University of Alberta

Over-fitting in Generalized Linear Evaluation Models

by

Siddhartha Chinthapally ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2006

Your file Votre référence
ISBN: 0-494-13804-1
Our file Notre référence
ISBN: 0-494-13804-1

To my parents and teachers

Abstract

We consider the problem of parameter over-fitting in sparse generalized linear models and empirically compare early stopping, regularization, weight clamping, and combinations thereof in the context of the backprop algorithm. New insights into the problem of over-fitting are given based on feature frequency. Results obtained for optimizing parameters of function approximators for synthetic data sets and a popular board game suggest that weight clamping depending on feature frequency combined with early stopping can outperform the other considered techniques in terms of test-set over-fitting.

Acknowledgements

I extend my gratitude to all the people who made this thesis possible. First and foremost, I thank my supervisor, Michael Buro, for getting this thesis started and for the constant guidance and support over the duration of the thesis. Throughout this period he provided me with lots of ideas, helped me whenever I got stuck, taught me lots of new things, enouraged me whenever I was down and has been a great company. This thesis would not have been possible without him.

I am also grateful to the GAMES group for the various interesting lectures and the games parties. I also thank all my teachers here at the graduate school for the wonderful courses. I thank Herb Yang, Martin Müller, Michael Bowling, Michael Buro and Guohui Lin for introducing me to various interesting fields in computer science.

I gratefully acknowledge the financial support provided the University of Alberta, the Department of Computing Science and iCORE.

I thank Shubhankar Chatterjee for proof-reading earlier drafts of this thesis and for his useful comments. I also thank my friends in Edmonton — Shallu, Raj, Biswaroop, Divya — for their moral support over the past two years.

Table of Contents

List of Tables

List of Figures

...

Chapter 1

Introduction

1.1 Machine Learning

Machine learning is the subject that deals with making machines (computers) learn. Machine learning is quite widespread these days and machine learning techniques have been successfully applied in varied fields of applications like manufacturing, banking, genetics, medical applications, control systems, computer games etc. One of the earliest applications of machine learning has been in Samuel's checker games [Sam59][Sam67]. The field of machine learning is closely related to statistics as data analysis is at the core of both fields. However, machine learning deals with the design of algorithms to process this data to achieve learning. Machine learning is also related to the broad field of Artificial Intelligence. In fact, some of the successful applications of machine learning have been in computer games like Backgammon [Tes95] and Othello [Bur99][Bur97]. A detailed survey of machine learning techniques applied to games is presented in [Für01].

It is difficult to precisely define learning. Let us look at what comprises learning. Webster's dictionary defines learning as "to gain knowledge or information; to fix in mind". Machine learning derives some of its methods from biologically inspired models. If we adapt the dictionary definition of learning to machines we can say that a machine learns if it can modify itself by interacting with the environment so that it improves its performance or its ability to predict outcomes. Forms of interaction between a machine and the environment include data, reward and error feedback.

It is possible to make machines experts by transferring the human domain knowledge. But this method cannot be applied to all situations and machine learning is a strong candidate in those situations. Some of those situations are as follows:

- In some cases the relationship between inputs and outputs is difficult to define or not known, even though a large number of sample inputs and outputs are known. In other words, the function is not known and machine learning can be used to learn a function which approximates the unknown function.
- Machine learning can be used to discover hidden relationships between items. Data mining studies unearth interesting correlations in consumer behaviour. For example, shopping sites like Amazon.com suggest other useful purchases based on data mining studies.
- When the system is dynamic, we can use machine learning techniques to make the system adapt to the changes rather than redesign the system each time. Sometimes the specification of the environment may not be known at the time of designing. For example, in an elevator control design the usage details for the elevator like frequency of use etc are unknown at the design time and may change with time. If a busy user moves to sixth floor from third floor, the elevator would learn the new pattern and maximize its performance.

1.2 Types of Learning

Machine learning can be broadly classified into supervised learning, unsupervised learning and reinforcement learning. In supervised learning the system is given a set of training samples for which the outputs are known. The system has to make a hypothesis h which approximates the unknown function f we are trying to learn. In other words, the system has to learn to predict based on the samples provided. The most common types of problems encountered in supervised learning are curve-fitting, real valued function approximation, and classification. Curve fitting as the name suggests is the problem of fitting the outputs to a real valued function of the inputs. Classification deals with mapping each input vector to one of the several classes of outputs. The most popular method used for supervised learning is a form of gradient descent which is called linear regression in linear models and error backpropagation in neural networks.

In unsupervised learning, the system is given a set of unlabeled training samples. Usually the goal is to cluster the samples into various sets. So, no output exists for a single sample. Since there exist measures to calculate the goodness of a clustering no other outputs are required. The data points can be clustered based on how closely related samples of one group are (diameter of the cluster) and how distinct samples

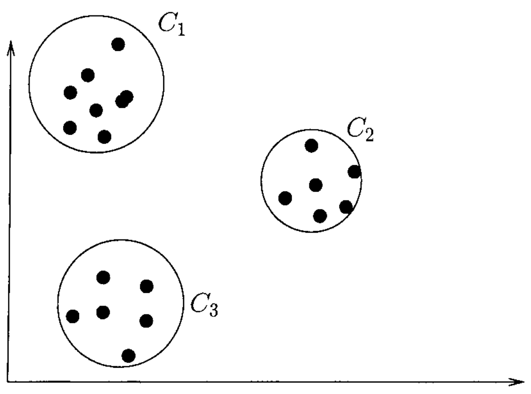


Figure 1.1: Clustering

from different clusters are (distance between clusters). In Figure 1.1 the data can be clustered into classes C_1, C_2 and C_3 as shown. For instance, unsupervised learning is used in the classification of species in bio-informatics.

In reinforcement learning, the learner has to choose a policy that determines the action to perform in a particular state. The learner is rewarded for its actions by the environment. The goal of reinforcement learning is to choose a policy that maximizes the total reward. The learner accomplishes this by judiciously exploring the action space and using the knowledge gained form previous experience to exploit in the current state. For an in-depth study of reinforcement learning see [SB98]. For example, reinforcement learning has been successfully applied to improve the performance of an elevator controller [CB96].

For a thorough discussion of machine learning, its relation to various fields and various types of learning refer to standard text books in machine learning [Nil96]. In this thesis, we restrict ourselves to supervised learning.

1.3 Over-fitting

One of the problems encountered when using supervised learning with neural networks or regression is over-fitting. In what follows, we refer to the sequence of input vectors with the corresponding labeled outputs which is used for training as *training data* or *training set* and the sequence of inputs with labeled outputs which is used for testing as *test data* or *test set*. Though the terms *training set* and *test set* are widely used, they are not sets in strict sense as they can have duplicate data. Strictly speaking, they are multi-sets.

We say a model over-fits if it approximates the function on training samples with low error while having a high error when tested for samples from an unseen

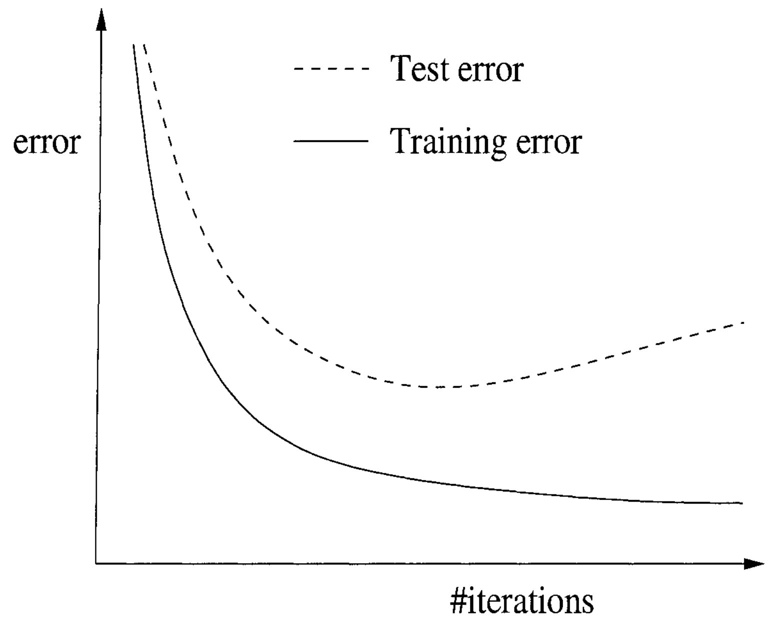


Figure 1.2: Over-fitting

test set (Figure 1.2). Over-fitting occurs when the system fits not just the signal but also the noise associated with the signal. The system adjusts itself so that it approximates the noisy training data but generalizes poorly on test data. Over-fitting leads to bad predictions even when trained on a large data set for a long time. In this thesis we look at ways to overcome over-fitting and compare the results to existing techniques.

1.4 Contribution of Thesis

The contributions of this thesis are the following

- New insights into the problem of over-fitting in *Generalized Linear Evaluation Models* (GLEM) based on feature frequency.
- We propose new techniques to actively prevent over-fitting in GLEM and compare their performance relative to existing techniques such as *regularization.*
- A novel over-fitting measure that uses a test set containing rare features.
- An improved model for constructing evaluation functions using patterns by extending GLEM to include non-linear functions.

- Software for performing training while preventing over-fitting for GLEM. The software can also be used to test new ideas for preventing over-fitting.

1.5 Organization of Thesis

The remainder of the thesis is organized as follows:

Chapter 2 deals with Linear Models, Generalized Linear Models and Generalized Linear Evaluation Models.

Chapter 3 discusses various issues involved in Supervised Learning. Enhancements to speed up learning and to improve the generalization of the system are discussed. The problem of over-fitting is also discussed. Chapter 3 also discusses the techniques used to overcome the problem of over-fitting.

Chapter 4 gives an overview of the software, details on the experimental setup and the results of simulations on a simple synthetic application and on Othello game.

Chapter 5 summarizes the results from the experiments, makes conclusions and suggests directions for further research.

Chapter 2

Generalized Linear Evaluation Model

In this thesis we focus on *Generalized Linear Evaluation Model* [Bur98] and its extensions. GLEM can be viewed as an extension of Linear Models and as a variation of Generalized Linear Models. Before describing GLEM we will discuss Linear Models and Generalized Linear Models in some detail.

2.1 Linear Models

Linear Models are statistical models that fit the measurements taken (outputs) as a linear combination of the observables (independent input variables) while minimizing the norm of the residual [1] vector. Let X be the vector of independent input variables and y be the function. Linear Models describe a linear relation between y and the x_is.

$$y = b_0 + b_1 x_1 + b_2 x_2 + \cdots + b_n x_n$$

The solution to this will be the vector b of parameters that determines the model. This can be solved exactly if the number of equations is equal to the number of unknowns and all the matrices have full rank. Usually this is not the case in practice. The system is over-specified and there are more samples than unknowns i.e more equations than variables. The solution now is to fit the observations into a model which minimizes the residual norm. For the i^{th} observation we can write the equation as

[1]We use the terms residual and error interchangeably throughout this thesis though their meanings are different in strict statistical sense. Error refers to the difference between an observation and its expected value (unobservable), whereas residual is an observable estimate of the error.

$$y^i = b_0 + b_1 x_1^i + b_2 x_2^i + \cdots + b_n x_n^i + e^i$$

where e^i is the residual corresponding to the i^{th} sample. The system of equations can be written as follows:

$$y^1 = b_0 + b_1 x_1^1 + b_2 x_2^1 + \cdots + b_n x_n^1 + e^1$$

$$y^2 = b_0 + b_1 x_1^2 + b_2 x_2^2 + \cdots + b_n x_n^2 + e^2$$

$$\vdots$$

$$y^N = b_0 + b_1 x_1^N + b_2 x_2^N + \cdots + b_n x_n^N + e^N$$

In matrix form :

$$\begin{pmatrix} y^1 \\ y^2 \\ \vdots \\ y^N \end{pmatrix} = \begin{pmatrix} 1 & x_1^1 & x_2^1 & \dots & x_n^1 \\ 1 & x_1^2 & x_2^2 & \dots & x_n^2 \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & x_1^n & x_2^n & \dots & x_n^N \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{pmatrix} + \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{pmatrix}$$

The whole system of equations can be compactly represented in matrix form as

$$Y = XB + E$$

where Y is an $N \times 1$ vector of measurements/outputs, X is $N \times (1 + n)$ matrix of inputs where each row is an input vector for one output in Y, B is an $(n + 1) \times 1$ matrix of parameters where each parameter corresponds to one input and E is an $N \times 1$ vector of residuals where each component is the error of the corresponding output.

The model is called linear because it is linear in the parameter vector B. For instance, the following model

$$y = ax_1^2 + bx_2 + c$$

is a Linear Model because it is linear in the parameters a, b and c while the model

$$y = ax_1 + bx_2 + ab$$

is not a Linear Model. Often, the parameter vector B is estimated using the maximum likelihood principle.

The maximum likelihood estimate finds a parameter vector which most likely results in the observed outputs. The following example explains the concept. A person has three race horses with the probability of winning a race equal to 0.1, 0.2 and 0.7 respectively. There are 10 races each day and only one of the 3 horses is entered in races on a particular day. Suppose a horse won exactly 4 races on a day. The question is which of the horses is it. Now, we calculate the likelihood of each horse and choose the one with maximum likelihood.

$$\text{Likelihood of 4 wins out of 10 given } (p = 0.1) = \binom{10}{4}(0.1)^4(0.9)^6 \approx 0.011$$

$$\text{Likelihood of 4 wins out of 10 given } (p = 0.2) = \binom{10}{4}(0.2)^4(0.8)^6 \approx 0.088$$

$$\text{Likelihood of 4 wins out of 10 given } (p = 0.7) = \binom{10}{4}(0.7)^4(0.3)^6 \approx 0.037$$

The maximum likelihood principle says that it is most likely horse number 2 was entered in the races on that day. Calculating the likelihood this way for large systems is impractical. Fortunately, under the Gauss-Markov conditions of *uncorrelated*, *homoescedastic* (all errors have the same variance) and *zero mean* errors the maximum likelihood method is equivalent to the least-squares optimization [PTVF92]. The ubiquitous least-squares method is discussed in the following sections.

2.2 Generalized Linear Models

Generalized Linear Models are an extension of Linear Models. A Linear Model is not sufficient in cases where the relationship between the output and input is not linear. For example, in chess, the material value is a good indicator of the chance of winning the game. The difference in the winning chance between having one Queen and having two Queens is larger than the difference between having three Queens and four Queens. In other words, the relationship between the number of excess Queens and winning chance is non-linear.

To account for this non-linear relationship between inputs and outputs, Generalized Linear Models apply a non-linear function to the output of a Linear Model. A Generalized Linear Model describes a non-linear relationship between inputs (x_is) and the output (y) in the following form

$$y = g(b_0 + b_1x_1 + b_2x_2 + \cdots + b_mx_m)$$

where g is a non-linear function usually S-shaped in the form $g(x) = \frac{1}{1+e^{-x}}$ if the output is restricted to $(0, 1)$. Least-squares optimization can be used for the generalized linear model as well.

2.3 Least-Squares Method

The least-squares method tries to minimize the sum of squares of the residuals and thus finds a function which fits the data. The goodness of the fit is measured by the error function which is minimized. It is the most commonly used optimization technique as it has nice theoretical properties and simple algorithms.

The problem can be formulated as follows. We are given a sequence of data points

$$D = (x_i, y_i), i = 1, 2, \ldots, N$$

and we are supposed to find a function f such that

$$f(x_i) \approx y_i$$

We now find a solution to the function (assuming we know the form of the function) in the parameter space which minimizes the sum of the squares of the residuals.

$$E = \sum_{i=1}^{N} (y_i - f(x_i))^2$$

If the function f is linear in the parameters to be optimized - like in the Linear Models - we use the linear least-square method which has an algebraic solution. If the function f is non-linear in the parameters to be optimized we use non-linear least-square methods such as gradient descent. Generalized Linear Models fall into this category.

2.3.1 Linear Least-Square Optimization

We want to find the solution to the system of equations

$$Y = XB + E$$

that minimizes $||E||$ where Y is an $m \times 1$ vector of outputs, X is $m \times (1+n)$ matrix of inputs where each row is an input vector for one output in Y, B is an $(n+1) \times 1$ matrix of parameters where each parameter corresponds to one input and E is an $m \times 1$ vector of residuals.

$$||E|| = ||XB - Y||$$

The l_2 norm of a vector V can be written as a the dot product of V^T and V.

$$\begin{aligned} ||E|| &= (XB - Y)^T (XB - Y) \\ &= (XB)^T XB - (XB)^T Y - Y^T XB + Y^T T \end{aligned}$$

At the minimum the derivative with respect to B is equal to zero. Differentiating both sides with B at the minimum gives

$$0 = 2X^TXB - 2X^TY$$

Therefore, a vector B that minimizes $||E||$ also solves the following equation

$$X^TXB = X^TY$$

The above is a system of linear equations with $n+1$ equations and $n+1$ unknowns. The system can be uniquely solved if X^TX has full rank. In which case, the solution can be found as

$$B = (X^TX)^{-1}X^TY$$

Example

Let us try to find a straight line $y = b_0 + b_1x$ which fits the following points the best. The points are : $(1, 1.5), (2, 1.5), (2.5, 2.6), (3.0, 2.95)$ and $(5.0, 4.0)$.

$$X = \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 2.5 \\ 1 & 3.0 \\ 1 & 5.0 \end{pmatrix}, \quad Y = \begin{pmatrix} 1.5 \\ 1.5 \\ 2.6 \\ 2.95 \\ 4.0 \end{pmatrix}$$

$$X^TX = \begin{pmatrix} 5 & 13.5 \\ 13.5 & 45.25 \end{pmatrix}, \quad X^TY = \begin{pmatrix} 12.55 \\ 39.35 \end{pmatrix}$$

$$(X^TX)^{-1} = \frac{1}{44}\begin{pmatrix} 45.25 & -13.5 \\ -13.5 & 5 \end{pmatrix}$$

$$B = (X^TX)^{-1}X^TY \approx \begin{pmatrix} 0.83 \\ 0.62 \end{pmatrix}$$

The equation of the best fit straight line is

$$y = 0.83 + 0.62 \cdot x$$

(see Figure 2.1). The same method can be used to find a best-fitting polynomial too as it is still linear in the parameters to be optimized.

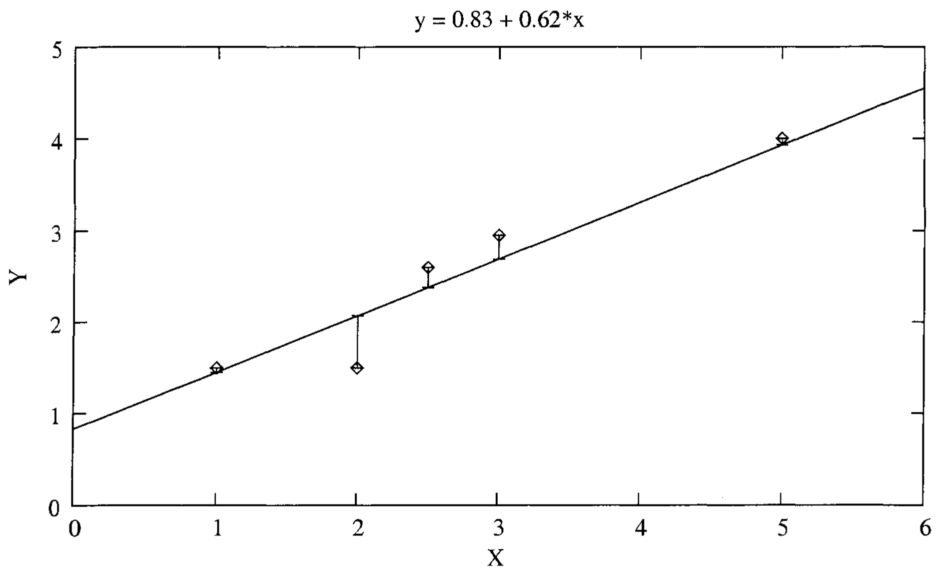


Figure 2.1: Linear Least-Square Example

2.3.2 Non-linear Least-Square Optimization

Non-linear least-square optimization methods usually use some form of gradient to direct their search for a minimum. Methods which use just the first derivative like *gradient descent* are called first-order methods. Methods which use higher-order derivatives also are called higher-order methods. Conjugate gradient, Gauß-Newton and Levenberg-Marquardt methods are examples of second-order methods. Gauß-Newton and Levenberg-Marquardt methods are known to work very well for a small number of weights. However, their space requirements are quadratic in the number of weights and hence they are infeasible for larger number of weights. They also involve matrix inversions which are expensive operations. Conjugate gradient method has a space requirement that is linear in the number of weights. A theoretical analysis presented in [Sch94] indicates that the conjugate gradient method takes fewer iterations to converge than gradient descent. On the other hand, conjugate gradient methods perform more computation in one iteration compared to gradient descent methods. An empirical study [SL94] of standard gradient descent and conjugate gradient methods in neural networks on N2N encoder benchmark problem shows that both methods have a near equal median time complexity. In the following we present the gradient descent method.

If $E(w)$ is differentiable at the current point $\mathbf{W}_{\mathbf{cur}}$

$$E(\mathbf{W}) = E(\mathbf{W}_{\mathbf{cur}}) + \nabla E(\mathbf{W}_{\mathbf{cur}})^T(\mathbf{W} - \mathbf{W}_{\mathbf{cur}}) + \text{ higher-order terms}$$

Ignoring the higher-order terms and differentiating both sides with respect to $\mathbf{W}$

$$\nabla E(\mathbf{W}) = \nabla E(\mathbf{W_{cur}}) + \nabla^2 E(\mathbf{W_{cur}})(W - W_{cur})$$

At $W = W_{min}$ the gradient is zero, which implies

$$W_{min} = W_{cur} - (\nabla^2 E(\mathbf{W_{cur}}))^{-1} \nabla E(\mathbf{W_{cur}})$$

where $\nabla^2 E$ is the Hessian matrix of E. If E is quadratic in w then the components of the Hessian would be constants and we can write the equation in the more familiar form.

$$W_{min} = W_{cur} - \beta \nabla E(\mathbf{W_{cur}})$$

where β is the learning rate.

Let us consider a simple case where we have two variables to be optimized.

$$f(x) = ax + b$$

$$E = \sum_{i=1}^{N} (y_i - f(x_i))^2$$

$$E = \sum_{i=1}^{N} (y_i - ax_i - b)^2$$

$$\frac{\partial^2 E}{\partial a^2} = 2\sum_{i=1}^{N} x_i^2, \quad \frac{\partial^2 E}{\partial b^2} = 2N, \text{ and } \frac{\partial^2 E}{\partial a \partial b} = \frac{\partial^2 E}{\partial a \partial b} = 2\sum_{i=1}^{N} x_i$$

The determinant of the Hessian is

$$||H|| = 4\left(N\sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{N} x_i\right)^2\right)$$

which can be shown to be > 0 using induction. Therefore, the error surface is convex and gradient surface can be used to find a minimum as shown in Figure 2.2 (only one dimension is shown). The parameter weights are updated as follows.

$$a := a + \delta a$$

$$\delta a = -\frac{\partial E}{\partial a} \cdot \beta$$

where β is the learning rate parameter which controls the speed of convergence. Parameter b is updated similarly. In each iteration the weight moves in the direction where the magnitude of the gradient decreases. Finally, the weight a reaches the

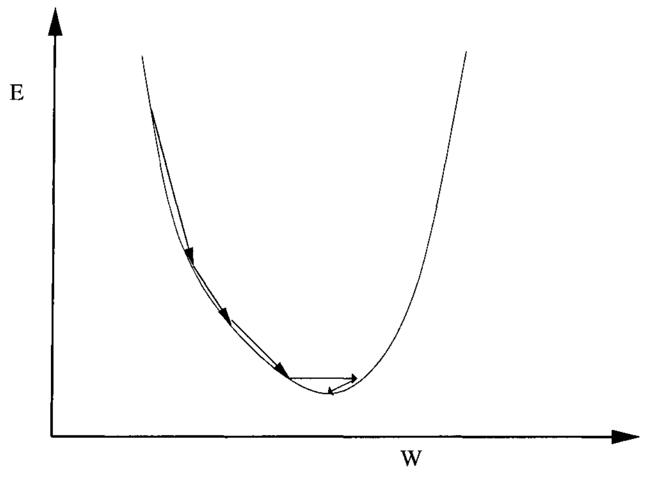


Figure 2.2: Gradient Descent

minimum where the gradient is 0 and therefore there won't be any more changes to a once it reaches the minimum. The learning rate β controls the effect of gradient on the weight update. A small value for β results in slow convergence while a large value might result in divergence. Techniques to speed up learning and adapt learning rates are discussed in Chapter 3.

2.4 Generalized Linear Evaluation Model (GLEM)

The Generalized Linear Evaluation Model [Bur98] has its roots in the Generalized Linear Model. This is a model which goes beyond just the optimization of the weights and also tries to generate the features. The GLEM framework has been largely inspired by the work done while developing a world-class Othello program, LOGISTELLO [Bur97]. The model is specified as follows. Let P denote the sequence of data points or samples from which data is extracted. Let A be a finite set of integer valued *atomic* features. Let

$$R_A = \{f(\cdot) = k | f \in A, k \in \mathbb{Z}\}$$

be the set of relations over A that maps features to integers. Configurations are combinations of these relations in R_A. In case of Othello (rules described in Appendix), an example of a configuration could be *white disc on A4 and black disc on A5*.

For a position $p \in P$ we define the value of a configuration $c = r_1 \wedge \cdots \wedge r_l$ as

$$val(c(p)) = \begin{cases} 1 & \text{if } r_1 \wedge \cdots \wedge r_l = \text{ true} \\ 0 & \text{if } r_1 \wedge \cdots \wedge r_l = \text{ false} \end{cases}$$

A configuration is said to be *active* in a position if its value is 1 in that position. In the above example, the value of the configuration is 1 if a game position actually has a white disc on A4 and a black disc on A5, otherwise the value is 0. Restricting the configuration values to 0, 1 has computational advantages as we see later.

We define the Generalized Linear Evaluation Model - GLEM (P, A, g) below. The evaluation functions in GLEM have the following form

$$f_w(p) = g\left(\sum_{i=1}^{n} w_i \cdot val(c_i(p))\right)$$

where $c_1, \ldots, c_n$ are configurations over R_A, $w_i \in \mathbb{R}$ are weights and $g : \mathbb{R} \longrightarrow \mathbb{R}$ is a differentiable and monotonic function.

Therefore, GLEM is a Generalized Linear Model over the active configurations in a position. The advantage of having binary values for configurations is that there is no need for performing explicit multiplication with weights. The value can be calculated by only using summation.

The weights are determined using a least-squares optimization. Given a sequence of labeled data points $(p_i, l_i)|i = 1, \cdots, N$ and a set of configurations $c_1, \ldots, c_n$ and a function g, the weights are chosen so that the sum of the squared error

$$E(w) = \sum_{i=1}^{N}(l_i - f_w(p_i))^2$$

is minimized.

In the discussion that follows we will use Othello as the application for GLEM to illustrate concepts. This model has the power to represent any evaluation function. For instance, by considering every possible board position as a configuration we can generate any evaluation function including the perfect evaluation function, at least in theory. This can be achieved by considering each different disc on a different square to be an atomic feature.

$$A = \{f_s | f_s(p) = \text{ contents of square } s \text{ in position } p\}$$

A feature need not be simple to be atomic, complex features could be made atomic. Using simpler atomic features results in a higher number of possible configurations and hence the model can distinguish a larger number of positions. Thus GLEM has the interesting feature that the expressive power of the model can be set by the user.

2.4.1 Advantages of GLEM

GLEM has various properties which are desirable when designing practical evaluation functions.

- Using the basic atomic features, various complex configurations can be built in the process leading to the discovery of important features. Atomic features need not be basic, they can be complex as well. This freedom allows us to generate evaluation functions of varying expressiveness naturally.
- By restricting the configuration values to binary values we can eliminate the time-consuming multiplication operation. Usually in games, speed is important and GLEM helps keep the time-overhead low.
- Non-linear effects can be modelled by combining various relations into a configuration. If we use a monotonic function we need not even compute g. In a game search tree we only need to compare between evaluation values and the actual evaluation values are not needed. So, if g is monotonic, we need not compute it.
- The weight-fitting process is efficient as the system is mostly linear. Even systems with large sets of weights of the order of a million could be fitted in a reasonable amount of time.

Evaluation functions for games need to be accurate and easy to compute. The following things have to be considered when selecting the training positions and configurations. Positions must be labelled with minimal error and they have to cover the type of positions the system is expected to encounter. For instance, using only games between grandmasters for training may not be a good idea as they hardly make blunders and this does not give the system a chance to learn simple but important features. The configuration set should be expressive enough to explain the data and should not be tailored too much to avoid over-fitting.

2.4.2 Configuration Selection

Configurations are conjunctions over the set of relations over the atomic feature set A. Configurations are built over atomic features. Therefore, atomic features should be simple enough so that the important features of the game can be easily constructed using these features. Moreover, these atomic features can be combined in several different ways to give rise to new features. Atomic features should be constructed on an abstracted layer. For example, in Othello many features of an evaluation function are based on the local board configurations. So for this game the abstraction needed is minimal, whereas in other games like chess a Queen at one end of the board can influence many parts of the board. The abstraction needed here is higher than in Othello. One abstraction here could be based on dependency graphs for pieces. The level of abstraction needed for atomic features is dependent on the application and the available time and space.

Automatically generating a set of configurations in a reasonable amount of time that does not under-fit or over-fit is a non-trivial task. Given an atomic feature set A, training position set P and a minimal match count k, [Bur98] gives a simple algorithm for generating configurations over A that appear at least k times in P. The idea is to iteratively increase the length of the configurations beginning with atomic features as configurations. At each step the algorithm tries to increase the size of the previously generated configurations by specializing them, until the match count drops below k.

In spite of the efficient ways to do this discussed in [Bur98], it is still a time and space consuming process. A large number of active configurations for each position slows down the evaluation as a larger number of configurations have to be computed. To counter this problem, the match count k can be increased. Increasing the match count reduces the number of active configurations but it causes the system to be less expressive.

One approach is to restrict the configurations to mutually exclusive sets called *Patterns*.

2.4.3 Patterns

A Pattern is a set of all possible most specific configurations over a subset of the atomic features.

$$pattern[f_1, \ldots, f_m] = \{r_{1,l_1} \wedge \ldots \wedge r_{m,l_m} | r_{i,l_i} = (f_i(.) = l_i), l_i \in range(f_i)\}$$

An example of a pattern (EDGE+2X - 10×1 block $A_1B_1C_1D_1E_1F_1G_1H_1B_2G_2$) in Othello is shown in Figure 2.3. This results in 3^{10} configurations. The set of these 3^{10} configurations is the pattern $A_1B_1C_1D_1E_1F_1G_1H_1B_2G_2$. Patterns are easy to build and easy to evaluate. However, it is possible that patterns may not capture all the essential features required. But their speed and ease of construction make them particularly attractive for use in game evaluation functions. There is a trade-off between using large patterns which capture a more complex evaluation function

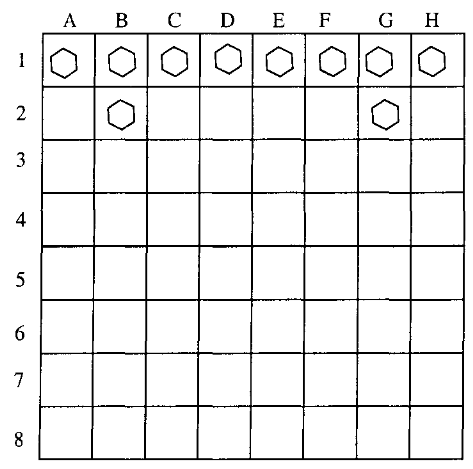


Figure 2.3: One of the patterns used in LOGISTELLO. Each hex represents an atomic feature which can take value {0, 1, 2} depending on whether the square is white, empty or black.

but are slow to be evaluated in game play and small patterns which are fast but inaccurate. Another problem associated with using large patterns is that they are sparse. Large patterns will have fewer training samples and over-fitting might set in as a result. The problem of over-fitting and techniques to overcome them are discussed in Chapter 3.

Chapter 3

Multi-Layer Perceptrons and Over-fitting

The linear models described in Chapter 2 are useful for approximating functions that are linear in the weights. In order to approximate non-linear functions we have to use a non-linear model like a Multi-Layer Perceptron (MLP). An MLP is a network of perceptrons arranged in a layered structure. The next section describes perceptrons and their computing power.

3.1 Perceptron

A perceptron is a multi-input, single output computing unit. The computation in a perceptron is divided into two stages - the accumulation stage and the activation stage as shown in Figure 3.1. In the accumulation stage a function (usually summation) is used on the inputs and a single output is calculated. In the activation stage a non-linear function like the sigmoid ($g(x) = \frac{1}{1+e^{-x}}$) or hyperbolic tangent

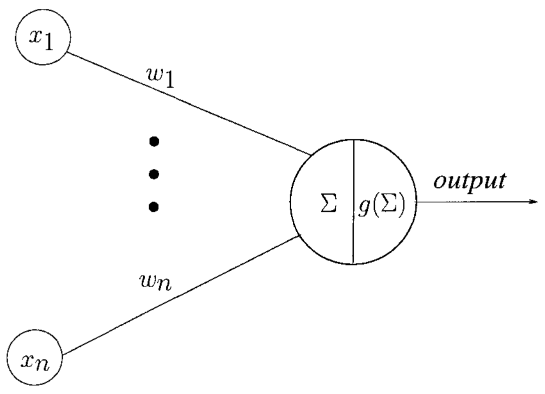


Figure 3.1: Perceptron

$(g(x) = \tanh(x))$ is applied to the sum to give a single output. Other commonly used activation functions include the *step* or threshold function and the *sign* function defined below.

$$step_t(x) = \begin{cases} 1 & \text{if } x \geq t \\ 0 & \text{if } x < t \end{cases}$$

$$sign(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

$$output = g\left(\sum_i w_i x_i\right)$$

A perceptron is a Generalized Linear Model. Therefore, the class of problems that can be learnt by a single perceptron is same as the class of problems that can be learnt by a Generalized Linear Model. A perceptron can only represent linearly separable functions. It is possible to learn any linearly separable function using perceptrons provided the samples are linearly separable.

3.2 Multi-Layer Perceptrons (MLP)

A Multi-layer Perceptron is a network of perceptrons. The perceptrons are networked in a layered structure with the outputs of one being the inputs to others. A network structure where all the links are forward or unidirectional, and there are no cycles is called a feed-forward network. In other words, a feed-forward network can be described by a directed acyclic graph. The other type of networks called recurrent networks allow all types of connections. In this thesis, we concentrate only on feed-forward network structures. Figure 3.2 shows a 3-layered, fully connected MLP. The function computed by an MLP is determined by the network structure and the weights on each link. Like the linear models described before, we can perform a gradient descent on the weight vector to learn a function using an MLP. This process is popularly known as error back-propagation or simply back-propagation.

3.3 Error Back Propagation

Let us define a few terms that are helpful in the derivation of the equations for back-propagation. We derive the equation for a single output case. The case for multiple outputs is analogous.

- I_k - Input vector for node k.

- W_k - Weight vector for node k.
- $s_k = I_k \cdot W_k$ - Weighted sum of the inputs of node k.
- $o(k) = g(s_k)$ - output for node k.
- UL(k) - $\{i|$the output of k is an input to $i\}$.

We can apply the same gradient descent algorithm here too. For a weight w_{kl} in the network the update rule would be

$$\Delta w_{kl} = -\eta \frac{\partial E(W)}{\partial w_{kl}}$$

w_{kl} is the weight corresponding to l^{th} input of node k. The difficult part is calculating the partial derivatives for each weight. It is possible to analytically calculate these derivatives for each weight, but it is cumbersome. We can make use of the network structure and the chain rule of calculus to overcome this problem.

Let p be the output node, and $D = \{X_i, y_i\}$ be the data set used for training. We can then write the error function as follows

$$E(W) = \frac{1}{2}\sum_i (y_i - o_i(p))^2$$

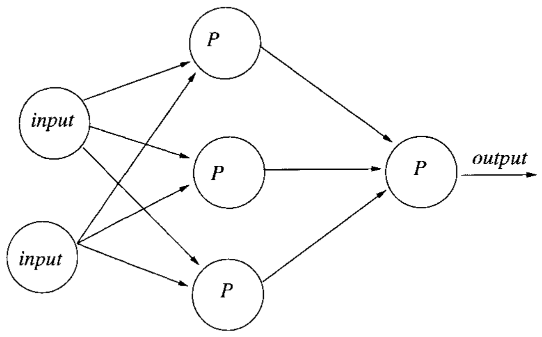


Figure 3.2: A Multi-layer Perceptron with a 2-node input layer, a 3-node hidden layer and an output node

We first consider the case where the node is an output node ($k = p$).

$$\begin{aligned}
err_i &= y_i - o_i(p) \\
\frac{\partial E}{\partial w_{pl}} &= \frac{1}{2}\frac{\partial}{\partial w_{pl}}\sum_i (y_i - o_i(p))^2 \\
&= \frac{1}{2}\sum_i \frac{\partial}{\partial w_{pl}}(y_i - o_i(p))^2 \\
&= -\sum_i (y_i - o_i(p))\,\frac{\partial o(p)}{\partial w_{pl}}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial o(p)}{\partial w_{pl}} &= \frac{\partial o(p)}{\partial s(p)}\frac{\partial s(p)}{\partial w_{pl}} \\
&= \frac{\partial g(s(p))}{\partial s(p)}\,x_{pl} \\
&= g'(s(p))\,x_{pl} \\
\frac{\partial E}{\partial w_{pl}} &= -\sum_i err_i\; g'(s(p))\,x_{pl_i} \\
\frac{\partial E}{\partial s(p)} &= -\sum_i err_i\; g'(s(p))
\end{aligned}$$

$$\therefore \Delta w_{pl} = \eta \sum_i err_i\; g'(s(p))\,x_{pl_i}$$

For a sigmoid $g(x) = \frac{1}{1+e^{-x}}$, $g'(x) = g(x)\,(1 - g(x))$ and for $g(x) = \tanh(x)$, $g'(x) = (1 - g(x)^2)$. If we use a sigmoid activation we can write the weight update as follows :

$$\Delta w_{pl} = \eta \sum_i err_i o_i(p)(1 - o_i(p))x_{pl_i}$$

Next we consider the case when the node is a hidden node. For any weight w_{kl},

$$\frac{\partial E}{\partial w_{kl}} = \frac{\partial E}{\partial s(k)}x_{kl}$$

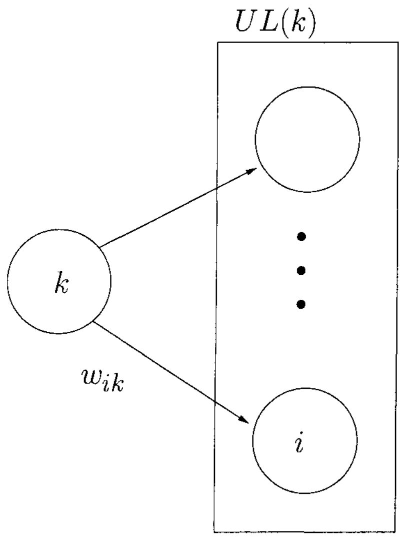


Figure 3.3: The upper layer of node k is shown. The error propagates from the upper layer to node k, while the function computation happens in the opposite direction.

For any node i above k, $s(i)$ is a function of $s(k)$. Therefore we can write

$$\begin{aligned}
\frac{\partial E}{\partial w_{kl}} &= \sum_{i \in UL(k)} \frac{\partial E}{\partial s(i)} \frac{\partial s(i)}{\partial o(k)} \frac{\partial o(k)}{\partial s(k)} \frac{\partial s(k)}{\partial w_k l} \\
&= \sum_{i \in UL(k)} \frac{\partial E}{\partial s(i)}\, w_{ik}\, \frac{\partial g(s(k))}{\partial s(k)}\, x_{kl} \\
&= g'(s(k)) \sum_{i \in UL(k)} \frac{\partial E}{\partial s(i)}\, w_{ik}\, x_{kl}
\end{aligned}$$

We get a relationship between the $\frac{\partial E}{\partial sk}$ and those of the nodes in its upper layer, which makes the calculations easy to perform.

$$\frac{\partial E}{\partial s(k)} = g'(s(k)) \sum_{i \in UL(k)} \frac{\partial E}{\partial s(i)}\, w_{ik}$$

We can now write the algorithm as follows:

1. For an input X, calculate output $o(k)$ for all the nodes.

2. Calculate the value $\frac{\partial E}{\partial s(k)}$ for each node as :

$$\begin{aligned}
err &= y - o(p) \\
\frac{\partial E}{\partial s(k)} &= -err\; g'(s(k)) && \text{(for output node)} \\
\frac{\partial E}{\partial s(k)} &= g'(s(k)) \sum_{i \in UL(k)} \frac{\partial E}{\partial s(i)}\, w_{ik} && \text{(for hidden node)}
\end{aligned}$$

3. The weight changes are calculated as follows:

$$\Delta w_{kl} = \eta \frac{\partial E}{\partial s(k)} x_{kl}$$

4. Finally the weights are updated. If the weights are updated after seeing each sample it is called online training. If they are updated after each epoch i.e after seeing all the training samples once, it is called stochastic or batch training. For batch training the weight changes from all samples are summed up before updating the weight.

$$w_{kl}^{(t+1)} = w_{kl}^{(t)} + \Delta w_{kl}^{(t)}$$

3.4 Enhanced GLEM

A natural way to enhance the computing power of the model is adding networked non-linear nodes on top of the linear core like in an MLP. In order to keep the network size manageable we have to use intelligent ways of dealing with large number of input configurations. If the number of configurations is large it would be infeasible to view each configurations as a potential input node. We have to work at some level of abstraction over the configurations. Patterns described in Chapter 2 are good candidates for this. Since the number of active configurations for each pattern is finite we can conveniently carry out the optimization process by storing the network weights at the lowest level in a table with each table corresponding to one pattern. The table entries store the weights for different pattern configurations. In Figure 3.4 the left half shows the network structure without tables and the right half shows the structure with tables. Since the inputs are binary we can store the weights in the tables and during computation we can retrieve the corresponding weight for a particular input by indexing into the table. We can use the outputs from these pattern tables as inputs to the network above. This gives the new model more computing power because of the non-linear nodes in the top level network. At the same time the extra computational cost is minimal due to the largely linear nature of the lower layers where most of the computation is done.

The two important factors to be considered while using any training method are the speed of convergence (if it converges) and the quality of the solution. The quality of the solution or generalization performance can be evaluated by measuring the error on a test set. In the following sections we look at ways to improve the speed of convergence and the generalization performance.

3.5 Learning Rate and Momentum

The learning rate used in the learning process affects the speed and convergence of learning. For some learning rates the weight vector diverges, some are slow to converge while an optimal rate can result in convergence to a minimum in one step. In this section we will look at how to find an optimal learning rate. In general, the error E is not quadratic in the weights and is multi-dimensional. Let us simplify the problem by assuming a one-dimensional quadratic error function E. Then there exists an optimal learning rate η_{opt} than can be analytically calculated. For a learning rate $\eta < \eta_{opt}$, the weight update is smaller than the optimum and convergence requires multiple steps. For a learning rate $\eta_{opt} < \eta < 2\eta_{opt}$, the weight update is larger than required. As a result the weight will overshoot the minimum and oscillate about it. This also results in multiple steps to reach a minimum. For a learning rate $\eta > \eta_{opt}$, the weight update is so large that it results in a higher error than before the update. This results in divergence as shown in Figure 3.5.

3.5.1 Optimal Learning Rate

Let us assume a one-dimensional, quadratic error function E. Then we can write the gradient descent equation as

$$w^{t+1} = w^t - \eta\frac{\partial E}{\partial w}$$

We can expand E using a Taylor series around the current weight w_{cur}.

$$E(w) = E(w_{cur}) + (w - w_{cur})\frac{\partial E(w_{cur})}{\partial w} + \frac{1}{2}(w - w_{cur})^2 K$$

Note that since E is quadratic, higher-order terms are zero and the second derivative is a constant. Differentiating with respect to w on both sides gives

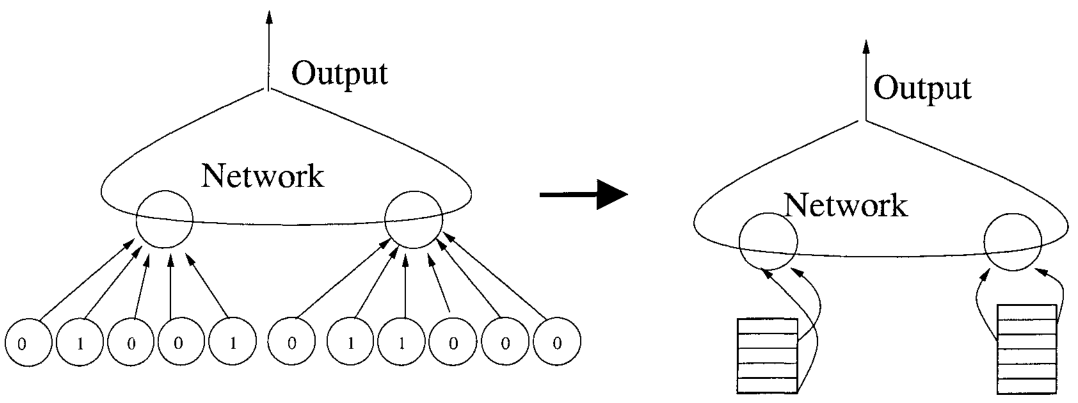


Figure 3.4: Enhanced version of GLEM

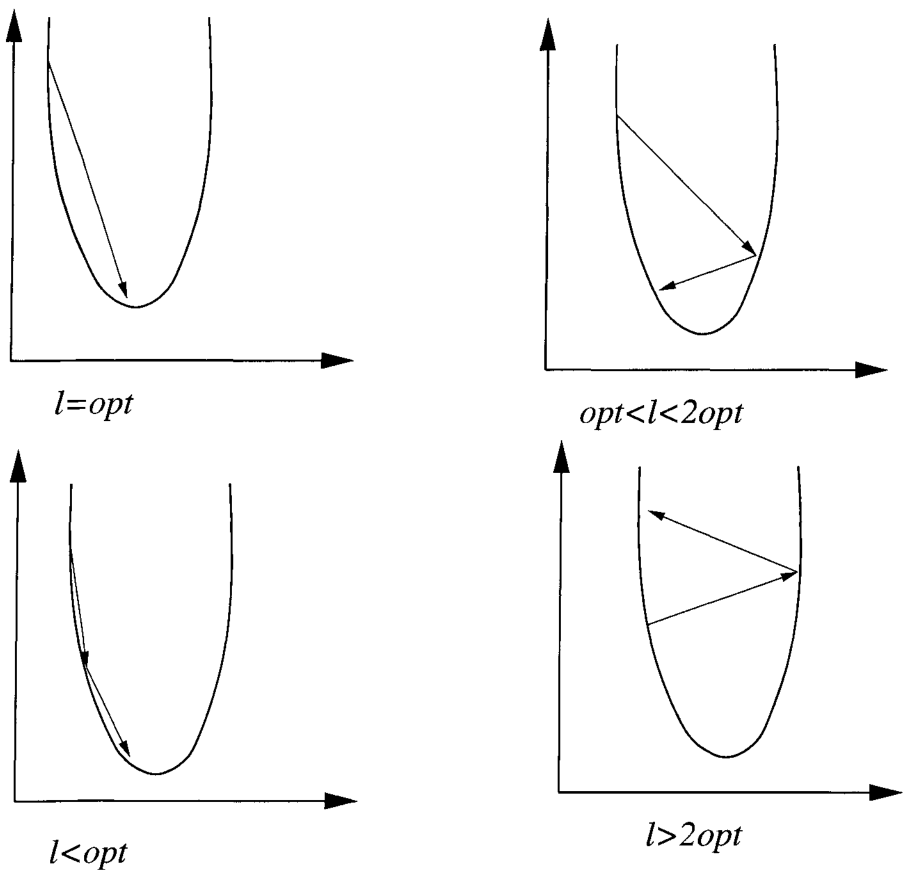


Figure 3.5: Different learning rates affecting speed of convergence

$$\frac{\partial E}{\partial w} = \frac{\partial E(w_{cur})}{\partial w} + (w - w_{cur})K$$

At the minimum point the derivative is zero. Substituting $w = w_{min}$ gives

$$\frac{\partial E(w_{cur})}{\partial w} + (w_{min} - w_{cur})K = 0$$

$$w_{min} = w_{cur} - \frac{1}{K}\frac{\partial E(w_{cur})}{\partial w}$$

Comparing this with the gradient descent equation yields

$$\eta_{opt} = \frac{1}{K} = \frac{1}{\frac{\partial^2}{\partial w^2}E(w_{cur})}$$

If E is not quadratic in the weight, then the higher-order terms may not be zero and the second derivative is not a constant. In such cases the η_{opt} calculated above is only an approximation and convergence is not achieved in a single step. However, it might serve as a good approximation if the higher-order terms are small. In practice the error function E is usually multi-dimensional. In such a case, determining n_{opt} for a quadratic error function requires more complex calculations.

We can expand the error function E around $\mathbf{W_{cur}}$ as follows

$$E(\mathbf{W}) = E(\mathbf{W_{cur}}) + \nabla E(\mathbf{W_{cur}})^T(\mathbf{W} - \mathbf{W_{cur}}) + \frac{1}{2}(\mathbf{W} - \mathbf{W_{cur}})^T H_{\mathbf{W_{cur}}}(\mathbf{W} - \mathbf{W_{cur}})$$

where $H_{\mathbf{W_{cur}}}$ is the Hessian matrix. In order to obtain a minimum in optimal number of steps we have to use different learning rates for different weights. It can be shown that the optimal learning rate for the i^{th} weight is $\eta^i_{opt} = \frac{1}{\lambda_i}$, where λ_i is the i^{th} eigenvalue of the Hessian matrix [LBOM98].

If we are constrained to choose a single learning rate then we must pick the smallest of all those learning rates so that we don't diverge.

$$\eta < \frac{2}{\lambda_{max}}$$

The ratio of the maximum to the minimum eigenvalues determines the number of steps required to converge. If the ratio is large then we will be taking small steps in the λ_{min} direction which actually requires large steps and as a result the convergence is slow.

3.5.2 Momentum

Another trick that is used to speed up learning is momentum [DPH86]. The previous weight change multiplied by a factor $0 < \mu < 1$, called *momentum*, is added to the current weight change to make the learning faster. The new weight update becomes

$$\Delta w_{t+1} = \eta \frac{\partial E}{\partial w} + \mu \Delta w_t$$

If a weight is oscillating around a minimum then its successive weight changes have opposite signs and momentum helps by decreasing the effective weight update and thus helps in reaching the minimum faster. If a weight is in a low curvature area, using momentum results in a larger update and thus helps move over the flat areas faster thereby speeding up convergence. The momentum factor μ controls the effect of the previous update in the present state. Having a large value for μ helps in covering the flat areas faster but might result in excessive oscillations in areas of high curvature. Conversely, having a small value for μ helps reduce the oscillations but is of not much use in flat areas of the error surface.

3.5.3 Adaptive Learning Rate

Calculating the optimal learning rate as described in the previous section is feasible if the error surface is known and the number of dimensions is small. Usually in real world problems, the number of dimensions is high and computing the Hessian is a cumbersome task. To tackle these problems we have to use a variable learning rate. The learning rate has to be adapted so as to make convergence faster. If we find that the current error is greater than the previous one, the error might be actually diverging and the learning rate should be decreased. If the error is decreasing then we can increase the learning rate as we seem to be moving in the right direction. There exist a number of different schemes for adapting the learning rate. A popular scheme is to multiply the learning rate by a factor $\eta^+ > 1$ when the error decreases in successive iterations and by a factor $\eta^- < 1$ when the error increases in successive iterations. It has been proved in [Lju77] that a learning rate that is asymptotically exponentially decreasing (i.e $\eta^{(t)} = \eta^{(0)} \cdot p^{-t}, 0 < p < 1$) is sufficient to guarantee convergence. Another scheme is to use a running average of the form $\eta^{(t)} = \frac{\eta^{(0)}}{1+t}$. This learning rate scheme performs well in the final stages. Although it guarantees convergence, it is slow in the beginning as the learning rate is decreased very drastically. Using a larger value for the initial rate can lead to instability in the beginning. It has been shown in [Gol87] that an learning rate that is asymptotically of the form $\eta^{(t)} = \frac{constant}{t}$ achieves convergence at an optimal rate.

Based on the above observations [DM90] proposed the following scheme known as Search-Then-Converge schedule.

$$\eta^{(t)} = \frac{K}{C+t}$$

$$\eta^{(t)} = \eta^{(0)} \frac{1}{1+\frac{t}{C}}$$

where K and C are constants and t is the current epoch number. The idea behind this scheme is to keep the learning rate high initially (search phase, $t \leq C$) so that the weights reach close to a minimum in this phase. In the next phase (converge phase, $t > C$), the learning rate decreases almost inversely proportional to t and the parameters converge at a near optimal rate. A more advanced Search-Then-Converge schedule presented in [DCM92] automatically finds the parameter C.

3.6 Over-fitting

The problem of over-fitting is encountered in most optimization processes when data is not accurate. Over-fitting occurs when the optimization process fits not only the output but also any errors that may have been present in the output. For example, some outputs could be erroneously labelled and the optimization process fits the data to those erroneous outputs as well. The consequence of over-fitting is poor generalization. When such a system is tested for performance on a test set not used in optimization, the performance of the system can be seen to degrade after a certain point. Over-fitting can also occur as a result of an overly complicated system trying to fit a simple function. The converse case of over-fitting is under-fitting where the system is not complex enough to capture the relation between input and output.

Over and under-fitting can be explained in terms of statistical bias and variance. Statistical bias is the difference between the mean value and the true value. Statistical variance is the variance of the function with respect to its mean. Let is consider the following setup where $f^*(x)$ is the true unknown function to be approximated, $D = (x_i, t_i)|i = 1, \cdots, N$ be the sequence of data points given for training where $t_i = f^*(x_i) + e_i$. Let us assume the e_is are normally distributed with zero mean and a variance σ^2. Let $y(x)$ be the function that is produced after training the model over D. We use y_i to denote $y(x_i)$. We can write the mean-squared error as

$$E = \frac{1}{N}\sum_{i=1}^{N}(t_i - y_i)^2$$

The true generalization error of this approximate function y is given by the expectation of the mean-squared error E.

$$\begin{aligned}
\langle E\rangle &= \frac{1}{N}\sum_{i=1}^{N}\langle(t_i - y_i)^2\rangle \\
\langle(t_i - y_i)^2\rangle &= \langle(t_i - f_i + f_i - y_i)^2\rangle \\
&= \langle e_i^2\rangle + \langle(f_i - y_i)^2\rangle + \cancel{2\langle e_i\rangle\langle f_i - y_i\rangle} \quad \text{(Since } \langle e_i\rangle = 0) \\
&= \sigma^2 + \langle(f_i - \langle y_i\rangle + \langle y_i\rangle - y_i)^2\rangle \\
&= \sigma^2 + \langle(f_i - \langle y_i\rangle)^2\rangle + \langle(\langle y_i\rangle - y_i)^2\rangle + \cancel{2\langle f_i - \langle y_i\rangle\rangle\langle\langle y_i\rangle - y_i\rangle} \\
\langle\langle y_i\rangle - y_i\rangle &= \langle y_i\rangle - \langle y_i\rangle = 0 \\
\langle(t_i - y_i)^2\rangle &= \sigma^2 + bias^2(y) + variance(y)
\end{aligned}$$

In other words, the generalization error is the sum of error variance, variance of the model and square of the model bias. An under-fitting model has a high bias and an over-fitting model has high variance both resulting in a high generalization error. This problem is often referred to as bias-variance trade-off. As the model starts fitting the data both bias and variance reduce. After a certain point the bias decreases and variance increases as the function becomes more complex. A function with high curvature has a high variance as the function value changes rapidly. Therefore, the problem of obtaining a model which has good generalization can be viewed as a trade-off between the bias and variance of the model.

The following example illustrates the concepts of over-fitting and under-fitting using polynomial fitting. The points were generated using a quadratic $f(x) = 0.4x^2 + 0.86x - 3$ and a normally distributed error with 0 mean and standard deviation 10 was added. Then these points were fitted with various polynomials starting with a linear fit (which clearly under-fits as it is not complex enough to capture the data) till a polynomial of degree five where all the points are exactly on the curve. The polynomials are shown in Figure 3.6.

The resulting functions were tested against a set of 5 perfectly labelled points $\{(-2, -3.12), (-1, -3.46), (0, -3.0), (1, -1.74), (2, 0.32)\}$. Figure 3.11 shows the l_2 test errors of these functions against this set. As can be seen from the figure functions f_2, f_3 are the best approximators as expected. The linear function f_1 under-fits while f_4, f_5 over-fit. f_5 has high variance as the function value changes quite rapidly over short distances in x direction. Consequently, it results in poor generalization as seen in Figure 3.11.

3.7 How to Overcome Over-fitting?

There are ways to overcome the problem of over-fitting. The easiest way is by having more training data. This method works when the model is fixed and there is a way to get more data. But it is not always possible to generate more data easily, for example in medical applications. Also, in some cases the chosen model might be dependent on the amount of data available. Having more data results in choosing a more complex model which has higher expressive power. For example, in the case of Othello, having more data would allow us to use larger patterns, which could result in a better evaluation function. But these larger patterns also require larger amounts of data and we end up having almost the same ratio of data and parameters. Other approaches to avoiding over-fitting when having a fixed amount of training data are described in the next sections.

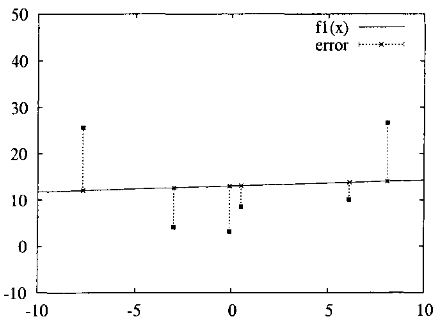


Fig. 3.6: $f_1(x) = 13.026 + 0.126x$

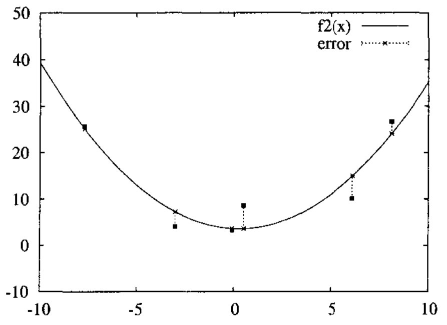


Fig. 3.7: $f_2(x) = 3.60 - 0.204x + 0.337x^2$

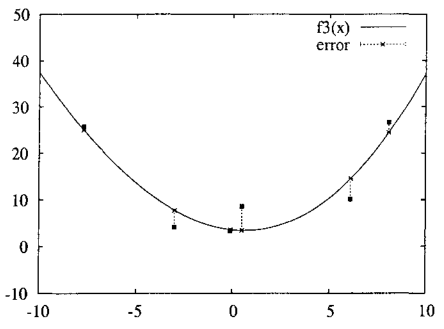


Fig. 3.8: $f_3(x) = 3.54 - 0.427x + 0.338x^2 + 0.004x^3$

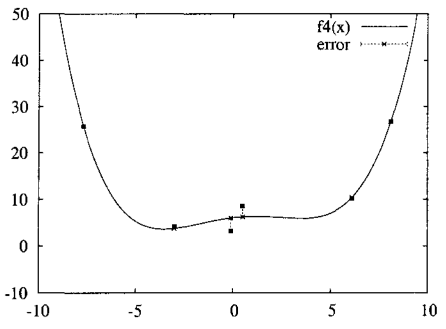


Fig. 3.9: $f_4(x) = 6.03 + 0.475x - 0.21x^2 - 0.012x^3 + 0.008x^4$

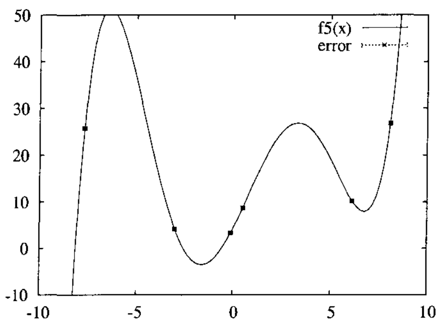


Fig. 3.10: $f_5(x) = 4.16 + 8.34x + 1.41x^2 - 0.588x^3 - 0.019x^4 + 0.007x^5$ over-fits the data

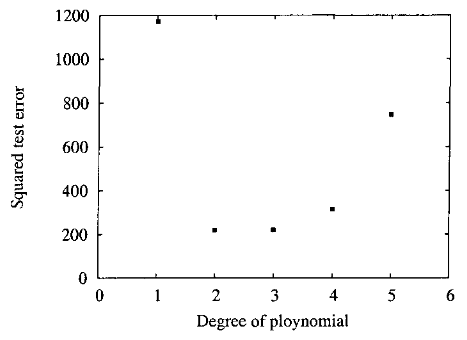


Fig. 3.11: Test error of the functions f_1, f_2, f_3, f_4, f_5

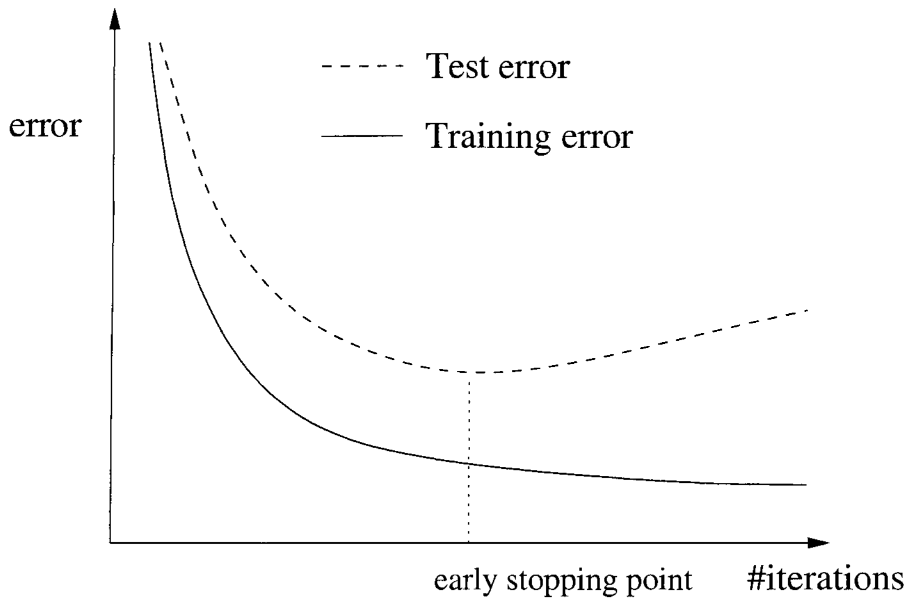


Figure 3.12: Early Stopping

3.8 Early Stopping

Early stopping[Pre98] attacks at the core of over-fitting. If we assume that the test set is representative of the data that would be actually encountered then the performance on the test set reflects the generalization performance closely. Therefore, the model has the best generalization when the error on the test set is minimal. Early stopping suggests to stop the training when the test error begins to increase as indicated in Figure 3.12. Applying early stopping to the example in Figure 3.11 suggests that the best function to approximate the unknown function is f_2, a quadratic as expected. Early stopping also helps in reducing the training time. There exist many empirical approaches to early stopping [Pre98]. For instance, instead of taking the error of one iteration we could average over a window period of k iterations. Another approach is to stop training only when the ratio of the current error to the minimal error found exceeds a predetermined threshold. However, empirical studies [Pre98] have shown that these complex stopping criteria did not result in significant ($\sim$ 4%) improvement in generalization error while being much slower (4 times) compared to the simple method. Early stopping does not change the way the optimization process is done, it is a passive way of avoiding over-fitting. There are other methods to overcome over-fitting by changing the error function. These methods are discussed in the following sections.

3.9 Regularization

Regularization is a penalty-based method for overcoming over-fitting. In these methods a penalty term R (also called regularizer) is added to the error function E to form a new error function E_R.

$$E_R = E + R$$

The idea behind regularization is that the penalty term captures the causes of over-fitting. Thus, by adding a penalty term to the error the optimization process would minimize the penalty term too thereby minimizing over-fitting. Usually, over-fitted models have high curvature in some areas (in order to fit the error associated with the data closely) which requires large feature weights. For example, in the polynomial fitting discussed above f_5 has the highest L_2 norm among the five. Therefore, by restricting the weight, we can reduce the curvature and hope to reduce over-fitting. By adding the norm of the weight vector as a penalty term to the error function we can hope to avoid large weights. This is also known in literature as *ridge regression* or *weight decay*.

$$E_R = E + \alpha \cdot ||W||^2$$

Then the gradient descent equation becomes

$$\Delta w^{(R)} = \Delta w - \alpha w$$

where $\Delta w^{(R)}$ is the weight change in the regularization algorithm Δw is the weight change in the usual algorithm and α is the regularization parameter. As the regularization algorithm is equivalent to removing the actual update by a portion of the current weight, it is also called weight decay.

3.10 TRI : A Metric-Based Approach to Model Selection

A different approach to model selection to counter over-fitting has been discussed in [Sch97]. This approach assumes that the model spaces from which a model has to be selected form a nested sequence; that is, $H_0 \subset H_1 \subset H_2 \cdots$ see Figure 3.13. Then choose the best approximator of the given samples from each class $h_0^*, h_1^*, h_2^*, \ldots$. Finally, use some measure to pick one function from these. The idea

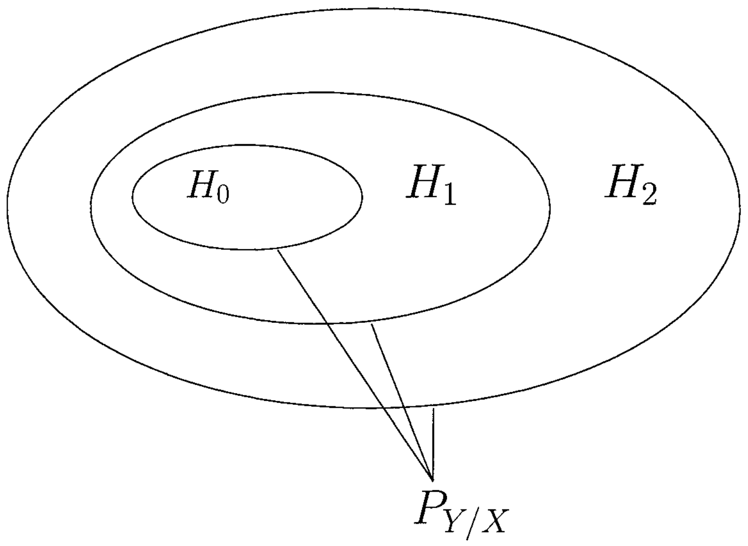


Figure 3.13: Geometric view of model selection.

is as follows: The training and test samples can be assumed to be random observations drawn from the joint probability distribution P_{XY} of samples $(X) \times$ labels (Y). This distribution can be decomposed into the conditional distribution $P_{Y/X}$ (which we are trying to learn) and the marginal distribution P_X. Using P_X, a distance measure between two hypotheses can be defined as the average discrepancy of the hypotheses on random x-objects. We can also extend this distance definition to the condition distribution $P_{Y/X}$. It can be noted from Figure 3.13 that the sequence of empirically closest functions have decreasing distances to the target. Now, we know the distance of each of the empirically closest functions $h_0^*, h_1^*, h_2^* \dots$ to the target conditional and also the distance between the functions in the sequence. Using this information we can detect over-fitting. The intuition is that if two successive functions in the sequence are both individually close to the target but have large distance between them, then it can be seen from simple geometry that one of them is wrong. We choose the one which has lower complexity, i.e the earlier one. The rule can be summarized as : "choose the last function in the sequence that does not violate the triangle inequality with any of the preceeding functions".

This method is reported to work well when choosing a model from a hirerachy of models. In our case, we don't know how training iterations correspond to complexity of the model.

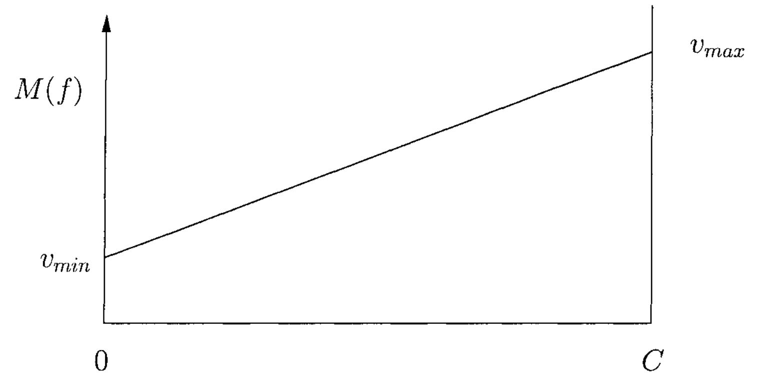


Figure 3.14: Clamping using a ramp function. The absolute value a feature weight can take is clamped by a function $M(f)$ of the feature frequency f.

3.11 Clamping

We propose another way of looking at over-fitting by concentrating on rare features. Over-fitting is also caused by having rare features in samples with errors (by 'rare' we refer to the frequency of occurrence in the training set). The intuition here is that when these rare features appear in training and the label has error, a lot of the error is blamed on these features. Rare features by definition being infrequent would have shared a large portion of the errors by the time training is done. When these rare features appear in the test set they produce bad predictions causing over-fitting. Clamping reduces this error by restricting the value a weight can take depending on the number of occurrences. The absolute value a weight can take is clamped by the clamping function $M(f)$, where f is the frequency of the feature. In Figure 3.14 the weights for features appearing at most C times are clamped by a ramp function running from value v_{min} to v_{max}. We call the value C *critical number*. Therefore, the maximum absolute weight, $M(f)$, attributed to a feature appearing f times is

$$M(f) = \begin{cases} v_{min} + \frac{f}{C}(v_{max} - v_{min}) & \text{if } f \leq C \\ \infty & \text{for } f > C \end{cases}$$

The ramp function used above is just one example. The function that is used for clamping should be monotone and should approach ∞ for large values. The intuition behind these requirements is that a feature appearing more frequently will probably acquire a more accurate weight after the training is done and therefore it should be less constrained which can be achieved by having a higher clamp value. This requires the function to be increasing. Furthermore, if a feature occurs frequently in training it is probably best not to constrain it. This implies the clamping function should approach ∞ for large values. Other functions that satisfy the above properties can used instead of the ramp function depending on the domain. It is not known whether there exists an optimal clamping function which results in the best generalization error. Experimental results in Chapter 4 show that clamping is a viable alternative to be considered while training.

3.12 Weighted Regularization

Weighted regularization combines the ideas of clamping and regularization. Here the weights that are added to the penalty term are weighted by a factor inversely proportional to the frequency of their occurrence. The error function E for weighted regularization is given by:

$$E_{WR} = E + \sum_{i} f(i) w_i^2$$

where $f(i)$ is a function of the frequency of feature i. When all the features have similar frequencies, weighted regularization approaches regularization. By choosing a function that has a higher value for lower frequency features we are forcing the corresponding weight of that feature to be small. This idea is similar in spirit to the idea behind clamping. The exact form of the function $f(i)$ that results in an optimal generalization performance is not known.

Clamping and regularization are independent of early stopping. So these methods can be applied in conjunction with it. Early stopping can be viewed as a passive way of countering over-fitting while clamping and regularization can be viewed as active approaches.

3.13 Rare-Feature Sets

We propose another way of detecting over-fitting in Generalized Linear Evaluation Models based on feature frequency. We define a rare-feature set RS_k as a multiset of data points selected from a test set. The criterion for selection is that at least k features of a sample must be rare (defined with respect to frequency of occurrence in the training set).

$$\forall (x_i, y_i) \in T \;\; (x_i, y_i) \in RS_k \text{ iff } rc(i) \geq k$$

where $rc(i)$ is the number of rare features in sample i and T is a test set. We define $RS_k(c)$ to be a rare-feature set with k rare features and the frequency of rare

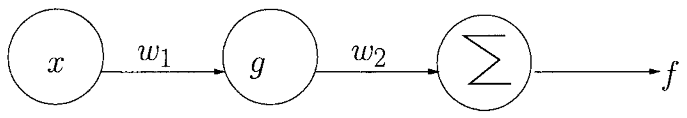


Figure 3.15: Function to be learnt, represented as a network

features being at most c. Though we use the set notation in the above definition, we mean multisets. Our claim is that the model prediction for these samples worsens as it starts to over-fit. Focusing solely on the RS_1 results in pronounced over-fitting effects. Rare-feature sets can be used when the test set does not show over-fitting but the performance in real world worsens and over-fitting is suspected. Even pronounced over-fitting can be observed by focusing on $RS_k, k > 1$. Using rare-sets we have been able to detect over-fitting in a GLEM based model for optimizing Othello weights, which did not show over-fitting on a usual test set. The over-fitting effect on the rare-featured samples was washed out by the small improvement on a large number of other samples.

3.14 Top Multiplier

One of the problems we have encountered during our experiments was the inability to train the top level parameter which we call top multiplier. A top multiplier is used when the range of the function computed is different from the range required. For instance, in a network when a tanh function is used at the top of the network, the range of the function is $(-1, 1)$. But if this network has to be used to train weights for an application like Othello in which the sample labels are in the range $[-64, 64]$ a multiplier is needed. Another way to deal with this problem is to normalize the data so that the range is between $(-1, 1)$.

Thus far we have inherently assumed the error function to be convex in the weight space. If the error function is not convex there might be multiple minima or no minimum at all. It would be difficult to find a weight vector which minimizes the error using gradient descent. In this section we determine whether the error is actually convex for a simple network with a top multiplier as shown in Figure 3.15 and if it has multiple local minima. For example, we are trying to learn a function of the type

$$f(x) = w_2 g(w_1 x)$$

where g is a monotonic function (assume increasing function) like $tanh(x)$ or $\frac{1}{1+e^{-x}}$.

Given a data sequence $D = \{(x_i, y_i)\}$, the problem is to find a weight vector $w =< w_1^{opt}, w_2^{opt} >$ that minimizes the l_2 error.

$$Err(w) = \frac{1}{2}\sum_i (y_i - f(x_i))^2$$

If the error is convex in the parameter w_1, w_2 then there exists a vector $< w_1^{opt}, w_2^{opt}$ which corresponds to a minimum of the error surface and thus can be learnt using gradient descent algorithm. The second derivative test for multiple dimensions involves checking if the Hessian is positive definite. For the sake of brevity we use g_i to denote $g(w_1 x_i)$.

$$\frac{\partial E}{\partial w_2} = \sum_i (w_2 g_i - y_i) g_i$$

$$\frac{\partial E}{\partial w_1} = \sum_i (w_2 g_i - y_i) \frac{\partial g_i}{\partial w_1}$$

Let $\delta_i = f(x_i) - y_i$

$$\frac{\partial^2 E}{\partial w_2^2} = \sum_i g_i^2$$

$$\frac{\partial^2 E}{\partial w_1^2} = \sum_i (w_2 g_i - y_i) \frac{\partial^2 g_i}{\partial w_1^2} + w_2 (\frac{\partial g_i}{\partial w_1})^2$$

$$= \sum_i (\delta_i) \frac{\partial^2 g_i}{\partial w_1^2} + w_2 (\frac{\partial g_i}{\partial w_1})^2$$

$$\frac{\partial^2 E}{\partial w_1 \partial w_2} = \sum_i (2 w_2 g_i - y_i) \frac{\partial g_i}{\partial w_1}$$

$$= \sum_i (w_2 g_i + \delta_i) \frac{\partial g_i}{\partial w_1}$$

Since the Hessian is a symmetric matrix, we can now write down the Hessian.

$$H(w_1, w_2) = \begin{pmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} \\ \frac{\partial^2 E}{\partial w_2 \partial w_1} & \frac{\partial^2 E}{\partial w_2^2} \end{pmatrix}$$

Assuming the δ_i's are negligible and substituting the values we get

$$H(w_1, w_2) = \begin{pmatrix} \sum_i w_2^2 (\frac{\partial g_i}{\partial w_1})^2 & \sum_i w_2 g_i \frac{\partial g_i}{\partial w_1} \\ \sum_i w_2 g_i \frac{\partial g_i}{\partial w_1} & \sum_i g_i^2 \end{pmatrix}$$

The determinant of $H(w_1, w_2)$

$$||H|| = w_2^2 \sum_i \left(\frac{\partial g_i}{\partial w_1}\right)^2 \sum_i g_i^2 - w_2^2 \left(\sum_i g_i \frac{\partial g_i}{\partial w_1}\right)^2$$

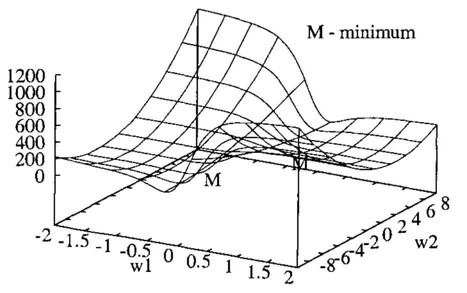


Figure 3.16: Error surface with 2 minima

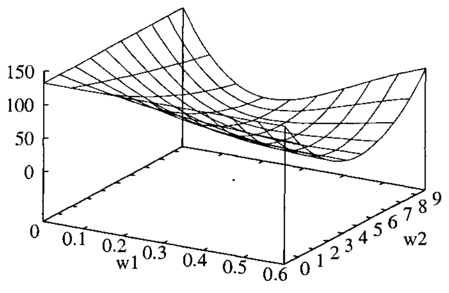


Figure 3.17: Error surface in one half of the space

Using the identity

$$\sum_i a_i^2 \sum_i b_i^2 \geq \left(\sum_i a_i b_i\right)^2$$

(which can be easily proved using induction) we can show

$$||H|| \geq 0$$

Therefore, we have proved H is positive semi-definite when the errors are small. The error function for the data sequence

$$D = \{(0.5, 0.53), (1.5, 1.48), (2.5, 2.15), (3.5, 2.55), (-5.5, -2.89), (-9.0, 2.99)\}$$

and function

$$f = w_2 \tanh(w_1 x)$$

is shown in the Figure 3.16. The function has two minima as the function can be written as

$$f = w_2 \tanh(w_1 x) = -w_2 \tanh(-w_1 x)$$

Therefore, for every minimum located at (w_1, w_2) there is a corresponding minimum at $(-w_1, -w_2)$. If we restrict the weight vector to one half of the space then the function is convex as seen in Figure 3.17 and the minimum is at $(0.36, 3)$.

The above simple example shows that there exist multiple minima and the error surface is convex only close to a minimum. The weight vector did not converge when the starting weights were in between the minima. We have experienced a similar problem while trying to learn the top multiplier. The model converged when the top multiplier was fixed or kept almost fixed by using a very small learning rate. With a fixed top multiplier the error surface became convex and we were able to learn the weights to get to a minimum.

Chapter 4

Software, Experiments, and Results

In this chapter an overview of the software and experimental setup is presented and the results are discussed. We have developed software that is domain-independent and can be used for training weights for any GLEM. The software [CB05] can also be used as a neural network simulator by just using the numerical features and omitting tables. The GLEM optimization software consists of two major parts - *optimizer* (the domain-independent part) and *application* (the domain-specific part). The application provides the interface to read the data in the application format and to present it to the optimizer in a format that the optimizer recognizes. The input to the optimizer consists of a set of table indices, label or output of the sample and the index of the evaluation function (t_indices, label and e_index respectively as shown in Figure 4.1), as there could be more than one evaluation function that are being fitted simultaneously. Using command line options, the user can specify the evaluation function to be used, the algorithm to be used for optimization and other parameters. The application provides the optimizer with the evaluation function definition and the input as shown in Figure 4.1. The application in turn gets the evaluation function definition form the user or a function that can be incorporated into the application.

4.1 Evaluation Function Definition Format

The evaluation function definition consists of two parts — topology and features. Topology refers to the structure of the non-linear network on top of the linear features. Features refer to the tables and the numerical features of the function. The lines beginning with # are comments and are ignored by the software. The format of the file is as follows

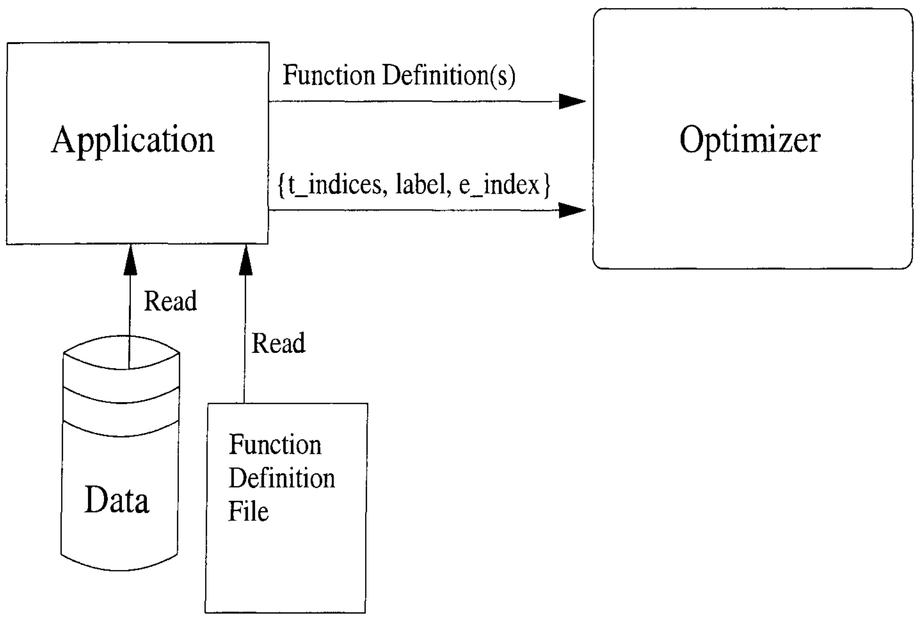


Figure 4.1: Overview of the data flow in the optimization process

```
;TOPOLOGY
<topology>
;FEATURES
<features>
#Comments can occur any where in the file
```

4.1.1 Topology

In the topology section the structure of the network is specified. Each line corresponds to one node and has the node number, type and the list of its children. A node i is called the child of node j if the output of i is a direct input of j. Each node is identified by a unique number assigned to it.

There are two types of nodes — *accumulator* nodes and *activation* nodes. An accumulator is a multi-input single output type of node while an activation node is a single input single output node. Currently, there is only one type of accumulator node — *summation*, for which the output is the sum of its inputs. Other types of accumulator nodes – like an averaging node or a multiplicator node – can be implemented within this framework with minimal effort. There are three types of pre-defined activation nodes, namely - *sigmoid*, *tanh* and *identity*. The outputs for these nodes are the sigmoid, tanh and identity functions applied to the inputs respectively. Other activation nodes such as the simple threshold and sign functions can be added easily. The symbols in the definition file for these nodes are as follows: 'sum' for summation, 'sig' for sigmoid, 'tnh' for tanh, 'ide' for identity.

The input to an activation node is always from the output of an accumulator node. However, an accumulator node can have inputs from activation nodes, tables or numerical features. Since back propagation involves calculating derivatives these functions have to be differentiable in the domain they are used.

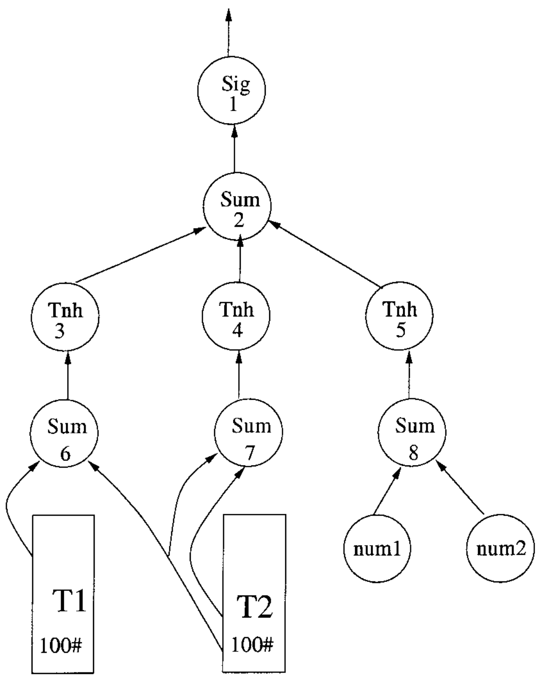
4.1.2 Features

The other major section of the function definition is the features. Features represent the configurations in GLEM. Since patterns are used, the features can be grouped together conveniently into tables. There are two types of features — *tables* and *numerical features*. Tables store pattern configuration weights. Numerical features are not restricted to binary values and thus can take any real value.

Tables

A table definition starts with the letter 'T' indicating it is a table, followed by the table name, number of transformations and the size of the table (number of configurations). A transformation is obtained by applying a symmetric transformation (as defined by the application) to the application data. For example, in Othello the symmetries are defined by geometry. Therefore, the transformations here are mirroring and rotation. A pattern consisting of a 3×3 top corner in Othello thus has 4 transformations. Hence, we can treat the 4 corners as being geometric transformations of the same instead of viewing them as 4 different tables. Transformations help to reduce the size of tables. The lines following the table definition line contain the names of transformations and the node each connects to. It is possible for different transformations to connect to different nodes in the topology.

A numerical feature line starts with the letter 'N' and is followed by its name and the node it connects to. A small example covering most of the details is presented below. The evaluation function definition file and the corresponding structure are shown in the Table 4.1. The numbers on the nodes in Table 4.1 correspond to those in the text definition.



```
;TOPOLOGY
1 sig 2
2 sum 3 4 5
3 tnh 6
6 sum
4 tnh 7
7 sum
5 tnh 8
8 sum

;FEATURES
T T1 1 100
   T1-Transformation-1 6

T T2 2 100
   T2-Transformation-1 7
   T2-Transformation-2 7
   T2-Transformation-2 6

N numerical-feature-1 8
N numerical-feature-2 8
```

Table 4.1: Network corresponding to the definition

4.2 Optimization Process

The optimization process is started by calling the optimizer's run method from within the application. The optimizer contains a pointer to the application. The run method gets the evaluation function from the application and creates an iterator for reading the data as shown in Figure 4.2.

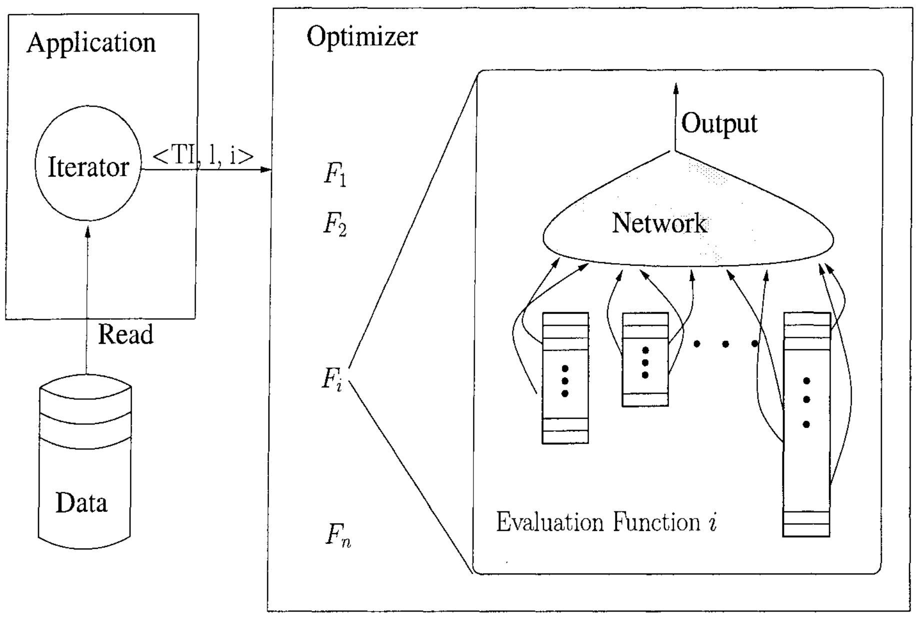


Figure 4.2: Overview of the Optimization Process

4.2.1 Iterator

The interface between the optimizer and the application is minimal. They communicate mostly through a data structure called an *iterator*. An iterator is a data structure that allows the optimizer to read data which is present in the application format. In other words, the iterator abstracts the application details from the optimizer. An iterator reads the data from the source and outputs it to the optimizer in the form of a vector containing table indices and the corresponding sample label. The iterator also indicates the evaluation function index and returns false upon reaching the end of sample sequence when called to get the next sample.

One optimizer can fit more than one evaluation function simultaneously. In fact, it is recommended to do this in some cases. For example, in LOGISTELLO each game stage (defined by the number of disks on board) has its own evaluation function. However, to provide smooth transitions between game stages, data from adjacent game stages are also used while training. Having the facility to train multiple functions simultaneously makes it possible to train all the evaluation functions simultaneously using the same data sequence. Otherwise, data has to be split into multiple sets which results in unwanted duplication of data. The problem becomes more relevant when the size of the data sequence increases. A command line parameter tells the Optimizer to interpolate the data between stages. A value of n results in a sample being interpolated to n stages each on either side of the the actual stage. This increases the effective amount of data roughly by a factor of $2n$. The learning rates, momentum, number of iterations and other details of the optimization process can be given as command line parameters.

4.3 Experimental Setup

Using the software that was described in the previous section we ran experiments to test the performance of the algorithms described in Chapter 3 on Othello data and on synthetic data. We also tested the performance on rare-feature sets. For each run the performance was tested against five test sets. The test sets were of two types. In *plain* test sets the frequency of features follows the same distribution as in the training set. *Rare* test sets are created from *plain* test set by extracting samples which contain rare features that also appear in the training set. Rarity is defined by the frequency of occurrence of features in the training set. Using different thresholds for rarity results in different rare-feature sets. We have used four thresholds for rarity to extract four rare-feature sets in each of the applications.

4.4 Synthetic Data

In this section we describe our experiments with synthetic data sequences. We used two different evaluation functions M1 and M2 as shown in Figure 4.3.

$$\begin{aligned} f(i,j,k) &= t_1[i] + t_2[j] + t_3[k] \quad \text{for M1} \\ f(i_1,i_2,j_1,j_2,k_1,k_2) &= \tanh(t_1[i_1] + t_1[i_2]) + \tanh(t_2[j_1] + t_2[j_2]) + \\ &\quad \tanh(t_3[k_1] + t_3[k_2]) \quad \text{for M2} \end{aligned}$$

First, a model was chosen and then random configuration weights were stored in the tables. These random values were generated using the rand() function of C and were uniformly distributed between $[-1, 1]$. The few network weights were set to fixed values. A random normal noise with mean 0 and standard deviation σ was

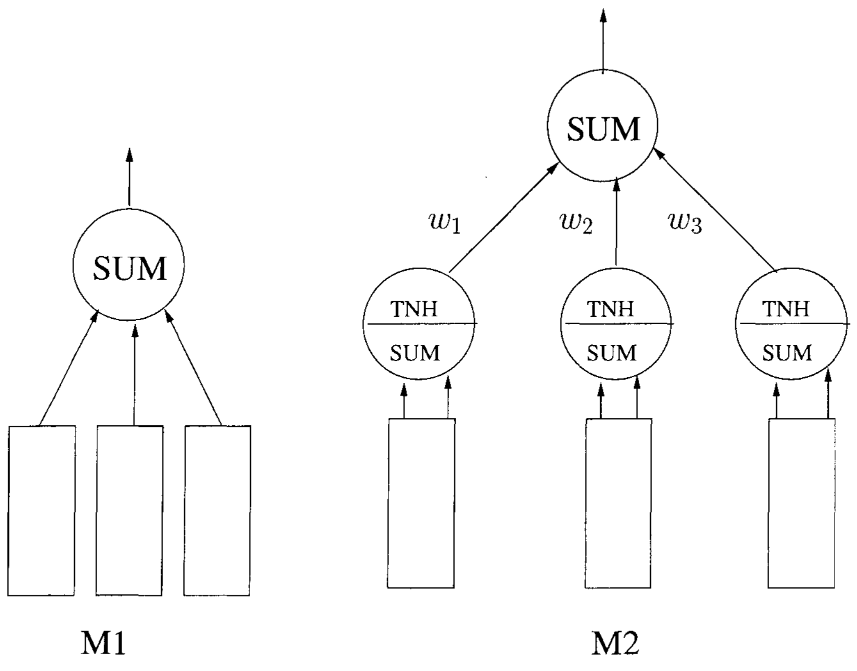


Figure 4.3: Different models used for the synthetic data experiments

added to the output to account for labeling errors. Box-Müller transform [Wei99] was used to generate a normal distribution. We assume there is no error in the input variables. The functions used for generating data can be written as

$$y = f + N(0, \sigma^2)$$

The index generation scheme is described later in this section. Then the same evaluation model that was used to generate the data was used to fit this data. The table weights were all set to zero before starting the optimization process. The network weights were all set to 1.0 and bias weights were set to 0.01. We used three tables containing 100 configuration weights each. Model M1 had one transformation per table while model M2 has two transformations.

Experiments were performed with these two models each with two values of σ and for each of the settings 100 data sequences were created. Each data sequence consisted to 1000 training samples, 100 test samples (which was used for performing early stopping) and 2000 validation samples which were used to report the final generalization error. Another dimension in the set of experiments was the presence of rare features. In one setting, 80% of the features were rare and each rare feature occurred about four times on average in the training set. In another setting only 20% of the features were rare and each rare feature appeared about seven times on an average. The algorithms were run on these eight settings (each with 100 data sequences) and their performance was tested against five test sets for each run. The experiments done are shown in a tree format in Figure 4.4. The numbers at the leaves represent the number of different settings tried for each algorithm.

Index Generation

The synthetic data was generated in the following manner. There were two different settings for rarity and percentage of rare features chosen. The first setting had 80% rare features and the rare features appeared on an average about four times in the training set. The second setting had 20% rare features and the rare features appeared on an average about seven times in the training set. There exists a relation between the percentage of rare features p, frequency of occurrence of a rare feature k, frequency of occurrence a non-rare feature l, the number of weights in the table n_t and the total number of samples n. The average frequency is given by $\frac{n}{n_t} = pk + (1-p)l$.

$$l = \frac{1}{1-p}(\frac{n}{n_t} - pk)$$

The specified percentage of rare features is achieved in the following way. An array A of size n_t is created and $A[i]$ is set to k for $i \leq pn_t$ and $A[i]$ is set to l for $i > pn_t$. Another vector B of size n_t is created such that

$$B[i] = \sum_{j=1}^{i} A[j]$$

Then a random number r is generated between 1 and $B[n_t]$. This random number r is used to generate an i such that $B[i] \leq r < B[i+1]$. The generated index i is a close approximation of the distribution of configuration indices desired. Using the above method data for eight different settings was generated. The set of experiments is shown in a tree form in Figure 4.4.

The parameters for learning rates and momentum were chosen by experimentation on a randomly chosen data sequence of size 1000. The learning rate and momentum parameters were found by varying both of them simultaneously. First, a reasonable range of learning rate was obtained by not taking momentum into account. This was done as we already have some idea about the range of the momentum factor which is usually between 0 and 1[DPH86]. Then the learning rate was varied by a step size of one tenth of the range and momentum by 0.1 and various pairs of values were tested. Finally, the pair which converged fastest was chosen. In case of complex models where a multilayer network is present on top of the tables

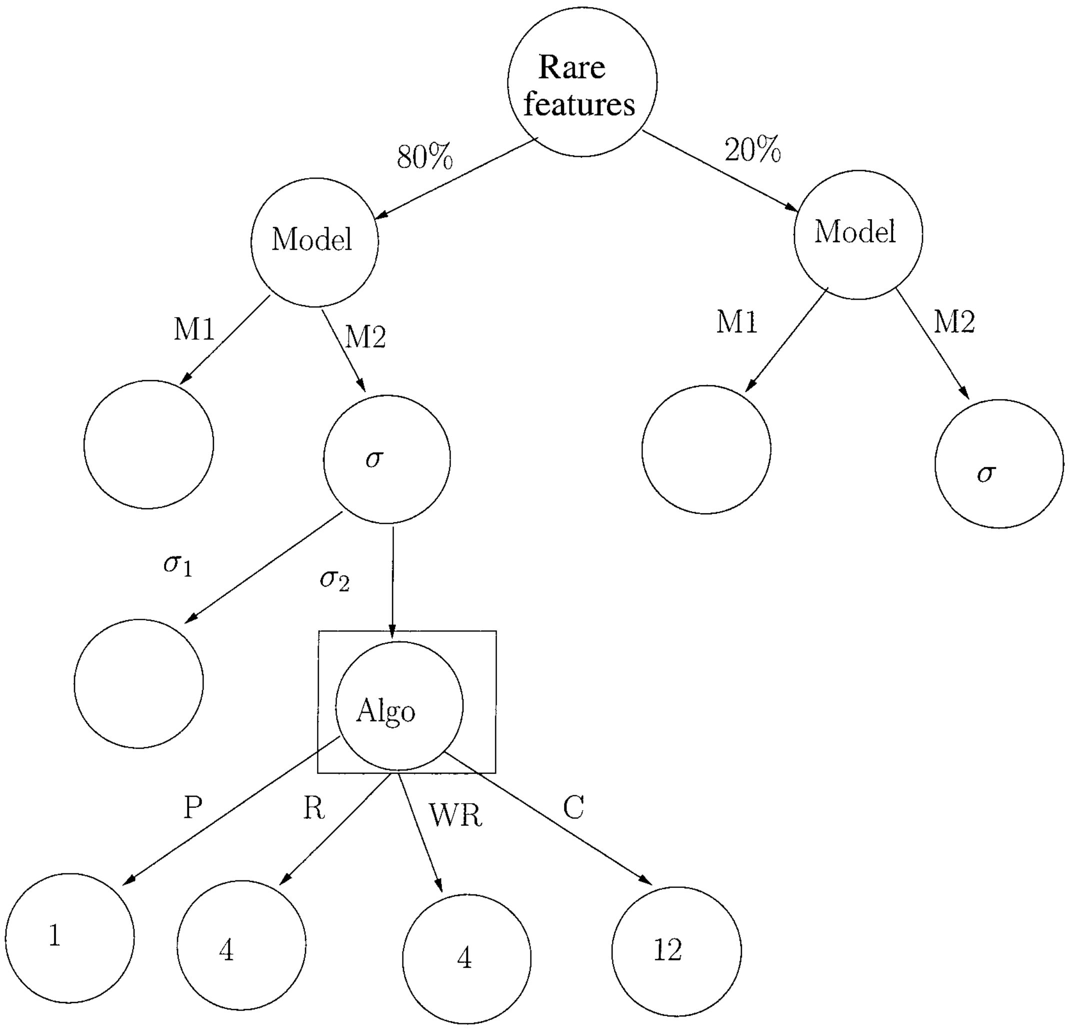


Figure 4.4: The set of experiments depicted in a tree form. The boxed node represents one of the eight experimental settings. The leaves of the tree represent the experiments performed and the number indicates the number of different settings tried. P - plain, R - Regularization, WR - Weighted Regularization, C - Clamping. M1 and M2 are different evaluation models and σ_1, σ_2 are the standard deviations of the zero-mean normally distributed errors that are added to give a noisy label.

we used two learning rates. One learning rate η for the lower layers (tables) and a different learning rate η_t for the topology part. In order to avoid recomputing the parameters for learning rate and momentum every time the size of the training set changes, we use the average error instead of total error. This helps to keep the learning rate independent of the size of the training set. One disadvantage of averaging the error over all the samples is that features that are relatively less frequent are learnt slower. We overcame this problem by dividing the error by feature frequency instead of the training set size. The learning rate and momentum parameters that are used in one particular setting are reported in Table 4.2.

Model	η	η_t	μ
M1	2.0	-	0.1
M2	5.0	2.0	0.1

Table 4.2: Learning rate and momentum values used in synthetic data experiments.

4.5 Results

In this section we provide empirical evidence of our claims for cause of over-fitting, performance against rare-feature sets and compare the clamping technique we proposed against regularization. Henceforth, unless explicitly specified otherwise, by test set performance we mean the value obtained by applying early stopping in conjunction with the algorithm being used.

4.5.1 Effects of Rare Features and Rare-Feature Sets

Over-fitting is higher when there are more rare features. In, other words a data sequence with a higher percentage of rare features is more prone to over-fitting. This can be seen in Figure 4.5. The data sequence with 80% rare features exhibits over-fitting while over-fitting in the data sequence with 20% rare features is negligible. Though there is over-fitting in the 20% case too, it is not visible. However, even a small amount of over-fitting can be detected by using an appropriate rare-feature set. Figure 4.6 shows the performance of the same algorithm with the same weights on a rare-feature set and a regular test set. The model shows significant over-fitting in case of the rare-feature set but almost none for the usual regular set. This is due to the fact that the rare features are small in number in this case ($\approx$ 20%) and the significant over-fitting on these samples is averaged out by a small improvement in

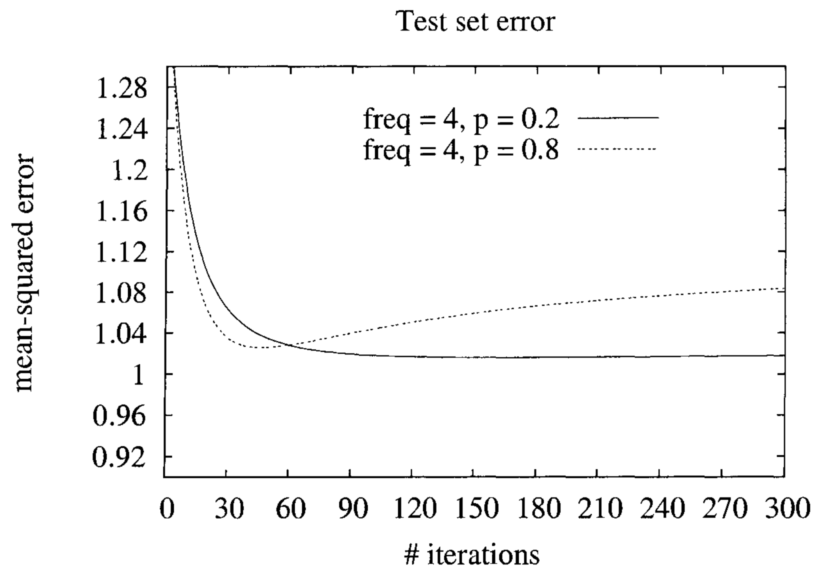


Figure 4.5: Test set error comparison between two data sequences with different percentage of rare features. The test set with a high percentage of rare features $p = 0.8$ exhibits over-fitting while the other does not. The model used is M1.

prediction of the rest of the test set. Using a rare-feature set with lower threshold criterion for rare features results in more pronounced over-fitting.

In cases such as medial applications where bad predictions are totally unacceptable, this approach can be considered to detect over-fitting. Another approach to detect small over-fitting is to use the L_{2n} norm ($n > 1$) instead of the usual L_2 norm while testing. The idea here is that as n is increases the samples with higher errors dominate the error value and consequently even a small over-fitting can be detected by using a sufficiently large n. One extreme is to use L_∞ which focuses only on the largest error. Figure 4.7 shows the test set error calculated using different L_{2n} norms for the same samples and algorithm. Although the values obtained by using different norms can not be compared, there is a higher over-fitting when using higher-order norms. This method is computationally expensive as it computes the error for all the samples in the test set and uses more multiplications compared to using a rare-feature set.

4.5.2 Empirical Comparison of Techniques

In this section we compare the clamping, regularization and weighted regularization algorithms empirically. We have tried four settings for regularization with α values set to $0.1, 0.01, 0.001,$ and 0.0001. We have tried twelve settings for clamping. We

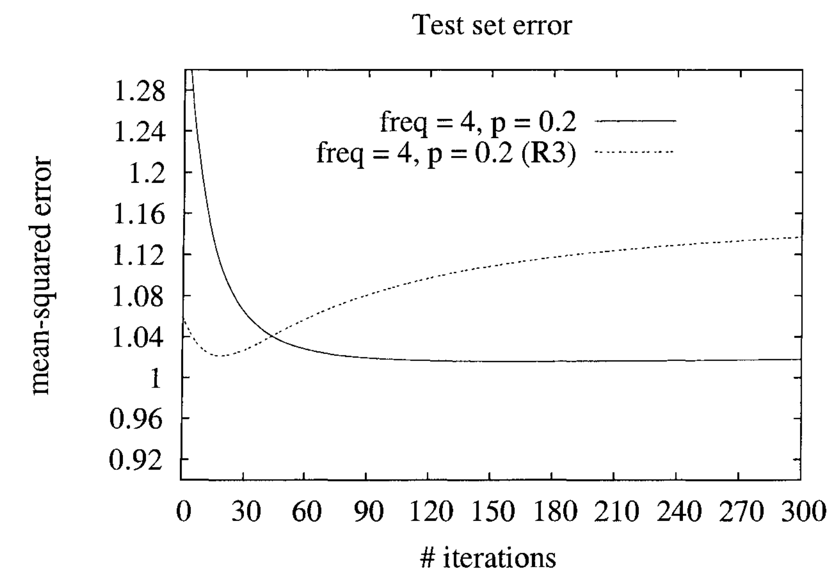


Figure 4.6: Test set error comparison between two data sequences using different test sets. There is no over-fitting when a regular test set is used. But using a rare-feature set (R3, criterion - RS_1 with feature frequency at most seven) exhibits over-fitting. The model used is M1.

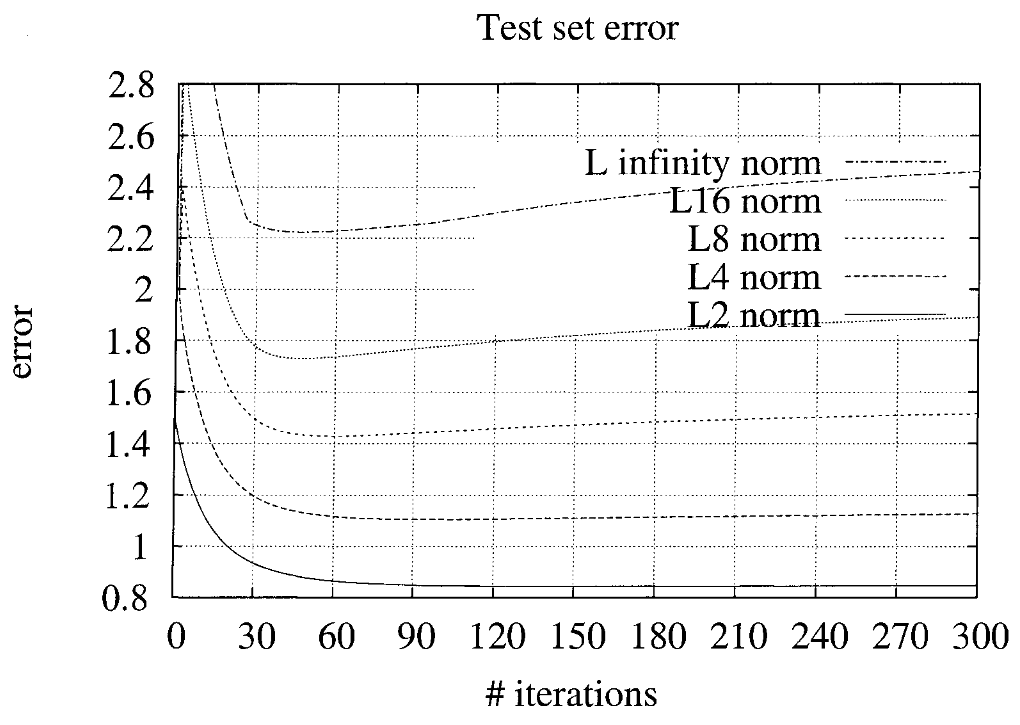


Figure 4.7: Test set error using different norms.

used a simple ramp function with v_{min} set to 0. We used four values for v_{max} (one, two, three, and four) and three values for the critical frequency depending upon the experimental setting. The range of configuration weights used for generating data is $[-1, 1]$ and the functions have a range of $[-4, 4]$. We assume that we do not know the range of configuration weights and we can only observe the range of the function. Therefore, v_{max}'s settings were chosen to cover the range of the function. In case the range of the configuration weights is known, it can be used to set more informed values for v_{max}. For weighted regularization we used the function $f(i) = \frac{1}{1+\frac{\exp(i)}{K}}$ where K is a constant. This function was used as the average frequency of a feature in this setting is ten and we wanted the function to have a negligible value for $i = 10$. The different α's tried were $0.1, 0.01, 0.001,$ and 0.0001. The settings used for generating data, clamping and generating rare-feature sets are shown in Table 4.3

p_r, f	Model	σ	Clamping Settings	$R1, R2, R3, R4$
$80\%, 4$	M1	0.5	$C \in \{4, 7, 10\}$	$RS_1(j), j \in \{2, 3, 4, 5\}$
$80\%, 4$	M1	1.5	$C \in \{4, 7, 10\}$	$RS_1(j), j \in \{2, 3, 4, 5\}$
$20\%, 7$	M1	0.5	$C \in \{6, 8, 10\}$	$RS_1(j), j \in \{5, 6, 7, 8\}$
$20\%, 7$	M1	1.5	$C \in \{6, 8, 10\}$	$RS_1(j), j \in \{5, 6, 7, 8\}$
$80\%, 4$	M2	0.5	$C \in \{4, 7, 10\}$	$RS_1(j), j \in \{6, 7, 8, 9\}$
$80\%, 4$	M2	1.5	$C \in \{4, 7, 10\}$	$RS_1(j), j \in \{6, 7, 8, 9\}$
$20\%, 7$	M2	0.5	$C \in \{12, 16, 20\}$	$RS_1(j), j \in \{2, 3, 4, 5\}$
$20\%, 7$	M2	1.5	$C \in \{12, 16, 20\}$	$RS_1(j), j \in \{2, 3, 4, 5\}$

Table 4.3: Parameters used for generating the data sets, rare-feature sets and the corresponding parameters used for clamping, $v_{max} \in \{1, 2, 3, 4\}$. p_r - percentage of rare features, f - average frequency of a rare feature.

A typical distribution of test set performance over the 100 different data sequences is shown in Figure 4.8. The distribution is close to normal as can be seen from the figure. Standard statistical tests like Student's t-test can be used to check if the means of two random normal variables are different.

Boxplot

We use boxplots to represent the performance over 100 data sequences. A box plot is an efficient method of displaying the median, upper and lower quartiles and the maximum and minimum of a distribution. The box part of the plot represents the middle 50% of the data. The upper edge of the box represents upper quartile (75th percentile) and the lower edge of the box represents lower quartile (25th percentile).

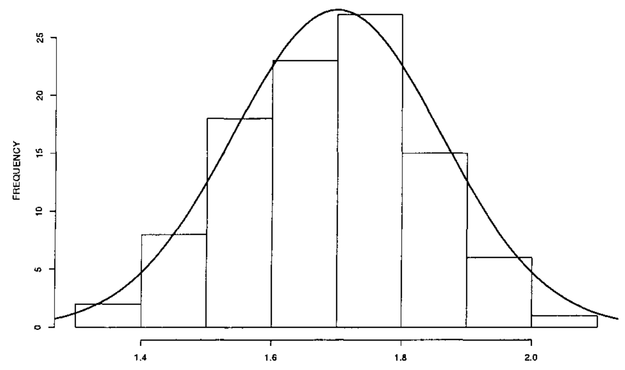


Figure 4.8: A typical distribution of mean-squared test error (early stopping value) of an algorithm over 100 data sequences.

The middle line or the notch represents the median. The whiskers extend for 1.5 times the inter-quartile range or till the maximum on the upper side or the minimum on the lower side (whichever results in a smaller whisker). Circles beyond the whiskers represent potential outliers. Boxplots help in quickly grasping the spread of the variable. By drawing boxplots for two variables side-by-side comparisons can be made easily. Figure 4.9 shows a few boxplots. We use p_r to represent the percentage of rare features and f to indicate the average frequency of rare features.

The amount of over-fitting can be gauged by comparing generalization errors after performing 300 iterations. The amount of over-fitting in a clamping setting with a fixed critical frequency increases as the clamp value is increased. This can be seen in Figure 4.9 where plots 10–13 represent clamping with critical number 4 and v_{max} equal to $1, 2, 3$ and 4 respectively. For plots 14–17 the critical frequency is 7 and for plots 18–21 it is 10. It can be seen that for the same critical number the mean increases as the value v_{max} is increased. This is because the features are less restricted and over-fitting sets is more pronounced. Also, for a fixed v_{max} the mean decreases as the critical frequency is increased. This is because restricting more number of rare features decreases over-fitting. This can be observed in Figure 4.9 by the decreasing mean of groups of four boxes with the same critical frequency.

We compare the above mentioned algorithms on the validation error. Each of the algorithms were trained on a sample set of size 1000 and a test set of size 100 was used for locating the early stopping value. Finally, the generalization error was estimated by testing the performance on a validation set of size 2000 samples. In what follows, $C(n, v)$ refers to a clamping algorithm with critical frequency n

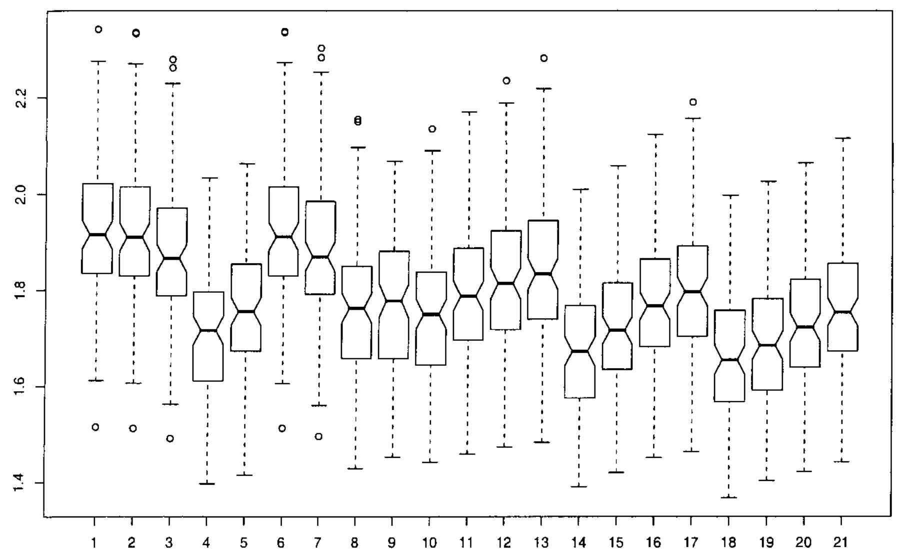


Figure 4.9: Boxplots of the test set performances after 300 iterations. 1 - plain, 2–5 - regularization $(0.0001, 0.001, 0.01, 0.1)$, 6–9 - weighted regularization $(0.0001, 0.001, 0.01, 0.1)$, 10–21 - clamping. Model - M1, $\sigma = 1.5, p_r = 80\%, f = 4$

and value v; $R(\alpha)$ and $WR(\alpha)$ refer to regularization and weighted regularization respectively with regularization parameter α and P refers to the plain version. All the algorithms use early stopping. The results are reported in Table 4.4, Table 4.5, Table 4.6, Table 4.7, Table 4.8,Table 4.9, Table 4.10 and Table 4.11. The tables show the number of times a particular algorithm was ranked first, second, third, fourth and fifth out of the 21 algorithms on the 100 different data sets. The columns are ordered based on the ranks of the algorithms. Only the top six algorithms are shown for each case.

From the above experiments we can see that clamping outperforms other algorithms on generalization error. In particular, clamping performs better when the percentage of rare features is high and the threshold is low (see Table 4.4, Table 4.5, Table 4.8 and Table 4.9). In this setting clamping performs better even when the error associated with the labels is higher. Actually, as the sample error is increased clamping begins to dominate (see Table 4.5 and Table 4.9) with $C(10, 1)$ ranking first 70 and 56 times respectively. Also, clamping algorithms have lesser over-fitting i.e. when over-trained the performance degradation is not as severe compared to regularization or plain algorithms.

	Algorithms →					
Rank ↓	C(7, 2)	C(10, 3)	C(10, 2)	C(10, 4)	C(7, 3)	R(0.01)
#1	25	17	17	9	6	6
#2	32	18	5	8	11	1
#3	8	24	6	19	8	4
#4	11	10	11	16	15	0
#5	1	9	3	17	14	1

Table 4.4: Model — M1, $\sigma = 0.5$, $p_r = 80\%$, $f = 4$. Each column indicates the number of times the corresponding algorithm ranked first, second, third, fourth and fifth with respect to the validation error. The best six algorithms sorted by performance are shown.

	Algorithms →					
Rank ↓	C(10, 1)	C(7, 1)	C(4, 1)	C(10, 2)	R(0.1)	C(7, 2)
#1	70	20	6	3	1	0
#2	17	55	7	15	1	1
#3	6	7	11	51	4	6
#4	0	4	36	6	6	16
#5	1	3	5	5	5	35

Table 4.5: Model — M1, $\sigma = 1.5$, $p_r = 80\%$, $f = 4$. Each column indicates the number of times the corresponding algorithm ranked first, second, third, fourth and fifth with respect to the validation error. The best six algorithms sorted by performance are shown.

	Algorithms →					
Rank ↓	R(0.001)	C(10, 1)	C(8, 1)	C(10, 2)	WR(0.001)	C(6, 1)
#1	35	19	18	6	5	5
#2	19	8	18	9	17	11
#3	11	5	7	11	7	10
#4	5	4	8	15	6	11
#5	9	3	1	7	1	1

Table 4.6: Model — M1, $\sigma = 0.5$, $p_r = 20\%$, $f = 7$. Each column indicates the number of times the corresponding algorithm ranked first, second, third, fourth and fifth with respect to the validation error. The best six algorithms sorted by performance are shown.

Rank ↓	Algorithms →					
	C(10, 1)	R(0.01)	C(8, 1)	R(0.1)	C(6, 1)	C(10, 2)
#1	37	16	12	11	7	7
#2	10	7	32	8	10	4
#3	8	13	5	7	17	7
#4	3	4	2	3	13	15
#5	1	5	1	2	4	14

Table 4.7: Model — M1, $\sigma = 1.5$, $p_r = 20\%$, $f = 7$. Each column indicates the number of times the corresponding algorithm ranked first, second, third, fourth and fifth with respect to the validation error. The best six algorithms sorted by performance are shown.

Rank ↓	Algorithms →					
	C(10, 1)	C(7,1)	C(10, 2)	R(0.0001)	C(7, 2)	R(0.001)
#1	53	18	11	6	4	3
#2	14	48	5	5	2	6
#3	5	2	47	6	4	5
#4	1	4	8	10	18	6
#5	0	2	7	5	27	2

Table 4.8: Model — M2, $\sigma = 0.5$, $p_r = 80\%$, $f = 4$. Each column indicates the number of times the corresponding algorithm ranked first, second, third, fourth and fifth with respect to the validation error. The best six algorithms sorted by performance are shown.

Rank ↓	Algorithms →					
	C(10, 1)	R(0.01)	C(7, 1)	C(10, 2)	C(7, 2)	C(10, 3)
#1	56	28	7	4	2	1
#2	9	24	34	3	2	4
#3	9	14	15	21	8	3
#4	4	5	12	12	7	4
#5	1	3	1	12	15	6

Table 4.9: Model — M2, $\sigma = 1.5$, $p_r = 80\%$, $f = 4$. Each column indicates the number of times the corresponding algorithm ranked first, second, third, fourth and fifth with respect to the validation error. The best six algorithms sorted by performance are shown.

Rank ↓	Algorithms →					
	C(20, 1)	WR(0.0001)	C(16, 1)	R(0.0001)	C(12, 1)	P
#1	45	19	11	7	6	5
#2	3	19	31	26	8	3
#3	8	17	5	24	15	12
#4	2	12	5	15	10	17
#5	2	9	1	6	16	25

Table 4.10: Model — M2, $\sigma = 0.5$, $p_r = 20\%$, $f = 7$. Each column indicates the number of times the corresponding algorithm ranked first, second, third, fourth and fifth with respect to the validation error. The best six algorithms sorted by performance are shown.

Rank ↓	Algorithms →					
	C(20, 1)	R(0.01)	WR(0.01)	C(16, 1)	P	R(0.001)
#1	34	21	17	7	7	3
#2	2	17	19	23	7	9
#3	13	2	2	5	17	11
#4	2	2	3	7	9	11
#5	2	1	1	1	13	8

Table 4.11: Model — M2, $\sigma = 1.5$, $p_r = 20\%$, $f = 7$. Each column indicates the number of times the corresponding algorithm ranked first, second, third, fourth and fifth with respect to the validation error. The best six algorithms sorted by performance are shown.

4.6 Othello

In this section we describe our experiments in a popular board game Othello. The rules of the game are described in Appendix B. The set of patterns used in this application are described in [Bur98]. The Othello data was randomly split into two sets – train data and test data. In the Othello application we divided the game into 13 stages based on the number of disks on the board. We used interpolation of data between game stages a described in [Bur98] to create smooth transitions between game stages.

The original $\sim$ 16 million labeled game positions were divided into two sets with $\sim$ 15.9 million labelled samples forming the training set and the remaining positions put into the test set. Four rare-feature sets R_1, R_2, R_3 and R_4 of type RS_1 were extracted from the test set with the rarity criteria being feature frequency at most $5, 10, 20$ and 50, respectively. We have tried five settings each for regularization and weighted regularization and 20 settings for clamping (clamp values $8, 16, 32, 64, 96 \times$ clamp frequencies $5, 10, 20, 50$). We have used a ramp function with v_1 set to 0 for the clamping. We report the experimental results for stage 1 of the game where the evaluation function is the weakest.

Testing the performance of these algorithms on a usual test set did not show significant over-fitting. The rare-feature set R_2 was used to detect over-fitting (see Figure 4.10). Figure 4.11 shows the performance of various algorithms on the rare test set R_2. In this example, clamping performs the best with respect to the minimum test error followed by regularization. Also, over-fitting is more pronounced in regularization compared to clamping. The plain algorithm with no clamping or regularization is the worst with respect to over-fitting. It can be seen in Figure 4.11 that over-fitting in clamping algorithms reduces as the clamping threshold is lowered. This is because a lower clamping threshold results in smaller weights for feature weights and hence over-fitting is lower. As the clamping threshold is increased, the algorithm behaves more like the plain algorithm because the feature weights are almost unrestricted.

Figure 4.12 shows the effect of different models. Both the models (M1 and M2, see Appendix B) result in almost the same test set error and show no over-fitting when tested on the regular test set. However, when tested on a rare-feature set both the models shows over-fitting. Model M1 results in higher over-fitting compared to M2. The downside of using M2 in game play is that it computationally more expensive than M1. The performance improvement obtained in the evaluation

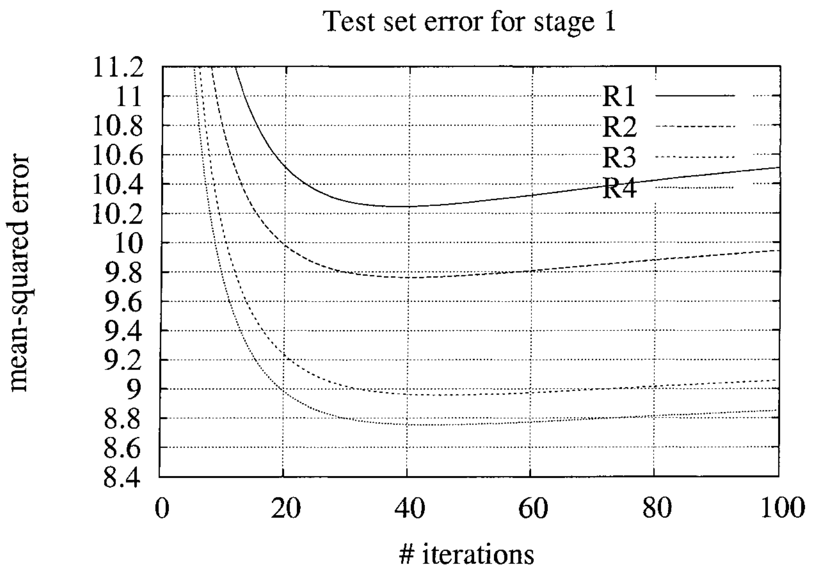


Figure 4.10: Performance of the plain algorithm on various Othello rare-feature sets for stage 1 and Model M1. Over-fitting is higher for the rare-feature set that has a lower average frequency of features.

function is compensated by slower computation. The net gain can be judged by using it in actual game play.

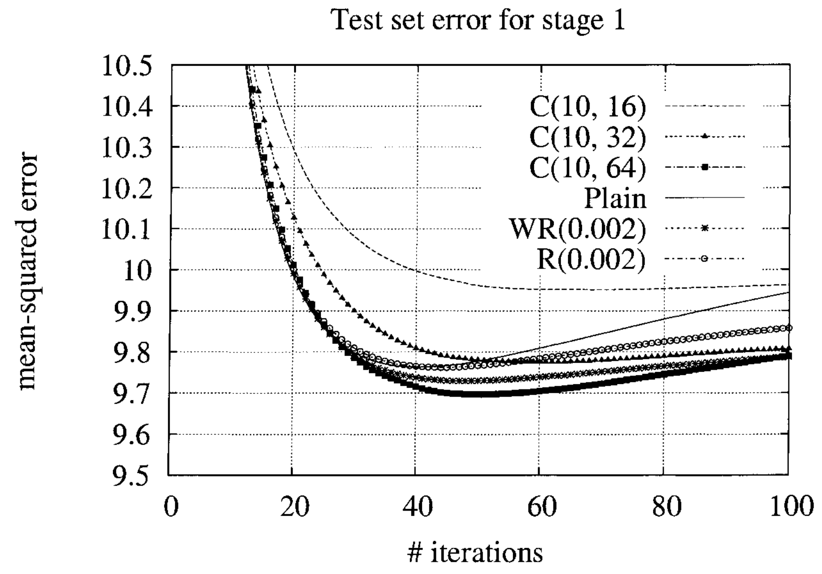


Figure 4.11: Performance of different algorithms on stage 1 of Othello on test set R_2. R - Regularization, $C(f, V)$ - Clamping with critical frequency f and $v_{max} = V$.

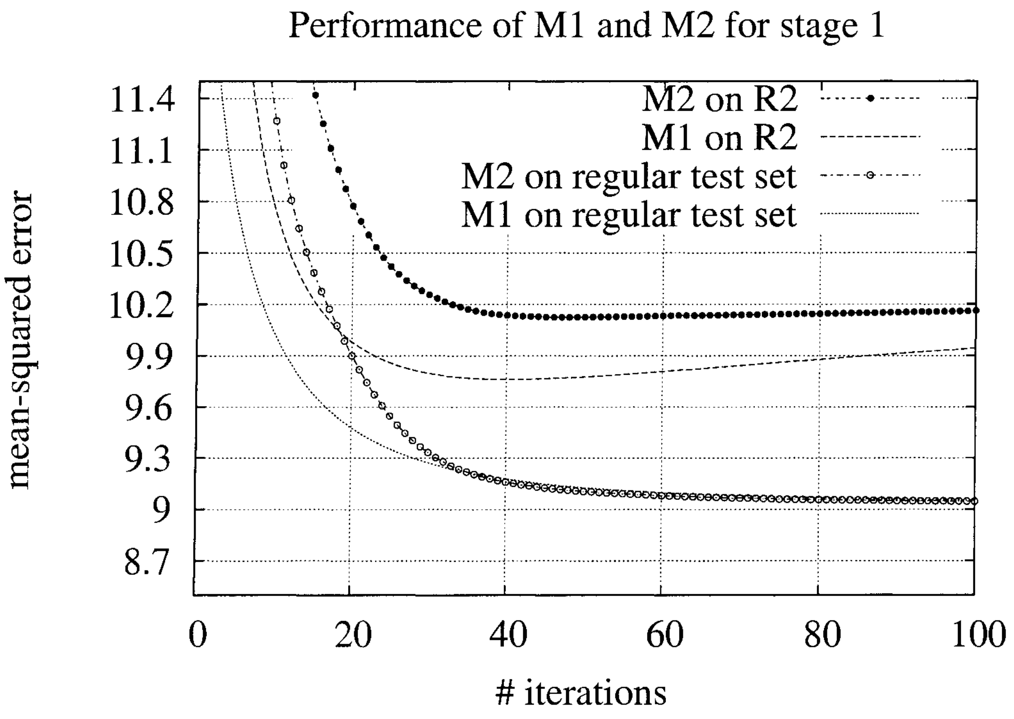


Figure 4.12: Performance of the plain algorithm on various Othello rare-feature sets for stage 1 using test sets R_2 and the regular test set using the plain algorithm.

Chapter 5

Conclusions and Future Work

GLEM presents a natural way to define evaluation functions with increasing complexity. We have proposed an enhancement to GLEM by incorporating non-linear layered network over the existing models.

In this thesis we have looked at how the presence of rare features affects over-fitting in GLEM. We have proposed a novel method of detecting over-fitting by using rare-feature sets. Using rare-feature sets we have detected over-fitting in both the application data we have considered — synthetic data and Othello data — which were not detected using other test sets. We have empirically studied how the percentage and frequency of rare features affects over-fitting. Results presented in previous chapter indicate that the presence of higher percentage of rare features in the training set results in over-fitting of parameters during training. We have also provided evidence that as the features become rarer over-fitting increases.

We have proposed new algorithms (clamping and weighted regularization) to tackle this problem based on frequency of features. We based our algorithms on the idea that a rare feature's weight must be limited by a function of its frequency. Weighted regularization did not perform much different from the plain regularization on the problems we considered. This is because the exact form of the function which results in optimal generalization performance is not known. But we know that function should have higher value for lower frequencies and should approach zero for higher frequencies. Results from the previous chapter show that clamping-based techniques result in better generalization than regularization in situations where there is a high percentage of rare features. Also, clamping-based techniques result in lesser over-fitting compared to regularization in this setting. Therefore, when the test set is not representative of the distribution of data encountered in real world (corresponding to rare-feature sets in our experiments) or bad predictions are unacceptable even for small portions of the data or there is a high percentage of rare features with labeling error, clamping-based techniques are strong candidates to be considered.

We have also seen in our synthetic data experiments that using early stopping results in very little difference between the performance of various algorithms in some cases. Therefore, early stopping should be used in conjunction with these algorithms whenever possible.

5.1 Future Work

Some ideas that can be explored in future:

- GLEM can be applied to other domains. Go is a strong candidate to be considered. Unlike Othello where the abstraction needed was minimal, Go requires a fair amount of abstraction as it is a complex game. However, state of the art Go programs already employ many abstractions. For example, it is common for Go programs to talk about abstractions like eyes, strings, liberties, blocks etc. These abstract features can serve as atomic features for GLEM.

- An ad-hoc ramp function was used for clamping in this thesis. A theoretical analysis should be conducted on the shape of the function to be used in clamping. Also, we do not know of a good function to be used for weighted regularization. However, we believe that for clamping the function should be monotonically increasing and for weighted regularization the function should be monotonically decreasing with frequency.

- We have based our algorithms on gradient descent. The problem of over-fitting in GLEM when quadratic methods like conjugate gradients are used can be explored. Empirical evidence [CLG01] suggests that conjugate gradient methods might result in higher over-fitting. It would be interesting to see how frequency-based techniques can be applied here.

Bibliography

[Bur97] Michael Buro. The Othello match of the year, Takeshi Murakami vs Logistello. *ICCA*, 20:189–193, 1997.

[Bur98] Michael Buro. From Simple Features to Sophisticated Evaluation Functions. In H. J.van den Herik and H. Iida, editors, *Computers and Games: Proceedings CG'98. LNCS 1558*, pages 126–145. Tsukuba, Japan, 1998.

[Bur99] Michael Buro. How Machines have Learned to Play Othello. *IEEE Intelligent Systems*, 14(6):12–14, 1999.

[CB96] Robert H. Crites and Andrew G. Barto. Improving Elevator Performance Using Reinforcement Learning. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 1017–1023. The MIT Press, 1996.

[CB05] Siddhartha Chinthapally and Michael Buro. *GLEM Software, http://www.cs.ualberta.ca/ mburo/software/glem.tgz*, 2005.

[CLG01] Rich Caruana, Steve Lawrence, and C. Lee Giles. Overfitting in neural networks: Backpropagation, conjugate gradient, and early stopping. In *Advances in Neural Information Processing Systems*, Denver, Colorado, 2001.

[DCM92] Christian Darken, Joseph Chang, and John Moody. Learning rate schedules for faster stochastic gradient search. In *Proc. Neural Networks for Signal Processing 2*. IEEE Press, 1992.

[DM90] Christian Darken and John Moody. Note on Learning Rate Schedules for Stochastic Optimization. In J. E. Moody R. P. Lippman and D. S. Touretzky, editors, *Neural Information Processing Systems*. Morgan Kauffman, 1990.

[DPH86] S. Nowlan D. Plaut and G. Hinton. Experiments on Learning by Back Propagation. 1986. Technical Report CMU-CS-86-126, Computer Science Department, Carnegie-Mellon University.

[Für01] Johannes Fürnkranz. Machine learning in games: A survey. In J. Fürnkranz and M. Kubat, editors, *Machines that Learn to Play Games*, chapter 2, pages 11–59. Nova Science Publishers, Huntington, NY, 2001.

[Gol87] Larry Goldstein. Mean square optimality in the continuos time Robbins Munro procedure. 1987. Technical Report DRB-306, Department of Mathematics, University of Southern California.

[LBOM98] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Mueller. Efficient BackProp. *Lecture Notes in Computer Science*, 1524:9–50, 1998.

[Lju77] Lennart Ljung. Analysis of Recursive Stochastic Algorithms. *IEEE Transactions on Automatic Control*, 22:551–575, 1977.

[Nil96] Nils J. Nilsson. *Introduction to Machine Learning*. 1996.

[Pre98] Lutz Prechelt. Early Stopping - But When? *Lecture Notes in Computer Science*, 1524:55–69, 1998.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

[Sam59] A L Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3:211–229, 1959.

[Sam67] A L Samuel. Some studies in Machine Learning Using the Game of Checkers ii - Recent Progress. *IBM Journal of Research and Development*, 11(6):601–617, nov 1967.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT press, 1998.

[Sch94] Jonathan Richard Schewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. August 1994. Unpublished Draft.

[Sch97] Dale Schuurmans. A new metric-based approach to model selection. In *AAAI/IAAI*, pages 552–558, 1997.

[SL94] James V. Stone and Raymond Lister. On the Relative Time Complexities of Standard and Conjugate Gradient Back Propagation. *Neural Networks*, 1:84–87, 1994.

[Tes95] Gerald Tesauro. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, 38:58–68, 1995.

[Wei99] Eric W. Weisstein. *Box-Muller Transform, From MathWorld — A Wolfram Web Resource, http://mathworld.wolfram.com/Box-MullerTransformation.html*, 1999.

Appendix A

Synthetic Application Function Definition Files

```
# Model M1
;TOPOLOGY
1 ide 2
2 sum

;FEATURES
T TAB1      1   100
t1-Transform-a 2

T TAB2      1   100
t2-Transform-a 2

T TAB3      1   100
t3-Transform-a 2
```

```
# Model M2
;TOPOLOGY
1 ide 2
2 sum 3 4 5

3 tnh 6
6 sum

4 tnh 7
7 sum

5 tnh 8
8 sum

;FEATURES
T TAB1      2   100
t1-transform-a 6
t1-transform-b 6

T TAB2      2   100
t2-transform-a 7
t2-transform-b 7

T TAB3      2   100
t3-transform-a 8
t3-transform-b 8
```

Appendix B

Othello

Othello is a two player, zero-sum perfect information game. It is based on a game known as Reversi and is included in the games section of standard GNU/Linux distributions as KReversi. The game is played on an 8×8 board by placing disks of two colors — black and white. At the start of the game a player chooses one color and uses it throughout the game. The objective of the game is to have maximum number of disks on the board of your color at the end.

The game begins with the setup shown in Figure B.1 with black to move first. A move consists of placing a disk in one of the empty squares of the board. every move has to outflank your opponent's disks and replacing them with your disks. To outflank means to place a disk on the board so that one or more of your opponent's continuous *string* of disks are bordered at each end by your disks. A string can be horizontal, vertical or diagonal. After the disk is placed all those opponent's disks that are outflanked by the recently placed disk are replaced by your disks. Figure B.1 shows the board situation after white makes his move. If a player can not outflank any of his opponents disks then his turn is also and his opponent moves again. A player can not forfeit his turn if a move is available. A player can not move over his own disks to outflank an opponents disk (see the diagonal box in Figure B.1). The game ends when both the players have no more moves to make. And the player with the maximum number of disks on the board is declared the winner. Usually the game ends when the board fills up, but it is possible for the game to end before all the 64 squares are filled up.

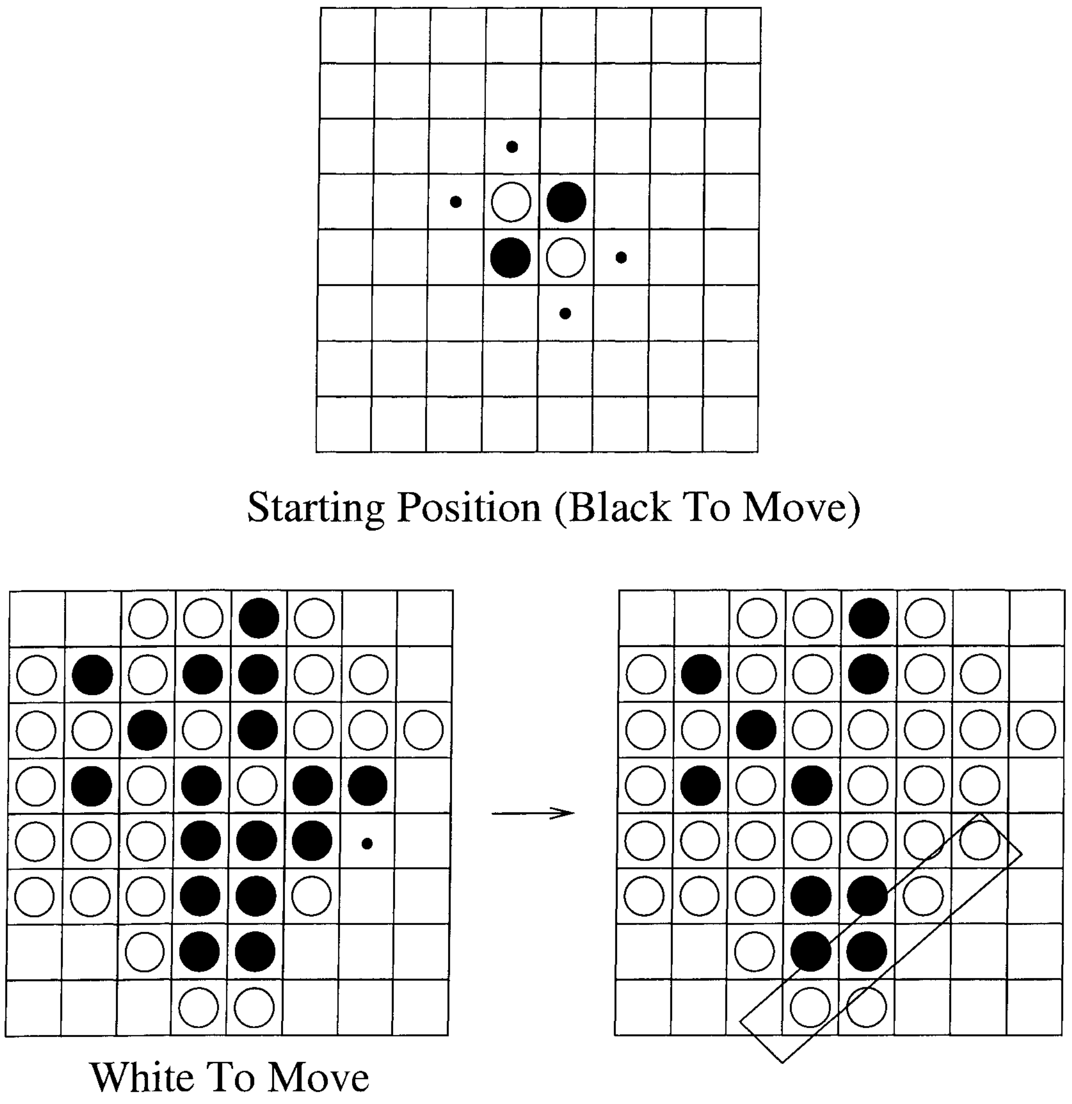


Figure B.1: Othello rules

B.1 Function Definition File for Othello Application, Model M1

```
;TOPOLOGY

1 ide 2
2 sum

;FEATURES

T 2x5 8 59049
A1B1C1D1A2B2C2D2E1E2 2
H1G1F1E1H2G2F2E2D1D2 2
A8B8C8D8A7B7C7D7E8E7 2
H8G8F8E8H7G7F7E7D8D7 2
A1A2A3A4B1B2B3B4A5B5 2
H1H2H3H4G1G2G3G4H5G5 2
A8A7A6A5B8B7B6B5A4B4 2
H8H7H6H5G8G7G6G5H4G4 2

T EDGE+2X 4 29646
A1B1C1D1E1F1G1H1B2G2 2
A8B8C8D8E8F8G8H8B7G7 2
A1A2A3A4A5A6A7A8B2B7 2
H1H2H3H4H5H6H7H8G2G7 2

T 3x3 4 10206
A1B1C1A2B2C2A3B3C3 2
H1G1F1H2G2F2H3G3F3 2
A8B8C8A7B7C7A6B6C6 2
H8G8F8H7G7F7H6G6F6 2

T HV2 4 3321
A2B2C2D2E2F2G2H2 2
A7B7C7D7E7F7G7H7 2
B1B2B3B4B5B6B7B8 2
G1G2G3G4G5G6G7G8 2

T HV3 4 3321
A3B3C3D3E3F3G3H3 2
A6B6C6D6E6F6G6H6 2
C1C2C3C4C5C6C7C8 2
F1F2F3F4F5F6F7F8 2

T HV4 4 3321
A4B4C4D4E4F4G4H4 2
A5B5C5D5E5F5G5H5 2
D1D2D3D4D5D6D7D8 2
E1E2E3E4E5E6E7E8 2

T D8 2 3321
A1B2C3D4E5F6G7H8 2
H1G2F3E4D5C6B7A8 2

T D7 4 1134
A2B3C4D5E6F7G8 2
H2G3F4E5D6C7B8 2
A7B6C5D4E3F2G1 2
H7G6F5E4D3C2B1 2

T D6 4 378
A3B4C5D6E7F8 2
H3G4F5E6D7C8 2
A6B5C4D3E2F1 2
H6G5F4E3D2C1 2

T D5 4 135
A4B5C6D7E8 2
H4G5F6E7D8 2
A5B4C3D2E1 2
H5G4F3E2D1 2

T D4 4 45
A5B6C7D8 2
H5G6F7E8 2
A4B3C2D1 2
H4G3F2E1 2

T PARITY 1 2
PARITY 2
```

B.2 Function Definition File for Othello Application, Model M2

```
;TOPOLOGY

1 ide 2
2 sum 3 4 5 6 7 8 9 10 11 12 13 14

# 2X5
3 tnh 15
15 sum

# EDGE+2X
4 tnh 16
16 sum

# 3X3
5 tnh 17
17 sum

# HV2
6 tnh 18
18 sum

# HV3
7 tnh 19
19 sum

# HV4
8 tnh 20
20 sum

# D8
9 tnh 21
21 sum

# D7
10 tnh 22
22 sum

# D6
11 tnh 23
23 sum

# D5
12 tnh 24
24 sum

# D4
13 tnh 25
25 sum

# PARITY
14 tnh 26
26 sum
```

```
;FEATURES

T 2x5 8 59049
A1B1C1D1A2B2C2D2E1E2 15
H1G1F1E1H2G2F2E2D1D2 15
A8B8C8D8A7B7C7D7E8E7 15
H8G8F8E8H7G7F7E7D8D7 15
A1A2A3A4B1B2B3B4A5B5 15
H1H2H3H4G1G2G3G4H5G5 15
A8A7A6A5B8B7B6B5A4B4 15
H8H7H6H5G8G7G6G5H4G4 15

T EDGE+2X 4 29646
A1B1C1D1E1F1G1H1B2G2 16
A8B8C8D8E8F8G8H8B7G7 16
A1A2A3A4A5A6A7A8B2B7 16
H1H2H3H4H5H6H7H8G2G7 16

T 3x3 4 10206
A1B1C1A2B2C2A3B3C3 17
H1G1F1H2G2F2H3G3F3 17
A8B8C8A7B7C7A6B6C6 17
H8G8F8H7G7F7H6G6F6 17

T HV2 4 3321
A2B2C2D2E2F2G2H2 18
A7B7C7D7E7F7G7H7 18
B1B2B3B4B5B6B7B8 18
G1G2G3G4G5G6G7G8 18

T HV3 4 3321
A3B3C3D3E3F3G3H3 19
A6B6C6D6E6F6G6H6 19
C1C2C3C4C5C6C7C8 19
F1F2F3F4F5F6F7F8 19

T HV4 4 3321
A4B4C4D4E4F4G4H4 20
A5B5C5D5E5F5G5H5 20
D1D2D3D4D5D6D7D8 20
E1E2E3E4E5E6E7E8 20

T D8 2 3321
A1B2C3D4E5F6G7H8 21
H1G2F3E4D5C6B7A8 21

T D7 4 1134
A2B3C4D5E6F7G8 22
H2G3F4E5D6C7B8 22
A7B6C5D4E3F2G1 22
H7G6F5E4D3C2B1 22

T D6 4 378
A3B4C5D6E7F8 23
H3G4F5E6D7C8 23
A6B5C4D3E2F1 23
H6G5F4E3D2C1 23

T D5 4 135
A4B5C6D7E8 24
H4G5F6E7D8 24
A5B4C3D2E1 24
H5G4F3E2D1 24

T D4 4 45
A5B6C7D8 25
H5G6F7E8 25
A4B3C2D1 25
H4G3F2E1 25

T PARITY 1 2
PARITY 26
```