES(50) limits the generalizability of these results. Clearly in the future it would also be desireable to perform these tests over a larger number of preconditioners, iterative solvers, and matrices.

Acknowledgements. This work was funded in part by the National Science Foundation under grant #CCF-0446604. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. T.-Y. Chen.: Preconditioning sparse matrices for computing eigenvalues and solving linear systems of equations. PhD thesis, University of California at Berkeley (December 2001)
2. E. Chow and Y. Saad.: Experimental study of ILU preconditioners for indefinite matrices. J. Comp. and Appl. Math. **16** (1997) 387–414
3. S. Xu, E. Lee, and J. Zhang.: An interim report on preconditioners and matrices. Technical Report Technical Report 388-03, Department of Computer Science, University of Kentucky (2003)
4. S. Bhowmick, V. Eijkhout, Y. Freund, E. Fuentes, and D. Keyes.: Application of machine learning to the selection of sparse linear solvers. Submitted (2006)
5. S. Xu and J. Zhang.: Solvability prediction of sparse matrices with matrix structure-based preconditioners. In Proceedings of Preconditioning 2005, Atlanta, Georgia (May 2005)
6. T. M. Mitchell.: Machine Learning. McGraw-Hill (1997)
7. T. Davis.: University of Florida sparse matrix collection. NA Digest, Vol .92,No. 42 (Oct. 16, 1994) and NA Digest, Vol. 96, No. 28 (Jul. 23, 1996) and NA Digest, Vol. 97, No. 23 (June 7, 1997) Available at: http://www.cise.ufl.edu/research/sparse/matrices/.
8. T.-Y. Chen.: ILUTP_Mem: A space-efficient incomplete LU preconditioner. In A.Laganà,M. L. Gavrilova, V. Kumar, Y. Mun, C. J. K. Tan, and O. Gervasi (eds.), Proceedings of the 2004 International Conference on Computational Science and its Applications. volume 3046 of LNCS. (2004) 31-39
9. Y. Saad and M. H. Schultz.: GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J. Sci. Stat. Comput. **7(3)** (July 1986) 856–869
10. I. S. Duff and J. Koster.: The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. SIAM J. Matrix Anal. Appl. **20(4)** (1999) 889–901
11. I. S. Duff and J. Koster.: On algorithms for permuting large entries to the diagonal of a sparse matrix. SIAM J. Matrix Anal. Appl. **22(4)**(2001) 973–996, .
12. T. Davis, J. Gilbert, S. Larimore, and E. Ng.: Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. ACM Trans. on Math. Softw. **30(3)** (September 2004) 377–380
13. T. Davis, J. Gilbert, S. Larimore, and E. Ng. A column approximate minimum degree ordering algorithm. ACM Trans. on Math. Softw. **30(3)** (September 2004) 353–376