Neural Network Based Prediction and Input Saliency Determination in a
Blast Furnace Environment

by

Ravindra K. Sarma

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 22, 2000

Copyright 2000 M.I.T. All rights reserved.
Neural Network Based Prediction and Input Saliency Determination in a Blast Furnace Environment

by

Ravindra K. Sarma

Submitted to the
Department of Electrical Engineering and Computer Science

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 22, 2000

ABSTRACT

This paper examines the ability of feed-forward neural networks to predict future hot metal temperature (HMT) of pig iron based on 17 input parameters that reflect the current and past conditions of the blast furnace. Various data challenges and solutions are presented, as well as a neural-network training suite that automatically finds the best-performing neural multi-layer perceptron (MLP) network. The paper then analyzes how prediction accuracy depends on the network architecture, data pre-processing, and input-output lag selection. The second part of the paper focuses on methods for determining the saliency of input variables. Given a trained neural network, the goal is to rank the input variables based on importance in affecting the output HMT. This exercise is useful as a means of eliminating redundant inputs as well as for choosing candidate control parameters. Four methods of saliency determination are compared. Two of them, sensitivity derivative analysis and weight-interpretation, which have been presented in previous literature, are applied to networks over different prediction horizons. In addition this paper presents two novel modifications to existing ranking algorithms. One of them, the weight derivative method, substitutes the weight magnitude by the derivative of the error with respect to the weight as a means of determining the weight's overall importance. After describing the theory involved, the paper compares these techniques, and discusses ways to improve their performance.

Thesis Supervisor: Dr. Amar Gupta
Title: Co-Director, Productivity from Information Technology (PROFIT) Initiative, MIT Sloan School of Management.
Acknowledgements

The author would like to sincerely thank the following people:

Prof. Amar Gupta, for providing guidance and direction on this project, and in general for being a supportive and helpful advisor.

Dr. Debashish Bhattacharjee, Mr. Ajit Kothari and Dr. S.K. Bag for all for their efforts to collect data for my research, and to provide answers to my questions regarding blast furnace operations.

All the members of the MIT/Sloan data mining research group, including Anjali Dhond, Michael Leonida, Ashish Mishra, Natasha Illiouchina, Benjamin Fang, Katherine Koch and Victor Chang for countless hours of processing data, running models, writing and debugging code that has made this thesis into a reality.

And finally, my deep personal gratitude and appreciation goes to my parents, Dr. Someswara Sarma and Dr. Lokamatha Sarma, and to my older brother, Shankar Sarma. They have provided me with motivation, inspiration and support in good times and bad. Without their help, this thesis simply would not have been possible.
Table of Contents

1. INTRODUCTION .................................................................................................................. 5

2. BACKGROUND .................................................................................................................. 9

   DOMAIN BACKGROUND ................................................................................................. 9
   MOTIVATION FOR USING NEURAL NETWORKS ............................................................ 11
   ANN PREDICTION AND CONTROL IN THE BLAST FURNACE ........................................ 14
   NEURAL NETWORKS AND INPUT SALIENCY DETERMINATION .................................... 19

3. DESCRIPTION OF PREDICTIVE MODEL ......................................................................... 24

   NETWORK ARCHITECTURE .............................................................................................. 24
   CAPTURING THE INPUT-OUTPUT LAGS ........................................................................... 25

4. NNRUN: AN AUTOMATED ANN TRAINING SUITE ......................................................... 31

   DATA MANIPULATION ..................................................................................................... 31
   AUTOMATED SEARCH FOR THE BEST NETWORK ......................................................... 32

5. ANN PREDICTION RESULTS USING MOVING WINDOW DATA .................................... 37

6. PITFALLS OF MOVING WINDOW AVERAGING ............................................................. 46

   THE PROBLEMS OF MOVING WINDOW AVERAGING ................................................... 46
   COMPARISON OF MOVING WINDOW WITH DIRECT HOURLY AVERAGING .................... 47
   COMPARISON OF NEW RESULTS WITH PREVIOUS WORK .......................................... 52

7. COMPARING ANNS TO STANDARD STATISTICAL ANALYSIS ........................................ 56

   THE LINEAR REGRESSION ............................................................................................... 56
   IMPLEMENTATION AND COMPARISON OF RESULTS .................................................... 57

8. METHODS FOR INPUT SALIENCY DETERMINATION ...................................................... 61

   MOTIVATION .................................................................................................................... 61
   INPUT-OUTPUT DERIVATIVE ANALYSIS ....................................................................... 61
       Theoretical Analysis .................................................................................................... 61
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation and Interpretation</td>
<td>65</td>
</tr>
<tr>
<td>Application of Derivatives</td>
<td>66</td>
</tr>
<tr>
<td>WEIGHT-INTERPRETATION ANALYSIS</td>
<td>70</td>
</tr>
<tr>
<td>Original Algorithm</td>
<td>70</td>
</tr>
<tr>
<td>Modification to Existing Algorithm</td>
<td>73</td>
</tr>
<tr>
<td>WEIGHT DERIVATIVE ANALYSIS</td>
<td>75</td>
</tr>
<tr>
<td>Weight Saliency</td>
<td>75</td>
</tr>
<tr>
<td>Revising the Measure of Saliency</td>
<td>78</td>
</tr>
<tr>
<td>New Weight Derivative Input Ranking Algorithm</td>
<td>80</td>
</tr>
<tr>
<td>Drawbacks to the Weight Derivative Method</td>
<td>82</td>
</tr>
<tr>
<td>9. SALIENCY RESULTS AND FUTURE WORK</td>
<td>83</td>
</tr>
<tr>
<td>RANKING COMPARISON AND INITIAL PRUNING</td>
<td>83</td>
</tr>
<tr>
<td>NARROWING DOWN THE VARIABLE SET FURTHER: CONTROL PARAMETERS</td>
<td>92</td>
</tr>
<tr>
<td>FURTHER WORK</td>
<td>98</td>
</tr>
<tr>
<td>10. COMPARISON WITH OTHER DATA MINING WORK</td>
<td>102</td>
</tr>
<tr>
<td>11. CONCLUSIONS</td>
<td>112</td>
</tr>
<tr>
<td>12. REFERENCES</td>
<td>117</td>
</tr>
<tr>
<td>13. APPENDIX</td>
<td>122</td>
</tr>
<tr>
<td>APPENDIX 1: DERIVATION OF THE HESSIAN MATRIX</td>
<td>122</td>
</tr>
<tr>
<td>APPENDIX 2: MATLAB SOURCE CODE FOR INPUT RANKING ALGORITHMS</td>
<td>128</td>
</tr>
</tbody>
</table>
Section 1: Introduction

With advances in database technologies, companies have started storing large amounts of historical data. Over the past decade, there has been an increasing interest in using artificial neural networks (ANNs) to mine these large databases to extract useful patterns and relationships. ANN based data mining has been applied in a multitude of domains, including finance [1], inventory reduction [2] [3] [4], electronic commerce [5] and medicine [6].

ANNs have also been used in industrial applications for tasks such as modeling and predicting complex industrial processes. For the most part, ANNs have been used for their ability to capture arbitrary non-linear and highly complex relationships between the inputs and outputs of processes. In addition, neural networks are becoming increasingly attractive in data mining of large corporate databases because they require no prior domain knowledge and because they are computationally efficient [7].

The focus of this paper is to analyze and interpret the use of ANNs in the context of a steel mill's blast furnace operations. The blast furnace is where inputs such as coke, iron ore and sinter are combined in a complex chemical process, to yield liquid pig iron, the pre-cursor of steel. The hot metal temperature (HMT) is an important indicator of the internal state of a blast furnace as well as of the quality of the pig iron being produced [8]. Hence, steel-makers and blast furnace operators would like to predict the HMT based on current and past conditions of the furnace as well as on the levels of the various input materials used. This would provide them with information
regarding the quality of the steel produced, and would also allow them to take corrective action if future HMT predictions indicate sub-optimal operating conditions.

Despite significant effort over a period of many years, researchers have not been able to find a precise function mapping these input variables to HMT. The production of pig iron involves complicated heat and mass transfers and introduces complex relationships between the various raw materials and chemicals used. Most of the time, these relationships are non-linear and therefore cannot be accurately estimated by standard statistical techniques or even mass-transfer equations [9]. As such, a number of researchers have turned to ANNs for modeling these complex inter-variable relationships. In particular, attempts have been made by researchers to use ANNs to predict the silicon content of pig iron produced from a blast furnace [8] [10] [13] [48]. Due to the success of this previous work, and the fact that silicon content is directly related to HMT [48], it seems natural to use neural networks in order to predict HMT.

The purpose of this paper is to examine the ability of feed-forward neural networks to predict the hot metal temperature of pig iron based on 17 input parameters that reflect the current and past conditions of the blast furnace. The prediction horizons for the networks presented here range from one to eight hours. The data are taken from a large industrial blast furnace in Asia. Throughout the paper, this steel company is referred to as “Steelcorp” where appropriate. Along the way, various data challenges and solutions are presented, as well as a neural network-training suite that automatically finds the best-performing neural multi-layer perceptron (MLP) network.

This paper also attempts to explain how the quality of prediction depends on the number of hidden nodes and layers, as well as on whether one uses automatic lag
detection or manually sets input variable lags based on domain knowledge. The relationship between prediction accuracy and prediction horizon is also discussed. In addition, the merit of moving window averaging, a technique for averaging data sets without significantly reducing their size, is discussed and its results are compared with those of direct hourly averaging.

The second part of this paper focuses on methods for determining the saliency of input variables. Given an optimally trained neural network, the goal is to rank its input variables based on importance in effecting the output HMT. This exercise is useful, firstly, as a means for eliminating redundant or noisy inputs that reduce prediction accuracy. Secondly, in terms of control, one would like to narrow down the choice of control parameters significantly. Since techniques such as the use of neuro-controllers and neural network inversion require a small set of inputs to be used as control parameters [11] [37] [40], it might be necessary to select the inputs that have the largest effect on the output. These highly important inputs could make up the desired set of control parameters.

In all, four methods of saliency determination are compared. Two of them, sensitivity derivative analysis and weight-interpretation, which have been presented in previous literature, are applied to networks over different prediction horizons. The two other techniques described in this paper present modifications to existing ranking algorithms. One of them, the weight derivative method, substitutes the weight magnitude by the derivative of the error with respect to the weight as a means of determining the weight's overall importance. These weight rankings are then translated into input saliencies. After describing the theory involved, the paper presents a
comparison of these techniques, as well as a discussion on ways to improve the current set of results.
Section 2: Background

2.1 Domain Background

The production of steel encompasses many different processes involving different plants. The first stage of steel production involves the creation of *pig iron*. This liquid iron is the raw material used in steel plants. Iron oxide pellets, or sintered ore, along with coke and limestone, are charged into layers in the *blast furnace* in order to produce pig iron. Inside the blast furnace, the oxygen from the iron oxides is removed to yield nearly pure liquid iron. Generally, blast furnaces can hold as much as several thousand cubic meters of raw material [12].

The whole process begins by sending coke, sintered ore and limestone through the top of the blast furnace, and waiting for the charge to gradually descend through the rest of the shaft. This process usually takes up to eight hours. Within the blast furnace, there are massive heat transfers and complex chemical phenomena governing the creation of the pig iron. These heat exchanges and reductions are reflected by the furnace conditions and the heat generated in the furnace. Often, irregular temperatures in the furnace and changing profiles in the furnace body can signal sub-optimal operation.

Once the raw materials are allowed to charge through the blast furnace, the resulting liquid iron is tapped from the furnace and sent via special *torpedo cars* to the steel-making shop. Here, *basic oxygen furnaces* (BOF) refine the composition of the iron steel by blowing oxygen into the hot metal, thereby oxidizing impurities [12].
The inherent issues of pig iron quality and means for its control have necessitated the use of expert systems within the blast furnace in order to improve iron production. The main thrust of quality improvement efforts within the blast furnace has been to regulate the silicon content variability in the hot metal produced. The heat level inside the furnace controls this. If the temperature within the furnace is too high, then the amount of silicon will increase. If, on the other hand, the temperature is too low, the level of silicon in the hot metal will be too low. Therefore, some researchers have used the blast furnace's hot metal temperature (HMT) as a means of understanding the state of the furnace and the amount of silicon in the resulting pig iron [8] [13].

One of the most common uses of expert systems in blast furnace operations is to help predict abnormal situations in the furnace, such as slips. Slips are characterized by an unusual descending in the raw materials charged through the furnace. Another major application of these expert systems is to build systems for prediction and control of the furnace’s internal temperature, as well as the pig iron’s hot metal temperature [8] [13] [21] [50] [51].

The second application is highlighted in this paper. Traditionally, it has been the role of the human operator to continually monitor and adjust certain variables in order to maintain the hot metal temperature at desired levels. Some of the variables used to perform these control operations include ore-by-coke ratio, fuel injection levels and blast temperatures [12]. The importance in being able to predict and control HMT is that drops in the all-important hot metal silicon levels below the desired level are synonymous with a drop in the HMT below its desired level. In such a scenario, the
operator will have to take actions to increase the HMT so that silicon content remains consistent.

Thus, if the blast furnace operator knows that HMT will be too low four hours from now, he or she may be able to alter the amount of coke or sinter in order to raise the future HMT value to a more optimal level. In addition, the further into the future one can predict HMT, the more time the operator will have to use cheaper strategies to correct the HMT level. Hence, accurate future prediction of HMT given current and past furnace conditions, as well as a knowledge of which input parameters are the most influential in determining HMT, are extremely important in attaining a more optimal production process at the blast furnace.

One complication to this system is that different raw materials into the blast furnace have different reaction times [8] [13] [14]. That is, if one alters the ore-by-coke ratio right now, its effect on hot metal temperature may not be seen until six hours from now. Therefore, trying to find solutions to problems with shorter control horizons is tougher than trying to control HMT further into the future.

2.2 Motivation for Using Neural Networks

The production of steel is a very complex process and the reactions that take place within the blast furnace are highly nonlinear. For decades, chemical engineers have tried to represent these processes by building deterministic models and equations based on the chemistry involved [14] [15] [48] [49]. These equations involve complex thermodynamics and mass-transfer relationships. Unfortunately, when used for prediction or control, these models have provided little success. In particular, Biswas
states that the accurate prediction and control of variables such as HMT and silicon content are extremely difficult to carry out based solely on domain level models [9]. One of the main problems is that the validity of the chemical equations presented in the literature is only as good as the accuracy with which one is able to measure the individual variables in the equation. Frequently, due to the extreme temperature conditions within the blast furnace, it is almost impossible to obtain such accurate input measurements.

Torssell et al. [49] attempted to use mathematical models to model the blast furnace. Specifically, they developed equations based on simulation of the mass-transfer equations believed to exist within the blast furnace. The resulting model was designed to approximate the furnace using variables such as blast temperature, oxygen enrichment, coal injection and coke moisture, a subset of the input variables used in this paper. Although decent results were obtained, these results still indicated that blast furnace parameters could not be predicted using simulations and chemistry alone. Singh et al [48] confirmed this result by examining conventional models for predicting the silicon content of pig iron, a variable closely related to hot metal temperature. According to Singh, these conventional models are not enough to predict silicon content with the required degree of accuracy.

As Ullman [16] explains, there exist other input uncertainties in the modeling process. These include the effects of interactions between different input variables, the effects of random fluctuations within the operating region of the blast furnace, as well as variations in the composition of raw materials. Therefore, much of the previous domain specific work [9] [14] [16] [48] [49] has concluded that reliable prediction and
control of different parameters in the blast furnace operations could not really be achieved by using deterministic chemical equations alone. This obviates the need for a different, data-driven modeling approach.

There are a few reasons why many have regarded neural network data mining as the technique of choice when modeling the chemical processes of steel-making plants [10]. First of all, neural networks can model highly non-linear relationships between variables. While traditional regression techniques are often limited to linear input-output mappings, neural network models have been shown to be capable of capturing more complex relationships [4] [7] [17]. It has also been proven in theory and shown in practice that neural networks with one or more hidden layers are capable of mapping any arbitrary mapping between many inputs and an output, given the appropriate number of hidden nodes [17] [18].

Secondly, neural networks help quicken model development. Rather than having to focus solely on domain knowledge and mathematical relationships derived from chemistry, neural networks learn by looking at examples and can be trained in a relatively short amount of time. Development of analytical models can take large amounts of time and intricate domain knowledge. On the other hand, neural networks do not assume anything special about the domain and provide an automated way to capture inherently non-linear variable relationships by automatically training ANNs on the computer. Thirdly, neural networks can be adjusted quickly in order to adapt to changes in operating conditions. In the case of steel plants, there often occurs daily fluctuations or drifts in processing. Researchers have devised ways to adapt the neural network to these long-term fluctuations in an efficient way [19] [51]. Such models that
can be adapted on-line are much more valuable than static regression or analytical models.

Finally, neural networks have been used successfully to model relationships in a wide range of application domains. At MIT, the data mining research group led by Dr. Amar Gupta has applied MLP models to forecast demand in order to optimize inventory levels [2] [4]. The use of these techniques helped reduce the inventory that a major drug company had to maintain from $1 billion to $500 million, while still ensuring that there was a 95% chance that a random customer looking for a random drug would find it in stock. The group has also applied ANN technology to recognition of handwritten numerals, again with a high rate of success [20].

2.3 ANN Prediction and Control in the Blast Furnace

In terms of iron making, attempts to predict HMT using standard statistical time series analysis have yielded poor results [12] [19] [50] [51]. Inherent complexities and non-linearities exist between the different parameters used in pig iron (hot metal) production. Given the highly desirable qualities of neural networks in regards to optimization of steel production, researchers have successfully applied these models to problems dealing with the blast furnace. Debashish Bhattacharjee et al [21] used feed-forward neural networks to predict several quality parameters, including HMT, of iron produced at an operational blast furnace in India. An initial set of 24 inputs was reduced to 15 based on a method that measures the entropy of different input variables in categorizing the output HMT [30]. Results indicated that multi-layer perceptron networks, with one hidden layer and employing back-propagation learning, were
capable of predicting trends in daily HMT values. Correlation values between the actual and predicted values were relatively high, equaling 0.78 in some cases. Although promising, the networks in [21] predict daily, rather than hourly HMT values. Therefore, unlike this example, the networks of the current paper must correctly take into account the lags between certain inputs and HMT, because these lags are on the order of one to eight hours. In addition, the prediction problem of the current paper may be more challenging than that of [21] because hourly data can fluctuate more frequently than daily data.

More recently, researchers at the same steel plant have applied neural network data mining to model the complex chemical processes of the air separation unit of an oxygen plant [24]. The steel plant’s Air Separation Unit (ASU) produces high purity oxygen, nitrogen and argon while also producing low purity waste nitrogen. The model developed predicts the gaseous oxygen (GOX) production rate for different reflux rates and streams. These researchers trained and validated networks with one hidden layer and three inputs. Initial results demonstrated a high capability of neural networks to predict GOX.

Neural networks have been applied to predict and control HMT and hot metal silicon content in blast furnaces all over the world. At the Koverhar blast furnace in Finland, Nikus et Al [50] used neural networks to predict the thermal conditions of the blast furnace. In particular, data measured in minute intervals were analyzed, and prediction horizons ranged between 1 to 20 minutes into the future. Single hidden-layer networks were chosen, and much time was spent identifying the optimal number of hidden nodes. A network with five hidden nodes and seven inputs performed the best.
In addition, lagged predicted values were fed into the network as additional inputs. The mean squared errors of the predictions ranged from 0.0036 to 0.0051 on the testing data. Although the granularity of the data set is different from the hourly data used in the current paper, the results of [50] provide reason for increased optimism that ANNs might be successful in the current work.

Researchers at the No. 2 blast furnace of the Lulea Works steel mill in Sweden have also used neural network techniques to model blast furnace processes [51]. The interesting difference here is the use of self-organizing networks in conjunction with MLP back-propagation networks in order to yield hot metal silicon content predictions two hours into the future. MLP networks were first used as a means of predicting silicon content based on recent training data. However, it is often the case that the operating conditions within the blast furnace change, thus requiring a new static network to predict silicon content for the next few hours. Therefore, self-organizing networks were used to define operating states of the furnace in such a way that separate prediction MLP networks could be trained for each state. This combination of self-organizing networks and MLPs formed a hybrid dynamic network that changed based upon the operating state of the blast furnace. Since silicon content is related to hot metal temperature, the success here in predicting silicon content using ANNs provides another indication that neural networks may also be useful in the prediction of hot metal temperature.

Another example of neural network applications at a steel plant is part of the work of the Industrial and Building Systems Group at Siemens. In their electric arc furnace, neural networks serve as the basic points of control [19]. Neural networks
have also been used in hot rolling mills to predict parameters such as rolling force and rolling-stock temperature.

Bloch et al used neural networks to control specific processes in a steel plant, such as the strip temperature of the plant’s induction furnace [25]. Here, a multi-layered neural network was used to model the inverse of the induction furnace process. That is, given the current strip temperature and inputs, how should one change the input levels so as to arrive at a different, desired temperature level? This is in fact the inverse of the temperature prediction problem. Although initial results look promising, more work in this area needs to be performed.

Researchers at British Steel [53] have used neural networks for the purposes of control at five separate blast furnaces in England. In particular, neural network controllers have been used as a means of controlling the burden distribution of the different blast furnaces. In addition, neural network models have been developed to forecast the quality of hot metal produced. Specifically, on-line models have been constructed to predict hot metal temperature and silicon content. The results of these prediction models have been integrated into the control process at the furnace in order to provide an automated prediction and control of hot metal quality. A separate statistical analysis of time-dependence of the different inputs was first performed before creating the new models. This case example illustrates the successful online integration of expert systems to the workings of an iron blast furnace.

Bulsary et al [10] observed promising results when using multi-layered feed-forward neural networks to predict the silicon content of hot metal from a blast furnace. Feed-forward networks with 1 and 2 hidden layers were trained and the method used to
update the neural network's weights was a non-linear variant of the traditional back-propagation learning algorithm. This method, the Levenberg-Marquardt training method, differs from standard back-propagation in that the direction of the change in a weight's value, during a training epoch, is actually an interpolation between the directions given by the steepest descent method and Gauss-Newton method. Due to the increased speed of this training algorithm [18], the networks in the current paper are also trained using the Levenberg-Marquardt improvement.

The results of Bulsary's paper show that, for this application, the feed-forward ANNs produced considerably better results than standard linear time-series prediction. The paper also states that networks with two hidden layers did not perform much better than networks with one hidden layer. In addition, two hidden-layer networks took much longer to train and found it much harder to arrive at an error minimum in the weight space.

Bulsary and Saxen used neural networks again in order to classify probe temperatures within a blast furnace [22]. Often, blast furnace operators require practical information regarding the radial distribution of gas in the furnace shaft. This information is used to make judgments regarding the distribution and flow conditions of the furnace's burden and the state of its cohesive zone [22]. Traditionally, operators make these judgments based on the measurement of gas temperatures from horizontally placed probes with the blast furnace, and on their own previous experience. Bulsary and Saxen built an expert system that was able to classify the gas distributions within the furnace based on these probe measurements. Feed-forward neural networks with sigmoid transfer functions in the hidden layer(s) and output layer were used to provide
these classifications. The best performing network contained only one hidden layer. Bulsary and Saxen also discovered that networks with too few hidden nodes were simply unable to capture the required relationships accurately, while larger networks were more likely to be guilty of over-fitting. This system was then implemented at the blast furnace and a user interface was designed to provide the operators with the classification information in a meaningful way.

Zuo et al. used feed-forward neural networks in order to predict furnace irregularities based on operational data [23]. Separate networks were trained to classify various top gas distribution profiles, to recognize different heat flux distributions, as well as to predict slips or abnormalities in the blast furnace. The paper presented a neural network-based supervision system that performed the classifications and predictions mentioned above, triggering an alarm to the operator if a dangerous situation within the furnace was about to occur. Feed-forward networks were used for slip prediction while self-organized feature maps (SOFM) and learning vector quantizers (LVQ) were used for classification. The results once again showed that neural networks were capable of providing prediction within the blast furnace environment.

2.4 Neural Networks and Input Saliency Determination

The question of input saliency is important for many reasons. One would like to eliminate redundant or irrelevant input variables from a prediction problem. Inclusion of irrelevant inputs in the model contributes noise to the system and, hence, may skew the actual relationships between the inputs and the outputs during training [17] [18]. In
addition, high input dimensionality means increased complexity in the network, measured in terms of more weights and biases. Increased complexity often comes at the price of reduced generalization [18]. Therefore, one would like to narrow down the input set to possibly increase prediction accuracy and improve generalization. This requires techniques for determining the relative importance of different input variables.

Another reason for performing input saliency analysis is that it is desirable to narrow down the original set of input variables to a group that would be most effective in controlling the output variable (in the present case, HMT). Several researchers [11] [37] [40] have pointed out that in order for models such as neuro-controlers or neural network inversion to work effectively, the number of control parameters used must be narrowed down substantially. The inversion problem is nothing but the inverse of the neural network prediction problem: given that one wants a desired output, what combination of input changes will obtain the desired output level [26]?

Since this is basically a one-to-many mapping problem, it is crucial that one reduces the number of input or input changes that need to be calculated by the neural network inverter [11]. In such a scenario, the determination of the most effective input parameters is crucial.

Since this is a very common problem, there have been many attempts to find desirable solutions. In the traditional statistical domain, the importance of regression coefficients can be determined by their relative size and their t-statistics [27]. In addition, a technique often used is Discriminant Analysis [28]. In the k-dimensional form of this problem, one is presented with a set of input data and k categories to which different input combinations belong. One of the purposes of discriminant analysis would be to identify those variables that are most important in determining the category
of a data point. Given that the output variable in the current problem, HMT, is continuous, discriminant analysis does not seem very useful in this case.

Another common technique for input dimension reduction is Principal Components Analysis (PCA). PCA is used as a pre-processing technique of the input space that has been applied to both traditional statistical problems and neural network analysis. PCA consists of finding the eigenvectors of the data matrix and reducing the input space to a set of inputs where each new input is a linear combination of the original inputs. These new, transformed inputs, are orthogonal to each other and are provided with a relative measure of "importance." [29]. However, the problem here is that the new inputs are combinations of the original ones. This is not desirable in a control setting. For example, if a controller suggests that transformed input $x$ should be reduced by half, it would be hard to translate this change into changes of the individual original inputs.

Often, a much simple approach called dithering is used to determine input saliency [21] [52]. This method relies on the measurement of input sensitivity based on a trained neural network. In dithering, one varies a particular input variable from its minimum operating value to its maximum, while keep all other inputs constant, and then examines the corresponding change in the output variable. The larger the change in the output variable given the change in the particular input variable, the more important that input variable must be. There are, however, serious limitations to this approach. Klimasauskas [52] states that this method ignores the non-linearities that may be present in the data and the multi-input dependence of the output.
Many researchers have tried to create methods suitable for input reduction of neural networks. Most of these techniques involve examination of a network trained on the original larger set of input variables. Sung [31] provides a comparison of three different techniques often used for input variable ranking and selection. The first of these is Fuzzy Curves (FC), which were initially derived by Lin and Cunningham [32]. FCs attempt to use fuzzy logic to establish a relationship between the input variables and the output. Fuzzy curves are then constructed for each input variable and their shape is analyzed to determine the variable’s importance.

Sung also describes another, simpler method for determination of input saliency. This is called the Change of MSE (COM) method. The original network, with all inputs included, is trained and the mean squared error (MSE) is observed. Next, an input is removed from the model, the network is retrained, and the change in the MSE is noted. Consequently, the first variable is re-included, while a different variable is dropped. The new network is retrained and the change in MSE from the original, full network, is again calculated. This process is performed for all of the input variables, which are ranked by their change in MSE. The variables that cause larger increases in MSE after removal are considered to be more important. One drawback of this method is that it requires constant retraining of the networks and is therefore time-inefficient.

The third method discussed is Sensitivity Analysis (SA), which is also analyzed by Zurada et al. [33] and Engelbrecht et al. [34]. This method, which will be derived in detail in Section 8, calculates the partial derivative of the ANN’s output with respect to each input, and averages the effect of each input over all data patterns. The variables are then pruned based on the largest gap method, which is also described later. The result
of Sung’s experiment was that Fuzzy Curves performed better than SA and COM in most of the experiments conducted.

A final method, which will also be discussed in detail later in this paper, is the method of neural network weight interpretation introduced by Garson [35], and described in further detail by Ravinder Nath [36]. This method tries to determine the saliency of different inputs based on the weights that connect each input to nodes in the hidden layer, as well as on the weights that connect the hidden nodes to the output.
Section 3: Description of Predictive Model

3.1 Network Architecture

The standard multi-layer perceptron (MLP) model became the architecture of choice due to its overwhelming success in previous work. The feed-forward network has provided accurate prediction results both in the blast furnace domain and in general chemical processes [8] [19] [22] [23] [37]. Time-delay neural networks (TDNNs) were also examined, primarily because there exist time lags between the instant when an input is changed and when the effect of the change is manifested in the output HMT value. TDNNs are capable of capturing these lags during training, and storing this knowledge implicitly in the connections of the network. On the other hand, if one is to use an MLP model with back-propagation learning, the time lag between a particular input and HMT must be decided ahead of time and the variables in the training examples must be adjusted accordingly before learning begins. This corresponds to explicit memory, which is presented to the network by the inputs.

Prior research by the group in comparing the performance of TDNN and MLP networks in medical inventory optimization indicated that the TDNN performed only marginally better than the MLP [2]. In addition, time delay networks take a much longer time to learn. As will be described later, NNRUN, the automated tool used to find the best neural network, tries a multitude of hidden layer configurations before finding the best one; the use of TDNNs in this case would force NNRUN to take a much longer time to converge to its final solution.
Thus, the model that was constructed consisted of 17 input variables, each lagged in time from the output HMT by an appropriate amount (this appropriate amount is described in the next section). Fourteen of the input variables used are chemical inputs into the blast furnace as well as byproducts of the steel-making process. These included coal injection, heat flux, ore/coke ratio, oxygen enrichment, steam flow, wind flow, carbon dioxide, coke, nutcoke and hydrogen. Three additional binary variables (cast, cast1op2 and cast2op2) were used that indicate the specific sensor from which the HMT is measured. In addition to these 17 inputs, an extra input, the last measured HMT value, was also used as an input. According to past work and neural network theory [17] [18], it has been shown that a neural network with one hidden layer is capable of modeling any arbitrary function between a set of inputs and outputs. As the mapping function needs to become more and more complex, the number of hidden nodes in the layer must increase. That is, in theory, any n-hidden layer network can be transformed into a one-hidden layer network, as long as the number of nodes in the single hidden layer is large enough [18]. However, some researchers have noted that, in practice, two-hidden layer networks have been able to capture certain relationships with greater accuracy [8] [13] [19] [39]. Therefore, this paper examines both one-hidden layer and two-hidden layer networks and compares their results.

3.2 Capturing Input-Output Lags in MLP network

The next important step was to identify the appropriate input-output time lags in order to provide the network with the correct inter-variable relationships. One way to identify the correct lag for each input is to use a data-driven approach. That is, for each
input variable, one calculates the correlation between that input and the output HMT at every possible time lag. Finally, one implements the time lag that produces the highest input-output correlation for that variable. This method makes use of the Pearson coefficient:

\[
\text{Corr}(X, Y) = \frac{n \sum_{i=1}^{n} X_i Y_i - (\sum_{i=1}^{n} X_i)(\sum_{i=1}^{n} Y_i)}{\sqrt{n \sum_{i=1}^{n} X_i^2 - (\sum_{i=1}^{n} X_i)^2} \sqrt{n \sum_{i=1}^{n} Y_i^2 - (\sum_{i=1}^{n} Y_i)^2}},
\]

where \(i \in [1, n]\)

Thus, using the above measure for correlation, the following algorithm was developed to discover the optimal lag between each input variable and HMT:

**Obtaining the Optimal Input-Output Lag for \(X_i\):**

For each input variable \(X_i\):

For \(j\) varying from 0 to \(\text{max}_{-}\)\(_{\text{lag}}\) (the maximum lag to consider):

1. Implement lag of \(j\) hours between \(X_i\) and HMT
2. \(c_{orr_j} = \text{Corr}(X_i(t-j), \text{HMT}(t))\)

The optimal time lag between \(X_i\) and HMT is \(j^* \in [0, \text{max}._{\text{lag}}]\) \(\Rightarrow c_{orr_j} = \text{Max}[c_{orr_j}]\)

Repeat the above algorithm for all \(X_i, i \in [1, N]\) where \(N\) is the number of input variables.

Figure 1. Automatic lag calculation.

After one obtains the optimal time lag for each input variable, these lags are implemented in the training and testing data. After consulting domain experts, it was decided that the maximum input-output lag to consider, when automatically detecting lags, was 8 hours.
Rather than basing input-output lags on correlation coefficients, another available strategy is to set the input-output lags to specific values based on domain knowledge of the blast furnace process. The table below compares the time lags determined automatically by the system and those obtained by experienced furnace operators.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Lag Estimated (in hours) by</th>
<th>Automated Lag (in hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Domain Expert</td>
<td></td>
</tr>
<tr>
<td>Cast</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cast1op2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Cast2op2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Charge time</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Coal injection</td>
<td>3-4</td>
<td>3</td>
</tr>
<tr>
<td>Coke</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Group 2 Heat Flux</td>
<td>3-4</td>
<td>8</td>
</tr>
<tr>
<td>Group 1 Heat Flux</td>
<td>2-3</td>
<td>8</td>
</tr>
<tr>
<td>Hot Blast Temperature</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Nutcoke</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Ore/Coke</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Oxygen flow</td>
<td>1-4</td>
<td>8</td>
</tr>
<tr>
<td>Steam flow</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Top gas CO</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Top gas CO2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Top gas H2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Wind flow</td>
<td>1-3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Comparison of Time Lags
It is interesting that the lag(s) suggested by the domain expert match that determined by the system for only 6 of the 17 variables above. Out of the 11 variables for which lags did not match, some of the lags were close while others were very far apart. In the case of Ore/Coke, for example, the automated lag is 2 hours, while domain experts believe the lag to be closer to 6 hours.

Once this analysis was completed, the next step was to train networks that implemented the automated lags, train networks that implemented domain expert lags, and finally perform a model comparison. This is described in the Section 5. For some variables, experts could only provide a possible range of lag values. Therefore, the networks trained using the domain expert lags contained some variables multiple times in the input layer, with each instance of the same input lagged by a different value.

One complication arose when the prediction horizon was greater than one or more of the variable lags. For example, assume the goal is to predict HMT (t + 4). In addition, assume that there is a two-hour lag between steam and HMT. Therefore, ideally, one should use steam (t+2) to predict HMT (t+4). However, when predicting at the current time t, the value of steam two hours from now is not available. Therefore, since one is not able to model the true input-output relationship, the best alternative is to use steam (t).

This complication therefore affected the way in which multiple lags were implemented for a single variable in the network. It may be the case that for some prediction horizons, the full range of lags for a particular input variable cannot be expressed. For example, wind flow has a lag with HMT in the range of 1 to 3 hours. If the goal is to predict HMT (t+1), it would be desirable to use wind flow (t), wind flow
(t-1) and wind flow (t -2) as inputs. However, when predicting HMT (t+8), only one instance of wind flow, wind flow (t) can be used in the model. If, for example, one used wind flow (t-1), the effective lag between wind flow and HMT would equal 8 + 1 = 9 hours. As such, the best option is to use wind flow (t), for an effective lag equaling 0 +8 = 8 hours. Given this consideration, the input sets for the one-hour networks and eight-hour networks were different in size. Tables 2 and 3 below display the input sets for one-hour and eight-hour forced lag networks:

Table 2. The input set for 1-hour prediction.

<table>
<thead>
<tr>
<th>Cast(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Castlop2(t)</td>
</tr>
<tr>
<td>Cast2op2(t)</td>
</tr>
<tr>
<td>Charge time(t)</td>
</tr>
<tr>
<td>Coal injection(t-2)</td>
</tr>
<tr>
<td>Coal injection(t-3)</td>
</tr>
<tr>
<td>Coke total(t-5)</td>
</tr>
<tr>
<td>G2heatflux(t-2)</td>
</tr>
<tr>
<td>G2heatflux(t-3)</td>
</tr>
<tr>
<td>Grp1HeatFlux(t-1)</td>
</tr>
<tr>
<td>Grp1HeatFlux(t-2)</td>
</tr>
<tr>
<td>Hot Blast Temp(t)</td>
</tr>
<tr>
<td>Nutcoketotal(t-5)</td>
</tr>
<tr>
<td>Orebycoket(t-5)</td>
</tr>
<tr>
<td>Oxygen flow(t)</td>
</tr>
<tr>
<td>Oxygen flow(t-1)</td>
</tr>
<tr>
<td>Oxygen flow(t-2)</td>
</tr>
<tr>
<td>Oxygen flow(t-3)</td>
</tr>
<tr>
<td>Steam(t)</td>
</tr>
<tr>
<td>Topgas CO(t)</td>
</tr>
<tr>
<td>TopgasCO2(t)</td>
</tr>
<tr>
<td>TopgasH2(t)</td>
</tr>
<tr>
<td>WindFlow(t)</td>
</tr>
<tr>
<td>WindFlow(t-1)</td>
</tr>
<tr>
<td>WindFlow(t-2)</td>
</tr>
<tr>
<td>HMT(t)</td>
</tr>
</tbody>
</table>

Table 3. The input set for 8-hour prediction.

<table>
<thead>
<tr>
<th>Cast(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Castlop2(t)</td>
</tr>
<tr>
<td>Cast2op2(t)</td>
</tr>
</tbody>
</table>
Table 3. The input set for 8-hour prediction.

<table>
<thead>
<tr>
<th>Chargetime(t)</th>
<th>Coalinjection(t)</th>
<th>Coketotal(t)</th>
<th>G2heatflux(t)</th>
<th>Grp1HeatFlux(t)</th>
<th>HotBlastTemp(t)</th>
<th>Nutcoketotal(t)</th>
<th>Orebycok(t)</th>
<th>Oxygenflow(t)</th>
<th>Steam(t)</th>
<th>TopgasCO(t)</th>
<th>TopgasCO2(t)</th>
<th>TopgasH2(t)</th>
<th>WindFlow(t)</th>
<th>HMT(t)</th>
</tr>
</thead>
</table>

30
Section 4: NNRUN – An Automated Neural Network Training Suite

NNRUN provides a menu-driven interface to perform customized data cleansing and manipulation, as well as a means to automate the search for the optimal neural network, given a set of training and testing data. The primary work behind NNRUN, in terms of initial design and development, was performed by Vladislav Y Gabovich and Bradley Banks [8].

4.1 Data Manipulation

Raw furnace data must be transformed into a more useful form before it can be presented as learning examples to the neural network. These techniques try to overcome the problems of missing data, scaling and lack of data. The initial data set consisted of approximately 30,100 data points, measured at 1-minute intervals. After consulting the domain experts at the blast furnace, it was suggested that hourly data should be used to perform training. This type of hourly averaging can suppress some of the measurement noise that is present in the 1-minute data. However, averaging every contiguous group of 60 data points would leave only $30100/60 \approx 501$ data points to be split between testing and training. Therefore, moving window averaging was adopted. Every contiguous set of 20 data points is first averaged to create a data set of 1505 data points at 20-minute intervals. The first three 20-minute interval data points are averaged into one hourly data point. Next, this window is shifted down by one 20-minute data point, and the next three data points are averaged. If the initial 20-minute data set has $N$ number of rows, using a window of size 3 results in an averaged data set of size $N - 3 + \ldots$
Therefore, each new data point contains 20 minutes of new information. This is why 20-minute, as opposed to 1-minute, data was used for moving window averaging: each data point carries 20 minutes, instead of 1 minute, of new information. At the same time, potential measurement errors at the 20-minute level are suppressed by averaging the “new” 20-minute point with two other 20-minute data points. The size of the new dataset was \( 1505 - (3 - 1) = 1503 \), a significant improvement over the 501 data points mentioned earlier. Each of these new data points was considered as hourly averaged.

Neural networks, along with many other statistical learning algorithms, are sensitive to the scale and variance of the input variables. For this reason, normalizing of variables is often performed [18]. This normalization renders values for each variable between 0 and 1. The normalization formula used here was:

\[
X_u = \frac{\text{Min} \{X_i\} / (\text{Max} \{X_i\} - \text{Min} \{X_i\})}
\]

where \( X_u \) is \( j \)th value of the \( i \)th input, the value that is to be normalized. \( \text{Max} \{X_i\} \) and \( \text{Min} \{X_i\} \) are the maximum and minimum values of the \( i \)th input variable respectively. This gave all of the variables the same range of values, along with relatively similar means and variances. These methods allow the neural network to capture more easily the relationship between a change in inputs and the change in the output, avoiding distortion caused by wide differences in scale and variation.

### 4.2 Automated Search For Best Network

In order to find the best performing neural network for a given set of data, it is necessary to find the optimal MLP hidden-node configuration, defined to be the
network whose predictions on the validation sample exhibit the lowest mean square error (MSE). NNRUN provides the user a choice, through menus, to select the number of hidden layers that the network should have, as well as a minimum and maximum number of nodes for each hidden-layer that should be considered. Therefore, if one wants to find the best two-hidden layer network, and consider number of hidden nodes in the range \([1,10]\) for the first layer and nodes in the range \([1, 5]\) in the second layer, the total number of network configurations trained will be \(10 \cdot 5 = 50\).

Typically, the initial data set is divided into a training set, which comprises 80% of the data, and the testing or validation set, which makes up the other 20%. Assume for the moment that the number of data points in the testing set is relatively small. If the performance of various neural nets is compared using only one particular partition of the data, the best performing network will necessarily be the one that is best fit to describe the particular data subset, and not necessarily one that best describes the overall mapping. To alleviate this problem, NNRUN allows the user to set the number of random runs. This parameter allows the user to train the neural networks on a large number (up to the number of data points) of different testing/training partitions of the original data set, resulting in the selection of the ANN hidden layer node configuration with the best mean performance over different partitions. This ensures that the "best" configuration reflects the general properties of the mapping and not a particular choice of data partition. If the number of runs chosen is \(m\), then NNRUN chooses the best node configuration such that:

\[
(MSE_1 + MSE_2 + \ldots + MSE_m) / m \text{ is minimized,}
\]

where \(MSE_n\) is the mean square error over the \(n^{th}\) randomized testing data set.
In addition, the user can also specify whether the different partitions are selected randomly or sequentially from the main data set. Once the best network configuration is identified, it is retrained on the default training set in order to produce the "best mean" network. Based on the results of running NNRUN with different numbers of runs, it was confirmed that the ANN exhibiting the lowest error on the default data

For a specific hidden node configuration:

**State:**
- \( i \) = training epoch
- \( j \) = current number of consecutive over-fitted epochs
- \( k \) = number of consecutive over-fitted epochs after which the network is considered over-fitted

**Flowchart:**

- **Start**
  
  - \( i = 0, j = 0 \)
  
  \( i = i + 1 \)

  - *Carry out backpropagation for ANN over all examples for epoch \( i \)*

  - **Is \( i = 0 \)? OR Is MSE testing for epoch \( i \) < MSE testing for epoch \( i-1 \)?**

  - **YES**
    - **Set \( j = 0 \)**
    - **Save ANN version \( i \)**
    - **If \( i \geq k+1 \)**
    - **Delete ANN version \( i - (k + 1) \)**
    - **YES**
    - **Done**
      - Best ANN for this node configuration is ANN version \( i - (k+1) \)
    - **NO**
      - **Is \( j = k \)?**
  
  - **NO**
    - **\( j = j + 1 \)**

**Figure 2:** NNRUN Algorithm
partition rarely provides the best mean-performance ANN. Further, it was observed that, for the best mean network, the mean square error (MSE) indicators of performance are significantly higher for the default partition than the MSE for randomized train/test selections. It is possible that this discrepancy is the result of fundamental changes in the properties of the mapping in the time period covering the train/test partition.

The number of random runs typically used was 3. The reason for choosing this value was to set an upper bound on search time. If one takes the above example of restricting the first and second hidden-layer node configurations to [1, 10] and [1, 5] respectively, and one sets the number of random runs to 3, the total number of training/testing sessions that the system would have to endure is $10 \cdot 5 \cdot 3 + 1 = 151$.

Finally, NNRUN uses an algorithm (see Figure 2) in order to avoid over-fitting of the ANNs. It is derived from the notion of early stopping in training [18]. Over-fitting occurs when the network is memorizing rather than generalizing the training examples. A simple heuristic used to identify this situation involves comparing the trends in the mean squared errors on the training and testing data over the range of epochs. If the MSE consistently increases on the testing set while the MSE decreases on the training set over a successive set of training epochs, this means that the network is over-fitting the training data.

The variable $k$ from Figure 2 is the number of consecutive over-fitted epochs past which the system considers the network to be over-fitting during training. The system saves the latest $k + 1$ versions of the trained neural network. If NNRUN detects that during the last $k$ epochs the $MSE_{testing}$ is increasing while the $MSE_{training}$ is decreasing, the system decides that the last $k$ versions of the trained network are over-
fitted. Thus, the optimal network given the current node configuration must be the oldest version still in memory.

After experimenting with the size of $k$, the group discovered that, in all cases, setting $k$ equal to 5 epochs was appropriate. Once the number of consecutive over-fitting epochs reached 5, the trend would continue without bound. When this technique of training termination was compared with other techniques such as limiting the number of training epochs to some predefined value, it was clear that networks trained using this new heuristic performed 10-15% better than networks trained using other stopping techniques.
Section 5: ANN Prediction Results Using Moving Window Data

Although neural network theory has mathematically proven that a one hidden layer network (3-layer net) can approximate any input-output mapping \[17\] \[18\], some researchers have suggested that feed-forward networks with two hidden layers (four-layer nets) may offer increased prediction accuracy in practice \[8\] \[18\] \[38\]. Therefore, this paper examines the ability of both one and two hidden layer networks and compares their performance.

Three-layer as well as four-layer networks were trained in order to predict HMT one, two, three, four, five, six, seven and eight hours into the future. \mbox{NNRUN}, described in Section 4, was used to find the optimal networks for each prediction horizon. For three-layer networks, the size of the hidden layer was varied from 1 to 40 nodes. For four-layer networks, each of the two hidden layers ranged from 1 to 12 nodes. These boundaries were established in order to avoid excessive computation time. Allowing for two random runs and running on Sun Ultra 5 workstations, three-layer simulations would take 15 hours to complete and four-layer simulations would take 30 hours to finish.

There are three main measures that were used to evaluate network performance and have been cited in many papers \[2\] \[4\] \[17\] \[50\]. These include the Pearson correlation coefficient (3.1), the average error (AE) and the Normalized Mean Square Error (NMSE). The Pearson coefficient is useful because it measures how well the model predicts trends in the output variable. A coefficient of 1 indicates that the model is predicting trends with the highest possible accuracy, while a value of -1 means the
model is not predicting trends at all. The average error is just the average difference (in degrees C) between the actual HMT and that predicted by the network:

\[
AE = \frac{\sum_{t=1}^{n} |HMT_{\text{actual}}(t) - HMT_{\text{predicted}}(t)|}{n}
\]

(5.1)

where \(n\) is the number of data points in the testing set.

NMSE compares the accuracy of the model's predictions versus predicting the mean, and is defined as [39]:

\[
NMSE = \frac{\sum_{t=1}^{n} (HMT_{\text{actual}}(t) - HMT_{\text{predicted}}(t))^2}{\sum_{t=1}^{n} (HMT_{\text{actual}}(t) - \overline{HMT})^2}
\]

(5.2)

An NMSE value of less than 1 indicates that the model performs better than the mean, while a value greater than 1 suggests that the model performs worse than the mean. The performance of each trained model was evaluated on the last 20% of the entire dataset, which is the default validation data set.

After training both three-layer and four-layer networks on the same data, it was determined that the networks with two hidden layers did not provide improved accuracy over the single-hidden layer models. Figures 3 and 4 below compare the relative effectiveness of the three and four-layer models for various prediction horizons:
Figures 3 and 4 indicate that although two-hidden layer and one-hidden layer networks perform at relatively the same level of accuracy for shorter prediction horizons, the one-hidden layer networks do show greater accuracy once the prediction horizon is larger than 4 hours. This could mean that for predictions closer in the future,
a relatively less complex four-layer model is required to predict HMT accurately. However, once the prediction horizons increase past 4 hours, the complexity of the two-hidden layer network may lie outside the search space of possible node configurations set when training using NNRUN. In general, for one and two-hidden layer networks, the number of hidden nodes tends to increase as the prediction horizon increases. The reason is that as the model tries to predict further into the future, it cannot rely as heavily on the last known HMT value (which is used as an input) when predicting future HMT.

In general, the Pearson coefficient values for the best overall performing networks range from 0.856 for one-hour prediction to 0.722 for eight-hour prediction. NMSE ranges from 0.228 for the one-hour network to 0.516 for the eight-hour network. Average error ranges from 3.92 to 7.95 degrees Celsius. Thus, although the prediction accuracy deteriorates with increasing prediction horizon, it is clear that overall prediction is very favorable. The fact that the Pearson coefficient does not fall below 0.722 indicates that each network has significant ability to predict the trends of HMT. Below are three figures that illustrate the performance of networks used to predict HMT two hours, four hours and eight hours into the future:
Figure 5. 2-hour HMT results. Configuration: 1 hidden layer with 26 nodes.

NMSE: 0.3001 Correlation: 0.8332

Figure 6. 4-hour HMT results. Configuration: 1 hidden layer with 36 nodes.
While the two-hour and four-hour predictions are much smoother and more accurate, the eight-hour predictions appear to be more jagged and not as close to the actual values. Of course, even in the eight-hour case, the trends of the HMT are predicted fairly well. For the blast furnace operator, knowing the trends of future HMT can be as important as knowing the actual future HMT value when deciding upon what actions to take.

It is important to note that the complexity of the neural network configuration that performs best over all random partitions, or runs, is generally higher than the node configuration that performs best only on the default testing set. For example, for four-hour prediction, the best mean configuration has 36 hidden nodes, while the best default configuration contains only 20 hidden nodes. One reason for this finding could be that
the best mean network configuration requires increased generalization to perform well over test data sampled from many different regions of the overall dataset.

The final comparison of importance involves networks trained with data in which input-output lags were manually set based on domain experts (forced lags), and data in which lags were automatically calculated by the method described in Section 3 (auto lags). Comparisons of networks trained using auto and forced lags are illustrated in figures 8-10.

![NMSE for Automated vs. Forced Lag Networks](image)

Figure 8. NSME comparison of networks using forced lags vs. using auto lags.
Figure 9. Pearson comparison of networks using forced lags vs. using auto lags.

Figure 10. NSME comparison of networks using forced lags vs. using auto lags.
One can see from the above graphs that the networks with forced lags perform as well or even a little bit better than the networks with automated lags for very short prediction horizons up to three hours. However, for prediction horizons in the four to seven hour range, the networks with automated lags perform better. One reason may be that when the prediction horizon is larger than the input-output lag of a particular input variable, the smallest lag that can be implemented between the input and the output is equal to the prediction horizon. Most of the inputs have lags estimated by the experts to be within one to three hours, with two variables lagged at four hours and three at six hours. For horizons greater than three hours, most of the “optimal” lags specified by the domain experts cannot be implemented in the data due to the above issue. Therefore, in such situations, it may be more optimal to search for an automated lag for every variable where maximum lag \(\geq\) automated lag \(\geq\) prediction horizon \(>\) forced lag. This could be why auto lags outperform forced lags for prediction horizons in the four to seven hour range. However, since the maximum lag is eight hours, eight-hour predictions for both the forced lag and automated lag cases lead to networks where all the effective input-output lags are equal to eight hours, the prediction horizon. That is why the eight-hour networks in both cases perform nearly the same.
Section 6: The Pitfalls of Moving Window Averaging

6.1 The Problems of Moving Window Averaging

The earlier sections of the paper mentioned moving window averaging as a way to average data of finer granularity (i.e. 20-minute data) to data of more coarse granularity (i.e. hourly data). This method is particularly useful because the averaging is performed without a major loss in the size of the original data set.

However, there are problems with this approach. Intuitively, it does not seem possible to create more information given the information inherent within an existing data set. In the current example, transformed data are averaged at 20-minute intervals, and moving window averaging is applied to create hourly data by setting the size of the averaging window equal to 3. Upon closer inspection, it becomes clear that although this newly created data are treated as hourly data, in fact, each data point in the new data set contains only 20 minutes worth of new information with respect to the previous data point in the averaged time series. The data point at time \( t \) has an overlap of 40 minutes worth of information with the data point at time \( t-1 \). Therefore, in the case of one-hour networks, the network is not exactly predicting the full hour’s worth of new HMT information.

This overlap between data points also causes the resulting HMT time series to be much smoother in nature. This is because HMT is not changing as much from one data point to the next. Figure 11 below displays the time series of HMT when the data are averaged using moving window:
For relatively smaller prediction horizons, the smoothness of this curve provides more help to the ANN when predicting future HMT. This is because the network is able to rely more heavily on previous HMT values to predict future HMT.

6.2 Comparison of Moving Window With Direct Hourly Averaging

For purposes of comparison and in order to obtain a more accurate representation of network performance, the networks are retrained on newly obtained 5-minute HMT data that are hourly averaged in a straightforward manner. That is, every contiguous set of 12 5-minute data points is averaged to form a single hourly averaged data point. Starting with an initial set of 12600 5-minute data points, this method
results in 12600/12 = 1050 hourly averaged data points. This set is split between the training phase (80 %) and validation (20%).

Figure 12 below displays the HMT time series that results from hourly averaging of the 5-minute data:

![HMT Values From Direct Hourly Averaged Data](image)

Figure 12. HMT time series from direct hourly averaging of 5-minute data.

It is clear that in comparison to Figure 13, the above HMT time series is more jagged than its moving window counterpart. This type of series is much tougher for a neural network to predict.
Networks are created using the same input variables in Section 3, where the input-output lags were manually set based on ranges provided by domain experts. Figures 13-17 display the time series results for HMT prediction at different hours.

**1 Hour Predictions Using Direct Hourly Averaged Data**

![Graph of 1 hour predictions using direct hourly averaged data](image1)

Figure 13. 1-hour HMT hourly averaged. Configuration: 1 hidden layer with 29 nodes.

NMSE: 0.3166 Correlation: 0.8333

**2 Hour Predictions Using Direct Hourly Averaged Data**

![Graph of 2 hour predictions using direct hourly averaged data](image2)

Figure 14. 2-hour HMT hourly averaged. Configuration: 1 hidden layer with 29 nodes.

NMSE: 0.3854 Correlation: 0.7886
Figure 15. 4-hour HMT hourly averaged. Configuration: 1 hidden layer with 28 nodes.
NMSE: 0.3411  Correlation: 0.7216

Figure 16. 6-hour HMT hourly averaged. Configuration: 1 hidden layer with 24 nodes
NMSE: 0.3333  Correlation: 0.8266
Although the new HMT values are more difficult to predict, it turns out that the HMT prediction results do not worsen by much when transitioning from moving window data to direct hourly averaged data. In fact, prediction for horizons eight hours, seven hours, and four hours into the future are better when using the new data as opposed to moving window data. The reason why networks predicting at shorter time horizons perform better under moving window is that these networks can rely more heavily on the last known HMT value when predicting future HMT. This is because the change in HMT values from one data point to the next is less abrupt in the moving window data.

In addition, note the reduction in the size of the hidden layer for networks trained using the direct hourly averaged data. In the eight-hour case, for example, the number of hidden nodes needed to predict HMT at a higher accuracy has dropped from 34 to 18. This reduction in network complexity could be one of the reasons behind better generalization.
6.3 Comparison of New Results With Previous Work

This subsection compares the prediction results presented in this paper with those previously obtained by Steelcorp and by past researchers in the data mining research group. The goal is to examine the progress made in terms of predicting future HMT.

As mentioned in Section 2, previous work at Steelcorp focused primarily on predicting daily HMT averages as opposed to hourly changes [21]. That is, rather than predicting average HMT one, two or even eight hours into the future, the research at Steelcorp focused on predicting the average value of HMT one or two days into the future. Since the change in daily-averaged HMT from one day to the next is likely to be smaller than the change in hourly HMT from one hour to two, four or eight hours into the future, the problem of predicting HMT on an hourly basis may be tougher than predicting average daily values. Also, since daily averaged data may not change much from one day to the next, these predictions can depend more heavily on last known daily HMT averages.

Steelcorp's results from predicting the daily-averaged HMT values exhibited Pearson coefficients of 0.78. The results from the current work (using direct averaged data) show that HMT one hour into the future can be predicted with a Pearson coefficient of 0.8333, while correlations of 0.7866, 0.8266, and 0.7129 were obtained for two-hour, six-hour and eight-hour results respectively. It is therefore clear that HMT predictions one hour, two hours and even six hours into the future are better than or very close to the results for daily HMT prediction. From a practical point of view, the ability to predict hourly-averaged HMT six or eight hours ahead is much more
useful for short-term HMT control than being able to predict the average HMT value over the next day. Of course, daily HMT predictions can be quite useful for predicting longer-term trends in the blast furnace.

The next step is to compare work performed in this paper with work performed by past research group members. Brad Banks [8] and Angela Ge [13] were able to obtain favorable results for hourly HMT prediction. However, once the prediction horizon was extended beyond four hours, the results would drop off rapidly, making these predictions less and less tolerable. While one hour and two hour results exhibited mean square errors of 0.004 and 0.005 respectively, HMT predictions for four hours and eight hours were 0.017 and 0.016 respectively. In addition, it became clear for networks in the two to four hour prediction range, that the predicted values were heavily lagging the actual values. These neural networks were placing an extremely high degree of importance on the last known HMT value for future HMT prediction and hence were not fully understanding the true relationships between the other inputs and HMT.

While HMT values up to three hours into the future were relatively close to the last known HMT value, HMT values beyond this horizon were markedly different. This meant that if the network relied heavily on the last known HMT value for predictions up to three hours into the future, its predictions would be fairly accurate. However, once the prediction horizon was increased, the last known HMT value was no longer an accurate indicator of future HMT and hence prediction performance suffered.

The MSE of networks in the current paper, with prediction horizons ranging from one to eight hours, are 0.0102, 0.0138, 0.0147, 0.0148, 0.0172, 0.0145, 0.0188 and
0.0211 respectively. Although the MSE is higher here for prediction horizons one hour, two hours and even three hours into the future, networks predicting HMT four, six and seven hours into the future are marginally better than those presented in [8].

The question that one must now ask is: why were the networks presented by Brad Banks [8] able to rely on the last known HMT value and achieve better results than networks in this paper (in the case of one to three hour predictions)? In order to answer this question, one must look at the type of data used for training networks in the current work as compared to data used in [8]. While this section used direct, hourly averaged data for training, the networks constructed by Banks were trained with HMT data that were *linearly interpolated*. Due to the low frequency of HMT measurements with respect to input measurements, the output time series exhibited step-function qualities, making prediction very difficult. Therefore, linear interpolation between measurements of HMT was used to approximate values for the missing data points.

One undesirable side effect of this technique, however, was to introduce unrealistic smoothness to the HMT time series. After linear interpolation, many sets of consecutive HMT values were now on a straight line; this ensured that, in many cases, HMT values from one hour to the next did not change significantly. This is probably the reason why one-hour, two-hour and three-hour networks using linearly interpolated data outperform their counterparts trained on pure hourly averaged data. The networks trained using linear interpolation are able to rely on the last known HMT value more than networks trained on direct hourly averaged data. The networks trained in this paper are more robust; as the prediction horizon increases, these networks do not deteriorate in prediction performance as significantly as the networks presented in [8].
This indicates that networks trained here may have been more successful in learning the
real patterns inherent within blast furnace operations rather than depending heavily on
the last known HMT value.
Section 7: Comparing ANNs to Standard Statistical Analysis

7.1 The Linear Regression Model

Prior work has stated that standard linear modeling techniques have not performed well in terms of predicting blast furnace parameters. The purpose of this section is to compare the results obtained in the previous section using MLP networks with those of the linear regression. The linear regression is probably the most commonly used technique to approximate a linear function between a set of inputs and a specific output.

If one assumes that there are $p$ inputs into the system, one can create an $n$ by $p$ input matrix $X$, where each row corresponds to a specific combination of input values at a given time, and each column corresponds to a particular input. Since it is desirable to include a constant in the regression model, one column of the input matrix is set equal to all 1’s. One would like to find the regression coefficients that minimize the error of prediction, over all $n$ points in time. The vector that contains these coefficients, $\beta$, has 1 column and $p$ rows, where each element corresponds to the coefficient of a specific variable. The vector $y$ also has 1 column and $p$ rows, where each element represents the value of the output at a specific point in time. Therefore, the overall model and solution can be presented as follows from [27]:

$$y = X\beta + \varepsilon$$

$$b = \beta = (X'X)^{-1} X'y$$

$$\hat{y} = Xb = X(X'X)^{-1} X'y$$
where \( e \) is an \( n \) by 1 vector of random disturbances. While \( \beta \) is the coefficient matrix for the population, \( b \) is the estimate derived from the above method. \( \hat{y} \) is the predicted output of the model based on a set of inputs.

### 7.2 Implementation and Comparison of Results

The above regression analysis was performed on the forced-lag data in hopes of comparing the results of the linear models with those of neural networks. Therefore, for each prediction horizon, the same data sets from Section 6.2 were used to perform the least-squares regression analysis. The first 80\% of the data set was reserved for performing the regression analysis while the last 20\% comprised the validation set. The table below summarizes the normalized mean square error, Pearson coefficient and average error of the regression models for prediction horizons of one to eight hours.

<table>
<thead>
<tr>
<th>Prediction Horizon (in hours)</th>
<th>NMSE</th>
<th>Pearson Coeff.</th>
<th>AE (in degrees C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.4184</td>
<td>0.7311</td>
<td>12.5035</td>
</tr>
<tr>
<td>2</td>
<td>1.0610</td>
<td>0.2915</td>
<td>19.2361</td>
</tr>
<tr>
<td>3</td>
<td>1.2267</td>
<td>0.1984</td>
<td>21.2661</td>
</tr>
<tr>
<td>4</td>
<td>1.3808</td>
<td>0.0716</td>
<td>22.7317</td>
</tr>
<tr>
<td>5</td>
<td>1.4728</td>
<td>-0.0317</td>
<td>23.1390</td>
</tr>
<tr>
<td>6</td>
<td>1.4946</td>
<td>0.0254</td>
<td>22.2969</td>
</tr>
<tr>
<td>7</td>
<td>1.1732</td>
<td>0.1723</td>
<td>20.0651</td>
</tr>
<tr>
<td>8</td>
<td>1.5968</td>
<td>-0.0819</td>
<td>22.1469</td>
</tr>
</tbody>
</table>

Table 4. Performance of Linear Regression on direct hourly averaged data
From the above chart, it is clear that except for the one-hour prediction case, the regression NMSE for every prediction horizon is larger than 1. This means that in the range of two to eight hour prediction horizons, the linear regression models actually perform *worse* than if one used the mean HMT value as a predictor.

The linear model’s relatively better performance in the one-hour case can be understood by examining the regression coefficients. Since the input data is normalized between 0 and 1, one can compare the different regression coefficients and, based on the magnitude, derive some intuition about an input’s relative importance to the model. In the case of the one-hour regression, the coefficient for the last known HMT value (0.7291) is much higher than the coefficients for the other input variables. The one-hour regression produces relatively better predictions than the regressions for horizons two to eight hours because 1) the one-hour linear regression depends almost solely on the last known HMT value and 2) the one-hour future HMT value is closer to the last known HMT value than are future values at larger prediction horizons.

The graphs below provide a detailed comparison of performance (in terms of NMSE and Pearson coefficient) between the linear regression models and the MLP networks for horizons ranging from one to eight hours into the future.
From the above graphs, it is clear that the feed-forward neural networks outperform their linear counterparts for every prediction horizon. In terms of NMSE, the two model types are closer in performance at the one-hour prediction level, and then diverge drastically. While the neural network NMSE is always lower than the regression NMSE
and never rises above 0.5064 (eight-hour prediction), the NMSE of the linear regression models are constantly higher, reaching a peak of 1.5968 for eight-hour prediction. As mentioned earlier, the reason why one-hour predictions for the regression model are relatively closer to the ANN predictions is that the one-hour regression can more successfully rely on the last known HMT value for predicting future HMT. For horizons greater than two hours, this is not true, and, therefore, one sees a sharp increase in the error of the linear model. The regression model is unable to capture the non-linear relationships of the system and must therefore depend heavily on the last known HMT value.

In terms of the Pearson coefficient evaluation, the results are the same. The one-hour linear model is able to predict trends reasonably well with a Pearson coefficient of 0.7311. However, for larger prediction horizons, the trend prediction of the linear model deteriorates extremely rapidly, reaching its nadir at eight hours, when it equals –0.0819. The fact that many of the Pearson coefficients are close to zero or even negative suggest that the linear model is not capable of predicting any real trends in HMT. By contrast, the Pearson coefficients of the neural network predictions never fall below 0.7129. The conclusion of this section is clear: the feed-forward neural network is by far a superior method to the linear regression for modeling the complex, non-linear relationships of the blast furnace.
Section 8: Methods for Input Saliency Determination

8.1 Motivation

As described in Section 1, limiting the set of inputs used for prediction is desirable for different reasons. Reducing the size of the input space can lead to higher network performance. Networks that are too large (i.e. that contain a large number of inputs and a large number of hidden nodes) carry an abundance of free parameters, or weights. This can harm the generalization power of the network and thus hamper overall prediction accuracy [18]. Also, redundant inputs can introduce noise into the model and hurt generalization in that way.

Secondly, determination of input importance can help in the selection of the most useful input parameters for controlling HMT. Ideally, one would like to choose those input variables that have the strongest effect on the output as HMT control parameters [11] [40].

8.2 Input-Output Derivative Analysis

8.2.1 Theoretical Analysis

The first method that is applied consists of examining the input-output derivatives of a trained neural network over time. The trained network provides a function that maps each input value combination to an HMT value. The current aim is to find the relative importance of each input variable into the network. Given the generalized mapping function $g$, represented by a trained neural network, one would like to find $\partial Y/\partial X_n$, where $Y$ represents the output of the ANN (in the current example,
HMT) and $X_i$ represents the $i$th input. Generally, this derivative will have the following functional form:

$$\frac{\partial Y}{\partial X_i} = g(X_1, X_2, \ldots, X_n | w_1, w_2, \ldots, w_m) \quad (8.1)$$

where $m =$ number of weights

Thus, the partial derivative of the output with respect to each input is a function of the current set of inputs as well as the network’s weights. Since the weights are fully determined after training and remain static afterwards, one can think of the partial derivative as a function of the input combination. Thus, as the input combination changes, so does the effect of each input on the output. In the practical terms of the blast furnace, the importance of a particular input on HMT varies as the operating conditions change.

Since it was earlier determined (in Section 5) that a feed-forward neural network with one hidden layer was sufficient for HMT prediction, this section will focus on the derivation of the partial derivative for this type of network. A derivation for the two-hidden layer case easily follows from this analysis and is presented in [29].
A basic diagram of a network and useful terminology appears below with

**FEED-FORWARD NEURAL NETWORK WITH 1 HIDDEN LAYER**

Input Layer (Layer 1)  |  Hidden Layer (Layer 2)  |  Output Layer (Layer 3)

\[ X_1, X_2, \ldots, X_m \rightarrow w_{11}(1)X_1^{(1)}, w_{12}(1)X_2^{(1)} \]
\[ \vdots \]
\[ w_{n1}(1)X_n^{(1)} \rightarrow X_1^{(2)}, X_2^{(2)}, \ldots, X_m^{(2)}, y \]

Figure 20. The basic architecture of a feed-forward neural network with appropriately marked features.

the following definitions:

- \( X_i \) = the input to neuron \( i \) at the input layer
- \( X_i^{(j)} \) = the total input to neuron \( i \) in layer \( j \)
- \( Y_i^{(j)} \) = the output of node \( i \) in layer \( j \)
- \( B_i^{(j)} \) = the bias of node \( i \) in layer \( j \)
- \( W_{ij}^{(k)} \) = the weight from node \( i \) in layer \( k \) to node \( j \) in layer \( k+1 \)
- \( Y_i^{(I)} = X_i \)
\[ X_i(j) = \left( \sum_{k=1}^{n(j-1)} W_{ki}(j-1) Y_k(j-1) \right) + B_i(j) \]

\[ Y_i = f\{ X_i(j) \} \] where \( f \) is the transfer function of node \( i \) in layer \( j \)

Let \( Y \) be the output HMT. The input-output sensitivity analysis for an input \( X_i \) begins with the calculation of the partial derivative \( \partial Y / \partial X_i \). Using the chain rule, one obtains:

\[
\frac{\partial Y}{\partial X_i} = \frac{\partial Y}{\partial X_i(3)} \cdot \frac{\partial X_i(3)}{\partial X_i}
\]

\[ = \partial f\{ X_i(3) \} / \partial X_i(3) \cdot \frac{\partial X_i(3)}{\partial X_i} \]

In the present case, \( f \), the transfer function of the output neuron, is the logistic sigmoid function. Therefore, \( \partial f\{X_i(3)\} / \partial X_i(3) \) is simply

\[
\frac{\partial}{\partial n} \left( \frac{1}{1+e^{-n}} \right) \bigg|_{n=X_i(3)}
\]

The next step is to calculate the second part of the above expression, \( \partial X_i(3) / \partial X_i \).

This can be broken down in the following manner:

\[
\frac{\partial X_i(3)}{\partial X_i} = \partial \left( \sum_{p=1}^{n(2)} W_{pi}(2) \cdot Y_p(2) \right) / \partial X_i
\]

\[ = \partial \left( \sum_{p=1}^{n(2)} W_{pi}(2) \cdot f\{ X_p(2) \} \right) / \partial X_i
\]

\[ = \sum_{p=1}^{n(2)} W_{pi}(2) \cdot \frac{\partial f\{ X_p(2) \}}{\partial X_p(2)} \cdot \frac{\partial X_p(2)}{\partial X_i}
\]

\[ = \sum_{p=1}^{n(2)} W_{pi}(2) \cdot \frac{\partial}{\partial n} \left( \frac{1}{1+e^{-n}} \right) \bigg|_{n=X_p(2)} \cdot \frac{\partial X_p(2)}{\partial X_i}
\]

(8.2)

Since the input nodes possess the transfer function \( f(a) = a \), it is obvious that \( \partial X_p(2) / \partial X_i = W_{ip}(1) \). Therefore, the total equation for the partial derivative becomes:
\[
\frac{\partial Y}{\partial X_i} = \frac{\partial}{\partial n} \left( \frac{1}{1 + e^{-n}} \right)_{n=X(3)} \cdot \sum_{\rho = 1}^{n(2)} W_{p1}(2) \cdot \frac{\partial}{\partial n} \left( \frac{1}{1 + e^{-n}} \right)_{n=Xp(2)} \cdot W_{p}(1)
\]

(8.3)

It is clear that the derivative of the output, \( Y \) with respect to the input \( X_i \), depends not only on the weights and biases of the trained neural network, but, also on the values of all the input variables. Therefore, this derivative is \textit{time dependent}. The importance of a particular input varies over time as the values of the input variables change.

\subsection*{8.2.2 Implementation and Interpretation}

Since the effect of \( X_i \) on HMT changes over time, one way to interpret these results is to graph, for each \( X_i \), \( \frac{\partial Y}{\partial X_i} \) over all data points in the training and testing set. Since the data is a time series, one can consider each data point as the combination of input values at a particular instant in time, and, thus, for each data point, (8.3) can be evaluated appropriately and graphed.

Since the trained neural networks are implemented using MATLAB, the above sensitivity calculation can also be numerically solved using MATLAB (see Appendix 2 for code). The required function, \texttt{getDerivativeMatrix(net, dataMatrix)}, takes the trained neural network object \( \texttt{net} \) and the training plus testing dataset \( \texttt{dataMatrix} \) as arguments. The function returns a matrix \( M \) with the number of columns equaling the number of input variables and the number of rows equaling the number of data points in the training plus testing data set. Therefore, for each neural network one can define a \textit{derivative matrix} \( M \):
where $X_k^i$ corresponds to the $i$th value of variable $X_k$

By examining the $i$th column of a derivative matrix $M$, one can observe how $\partial Y/\partial X_i$ varies over time (the first row corresponding to $t=1$, the second $t=2$ and the $n$th row representing $t=n$).

### 8.2.3 Application of Derivatives

Since the partial derivative of the output with respect to each variable varies over time, there must be a way to consolidate the temporal information into a value that could provide an overall ranking for the particular variable in question.

The analysis begins by examining, for each neural network, the behavior of each input’s derivative over time. These *derivative graphs* are easily calculated by graphing, for a particular input $X_k$, the $k$th column of the corresponding derivative matrix vs. time. The goal is to get a qualitative sense of how the derivative for a particular variable fluctuates over time. Is the trend consistent, fluctuating around some central value, or, is the derivative of each input highly volatile over time, thus exhibiting no clear trend? Some examples of these derivative graphs are shown below.
Figure 21. Time series derivative for coke (t-5) from the 1-hour network.

Figure 22. Time series derivative for HMT(t) from the 1-hour network.
Each of the above graphs can be classified into a different category. The derivative values in Figure 21 are for the variable coke used for predicting HMT one hour into the future. In this case, the derivative values are relatively consistent over time, showing only a few “spikes,” or extreme values. Figure 22 depicts the derivative time series for the last known HMT value for the same one-hour network. Here, the derivative graph contains a large number of spikes and thus has a high variance. However, the sign of the derivative is consistently positive. As such, the overall effect of the last known HMT value in predicting HMT one hour from now is overwhelmingly positive. This makes intuitive sense, because, as mentioned earlier in the paper, ANNs that predict closer into the future have a markedly higher dependence on using the last known HMT value when predicting future HMT. The research team found that as the time horizon increases, this reliance on the last known HMT value falls.
Figure 23 displays the derivative of HMT (t+1) with respect to oxygen (t-3). Here, unlike the other two cases, the derivative varies from positive to negative values fairly often. The derivative has spikes that are both negative and positive. This means that oxygen sometimes has a positive effect on HMT, while other times the effect is more negative. Since the current problem deals with the complex, dynamic process of a blast furnace, this type of behavior is understandable.

Since the variance of many of the derivatives is high, and since positive and negative spikes can occur quite frequently, it is difficult to provide a temporal-based analysis of the derivative time series. That is why researchers have proposed measures such as examining the minimum or maximum derivative values over time, and ranking the given variable based on these values [31]. Instead, this paper adopts the formulation described by Zurada et al. [33], one that examines the average magnitude of the derivative over time for each variable, and ranks each variable based on this value. Consequently:

\[
\text{The sensitivity of } X_i = \left( \frac{1}{n} \right) \sum_{t=1}^{n} |(\partial Y / \partial X_i)_t'| \quad (8.4)
\]

where \((\partial Y / \partial X_i)_t'\) is the partial derivative of \(Y\) with respect to \(X_i\) evaluated at input combination \(t\). The absolute value is used, because, it is the magnitude of the variable’s derivative that is most important in the current discussion rather than its sign.

In terms of determining whether a particular input variable is a good candidate for being a control parameter of the output, one might also be interested in the variable’s derivative variance over time. A parameter that maintains a moderate, consistent derivative value over time will have a more predictable effect on the output as compared to a variable whose derivative values are volatile and contain many spikes.
8.3 Weight-Interpretation Analysis

8.3.1 Original Algorithm

Rather than characterizing the neural network as a black box, researchers have tried to determine input saliency by looking at the architecture of the neural network itself and defining intuitive measures. Specifically, Carson [35] introduces the concept of “partitioning” the importance conveyed by the ANN’s weights among each of the network’s inputs. Generally, for each input node, his method looks at the magnitude of the weights that connect it to the nodes in the hidden layer. In addition, each node in the hidden layer is also evaluated for importance based on the magnitude of the weight that connects it to the output node. Therefore, an input that has a very large weight connecting it to a hidden node, is not as important if that particular hidden node is connected by a small weight to the output node.

Carson’s reasoning can be illustrated with a simple example. Figure 24 below presents a small feed-forward network with two inputs, two hidden nodes and one output.
One can measure $X_i$'s importance by looking at how it connects to the nodes in the hidden layer. The amount of $X_i$'s activation that reaches the nodes in the hidden layer depends on the weights that connect it to these nodes. Intuitively:

$$X_i's \text{ effect on } Y = (X_i's \text{ relative connectivity to Node } 3) \cdot (\text{Node } 3's \text{ effect on } Y) + (X_i's \text{ relative connectivity to Node } 4) \cdot (\text{Node } 4's \text{ effect on } Y)$$

$$X_2's \text{ effect on } Y = (X_2's \text{ relative connectivity to Node } 3) \cdot (\text{Node } 3's \text{ effect on } Y) + (X_2's \text{ relative connectivity to Node } 4) \cdot (\text{Node } 4's \text{ effect on } Y) \quad (8.5)$$

The larger the weights are that are connecting $X_i$ to a particular hidden node when compared to $X_2$, the more importance should be given to $X_i$'s inputs into that hidden node. Thus, with respect to Node 3, $X_i$ is more important than $X_2$ if $X_i$'s connections make up a larger fraction of the total weight into Node 3. This is why one must look at
$X_1$’s and $X_2$’s relative connectivity to different hidden nodes. Therefore, expanding on (8.5), the following is true:

- $X_1$’s relative connectivity to Node 3 = $w_{11}(1) / (w_{11}(1) + w_{21}(1))$
- $X_1$’s relative connectivity to Node 4 = $w_{21}(1) / (w_{21}(1) + w_{22}(1))$
- $X_2$’s relative connectivity to Node 3 = $w_{12}(1) / (w_{12}(1) + w_{11}(1))$
- $X_2$’s relative connectivity to Node 4 = $w_{22}(1) / (w_{22}(1) + w_{21}(1))$ (8.6)

The next step is to evaluate the effect of a hidden node on $Y$. Since a hidden node is directly connected to the output node by a weight, this weight determines how much of the hidden node’s output is translated into the ANN’s overall output. Thus, it follows:

- Node 3’s effect on $Y = w_{11}(2)$
- Node 4’s effect on $Y = w_{12}(2)$ (8.7)

In practice, Carson [35] and Nath [36] use the magnitude of the weights in the above reasoning, indicating that even highly negative weights are considered important. Ravinder Nath describes the above logic as an algorithm for evaluating input saliencies:

1. Each hidden-to-output weight $w_{ko}$, irrespective of sign, is incorporated into the input-to-hidden weights $w_{ij}$ using the following expression:

$$w_{ij}^* = \{|w_{ij}| / S_j\} \cdot |w_{jO}|$$

where $i = 1, 2, \ldots, z; j = 1, 2, \ldots, h; z =$ number of inputs, $h =$ number of hidden nodes and

$$S_j = \sum_{i=1}^z |w_{ij}|$$

2. For each input node, the newly adjusted weights $w_{ij}^*$ are summed over all input nodes. This sum is then converted to a percentage of the total of all the inputs. This percentage represents the saliency of the input variable.

Figure 25. The original weight-interpretation algorithm.
8.3.2 Modification to Existing Algorithm

In many modeling situations, the inputs are normalized to non-negative values. In the current problem, each input has been normalized to a value between 0 and 1. Also, the logistic sigmoid function, displayed in Figure 26, is often used as the transfer function for the hidden and output nodes. The logistic sigmoid maps all input values to numbers between 0 and 1.

\[ a = \text{logsig}(n) \]

Log-Sigmoid Transfer Function

Therefore, under these conditions, if a positive input is connected to a hidden node by a negative weight, this indicates that the input is less important with respect to the particular hidden node. Consider an extreme example for purposes of illustration. Imagine that all the weights into a hidden node are highly negative. Since all inputs are positive, the total input into the hidden node will be very negative. Since the transfer function in the hidden node is a logistic sigmoid, it is clear from above that the negative input to the node will result in a near-zero activation of the node.

Given these conditions, one can try to employ a transformation of the weights in the above algorithm different from the absolute value, one that will penalize negative weights and make their magnitude small and sign positive. One alternative to the absolute value function that meets these requirements is the exponential. However, one
would prefer a transformation of the weight that yields a value that is, in regards to size, on the same scale as the original weight. Therefore, the transformation function chosen is the following:

\[ f(w_{ij}) = \frac{1}{1 + e^{-w_{ij}}} \]  

(8.8)

Rather than using the exponential, which could skew the original relationship between the weight magnitudes, the above function tries to maintain, relatively, the same scale of magnitude as the weight that it transforms. The magnitude is multiplied by a penalty term, which is the logistic sigmoid evaluated at the value of the weight. Intuitively, if a weight is highly negative, the absolute value of the weight will be large, while the evaluation using the logistic sigmoid will be near zero, curbing the final value. Thus, negative weights are transformed to smaller positive values while more positive weights are transformed into relatively larger positive values. The updated algorithm for input saliency determination is now as follows:

1. Each hidden-to-output weight \( w_{ko} \), irrespective of sign, is incorporated into the input-to-hidden weights \( w_{ij} \) using the following expression:

\[ w_{ij}^* = \frac{f(w_{ij})}{S_j} \cdot f(w_{jo}) \]

where \( i = 1, 2, \ldots, z; j = 1, 2, \ldots, h \); \( z \) = number of inputs, \( h \) = number of hidden nodes,

\[ S_j = \sum_{i=1}^{z} f(w_{ij}) \text{ and } f = f(w_{ij}) = |w_{ij}| \cdot 1/(1 + e^{-w_{ij}}) \]

2. For each input node, the newly adjusted weights \( w_{ij}^* \) are summed over all input nodes. This sum is then converted to a percentage of the total of all the inputs. This percentage represents the saliency of the input variable.

Figure 27. The modified weight-interpretation algorithm.
The MATLAB code that implements this modified algorithm can be found in Appendix 2.

8.4 Weight-Derivative Ranking Analysis

8.4.1 Weight Saliency

The third method presented is a new technique that can be thought of as a hybrid of weight-interpretation and ANN weight sensitivity. Le Cun [29] describes a method, commonly referred to as Optimal Brain Damage, which attempts to measure the saliency of each weight in the neural network. The general notion explored here is that it might be possible to represent the importance of a weight by a means other than its magnitude alone. In fact, Bishop claims that this sort of magnitude-derived importance of weights may be inaccurate in some cases [18]. Therefore, this paper searches for an alternate measure of weight importance.

Assume that the goal is to measure the change in error resulting from the removal of a particular weight. A brute force method might consist of manually setting a weight to zero, retraining the partially connected network, and then observing the change in the mean squared error. This same experiment would be carried out for all weights in the network. One would then want to remove the weights whose removal caused the smallest increase in error. Given a set of $m$ weights, one would have to retrain the network $m$ times in order to see which weights have the largest effects on MSE.

Fortunately, Le Cun [41] devised an analytical way to approximate the change in error due to the removal of a particular weight. This method is based on the second
derivatives of the error with respect to each of the weights. Consider the change in error function resulting from changing the weight \( w_i \) to \( w_i + \delta w_i \). The total differential of the error can be written as:

\[
\delta E = \sum_i \frac{\partial E}{\partial w_i} \delta w_i + \frac{1}{2} \sum_i \sum_j H_{ij} \delta w_i \delta w_j + O(\delta w^3) \tag{8.9}
\]

where \( H_{ij} \) is an element of the Hessian matrix. The Hessian Matrix is a matrix of second derivatives of the error with respect to the weights. In general:

\[
H_{ij} = \frac{\partial^2 E}{\partial w_i \partial w_j} \tag{8.10}
\]

Le Cun et al. assume that their trained network has completely converged. This means that the partial derivatives of the error with respect to the weights are zero, erasing the first expression on the right side of (8.9). Ignoring third order derivatives, the equation becomes:

\[
\delta E = \frac{1}{2} \sum_i \sum_j H_{ij} \delta w_i \delta w_j \tag{8.11}
\]

Since the current interest is to change one weight at a time, Le Cun chooses to ignore the cross-derivatives and thus assumes that non-diagonal terms of the Hessian matrix are zero. The author uses an efficient approximation for calculation of the diagonal Hessian [18]. Thus, \( \delta E = \frac{1}{2} \sum_i \sum_j H_{ij} \delta w_i \delta w_j \) above becomes

\[
\delta E = \frac{1}{2} \sum_i H_{ii} (\delta w_i)^2 \]

If one assumes that the initial value of the weight in the trained network is \( w_i \), and the aim is to see the change in the error when the weight is dropped
to zero, one can set $\delta w_i = w_i$, transforming the measure of saliency of weight $w_i$ into the following:

$$\delta E_i = \frac{1}{2} H_{ii} \times w_i^2 \quad (8.12)$$

This measurement is the approximate *increase in error* that will be incurred by removing weight $w_i$ from the network. Le Cun's algorithm consists of evaluating the saliency of each weight using (8.12), sorting the weights based on saliency, then removing the least important weight (the weight resulting in the smallest increase in error). The network is then retrained without this weight and the whole process is continued until some stopping criterion is reached.

Hassibi and Stork [43] extend this idea further by analytically calculating the readjustment to the remaining weights after the removal of a single weight. Thus, successive retraining of the network is not required after each weight deletion. The authors use an iterative approach to recalculate the inverse Hessian after each weight removal and update of the remaining weights. At each pass, the weight that is removed is the one that has the least saliency. Again, this process ends after some stopping criterion is reached. One nice thing about this method is that the Hessian matrix is not assumed to be diagonal.

As mentioned earlier, the weight-interpretation ranking method relies heavily on the magnitude of the weights as a reflection of a connection's importance. However, it has been pointed out in the literature that this evaluation method may not be as optimal as using the Hessian-based ranking described [18]. Therefore, it is natural to think about using the basic ideas of how node connectivity determines input importance, introduced
in weight-interpretation, and substituting weight magnitude with the saliency measures described by Le Cun and Hassibi et al.

Due to time and complexity constraints, it might be more useful to follow a method similar to Le Cun [41]. One would first come up with a measurement of saliency for each weight, and then employ Carson’s weight partitioning heuristics to determine input saliency.

8.4.2 Revising the Measure of Weight Saliency

This section begins with a modification to Le Cun’s technique. Since Le Cun assumes absolute convergence of the trained ANN, he ignores the following term in his calculation of \( \delta E_i \), the first-order increase in error due to removal of weight \( w_i \):

\[
\sum_i \frac{\partial E}{\partial w_i} \delta w_i \quad (8.13)
\]

After considering the early-stopping algorithm used to halt training in order to prevent over-fitting (Section 4), it may not be safe to assume that the partial derivative of the error with respect to each weight is equal to zero. Therefore, in the current situation it is not accurate to exclude this first derivative. (One may note that Hassibi and Stork’s algorithm inherently assume convergence as well - this again would be a problem for the current application).

Can one easily substitute \( \delta w_i = w_i \) as Le Cun had done earlier to come up with (8.12)? Since the weight term is no longer squared, the sign of \( w_i \) does matter. One must make sure that whatever term is added to (8.13) can still be interpreted as the increase in error due to the dropping of \( w_i \). Since the weight is initially \( w_i \), it is more accurate to set \( \delta w_i = -w_i \). This issue did not arise earlier because the weight term was
squared and Le Cun never considered the first derivative term. Thus, the new saliency term becomes:

$$\delta E_i = -\frac{\partial E}{\partial w_i} \times w_i + \frac{1}{2} H_{ii} \times w_i^2 \quad (8.14)$$

Intuitively, this makes sense because a weight that is initially positive will have a negative change when set to zero, while a weight that is initially negative will have a positive change in value when it is set to zero. Figure 28, on the next page, illustrates this subtle point.

<table>
<thead>
<tr>
<th>Case I: $\frac{\partial E}{\partial w_i}$ is negative and $w_i$ is positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>The derivative’s sign implies that a decrease in the weight leads to an increase in the error. Thus, $\delta E_i$ should be positive if $w_i$ is dropped to zero. Given (8.13), the following is true:</td>
</tr>
<tr>
<td>$\delta E_i = -(\text{negative}) \times \text{positive} = \text{positive}$ ✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case II: $\frac{\partial E}{\partial w_i}$ is negative and $w_i$ is negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>The derivative’s sign implies that an increase in the weight leads to a decrease in the error. Thus, $\delta E_i$ should be negative if $w_i$ is increased to zero. (8.13) gives the following:</td>
</tr>
<tr>
<td>$\delta E_i = -(\text{negative}) \times \text{negative} = \text{negative}$ ✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case III: $\frac{\partial E}{\partial w_i}$ is positive and $w_i$ is positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>The derivative’s sign implies that a decrease in the weight means a decrease in the error. Thus, $\delta E_i$ should be negative if $w_i$ is dropped to zero. (8.13) results in the following:</td>
</tr>
<tr>
<td>$\delta E_i = -(\text{positive}) \times \text{positive} = \text{negative}$ ✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case IV: $\frac{\partial E}{\partial w_i}$ is positive and $w_i$ is negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>The derivative’s sign implies that an increase in the weight means an increase in the error. Thus, $\delta E_i$ should be positive if $w_i$ is increased to zero. (8.13) results in the following:</td>
</tr>
<tr>
<td>$\delta E_i = -(\text{positive}) \times \text{negative} = \text{positive}$ ✓</td>
</tr>
</tbody>
</table>

Figure 28. Validation of first-order weight derivative term
Unlike in [41], this paper performed an exact calculation of the diagonal terms of the Hessian matrix in order to evaluate the saliency of the individual weights. The technique used is similar to Chris Bishop’s formulation [42] and can be found in Appendix 1.

8.4.3 New Weight Derivative Input Ranking Algorithm

Now that the concept of weight ranking and the key relationships that determine the Hessian have been derived, the next step is to combine these techniques into a new algorithm for ranking input variables. These weight rankings must be translated into input saliencies. Again, the basic logic of weight-interpretation is employed. Intuitively, the saliency of an input is determined by the relative “importance” of the weights that connect the input to each hidden node, and, the relative “importance” of the weights that connect each of the hidden nodes to the output. While Carson [35] uses the magnitude of each weight to represent its importance, this section uses the $\delta E_i$ calculated for each weight $w_i$ using the Hessian. The new ranking algorithm becomes:
1. The saliency of each hidden-to-output weight $w_{ko}$, $\delta E_{ko}$, is incorporated into the input-to-hidden weight saliencies $\delta E_{ij}$ using the following expression:

$$w_{ij}^* = \left\{ \frac{f(\delta E_{ij})}{S_j} \right\} \cdot f(\delta E_{ko})$$

where $i = 1, 2, ..., z; j = 1, 2, ..., h; z =$ number of inputs, $h =$ number of hidden nodes,

$$\delta E_{ab} = -\frac{\partial E}{\partial w_{ab}} \cdot w_{ab} + \frac{1}{2} \cdot H_{w_{ab}} \cdot w_{ab}^2$$

$$S_j = \sum_{i=1}^{z} f(\delta E_{ij}), \quad f(\delta E_{ij}) = |\delta E_{ij}| \cdot \frac{1}{1 + e^{-\delta E_{ij}}}$$

and $H_{w_{ab}} = \frac{\partial^2 E}{\partial w_{ab}^2}$

2. For each input node, the newly adjusted weights $w_{ij}^*$ are summed over all input nodes. This sum is then converted to a percentage of the total of all the inputs. This percentage represents the saliency of the input variable.

Figure 29. New input ranking algorithm based on weight derivatives (weight derivative method).

The code implementing this algorithm can be found in Appendix 2 of this paper.

One issue worth noting is that the same transformation that is performed in the modified weight-interpretation algorithm function is also performed on $\delta E_{ij}$ here. This is done in order to ensure that weights with negative $\delta E_{ij}$ are given less value, while keeping the saliency value itself positive. Since $\delta E_{ij}$ is interpreted to mean the increase in error caused by a removal of weight $w_{ij}$, a negative value for $\delta E_{ij}$ would mean that removal of the weight results in a decrease in error. Thus, when compared to
a weight with a positive $\delta E_{ij}$, this weight should have much less importance. For this reason, the absolute value transformation is not used.

8.4.4 Drawbacks to the Weight Derivative Method

The most obvious drawback to the weight derivative method is that the $\delta E_{ij}$s calculated may not be completely accurate. What happens in reality is that when a particular weight is removed, the other weight values change and compensate for the loss of the given weight. The $\delta E_{ij}$ calculated by Le Cun's method does not account for this. That is why Hassibi and Stork's [43] Optimal Brain Surgeon readjusts the weights after a weight removal. A more accurate, but significantly time consuming method would be to remove each weight separately and retrain the network to find the new change in error.

However, given time constraints, the method as presented in Section 8.4.3 is used. The next section applies the four ranking techniques above to the new, direct hourly averaged data, and attempts to evaluate the relative merits of the different ranking methods.
Section 9: Saliency Results and Future Work

Since the need to improve predictive performance is greater for networks trained on hourly averaged data than for those trained on moving window data, input saliency analysis is applied to the hourly averaged networks. Also, as described earlier, hourly averaged networks are used because they present a more accurate view of the blast furnace operations. This section focuses on the evaluation of the relative merits of the different ranking methods discussed in the previous section.

9.1 Ranking Comparison and Initial Pruning

The first goal is to perform a straightforward comparison of the different ranking methods, leaving all else constant. Therefore, rankings are first performed using the four techniques mentioned in Section 8 for networks that predict one to eight hours into the future.

The tables below show the input rankings created by the four different methods for a few different prediction horizons:

<table>
<thead>
<tr>
<th>Saliency Rank (1 being the highest)</th>
<th>Weight Interpretation</th>
<th>Weight Derivative Method</th>
<th>Input-Output Derivative Method</th>
<th>Modified Weight Int.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Oxygen(t-3)</td>
<td>TopgasH2(t)</td>
<td>HMT(t)</td>
<td>Coalinjection(t-3)</td>
</tr>
<tr>
<td>2</td>
<td>Hot Blast Temp (t)</td>
<td>Oxygen(t-3)</td>
<td>Cast(t)</td>
<td>G2heatflux(t-2)</td>
</tr>
<tr>
<td>3</td>
<td>Grp1HeatFlux(t-2)</td>
<td>TopgasCO2(t)</td>
<td>TopgasH2(t)</td>
<td>HMT(t)</td>
</tr>
<tr>
<td>4</td>
<td>Coalinjection(t-3)</td>
<td>Grp1HeatFlux(t-2)</td>
<td>Oxygen(t)</td>
<td>Oxygen(t)</td>
</tr>
<tr>
<td>5</td>
<td>HMT(t)</td>
<td>Hot Blast Temp (t)</td>
<td>Steam(t)</td>
<td>Ore by Coke(t-5)</td>
</tr>
<tr>
<td>6</td>
<td>WindFlow(t)</td>
<td>G2heatflux(t-3)</td>
<td>TopgasCO(t)</td>
<td>Steam(t)</td>
</tr>
</tbody>
</table>
Table 5. The input rankings for prediction of HMT (t+1).

<table>
<thead>
<tr>
<th>Rank</th>
<th>Interpretation</th>
<th>Method</th>
<th>Method</th>
<th>Derivative Method</th>
<th>Derivative Weight Int.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nutcoke(t-2)</td>
<td>Nutcoke(t-2)</td>
<td>Nutcoke(t-2)</td>
<td>Ore by Coke(t-2)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Group 1 Heat Flux</td>
<td>Oxygen</td>
<td>Topgas CO</td>
<td>Nutcoke(t-2)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Oxygen</td>
<td>Wind flow</td>
<td>Ore by Coke(t-2)</td>
<td>Oxygen</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Ore by Coke(t-2)</td>
<td>Ore by Coke(t-2)</td>
<td>current HMT</td>
<td>Group 1 Heat Flux</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Steam</td>
<td>Hot Blast Temp</td>
<td>Oxygen</td>
<td>Charge Time</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Cast2op2</td>
<td>Topgas CO</td>
<td>Hot Blast Temp</td>
<td>Cast2op2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Group 2 Heat Flux</td>
<td>Steam</td>
<td>Group 1 Heat Flux</td>
<td>Wind flow</td>
<td></td>
</tr>
</tbody>
</table>
Table 6. The input rankings for prediction of HMT (t+4). All of the above inputs are measured at current time t unless otherwise noted.

<table>
<thead>
<tr>
<th>Saliency Rank (1 being the highest)</th>
<th>Weight Interpretation</th>
<th>Weight Derivative Method</th>
<th>Input-Output Derivative Method</th>
<th>Modified Weight Int.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Group 1 Heat Flux</td>
<td>Coal Injection</td>
<td>Group 1 Heat Flux</td>
<td>Oxygen</td>
</tr>
<tr>
<td>2</td>
<td>Group 2 Heat Flux</td>
<td>Group 1 Heat Flux</td>
<td>HotBlastTemp</td>
<td>Group 1 Heat Flux</td>
</tr>
<tr>
<td>3</td>
<td>Topgas H2</td>
<td>Wind flow</td>
<td>Coke</td>
<td>Topgas H2</td>
</tr>
<tr>
<td>4</td>
<td>Ore by Coke</td>
<td>HotBlastTemp</td>
<td>Nutcoke</td>
<td>Cast</td>
</tr>
<tr>
<td>5</td>
<td>current HMT</td>
<td>Nutcoke</td>
<td>Group 2 Heat Flux</td>
<td>Ore by Coke</td>
</tr>
<tr>
<td>6</td>
<td>Nutcoke</td>
<td>Cast</td>
<td>current HMT</td>
<td>Charge Time</td>
</tr>
<tr>
<td>7</td>
<td>Coal Injection</td>
<td>Cast1op2</td>
<td>Topgas CO</td>
<td>Group 2 Heat Flux</td>
</tr>
<tr>
<td>8</td>
<td>Cast1op2</td>
<td>Topgas H2</td>
<td>Topgas H2</td>
<td>HotBlastTemp</td>
</tr>
<tr>
<td>9</td>
<td>Wind flow</td>
<td>Ore by Coke</td>
<td>Cast1op2</td>
<td>Nutcoke</td>
</tr>
<tr>
<td>10</td>
<td>HotBlastTemp</td>
<td>Oxygen</td>
<td>Wind flow</td>
<td>Wind flow</td>
</tr>
<tr>
<td>11</td>
<td>Oxygen</td>
<td>Group 2 Heat Flux</td>
<td>Ore by Coke</td>
<td>Topgas CO2</td>
</tr>
<tr>
<td>12</td>
<td>Coke</td>
<td>current HMT</td>
<td>Coal Injection</td>
<td>Coke</td>
</tr>
<tr>
<td>13</td>
<td>Cast</td>
<td>Topgas CO2</td>
<td>Cast</td>
<td>current HMT</td>
</tr>
<tr>
<td>14</td>
<td>Topgas CO2</td>
<td>Steam</td>
<td>Topgas CO2</td>
<td>Cast2op2</td>
</tr>
<tr>
<td>15</td>
<td>Charge Time</td>
<td>Charge Time</td>
<td>Cast2op2</td>
<td>Steam</td>
</tr>
<tr>
<td>16</td>
<td>Steam</td>
<td>Cast2op2</td>
<td>Oxygen</td>
<td>Topgas CO</td>
</tr>
</tbody>
</table>
Table 7. The input rankings for prediction of HMT(t+8). All of the above inputs are measured at current time t.

<table>
<thead>
<tr>
<th></th>
<th>Cast2op2</th>
<th>Topgas CO</th>
<th>Charge Time</th>
<th>Coal Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Topgas CO</td>
<td>Coke</td>
<td>Steam</td>
<td>Cast1op2</td>
</tr>
</tbody>
</table>

Although, for every given prediction horizon, the different methods have come up with different overall input rankings, it is somewhat reassuring that some similarities in results remain among the different saliency techniques. For example, in the case of one-hour prediction, oxygen, coal injection, the last known HMT value, and heat flux all seem to be important variables. In fact, the weight derivative method is the only one not to place previous HMT near the top of its rankings. Similarly, for four-hour prediction, it seems that nut coke, ore by coke and oxygen are seen by all ranking techniques as some of the most salient input parameters.

Once the above rankings were calculated, the next step was to use these rankings to choose subsets of the input set that contained the most important variables for each network. Finally, new networks were trained using NNRUN with data containing only the pruned input variables. Retraining the networks using NNRUN meant that the number of hidden nodes in the original networks was not preserved as NNRUN performed new searches to find the optimal node configurations for each of the resulting networks. Due to time and machine constraints, the number of randomized test/train partitions used by NNRUN was limited to two.

The number of inputs to keep for each network after pruning was based on the size of the network’s original input set and on prior experience in working with the networks. The one-hour network, which initially had 26 inputs, was reduced to 18. The two-hour network’s input space was reduced from 24 to 16. Networks predicting four hours or more into the future, each of which originally had 18 inputs, were pruned to
13. The tables below illustrate the NMSE, Pearson coefficient and average error for networks trained using variable subsets chosen by different ranking algorithms.

<table>
<thead>
<tr>
<th></th>
<th>NMSE.</th>
<th>Pearson Coeff.</th>
<th>AE (in degrees)</th>
<th># of hidden nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight-Interpretation</td>
<td>0.3527</td>
<td>0.8079</td>
<td>10.6884</td>
<td>11</td>
</tr>
<tr>
<td>Weight Derivative Method</td>
<td>0.4857</td>
<td>0.7216</td>
<td>12.9897</td>
<td>15</td>
</tr>
<tr>
<td>Input-Output Derivative Method</td>
<td>0.3536</td>
<td>0.8051</td>
<td>10.7710</td>
<td>11</td>
</tr>
<tr>
<td>Modified Weight-Int.</td>
<td>0.3814</td>
<td>0.7945</td>
<td>11.4462</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 8. Prediction performance after pruning the inputs of the 1-hour network from 26 to 18 using the four different input-ranking methods.

<table>
<thead>
<tr>
<th></th>
<th>NMSE.</th>
<th>Pearson Coeff.</th>
<th>AE (in degrees)</th>
<th># of hidden nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight-Interpretation</td>
<td>0.7274</td>
<td>0.5593</td>
<td>16.1083</td>
<td>21</td>
</tr>
<tr>
<td>Weight Derivative Method</td>
<td>0.6667</td>
<td>0.5867</td>
<td>15.3973</td>
<td>22</td>
</tr>
<tr>
<td>Input-Output Derivative Method</td>
<td>0.5603</td>
<td>0.6961</td>
<td>14.0487</td>
<td>27</td>
</tr>
<tr>
<td>Modified Weight-Int</td>
<td>0.5575</td>
<td>0.6668</td>
<td>13.7573</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 9. Prediction performance after pruning the inputs of the 2-hour network from 24 to 16 using the four different input-ranking methods.
### Table 10. Prediction performance after pruning the inputs of the 4-hour network from 18 to 13 using the four different input-ranking methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>NMSE</th>
<th>Pearson Coeff.</th>
<th>AE (in degrees)</th>
<th># of hidden nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input-Output</td>
<td>0.5168</td>
<td>0.7009</td>
<td>13.4032</td>
<td>14</td>
</tr>
<tr>
<td>Derivative Method</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modified</td>
<td>0.5365</td>
<td>0.6817</td>
<td>13.1648</td>
<td>5</td>
</tr>
<tr>
<td>Weight-Int.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 11. Prediction performance after pruning the inputs of the 6-hour network from 18 to 13 using the four different input-ranking methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>NMSE</th>
<th>Pearson Coeff.</th>
<th>AE (in degrees)</th>
<th># of hidden nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight-Interpretation</td>
<td>0.4219</td>
<td>0.7665</td>
<td>11.5357</td>
<td>18</td>
</tr>
<tr>
<td>Weight Derivative</td>
<td>0.4691</td>
<td>0.7484</td>
<td>12.1823</td>
<td>11</td>
</tr>
<tr>
<td>Method</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input-Output</td>
<td>0.3854</td>
<td>0.7849</td>
<td>11.4300</td>
<td>28</td>
</tr>
<tr>
<td>Derivative Method</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modified</td>
<td>0.4555</td>
<td>0.7388</td>
<td>12.5819</td>
<td>17</td>
</tr>
<tr>
<td>Weight-Int.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 12. Prediction performance after pruning the inputs of the 7-hour network from 18 to 13 using the four different input-ranking methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>NMSE</th>
<th>Pearson Coeff.</th>
<th>AE (in degrees)</th>
<th># of hidden nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight-Interpretation</td>
<td>0.5608</td>
<td>0.6982</td>
<td>13.6535</td>
<td>22</td>
</tr>
<tr>
<td>Weight Derivative</td>
<td>0.5864</td>
<td>0.6469</td>
<td>13.9920</td>
<td>15</td>
</tr>
<tr>
<td>Method</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input-Output</td>
<td>0.4893</td>
<td>0.7171</td>
<td>12.6635</td>
<td>7</td>
</tr>
<tr>
<td>Derivative Method</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modified</td>
<td>0.5764</td>
<td>0.6709</td>
<td>14.0705</td>
<td>6</td>
</tr>
<tr>
<td>Weight-Int.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

88
<table>
<thead>
<tr>
<th></th>
<th>NMSE.</th>
<th>Pearson Coeff.</th>
<th>AE (in degrees)</th>
<th># of hidden nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight-Interpretation</td>
<td>0.4440</td>
<td>0.7460</td>
<td>11.6443</td>
<td>19</td>
</tr>
<tr>
<td>Weight Derivative Method</td>
<td>0.3955</td>
<td>0.7788</td>
<td>11.2857</td>
<td>22</td>
</tr>
<tr>
<td>Input-Output Derivative Method</td>
<td>0.5137</td>
<td>0.7020</td>
<td>13.1864</td>
<td>14</td>
</tr>
<tr>
<td>Modified Weight-Int.</td>
<td>0.4705</td>
<td>0.7283</td>
<td>12.7044</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 13. Prediction performance after pruning the inputs of the 8-hour network from 18 to 13 using the four different input-ranking methods.

Before comparing the performance of the different ranking methods, it was desirable to see if pruning the inputs in the above manner actually led to a reduction, or minimal increase, in the prediction error of the new networks. One must remember that, besides trying to increase prediction accuracy, another goal was to reduce the overall number of free parameters (weights) in the network. Smaller network complexity can lead to better generalization [17] [18]. The figure below compares the performance of the pruned networks versus the original networks over different prediction horizons.
From the above graph, it is clear that a reduction in the number of inputs from 18 to 13 for the eight-hour network greatly enhanced prediction performance. The weight derivative method produced the best pruning results for the eight-hour network, reducing the NMSE from 0.5064 to 0.3955 and increasing the Pearson coefficient from 0.7129 to 0.7788. In this case, the weight derivative method outperformed the other three methods, with the original weight-interpretation method resulting in a new network with an NMSE of 0.4440.

However, input pruning does not introduce such drastic accuracy improvements for networks predicting at other horizons. The pruned one-hour network had an
accuracy close to that of the original larger network; the NMSE of the network chosen by input-output derivative analysis was 0.361, compared with 0.3331 for the original one-hour network. Pruning six-hour and seven-hour networks from an initial input size of 18 to 13 also produced new networks that are close (within 0.04 NMSE) in performance to the original networks. The four-hour network seemed to have the largest gap between pruned network performance and its original network performance.

The focus now shifts to a comparison of the different pruning algorithms. After this initial set of pruning attempts, no single ranking method stands out as being vastly superior to the others. While the weight derivative algorithm selected the best 13-input subset for the eight-hour network, networks trained on subsets chosen using this ranking algorithm were not the best performers for other prediction horizons. The input-output derivative method performed best for six and seven-hour networks, while the modified weight-interpretation method performed best for the two-hour network. The original weight-interpretation network was the best performer for the four-hour network.

From experience, it seems that the number of hidden nodes in the original network affects the performance of the weight derivative method relative to the other methods. When looking at the initial phase of pruning, the weight derivative method outperformed all others when the number of hidden nodes in the original network was less than 20. When the number of hidden nodes was equal to 24 or higher, the weight derivative method performed relatively poorly. This was also true in some of the group's initial work with the pruning algorithm. Generally, it seems that the effectiveness of the weight derivative method deteriorates as the original network's hidden layer size increases.
However, this phenomenon only seems to be true when the first, larger input subsets were selected. When the size of the original input space was pruned to 8, the eight-hour network trained on the subset chosen by the weight-derivative method performed extremely poorly, with an NMSE of 0.9602, only performing better than the modified weight-interpretation method. This result will be discussed further in the next subsection.

9.2 Narrowing Down the Variable Set Further: Control Parameters

As mentioned earlier, one of the reasons for performing this saliency analysis is to aid in selection of the optimal input parameters for controlling HMT. Therefore, one would like to narrow down the parameters even further from 18 or 13. This was accomplished by selecting the top 8 most salient variables suggested by the different ranking methods and comparing their performance after retraining networks using the suggested 8 variables. The reason why an even smaller set of variables was not chosen is that the models would have been severely under-defined.

The performance of the different ranking techniques was evaluated by training new networks using only the 8 most salient variables selected by each method. The resulting NMSE, AE and Pearson coefficients are listed below:

<table>
<thead>
<tr>
<th></th>
<th>NMSE</th>
<th>Pearson Coeff.</th>
<th>AE (in degrees)</th>
<th># of hidden nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight Interpretation</td>
<td>0.5019</td>
<td>0.7082</td>
<td>12.5087</td>
<td>15</td>
</tr>
<tr>
<td>Weight Derivative Method</td>
<td>0.6588</td>
<td>0.5950</td>
<td>14.9916</td>
<td>16</td>
</tr>
<tr>
<td>Input-Output</td>
<td>0.5562</td>
<td>0.6663</td>
<td>13.9432</td>
<td>15</td>
</tr>
<tr>
<td>Derivative Method</td>
<td>NMSE.</td>
<td>Pearson Coeff.</td>
<td>AE (in degrees)</td>
<td># of hidden nodes</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------</td>
<td>----------------</td>
<td>-----------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>Modified Weight-Int.</td>
<td>0.5411</td>
<td>0.6814</td>
<td>13.4275</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 14. Prediction performance after pruning the inputs of the 4-hour network from 18 to 8 using the four different input-ranking methods.

<table>
<thead>
<tr>
<th></th>
<th>NMSE.</th>
<th>Pearson Coeff.</th>
<th>AE (in degrees)</th>
<th># of hidden nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight-Interpretation</td>
<td>0.6785</td>
<td>0.5676</td>
<td>15.0849</td>
<td>9</td>
</tr>
<tr>
<td>Weight Derivative Method</td>
<td>0.6228</td>
<td>0.6175</td>
<td>14.5997</td>
<td>9</td>
</tr>
<tr>
<td>Input-Output Derivative Method</td>
<td>0.5487</td>
<td>0.6756</td>
<td>13.4739</td>
<td>20</td>
</tr>
<tr>
<td>Modified Weight-Int.</td>
<td>0.6936</td>
<td>0.5582</td>
<td>15.4459</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 15. Prediction performance after pruning the inputs of the 5-hour network from 18 to 8 using the four different input-ranking methods.

<table>
<thead>
<tr>
<th></th>
<th>NMSE.</th>
<th>Pearson Coeff.</th>
<th>AE (in degrees)</th>
<th># of hidden nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight-Interpretation</td>
<td>0.4739</td>
<td>0.7281</td>
<td>12.8388</td>
<td>9</td>
</tr>
<tr>
<td>Weight Derivative Method</td>
<td>0.7150</td>
<td>0.5340</td>
<td>15.9641</td>
<td>20</td>
</tr>
<tr>
<td>Input-Output Derivative Method</td>
<td>0.7329</td>
<td>0.5324</td>
<td>15.9615</td>
<td>11</td>
</tr>
<tr>
<td>Modified Weight-Int.</td>
<td>0.7874</td>
<td>0.4746</td>
<td>17.1994</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 16. Prediction performance after pruning the inputs of the 6-hour network from 18 to 8 using the four different input-ranking methods.
Table 17. Prediction performance after pruning the inputs of the 7-hour network from 18 to 8 using the four different input-ranking methods.

<table>
<thead>
<tr>
<th></th>
<th>NMSE</th>
<th>Pearson Coeff.</th>
<th>AE (in degrees)</th>
<th># of hidden nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight-Interpretation</td>
<td>0.5996</td>
<td>0.6656</td>
<td>14.3754</td>
<td>12</td>
</tr>
<tr>
<td>Weight Derivative Method</td>
<td>0.6716</td>
<td>0.5855</td>
<td>15.0452</td>
<td>19</td>
</tr>
<tr>
<td>Input-Output Derivative Method</td>
<td>0.5916</td>
<td>0.6607</td>
<td>14.1404</td>
<td>12</td>
</tr>
<tr>
<td>Modified Weight-Int.</td>
<td>0.7516</td>
<td>0.5275</td>
<td>16.0717</td>
<td>12</td>
</tr>
</tbody>
</table>

Since the goal was to compare the relative accuracy of the different ranking methods, the graph below presents a summary of performance, in terms of NMSE, of the different pruning methods for input horizons ranging from four to eight hours:
What is interesting is the relative drop in performance of the weight derivative method when its 8 most salient inputs are used for prediction, rather its 13 most salient inputs. As one may recall from the previous subsection, the eight-hour network pruned to 13 input variables using the weight derivative method outperformed all eight-hour networks pruned using other methods, and even led to a decrease in overall NMSE. However, when a network is trained using the weight derivative method’s 8 most salient inputs to predict HMT, its performance is far worse than the network of the best performing ranking method. This means that the weight derivative method is capable of selecting a relatively large set of important variables, but, when asked to select what it thinks are the few most important variables, it performs poorly.
In the 8-input pruning scenario, there seemed to be clearer "winners" in terms of prediction performance over all horizons. By looking at Figure 31, one can see that for four, six and eight hour horizons, the original weight-interpretation algorithm chose input subsets that led to the best performing new networks. Even for seven-hour prediction, the weight-interpretation method was extremely close in performance to the best ranking method. In the cases of five-hour and seven-hour predictions, the input-output derivative method was the best performing method.

The two modified algorithms (modified weight-interpretation and weight derivative analysis) seem to perform in a relatively similar fashion. Each performs well only in certain scenarios, and is not consistently strong when compared to the other two methods. One possible reason for this result is that the transfer function used for penalizing negative weight and weight derivatives in (8.8) may skew the relative relationship between the weight saliencies of the network. If many of the network weights in the original network are positive, the benefits of applying this penalizing function may be nullified or even overshadowed by the skewing effects of using the logistic sigmoid in order to transform the weights.

Tables 19 and 20 below summarize the best performing methods for both pruning phases presented in this paper:

<table>
<thead>
<tr>
<th>Prediction Horizon</th>
<th>Best Performing Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 hour</td>
<td>Original Weight-Interpretation</td>
</tr>
<tr>
<td>2 hours</td>
<td>Modified Weight-Interpretation</td>
</tr>
<tr>
<td>4 hours</td>
<td>Original Weight-Interpretation</td>
</tr>
<tr>
<td>6 hours</td>
<td>Input-Output Derivative method</td>
</tr>
</tbody>
</table>
From the above charts, it seems that the most reliable pruning method was the original weight-interpretation method, although there were prediction horizons when other methods performed better. Since each prediction horizon may be characterized by different dynamics in terms of input-output lags, it is not surprising that different methods had varying levels of accuracy over the spectrum of prediction horizons.

By experience, it seems that the weight derivative method developed in this paper works well for relatively smaller networks in terms of number of hidden nodes. Perhaps this is due to the fact that the weights of smaller networks more readily converge to minima in the error space as opposed to relatively larger networks.
9.3 Further Work

Unfortunately, the weight derivative method did not produce the kind of consistently strong results over all prediction horizons that the group had hoped for. Although it was the only method to actually narrow down the input space of a neural network while \textit{substantially increasing} its prediction accuracy, future efforts must be focused on improving its overall performance. Clearly, a major problem with this method in its current form is that it assumes that the cross derivative terms of the Hessian are zero. Although this assumption was made in order to reduce computational complexity, it may not be all that accurate. Even though the removal of a particular weight may cause an increase in the error, the removal of the weight might also cause a change in the values of the other weights in the network, as they try to compensate for the loss of the first weight. This means that the real change in the error resulting from a removal of the weight may not be as large as is reflected by the current method. Therefore, one possible improvement to the weight derivative ranking technique would be to incorporate the dynamic weight and corresponding error changes calculated by Hassibi and Stork [43] in the \textit{Optimal Brain Surgeon} method.

It is clear that the size one chooses for the pruned input set is important in determining whether prediction performance rises, falls or remains the same. In this paper, the choices of input subset sizes were based on previous experience. However, one could apply more rigorous techniques. The largest gap method (LGM) is one technique that has been suggested in the literature [33] [34] for use with input-output sensitivity (derivative) analysis.
Assume the input variables are ranked based on a measure of saliency $S_i$, where $i$ denotes the input variable for which the saliency is measured. One way to perform variable selection is to sort all variables according to their saliency values, and find a cutoff variable past which all lower ranking variables are eliminated from consideration. The largest gap method, mentioned in [33] and [34], can be stated as follows:

1. Sort all variables bases on saliency and number them, with 1 being the most salient and $n$ (the number of variables) being the least important.
2. Start at variable $i=2$.
   For each $i$:
   a. calculate $\phi_i = \frac{S_{i-1}}{S_i}$
   b. If $\phi_i < \phi_{\text{threshold}}$, then only retain variables numbered 1 through $i$, discarding all variables ranked below. If condition not satisfied, then move on to the next variable $i+1$ and go back to Step 2.

Figure 32. The largest gap method

The value $\phi_{\text{threshold}}$ is the smallest ratio of two consecutive variable saliencies past which all other variables are discarded as redundant. The lower the user sets $\phi_{\text{threshold}}$, the more stringent the requirements are for discarding the variables after variable $i$. Therefore, it is important to set $\phi_{\text{threshold}}$ to be small enough so that the requirements
for discarding variables are not too weak, while also choosing $\phi_{\text{threshold}}$ to be large enough so that some variables are dropped.

The selection of $\phi_{\text{threshold}}$ is quite difficult in the current problem for most of the ranking methods mentioned. In some cases, the largest drop in saliency occurs between the most important and second most important variables. Using LGM in this scenario would leave only one input in the pruned network. Therefore another method worth pursuing, for some ranking methods, could be one that the research team has derived - the fraction of maximum method. According to this method, a cutoff variable is chosen based on $S_1 \cdot \text{frac}$, where $\text{frac}$ is a fraction between 0 and 1, and $S_1$ is the saliency measurement of the most important variable. Variables are dropped only if their saliencies fall below a fraction of the maximum input saliency. The lower the value of $\text{frac}$, the stricter the conditions are for removal of inputs.

Further work can attempt to apply these two selection algorithms to the available rankings and determine suitable threshold values. The work of this current paper assumes that the size of pruned input sets for all methods, when applied to the same original network, are equal. This assumption is made merely for comparison purposes. When one is trying to find the optimal input subset choice, the above assumption may not be true.

In addition, one should apply these input saliency techniques to networks that have been trained using more than two random runs. The more random runs one employs while training, the more confident one can be regarding the validity of the resulting neural network and its results. In the current example, only two randomized runs were performed for training and testing due to time and computation constraints.
As a final suggestion, it would also be useful to validate ranking results with domain experts when examining the relative merits of the different rankings.
Section 10: Comparison with Other Work by Data Mining Research Group

The Data Mining Research Group at MIT has been involved in various projects relating to the prediction and control of hot metal temperature at Steelcorp. While the current paper focuses on the use of MLP networks for prediction and input saliency determination, other members of the research group have examined the ability of time-delay neural networks (TDNNs) [54] [55], classification and regression trees [46], genetic algorithms [56] and neural network inversion [57] to predict and/or control HMT. This section provides highlights of these ongoing projects as well as a comparison of their preliminary results with those of the current work.

Leonida [54] and Purville [55] applied time-delay neural networks (TDNNs) as a means of modeling the blast furnace and predicting hourly HMT. As described in Section 3, TDNNs are useful because they are able to automatically capture the time lags that exist between each input and the output. For example, assume that a TDNN is constructed to predict HMT (t), initially using the current values of all its inputs. If the delay size of the input layer is set to 3, this means that three different versions of each input are fed as separate inputs into the neural network; the values of each variable at times t, t-1 and t-2 are all considered inputs into the network [1]. The TDNN then employs a learning algorithm, such as back-propagation, to search for the optimal values of the network’s weights. In doing so, the network weighs different time-delayed inputs according to their importance in predicting the output; inappropriate time-delays for different inputs will be assigned relatively smaller connection weights. Unlike the MLP network case, the selection of the appropriate input delays is left to the TDNN rather than the individual constructing the network.
Leonida [54] and Purville [55] used TDNNs in order to forecast the HMT values that are predicted in the current paper. However, rather than setting input-output lags to those specified by domain experts, the TDNNs automatically determined the most relevant time delays. [54] and [55] began by using the moving window averaged data described in Sections 5 and 6. Although initial results were promising, Leonida [54] discovered a disturbing trend that invalidated most of their moving window results. For networks predicting at relatively short time horizons (one and two hours ahead), errors on the testing data (last 20% of data) were smaller than errors on the training data (first 80% of data). This indicated an undesirable and confusing situation as the trained networks performed worse on the data for which they were optimized to predict (the training set). TDNNs predicting three hours and beyond did not exhibit this trend.

After some analysis, Leonida [54] ascribed this training/testing error anomaly to moving window averaging of the data. As described in Section 6, consecutive data points in a moving window averaged data set contain a significant amount of overlap of information (in the current example, the overlap was forty minutes worth of information). Therefore, when predicting one hour into the future and using the last known HMT value as an input, the network was “cheating” because it was explicitly using future knowledge in order to predict future HMT [54]. However, this problem was only present for networks predicting HMT averages one and two hours into the future; once the prediction horizon was larger than two hours, the last known HMT value did not include explicit information about future HMT. Therefore, networks predicting three hours or more into the future using moving window data did not exhibit such strange error phenomenon. Leonida [54] confirmed this reasoning by dropping the
last known HMT value as an input and retraining the TDNNs. In this case, predictions of one-hour and two-hour HMT did not possess this training/testing error anomaly.

Therefore, the work of Leonida [54] confirmed the general result of the current paper that moving window averaging may not be a desirable data processing solution. However, the current work involving MLP network training with moving window averaged data did not show such training and testing error problems as exhibited in [54] and [55]. This might have been the case because the MLP networks did not rely as heavily on the last known HMT value as the TDNNs did for HMT prediction one and two hours into the future.

As such, results derived from direct hourly averaged HMT data provide the best means of comparing MLP and TDNN performance. Leonida [54] and Purville [55] performed TDNN predictions for horizons ranging from one to four hours; a comparison of their results with those of the current paper is outlined in Figure 33 below:

![NMSE Performance of MLP Networks vs. TDNNs using Hourly Averaged HMT Data](image)

Figure 33. NMSE comparison of MLP networks vs. TDNNs using hourly averaged HMT data.
It is clear from the above graph that the MLP networks performed better than their time-delay counterparts for all prediction horizons. What is also obvious is that the performance of the MLP networks did not fall as sharply as TDNN performance when the prediction horizon was increased. In fact, the quality of MLP predictions remained relatively constant over different prediction horizons, even increasing in performance when the horizon was moved from two to four hours. As described in Section 6 when comparing the current MLP networks to those of Brad Banks [8], the absence of forecast accuracy deterioration as the prediction horizon was increased signifies that the MLP networks were better able to understand the real relationships between the inputs and HMT. This is because the MLP networks did not rely as greatly on the last known HMT value as did the TDNNs when predicting future HMT. This has resulted in more robust MLP networks.

One interesting finding was the ability of Purville [55] to effectively reduce the input space for TDNN models from 18 to 11 with increased HMT prediction accuracy for one, two and three hour predictions. Purville employed a simple method of removing the seven input variables that were least correlated with the output. However, the pruning results in [54] should be viewed with cautious optimism. Since the TDNNs seem to have placed significant dependence on the last known HMT value when predicting future HMT, the network may have regarded other inputs (especially in the one, two and three-hour prediction cases) as unimportant and irrelevant in predicting HMT. Hence, although the input variable space was reduced from 18 to 11, this reduction might have been accomplished at the cost of placing an even greater dependence on the last known HMT value as an indicator of future HMT. On the other
hand, the input saliency work in the current paper was able to increase the prediction accuracy of eight-hour HMT networks by 10% while pruning the input space from 18 to 13. The fact that input pruning performed best for eight-hour prediction in the current paper demonstrates that the pruning algorithms presented here did not increase the dependence on last known HMT for prediction (this is because, in general, the last known HMT value provides little help in predicting HMT eight hours into the future).

Mu’taz Qubbaj [46] adopted a different approach to blast furnace modeling by using decision tree analysis to classify and predict hot metal temperature based on ranges of different input values. The ultimate goal in building decision trees was to explore the construction of a tree-based HMT control system. A decision tree is often used to classify a data point into a certain output category based on the values of the data point’s inputs [47]. Each node in the tree corresponds to a test of a specific input variable; the different results of the test lead to different nodes with variable tests of their own. An output classification is obtained once a leaf node is reached. The idea behind using such trees for control is that, once the tree is built, one will be able to start at a desired leaf node (HMT category or level) and trace back up the tree to the root [47]. The path traveled would constitute a valid input combination (control strategy) in order to reach the desired HMT category or level. Since there might be different paths connecting the desired leaf node to the root, Qubbaj outlined a basic strategy for incorporating outside constraints, such as input costs, in order to select the optimal control path.

Qubbaj used two techniques in order to build his trees. The first, See5 [56], was used to construct a decision tree classifier. Since See5 is a classification tree, the output
HMT values were first divided into 11 equally spaced bands, where each band spanned an HMT interval of 10 degrees Celsius. The tree constructed by See5 was then used to derive rules for HMT classification based on the input values. The other technique used, defined in the software program Cubist [56], was a multivariate piece-wise regression model. Cubist was used to build a set of rules based on the input values. Each rule was associated with a corresponding linear regression based on the different inputs to the system. This technique was used to build a continuous variable HMT predictor.

Since the work of the current paper focuses on prediction rather than classification of HMT, the most appropriate comparison involves the results of the current work and that of the Cubist in [46]. While the work of this paper looked at prediction of HMT in the range of one to eight hours in the future, Qubbaj used decision tree analysis to predict current HMT [46]. As such, Qubbaj did not make use of the last known HMT value as an input into the model. In addition, there was no explicit division of the original dataset into training and testing sets as was done in the current work. Prediction performance was thus evaluated by applying the constructed tree to the data in the training set. This may have presented a biased view of the tree's overall performance [18] because model evaluation was performed on the same data for which the tree was optimized (this is why models are usually evaluated on separate hold-out data, referred to as the testing or validation set). Finally, [46] used moving window averaged data instead of direct hourly averaged data when building its trees.

The initial Cubist results in [46] were fairly impressive. While the average prediction error was quite high (6.2840% for Cubist vs. 0.5-1.5% for MLP network predictions), the Pearson coefficient between actual and predicted values was 0.92 (as
compared to 0.86-0.72 for MLP networks using moving window data). However, as mentioned above, the fact that this performance was measured on the training data instead of on separate validation data indicates that the results in [46] are probably biased. As a next step, one should reevaluate the performance of the regression tree on new validation data. Once this is done, domain experts should be consulted in order to develop control strategies that incorporate information derived both from the classification/regression trees and domain specific input constraints.

Ashima Dua [57] applied a technique known as neural network inversion to a feed-forward network trained in the current paper. Dua was able to utilize a method that involves non-linear programming [26] in order to reverse the neural network mapping. The technique provides one of many feasible input combinations that could produce a given desired HMT output. The method that she had suggested (which was presented in [26]) finds a suitable input combination for the given HMT target, while applying a constraint that minimizes the change in the inputs from their current values. In a control sense, the goal was to suggest a strategy to arrive at a desired HMT level that would also minimize the input change required to get there.

This method was implemented in MATLAB and applied to the four-hour MLP network trained in the current paper using direct hourly averaged data. Validation of the inversion technique was performed by first specifying both the desired normalized output HMT value and the current normalized values of the inputs. Inversion of the four-hour network was then performed in order to determine the new set of input values necessary to obtain the desired output. Finally, this set of new inputs was fed into the network and the absolute value of the difference between the network’s output and the
desired HMT level was calculated. This was noted as the inversion error. Although promising, errors encountered were still quite high, ranging from 0.1697 to 0.1539 (using normalized HMT values) for the examples presented.

As a future improvement, one could use the saliency techniques presented in the current paper in order to narrow down the number of inputs for which the inversion is calculated. For example, one could hold a set of less salient variables constant, while calculating new values only for the group of input variables deemed to be most influential in effecting HMT. This reduction in the number of input values that must be computed would significantly reduce the complexity of the non-linear programming problem that has to be solved [26].

Danny Lai [58] applied genetic algorithms in order to develop a control system that returns a set of input values corresponding to a desired HMT value. The genomes were represented as matrices, where each column corresponded to a desired output HMT change while each row corresponded to a specific input. Thus, each column could be considered a gene and the matrix as a whole would be the genome. The function used to evaluate the fitness of a particular gene was the negative of the squared difference between the desired HMT value and the result of the four-hour network (derived in Section 6) evaluated at the set of input values encoded by the gene in question. In order to limit the size of the genome and make the control problem easier to solve, Lai reduced the number of input values from eighteen to fourteen by selecting the fourteen input variables that were most correlated with HMT. Genetic selection features such as crossover and mutation were also implemented in the evolution process.
Unfortunately, initial results were not very promising. In particular, the genomes were unable to converge to configurations with decent fitness values. Lai concluded that the set of inputs chosen using input correlation analysis might not have been the most important in determining HMT. As a suggestion for further work, the methods in [58] could be applied using the top fourteen input variables chosen by any of the four measures of saliency presented in the current paper.

The data mining research group at MIT has also been involved in predicting other variables of interest to Steelcorp, such as the silicon content and slag basicity of hot metal produced in the blast furnace [59]. Dhond [59] was able to obtain promising results for silicon content prediction up to four hours into the future, with mean square errors of prediction ranging from 0.004 to 0.020. She utilized MLP networks using one and two hidden layers. However, as in the case of Banks [8], the dataset used contained many missing output values and was therefore preprocessed using linear interpolation of the output values [8] [59]. This technique led to the creation of unrealistic blast furnace data, and, again, an overwhelming dependence on the last known output value in order to predict future silicon content. The dataset for slag basicity was even worse, leading to even more of a reliance on linear interpolation. Regardless of the data's shortcomings, however, it was clear that, at least in the case of hot metal silicon content, neural networks were able to predict blast furnace operations well. The results in [59] for silicon content mirror those presented for HMT prediction using linearly interpolated data in [8].

As a final note, Benjamin Fang [60] applied one and two-hidden layer MLP networks for the prediction of RDI and +10, parameters associated with sinter plant
operations. The data used consisted of daily RDI and +10 values. Unfortunately, weeks worth of data in the middle of each data set were missing, making it extremely difficult to use the last known RDI or +10 value as inputs to the networks. Hence, the resulting prediction MSEs were high, ranging from 0.021 for one-day +10 prediction to 0.059 for two-day RDI prediction [60]. The research group will soon try to make use of a new data system at Steelcorp’s sinter plant in order to obtain more consistent RDI and +10 data.
Section 11: Conclusions

This paper examines the ability of feed-forward neural networks to predict the hot metal temperature of pig iron based on 17 input parameters that reflect the current and past conditions of the blast furnace. Along the way, various real world data challenges and solutions are presented, as well as a neural network training suite that automatically finds the best-performing neural multi-layer perceptron (MLP) network. The paper then analyzes the importance of the size of the network as well and the network’s input-output lags in determining prediction performance. Prediction accuracy depends on the number of hidden nodes and layers, as well as on whether one uses automatic lag detection or manually sets input variable lags based on domain knowledge. The relationship between prediction accuracy and prediction horizon is also discussed. In addition, the merit of moving window averaging is discussed and its results are compared with those of direct hourly averaging.

The prediction results presented in this paper are very promising. The substitution of moving window data with direct hourly averaged data presents a more realistic picture of the blast furnace to the network during training. With the introduction of this new training and testing data, networks predicting four, seven and eight hours perform better than those trained using moving window averaged data. Networks trained using direct hourly averaged data also tend to produce networks with smaller numbers of hidden nodes, leading to less complexity and a possible increase in generalization.

A comparison of prediction results from ANNs and results from linear regression analysis clearly indicates that neural network techniques are far better at
capturing the complex, dynamic and non-linear relationships of blast furnace processes. While MLP networks using back-propagation yield Pearson coefficients ranging from 0.8333 to 0.7129, linear techniques deliver predictions with Pearson values between 0.7311 and -0.0819. Pearson values close to zero indicate that for horizons greater than two hours, linear regression analysis is incapable of predicting HMT trends with any sort of accuracy.

This paper extends the previous work at Steelcorp by predicting HMT on an hourly basis. The ability to predict hourly averaged HMT up to eight hours into the future, rather than predicting daily averaged HMT values alone, provides useful information to blast furnace operators who wish to control HMT within a relatively short time horizon. Finally, when comparing the work here with previous work by the group, it is clear that the new networks are more able to capture the real relationships between the input variables and HMT. Rather than relying heavily on the last known HMT value when predicting HMT, these networks try to find the actual variable relationships present in the data. This is reflected in the fact that network prediction performance in the current paper does not deteriorate as significantly as the performance of networks in [8] when the prediction horizon is increased from one to eight hours.

The second part of this paper focuses on methods for ranking input variables based on their relative importance in predicting the output. Given the optimally trained neural networks, the goal is to rank the input variables based on importance in effecting the output HMT. This exercise is useful, firstly, as a means of eliminating redundant or noisy inputs that reduce the accuracy of the current predictions. Secondly, in terms of
control, one would like to narrow down the choice of control parameters significantly. Since techniques such as the use of neuro-controllers and neural network inversion require a small set of inputs to be used as control parameters [11] [40], it is necessary to select the inputs that affect the output the most.

Therefore, four methods of saliency determination are compared. Two of them, sensitivity derivative analysis and weight-interpretation, have been presented earlier in literature [31] [33] [34] [35] and are applied here. This paper also presents several novel modifications of existing algorithms and compares their performance to those already found in literature. One of these new techniques, referred to as the weight derivative method, is a hybridization of the weight-interpretation method and the use of the Hessian matrix to determine weight saliency.

The paper's initial results demonstrate that for eight-hour prediction, the weight derivative method produces an optimal reduction in the size of the input space from 18 to 13. This reduction in input size is also characterized by a non-trivial increase in performance. In particular, the application of this new saliency measure to input pruning results in an 8-hour network that predicts almost 10% better than the original, larger network. Since eight-hour predictions are by far the most challenging to carry out, a performance increase coupled with an input reduction is highly significant in this case. Unfortunately, for shorter prediction horizons, the weight derivative method does not perform as well. The general trend seems to be that the weight derivative method performs worse on networks with a relatively large number of hidden nodes.

When the sizes of the pruned input sets are narrowed down to 8, the weight derivative method deteriorates significantly. This means that while the weight
derivative method is capable of selecting relatively larger input subsets, it still has trouble ranking the few most important variables. In this case, initial results show that the original weight-interpretation analysis and the input-output derivative method are more consistently accurate. Further work is proposed in order to improve the performance of the two new saliency methods presented here.

As a final note, this paper presents ongoing work related to Steelcorp, performed by other members of the data mining research group, as a means of comparing HMT prediction results from other techniques with those presented in the current paper. In particular, the MLP networks derived here outperform their time-delay counterparts [54] [55] for all prediction horizons surveyed. In addition, prediction accuracy remains robust over increasing prediction horizons when compared to TDNN predictions. The MLP network is also compared to a decision tree technique, Cubist [46] [56], which can be thought of as a piece-wise linear regression that is constructed using rule sets. Although the results of the tree analysis are promising and seem to match or even exceed the MLP’s prediction accuracy for very small prediction horizons, a more thorough evaluation of the regression tree on new testing data is essential for model validation and comparison.

This paper also explores the impact of using MLP networks derived in the current work as part of different HMT control techniques. Dua [57] applies non-linear programming methods [26] to invert the four-hour MLP network presented in Section 6, while Lai [58] incorporates the same MLP network into a fitness function for a genetic algorithm-based HMT controller. Results from these two applications indicate that more work in this area must be performed.
Section 10 ends by discussing the application of MLP networks to prediction of other steel-making parameters at Steelcorp, such as hot metal silicon content [59], slag basicity [59] and sinter plant parameters (RDI and +10) [60]. Despite significant data shortcomings, MLP networks demonstrate the ability to forecast these variables with reasonable degrees of accuracy. In particular, hourly hot metal silicon content can be predicted at a very high level of accuracy, nearly matching the quality of results obtained from HMT prediction.

It is clear that the application of neural network data mining has been able to accurately model blast furnace processes and yield the kind of prediction results that prior work [9] [14] [16] [48] [49] had claimed would be extremely difficult to obtain. Besides providing insights into the key techniques necessary to accomplish this successful modeling, this paper extends the work of neural networks even further to help determine the key input variables that could be used for controlling HMT in the blast furnace environment. As mentioned above, some of the ongoing work within the data mining research group is directed at using different machine-learning techniques for the purposes of controlling HMT. A logical next step in this research would be to apply these control techniques, making use of the neural networks developed in this paper as the “plant” in the control loop, as well as using the subset of optimal variables selected here as the parameters of control.
Section 12: References


Appendix 1: Derivation of the Hessian Matrix

The following is a derivation of the diagonal terms in the Hessian matrix. The methods used here are similar in procedure to Chris Bishop’s formulation [42].

One first realizes that in a single hidden layer network, there are weight matrices, \( W(1) \), representing all weights between the input and hidden layer, and \( W(2) \), comprising of all weights between the hidden and output layer. Therefore, the derivation is broken up into two parts, derivation of the diagonal Hessian term for a weight in \( W(2) \), and separately, for a weight in \( W(1) \).

For convenience purposes, the notation introduced earlier to derive the input-output derivatives is presented:

- \( X_i = \) the input to neuron i at the input layer
- \( X_i(j) = \) the total input to neuron i in layer j
- \( Y_i(j) = \) the output of node i in layer j
- \( W_{ij}(k) = \) the weight from node i in layer k to node j in layer k+1
- \( Y_j(1) = X_i \)
- \( X_i(j) = \sum_{k=1}^{n(j-1)} W_{ij}(j-1) Y_k(j-1) + B_i(j) \)
- \( Y_i = f(X_i(j)) \) where f is the transfer function of node i in layer j
- Input layer is layer 1, hidden layer is layer 2 and output layer is layer 3

Beginning with a weight \( W_{lm}(2) \), the objective is to compute \( \frac{\partial^2 E}{\partial W_{lm}^2} \), a diagonal term in the Hessian matrix \( H_{ij} \). Note that l is a node in layer 2 and m is the output layer.
node (since there is only one output, \( m = 1 \)). The derivation begins by applying the chain rule in the following manner:

\[
\frac{\partial^2 E}{\partial W_{lm}(2)} = \frac{\partial}{\partial W_{lm}(2)} \left( \frac{\partial E}{\partial W_{lm}(2)} \right)
\]

\[
= \frac{\partial X_m(3)}{\partial W_{lm}(2)} \cdot \frac{\partial}{\partial X_m(3)} \left( \frac{\partial E}{\partial W_{lm}(2)} \right) \quad (A1.1)
\]

\( X_m \) is the total input into the output node \( m \). Therefore \( \frac{\partial X_m(3)}{\partial W_{lm}(2)} = Y_l(2) \), the output of node \( l \).

\[
\therefore \frac{\partial^2 E}{\partial W_{lm}(2)} = Y_l(2) \cdot \frac{\partial}{\partial X_m(3)} \left( \frac{\partial E}{\partial W_{lm}(2)} \right) \quad (A1.2)
\]

The inner derivative can be expanded again making use of the chain rule:

\[
\therefore \frac{\partial^2 E}{\partial W_{lm}(2)} = Y_l(2) \cdot \frac{\partial}{\partial X_m(3)} \left( \frac{\partial X_m(3)}{\partial W_{lm}(2)} \cdot \frac{\partial E}{\partial X_m(3)} \right)
\]

\[
= Y_l(2) \cdot \frac{\partial}{\partial X_m(3)} \left( Y_l(2) \cdot \frac{\partial E}{\partial X_m(3)} \right) \quad (A1.3)
\]

\[
= Y_l(2) \cdot \frac{\partial}{\partial X_m(3)} \left( Y_l(2) \cdot \frac{\partial E}{\partial X_m(3)} \right)
\]

By applying the product rule, the following results:

\[
\frac{\partial^2 E}{\partial W_{lm}(2)} = Y_l(2) \cdot \frac{\partial Y_l(2)}{\partial X_m(3)} \cdot \frac{\partial E}{\partial X_m(3)} + (Y_l(2))^2 \cdot \frac{\partial}{\partial X_m(3)} \left( \frac{\partial E}{\partial X_m(3)} \right) \quad (A1.4)
\]
However, by noting that the input into output node $m$ does not change the behavior of the output of hidden node $l$, it is clear that $\frac{\partial Y_i(2)}{\partial X_m(3)} = 0$. Therefore, the entire left hand part of the right-hand sum vanishes and one is left with equation (A1.5).

$$\frac{\partial^2 E}{\partial W_{lm}^2 (2)} = (Y_i(2))^2 \cdot \frac{\partial}{\partial X_m(3)} \left( \frac{\partial E}{\partial X_m(3)} \right) \quad (A1.5)$$

If one defines $b_{gh}(g, h) = \frac{\partial}{\partial X_g(g)} \left( \frac{\partial E}{\partial X_f(h)} \right)$, then (A1.5) becomes:

$$\frac{\partial^2 E}{\partial W_{lm}^2 (2)} = (Y_i(2))^2 \cdot b_{mm}(3, 3) \quad (A1.6)$$

The derivation of $b_{fg}(h, i)$ will be elaborated after the derivative analysis of weights in $W(l)$.

Using the above methodology, and given the currently defined variables, a derivation of $\frac{\partial^2 E}{\partial W_{ij}^2 (1)}$ should be within reach, where $i$ is a node in the input layer and $j$ is a node in the hidden layer. Here is the derivation:

$$\frac{\partial^2 E}{\partial W_{ij}^2 (1)} = \frac{\partial}{\partial W_{ij} (1)} \left( \frac{\partial E}{\partial W_{ij} (1)} \right)$$

$$= \frac{\partial X_j(2)}{\partial W_{ij} (1)} \cdot \frac{\partial}{\partial X_j(2)} \left( \frac{\partial E}{\partial W_{ij} (1)} \right)$$

$$= Y_i(1) \cdot \frac{\partial}{\partial X_j(2)} \left( \frac{\partial E}{\partial W_{ij} (1)} \right)$$
Thus, the diagonal Hessian terms for weights in $W(1)$ and $W(2)$ have been derived in terms of $b_{fg}(h,i)$, the second error derivative in terms of node total inputs. $b_{j,j}(2,2)$ and $b_{mm}(3,3)$ are now derived in order to complete the discussion:

$$b_{mm}(3,3) = \frac{\partial}{\partial X_m(3)} \left( \frac{\partial E}{\partial Y_m(3)} \right)$$

$$= \frac{\partial}{\partial X_m(3)} \left( \frac{\partial Y_m(3)}{\partial X_m(3)} \cdot \frac{\partial E}{\partial Y_m(3)} \right)$$

$$= \frac{\partial}{\partial X_m(3)} \left( f'(X_m(3)) \cdot \frac{\partial E}{\partial Y_m(3)} \right)$$

$$= f''(X_m(3)) \cdot \frac{\partial E}{\partial Y_m(3)} + f'(X_m(3)) \cdot \frac{\partial}{\partial X_m(3)} \left( \frac{\partial E}{\partial Y_m(3)} \right)$$
(A1.8) specifies the second derivative of the error with respect to the total input into the output node \( m \). Since, in the current case, there is one output node, \( m=1 \). Since the error function used during training was the mean squared error, the error function for a specific data point, \( n \), equals:

\[
E^n = \frac{1}{N} (Y^n_m(3) - Y_{\text{actual}}^n)^2
\]

where \( Y_{\text{actual}}^n \) is a scalar representing the actual output of the \( n \)th datapoint.

Thus, if one assumes that the current discussion refers to the error with regard to a specific pattern in the dataset, then \( E = E^n \) and:

\[
\frac{\partial E}{\partial Y_m(3)} = \frac{2}{N} (Y_m(3) - Y_{\text{actual}}^n), \quad \frac{\partial^2 E}{\partial Y_m^2(3)} = \frac{2}{N}
\]

(A1.9)

One must keep in mind that all of the calculations must be performed iteratively for every pattern in the data set, and then added to find the total Hessian values.

The remaining problem to solve is to calculate \( b_{ji}(2,2) \). Here is the derivation:

\[
b_{ji}(2,2) = \frac{\partial}{\partial X_j(2)} \left( \frac{\partial E}{\partial X_j(2)} \right)
\]

\[
= \frac{\partial}{\partial X_j(2)} \left( \frac{\partial Y_j(2)}{\partial X_j(2)} \cdot \frac{\partial E}{\partial Y_j(2)} \cdot \frac{\partial E}{\partial X_m(3)} \right)
\]

\[
= \frac{\partial}{\partial X_j(2)} \left( f'(X_j(2)) \cdot W_{jm} \cdot \frac{\partial E}{\partial X_m(3)} \right)
\]
\[ W_{jm} \cdot \left[ f''(X_j(2)) \cdot \frac{\partial E}{\partial X_m(3)} + f'(X_j(2)) \cdot \frac{\partial X_m(3)}{\partial X_j(2)} \cdot \frac{\partial}{\partial X_m(3)} \left( \frac{\partial E}{\partial X_m(3)} \right) \right] \]

\[ = W_{jm} \cdot \left[ f''(X_j(2)) \cdot \frac{\partial E}{\partial X_m(3)} + [f'(X_j(2))]^2 \cdot W_{jm} \cdot \frac{\partial^2 E}{\partial X_m^2(3)} \right] \quad (A1.10) \]

where \( \frac{\partial^2 E}{\partial X_m^2(3)} = b_{mm}(3,3) \) from (A1.8) above.

The above calculations are performed iteratively for every pattern in the data set, and the total Hessian is calculated by summing up the partial Hessians derived using each data point.
Appendix 2: MATLAB Source Code for Input Ranking Algorithms

A2.1: Input-Output Derivative Analysis

function result = getSARankings(net, data)
    % Top level function to calculate rankings-see getDerivativeMatrix.m
    % net is a loaded network and data is training+testing dataset
    
    dMat = getDerivativeMatrixOne(net, data, 1);
    dMat = abs(dMat);
    [m n] = size(dMat);
    means = mean(dMat);
    means = means';
    means(:, 2) = [1:n]';
    result = sortrows(means, 1);

function derivMatrix = getDerivativeMatrixOne(net, data_matrix, lastColumnOutput)
    % This function calculates the derivative of the output with respect to
    % each input variable. Uses getDerivative.m. The output is the
    % derivative matrix for the network net given data data_matrix. This
    % function only applies to neural networks with one hidden layer
    
    weights = net.LW;
    input_weights = net.IW;
    
    weights_two_to_three = weights{2, 1};
    weights_one_to_two = input_weights{1};
[m n] = size(weights_two_to_three);
num_layer_two_nodes = n;

[num_rows num_columns] = size(data_matrix);

if(lastColumnOutput)
    num_columns = num_columns - 1;
end

for row = 1:num_rows
    disp('row:');
    disp(row);
    currentInputs = data_matrix(row, 1:num_columns);
    for i = 1:num_layer_two_nodes
        inputsLayerTwo(i) = getInputLayerTwo(i, net, currentInputs);
    end
    inputsLayerThree(1) = getInputLayerThree(1, net, currentInputs, inputsLayerTwo);

    for col = 1:num_columns
        %disp('col:');
        %disp(col);
        derivMatrix(row, col) = getDerivativeOne(col, net, currentInputs, inputsLayerTwo, inputsLayerThree);
    end
end

function result = getDerivativeOne(input_node_index, net, inputs, inputsLayerTwo, inputsLayerThree)

% Evaluates the derivative of output with respect to an input variable, at a particular datapoint (a set of input values)
% Inputs
%-------
% input_node_index: the index of the input for which we want to calculate the derivative
% net: the network used to calculate the derivative
% %inputs: a vector with the input values for which the derivative is to be evaluated
% %inputsLayerTwo: the inputs into layer 2
% %inputsLayerThree: the inputs into layer 3
% %Output: result is the derivative value of input input_node_index evaluates
%with the input values inputs

weights = net.IW; %weight matrix each element is another matrix
input_weights = net.IW;
weights_two_to_three = weights {2, 1}; % oneD matrix (array)
weights_one_to_two = input_weights{1};

%vector of biases
biases_two = net.b{1};
biases_three = net.b{2};

[m n] = size(weights_two_to_three);
num_layer_two_nodes = n;

sum = 0;
for layer_two_node = 1:num_layer_two_nodes
    arg = inputsLayerTwo(layer_two_node);
    deriv = dlogsig(arg, logsig(arg));
    sum = weights_two_to_three(1, layer_two_node) *
        weights_one_to_two(layer_two_node, input_node_index) * deriv + sum;
end

% now the summations results are stored in sum!

%sum represents d(x(3))/dx(i,1)

%Now must multiply by dY/dx(3) to get our answer!!
%dY/dx(3) = df(x(3))/dx(3)
%f is logsig
arg3 = inputsLayerThree_logsig(1);
final_deriv = dlogsig(arg3, logsig(arg3));
function result = getInputsLayerThree_logsig(net, inputsLayerTwo)
    %gets the inputs going into the output layer of logistic sigmoid network
    %inputsLayerTwo = inputs into layer two of this network
    %first thing we do is get the weights from layer 2 to layer 3!
    weights = net.LW;
    %weights going from layer 1 to layer 2
    weight_matrix = weights{2, 1};
    num_layer_three_nodes = size(weight_matrix, 1);
    num_layer_two_nodes = size(weight_matrix, 2);
    %each row of this matrix corresponds to a node in the "to" layer
    %each column corresponding to each node from the "from" layer
    %therefore, to get the weight from node i to j, we get
    weight_matrix(j, i)
    %we also need to know the biases for layer 3!
    bias_vector = net.b{2};
    for nodeIndex = 1:num_layer_three_nodes
        total = bias_vector(nodeIndex);
        %we need to know the number of nodes in layer two (layer 1 in matlab terms)
        [m n] = size(weight_matrix);
        %we know that the number of "from" nodes is the number of columns in the
        %weight matrix
        for layer_two_index = 1: num_layer_two_nodes
            arg = inputsLayerTwo(layer_two_index);
            %we need the weight from the layer two node to nodeIndex in layer3
            total = total + logsig(arg) * weight_matrix(nodeIndex, layer_two_index);
        end
        result(nodeIndex) = total;
    end
function result = getInputsLayerTwo(net, inputs)

% gets inputs into the first hidden layer of a network
% inputs - the vector of inputs into the input layer
%
% first thing we do is get the weights from layer 1 to 2
% input_weights = net.IW{1};

% we also need the biases!
bias_vector = net.b{1};

% get num nodes
num_nodes = size(input_weights, 1);

for nodelndex=1:numnodes

% we assume a normal weighting from input layer to next
% we want the weights from each input to the node indexed nodeIndex
num_inputs = net.inputs{1}.size;
total = bias_vector(nodeIndex);

for input_index = 1 : num_inputs
    total = total + input_weights(nodeIndex, input_index) * inputs(input_index);
end
result(nodeIndex) = total;
end

A2.2: Weight Derivative Analysis

function result = getHessianRankings_latest(net, data)

% top-level function to perform the weight derivative ranking
% method on a network net and using dataset data
% returns a list of input variables sorted by input saliency
res = getHessianDeltas_latest(net, data);

weights_one_to_two = res{1};
weights_two_to_three = res{2};
\[ \begin{align*} 
\text{[m n] &= \text{size(weights\_one\_to\_two);} } \\
\text{num\_inputs &= n;} \\
\text{[m n] &= \text{size(weights\_two\_to\_three);} } \\
\text{num\_layertwo\_nodes &= n;} \\
\end{align*} \]

%must loop through all inputs

for i=1:num\_inputs,

\text{sum} = 0;

%for each input, must calculate \text{w(ij)}* for each node in hidden layer

for j=1:num\_layertwo\_nodes,

\text{if(i == 1)}

%means that \text{S(j)}'s have not been calculated yet...
%Thus calculate them, store in the array and use if needed

\text{inner\_sum} = 0;
for k=1:num\_inputs,
\text{inner\_sum} = \text{inner\_sum} + \text{trans(weights\_one\_to\_two}(j, k));
end;
\text{hidden\_sum}(j) = \text{inner\_sum};
end

\text{sum} = \text{sum} + (\text{trans(weights\_one\_to\_two}(j, i))/\text{hidden\_sum}(j)) * \\
\text{trans(weights\_two\_to\_three}(i, j));
end
\text{result\_temp}(i) = \text{sum};
end

% now we have a vector with all the summed effects...
% let us now put the results into percentage form

disp(result\_temp);
disp(size(result\_temp));
%disp(sum(result\_temp, 2));

\text{sum\_vals} = 0;
for l=1:size(result\_temp, 2),
sum_vals = sum_vals + result_temp(1);
end

%sum_vals = sum(result_temp);
for m=1:size(result_temp, 2),
result(m) = (result_temp(m)/ sum_vals) * 100;
end
result = result';
result(:, 2) = [1:size(result, 1)]';
result = sortrows(result, 1);

function result = getHessianDeltas_latest(net, dataset)
% calculates the actual Error delta's based on 1st and 2nd error 
% derivatives of the weights. This function is used by 
% getHessianRankings_latest.m
% returns a matrix of error deltas for the weight matrices
%
num_datapts = size(dataset, 1);

w_1 = net.IW{1};
w_2 = net.LW{2, 1};

num_11 = size(w_1, 1);

l2_sderiv = zeros(1, num_11);
l2_deriv = zeros(1, num_11);

l1_sderiv = zeros(num_11, size(w_1, 2));
l1_deriv = zeros(num_11, size(w_1, 2));

for counter=1:size(dataset, 1)
    disp(counter);
    datapoint = dataset(counter, :);
    b_2 = zeros(1);
    b_lto2 = zeros(1, num_11);
    b_l = zeros(num_11, num_11);
    sig_2(1) = 0;
    sig_l = zeros(1, num_11);

    [m n] = size(datapoint);
    inputs = datapoint(:, 1:n-1);
    target = datapoint(:, n);
a_1 = getInputsLayerTwo_netlogsig(net, inputs);
a_2 = getInputsLayerThree_netlogsig(net, a_1);

z_1 = inputs;
z_2 = logsig(a_1);
z_3 = logsig(a_2);

num_inputs = size(inputs, 2);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%layer2

sig_2(1) = dlogsig(a_2(1), logsig(a_2(1))) * (2/num_datapts) * (z_3(1) - target);

%layer 1

for i=1:num_11
    sig_1(i) = dlogsig(a_1(i), logsig(a_1(i))) * w_2(1, i) * sig_2(1);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

f = logsig(a_2(1));
dderiv = f*(1-f)*(1-2*f);

H_m = dderiv * (2/num_datapts) * (z_3(1) - target) + ((dlogsig(a_2(1),
logsig(a_2(1))))^2) * (2/num_datapts);

% b 2->2

b_2 = H_m;

% b 1->2

for i=1:num_11
    b_1to2(i) = w_2(1, i) * dlogsig(a_1(i), logsig(a_1(i))) * H_m;
end

% b 1->1

for i =1:num_11
    f = logsig(a_1(i));
dderiv = f*(1-f)*(1-2*f);

    b_1(i) = w_2(1, i) * (dderiv*sig_2(1) + dlogsig(a_1(i),
    logsig(a_1(i)))) * b_1to2(i));
end
%compute layer 2 second derivs
for i=1:num_11
    12_sderiv(i) = 12_sderiv(i) + ((z_2(i))^2) * b_2(1);
end

%compute layer 1 derivs
for x = 1:num_11
    for y=1:num_inputs
        11_sderiv(x, y) = 11_sderiv(x, y) + ((z_1(y))^2) * b_1(x);
    end
end

%now we need to calculate 1st derivatives
%compute layer 2 first derivs
for i=1:num_11
    12_deriv(i) = 12_deriv(i) + z_2(i) * sig_2(1);
end

%compute layer 1 first derivs
for x = 1:num_11
    for y=1:num_inputs
        11_deriv(x, y) = 11_deriv(x, y) + z_1(y) * sig_1(x);
    end
end

end

%%%derivs now calculated

%calculate new weight matrices for evaluation ranking
%layer 2
for i = 1:num_11
    new_12(i) = 12_deriv(i) * w_2(i) + .5 * 12_sderiv(i) * ((w_2(i))^2)
end

% layer 1
for l = 1:num_11
for k = 1:num_inputs
    new_11(l, k) = -1 * ll_deriv(l, k) * w_1(l, k) + .5 * l1_sderiv(l, k) * ((w_1(l, k))^2);
end
result{1} = new_11;
result{2} = new_12;

function result = trans(weight)
% transfer function for the weights in weight derivative ranking
% analysis and weight interpretation analysis
result = abs(weight) * logsig(weight);

A2.3: Modified Weight Interpretation Algorithm

function result = getWeightRankingsOne_3(net)
% Calculates modified weight interpretation rankings based on the
% network %net
% returns a sorted matrix where first column is variable index and
% second column is saliency value. The matrix’s rows are sorted by
% saliency value
% to only be used with one-hidden layer networks
input_weights = net.IW;
weights = net.LW;

weights_one_to_two = input_weights{1};
[m n] = size(weights_one_to_two);
num_inputs = n;
weights_two_to_three = weights{2, 1};
[m n] = size(weights_two_to_three);
num_layer_two_nodes = n;

% must loop through all inputs

for i=1:num_inputs,
    sum = 0;
    %for each input, must calculate w(ij)* for each node in hidden layer
    for j=1:num_two_nodes,
        if(i == 1)
            %means that S(j)'s have not been calculated yet...
            %Thus calculate them, store in the array and use if needed
            inner_sum = 0;
            for k=1:num_inputs,
                inner_sum = inner_sum + trans(weights_one_to_two(j, k));
            end;
            hidden_sum(j) = inner_sum;
        end
        sum = sum + (trans(weights_one_to_two(j, i))/hidden_sum(j)) * 
        trans(weights_two_to_three(1, j));
    end
    result_temp(i) = sum;
end

% now we have a vector with all the summed effects...
% let us now put the results into percentage form

disp(result_temp);
disp(size(result_temp));
%disp(sum(result_temp, 2));

sum_vals = 0;
for l=1:size(result_temp, 2),
    sum_vals = sum_vals + result_temp(l);
end
%sum_vals = sum(result_temp);
for m=1:size(result_temp, 2),
    result(m) = (result_temp(m)/ sum_vals) * 100;
end
result = result';
result(:, 2) = [1:size(result, 1)]';
result = sortrows(result, 1);

A2.4: Original Weight Interpretation Algorithm

function result = getWeightRankingsOne_abs(net)
%Calculates original weight interpretation rankings based on the network %net
%returns a sorted matrix where first column is variable index and %second column is saliency value. The matrix's rows are sorted by %saliency value %to only be used with one-hidden layer networks

input_weights = net.IW;
weights = net.LW;

weights_one_to_two = input_weights{1};
[m n] = size(weights_one_to_two);
num_inputs = n;

weights_two_to_three = weights{2, 1};
[m n] = size(weights_two_to_three);
num_layer_two_nodes = n;

%must loop through all inputs
%
for i=1:num_inputs,
    sum = 0;

    %for each input, must calculate w(ij)* for each node in hidden layer
    for j=1:num_layer_two_nodes,
        if(i == 1)
            %means that S(j)'s have not been calculated yet...
            %Thus calculate them, store in the array and use if needed
            inner_sum = 0;
            for k=1:num_inputs,
                inner_sum = inner_sum + abs(weights_one_to_two(j, k));

        end

        sum = sum + abs(weights_one_to_two(i, j));
    end

    result(i, 2) = sum;
end

result(:, 2) = [1:size(result, 1)]';
result = sortrows(result, 1);
end;

    hidden_sum(j) = inner_sum;
end

    sum = sum + (abs(weights_one_to_two(j, i))/hidden_sum(j)) *
    abs(weights_two_to_three(l, j));
end

    result_temp(i) = sum;
end

% now we have a vector with all the summed effects...
% let us now put the results into percentage form

    disp(result_temp);
    disp(size(result_temp));
%disp(sum(result_temp, 2));

    sum_vals = 0;
    for l=1:size(result_temp, 2),
        sum_vals = sum_vals + result_temp(l);
    end
%sum_vals = sum(result_temp);
    for m=1:size(result_temp, 2),
        result(m) = (result_temp(m)/ sum_vals) * 100;
    end
    result = result';
    result(:, 2) = [1:size(result, 1)]';
    result = sortrows(result, 1);