EVALUATION OF GAUSSIAN PROCESSES AND OTHER METHODS FOR NON-LINEAR REGRESSION

Carl Edward Rasmussen

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy,
Graduate Department of Computer Science,
in the University of Toronto

Evaluation of Gaussian Processes and other Methods for Non-Linear Regression

Carl Edward Rasmussen

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy,
Graduate Department of Computer Science,
in the University of Toronto
Convocation of March 1997

Abstract

This thesis develops two Bayesian learning methods relying on Gaussian processes and a rigorous statistical approach for evaluating such methods. In these experimental designs the sources of uncertainty in the estimated generalisation performances due to both variation in training and test sets are accounted for. The framework allows for estimation of generalisation performance as well as statistical tests of significance for pairwise comparisons. Two experimental designs are recommended and supported by the DELVE software environment.

Two new non-parametric Bayesian learning methods relying on Gaussian process priors over functions are developed. These priors are controlled by hyperparameters which set the characteristic length scale for each input dimension. In the simplest method, these parameters are fit from the data using optimization. In the second, fully Bayesian method, a Markov chain Monte Carlo technique is used to integrate over the hyperparameters. One advantage of these Gaussian process methods is that the priors and hyperparameters of the trained models are easy to interpret.

The Gaussian process methods are benchmarked against several other methods, on regression tasks using both real data and data generated from realistic simulations. The experiments show that small datasets are unsuitable for benchmarking purposes because the uncertainties in performance measurements are large. A second set of experiments provide strong evidence that the bagging procedure is advantageous for the Multivariate Adaptive Regression Splines (MARS) method.

The simulated datasets have controlled characteristics which make them useful for understanding the relationship between properties of the dataset and the performance of different methods. The dependency of the performance on available computation time is also investigated. It is shown that a Bayesian approach to learning in multi-layer perceptron neural networks achieves better performance than the commonly used early stopping procedure, even for reasonably short amounts of computation time. The Gaussian process methods are shown to consistently outperform the more conventional methods.

Acknowledgments

Many thanks to Radford Neal and Geoffrey Hinton for sharing their insights and enthusiasm throughout my Ph.D. work. I hope that one day I will similarly be able to inspire people around me.

I also wish to thank past and present members and visitors to the neuron and DELVE groups as well as my committee, in particular Drew van Camp, Peter Dayan, Brendan Frey, Zoubin Ghahramani, David MacKay, Mike Revow, Rob Tibshirani and Chris Williams.

Thanks to the Freys and the Hardings for providing me with excellent meals at times when my domestic life was at an ebb. Lastly, I wish to thank Agnes Heydtmann for her continued encouragement and confidence.

During my studies in Toronto, I was supported by the Danish Research Academy, by the University of Toronto Open Fellowship and through grants to Geoffrey Hinton from the National Sciences and Engineering Research Council of Canada and the Institute for Robotics and Intelligent Systems.

Contents

Chapter 1

Introduction

The ability to learn relationships from examples is compelling and has attracted interest in many parts of science. Biologists and psychologists study learning in the context of animals interacting with their environment; mathematicians, statisticians and computer scientists often take a more theoretical approach, studying learning in more artificial contexts; people in artificial intelligence and engineering are often driven by the requirements of technological applications. The aim of this thesis is to contribute to the principles of measuring performance of learning methods and to demonstrate the effectiveness of a particular class of methods.

Traditionally, methods that learn from examples have been studied in the statistics community under the names of model fitting and parameter estimation. Recently there has been a huge interest in neural networks. The approaches taken in these two communities have differed substantially, as have the models that are studied. Statisticians are usually concerned primarily with interpretability of the models. This emphasis has led to a diminished interest in very complicated models. On the other hand, workers in the neural network field have embraced ever more complicated models and it is not unusual to find applications with very computation intensive models containing hundreds or thousands of parameters. These complex models are often designed entirely with predictive performance in mind.

Recently, these two approaches to learning have begun to converge. Workers in neural networks have "rediscovered" statistical principles and interest in non-parametric modeling has risen in the statistics community. This intensified focus on statistical aspects of non-parametric modeling has brought an explosive growth of available algorithms. Many of these new flexible models are not designed with particular learning tasks in mind, which introduces the problem of how to choose the best method for a particular task. All of these general purpose methods rely on various assumptions and approximations, and in many cases it is hard to know how well these are met in particular applications and how severe the consequences of breaking them are. There is an urgent need to provide an evaluation of these methods, both from a practical applicational point of view and in order to guide further research.

An interesting example of this kind of question is the long-standing debate as to whether Bayesian or frequentist methods are most desirable. Frequentists are often unhappy about the setting of priors, which is sometimes claimed to be “arbitrary”. Even if the Bayesian theory is accepted, it may be considered computationally impractical for real learning problems. On the other hand, Bayesians claim that their models may be superior and may avoid the computational burden involved in the use of cross-validation to set model complexity. It seems doubtful that these disputes will be settled by continued theoretical debate.

Empirical assessment of learning methods seems to be the most appealing way of choosing between them. If one method has been shown to outperform another on a series of learning problems that are judged to be representative, in some sense, of the applications that we are interested in, then this should be sufficient to settle the matter. However, measuring predictive performance in a realistic context is not a trivial task. Surprisingly, this is a tremendously neglected field. Only very seldom are conclusions from experimental work backed up by statistically compelling evidence from performance measurements.

This thesis is concerned with measuring and comparing the predictive performance of learning methods, and contains three main contributions: a theoretical discussion of how to perform statistically meaningful comparisons of learning methods in a practical way, the introduction of two novel methods relying on Gaussian processes, and a demonstration of the assessment framework through empirical comparison of the performance of Gaussian process methods with other methods. An elaboration of each of these topics will follow here.

I give a detailed theoretical discussion of issues involved in practical measurement of predictive performance. For example, many statisticians have been uneasy about the fact that many neural network methods involve random initializations, such that the result of learning is not a unique set of parameter values. However, once these issues are faced it is not difficult to give them a proper treatment. The discussion involves assessing the statistical significance of comparisons, developing a practical framework for doing comparisons and measures ensuring that results are reproducible. The focus of Chapter 2 is to make the goal of comparisons precise and to understand the uncertainties involved in empirical evaluations. The objective is to obtain a good tradeoff between the conflicting aims of statistical reliability and practical applicability of the framework to computationally intensive learning algorithms. These considerations lead to some guidelines for how to measure performance.

A software environment which implements these guidelines called DELVE — Data for Evaluating Learning in Valid Experiments — has been written by our research group headed by Geoffrey Hinton. DELVE is freely available on the world wide web[1]. DELVE contains the software necessary to perform statistical tests, datasets for evaluations, results of applying methods to these datasets, and precise descriptions of methods. Using DELVE one can make statistically well founded simulation experiments comparing the performance of learning methods. Chapter 2 contains a discussion of the design of DELVE, but implementational details are provided elsewhere [Rasmussen et al. 1996].

DELVE provides an environment within which methods can be compared. DELVE includes a number of standardisations that allow for easier comparisons with earlier work and attempts to provide a realistic setting for the methods. Most other attempts at making benchmark collections provide the data in an already preprocessed format in order to heighten reproducibility. However, this approach seems misguided, if one is attempting to measure the performance that could be achieved in a realistic setting, where the preprocessing could be tailored to the particular method. To allow for this, definitions of methods in DELVE must include descriptions of preprocessing. DELVE provides facilities for some common types of preprocessing, and also a "default" attribute encoding to be used by researchers who are not primarily interested in such issues.

In Chapter 3 detailed descriptions of several learning methods that emphasize reproducibility are given. The implementations of many of the more complicated methods involve choices that may not be easily justifiable from a theoretical point of view. For example, many neural networks are trained using iterative methods, which raises the question of how many iterations one should apply. Sometimes convergence cannot be reached within reasonable amount of computational effort, for example, and sometimes it may be preferable to stop training before convergence. Often these issues are not discussed very thoroughly in the articles describing new methods. Furthermore authors may have used preliminary simulations to set such parameters, thereby inadvertently opening up the possibility of bias in simulation results.

[1]The DELVE web address is: `http://www.cs.utoronto.ca/~delve`

In order to avoid these problems, the learning methods must be specified precisely. Methods that contain many parameters that are difficult to set should be recognized as having this handicap, and heuristic rules for setting their parameters must be developed. If these rules don't work well in practice, this may show up in the comparative studies, indicating that this learning method would not be expected to do well in an actual application. Naturally, some parameters may be set by some initial trials on the training data, in which case this would be considered a part of the training procedure. This precise level of specification is most easily met for "automatic" algorithms, which do not require human intervention in their application. In this thesis only such automatic methods will be considered.

The methods described in Chapter 3 include methods originating in the statistics community as well as neural network methods. Ideally, I had hoped to find descriptions and implementations of these methods in the literature, so that I could concentrate on testing and comparing them. Unfortunately, the descriptions found in the literature were rarely detailed enough to allow direct application. Most frequently details of the implementations are not mentioned, and in the rare cases where they are given they are often of an unsatisfactory nature. As an example, it may be mentioned that networks were trained for 100 epochs, but this hardly seems like a principle that should be applied universally. On the other hand it has proven extremely difficult to design heuristic rules that incorporate a researcher's "common sense". The methods described in Chapter 3 have been selected partially from considerations of how difficult it may be to invent such rules. The descriptions contain precise specifications as well as a commentary.

In Chapter 4 I develop a novel Bayesian method for learning relying on Gaussian processes. This model is especially suitable for learning on small data sets, since the computational requirements grow rapidly with the amount of available training data. The Gaussian process model is inspired by Neal's work [1996] on priors for infinite neural networks and provides a unifying framework for many models. The actual model is quite like a weighted nearest neighbor model with an adaptive distance metric.

A large body of experimental results has been generated using DELVE. Several neural network techniques and some statistical methods are evaluated and compared using several sources of data. In particular, it is shown that it is difficult to get statistically significant comparisons on datasets containing only a few hundred cases. This finding suggests that many previously published comparisons may not be statistically well-founded. Unfortunately, it seems hard to find suitable real datasets containing several thousand cases that could be used for assessments.

In an attempt to overcome this difficulty in DELVE we have generated large datasets from simulators of realistic phenomena. The large size of these simulated datasets provides a high degree of statistical significance. We hope that they are realistic enough that researchers will find performance on these data interesting. The simulators allow for generation of datasets with controlled attributes such as degree of non-linearity, input dimensionality and noise-level, which may help in determining which aspects of the datasets are important to various algorithms. In Chapter 5 I perform extensive simulations on large simulated datasets in DELVE. These simulations show that the Gaussian process methods consistently outperform the other methods.

Chapter 2

Evaluation and Comparison

In this chapter I discuss the design of experiments that test the predictive performance of learning methods. A large number of such learning methods have been proposed in the literature, but in practice the choice of method is often governed by tradition, familiarity and personal preference rather than comparative studies of performance. Naturally, predictive performance is only one aspect of a learning method; other characteristics such as interpretability and ease of use are also of concern. However, for predictive performance a well developed set of directly applicable statistical techniques exist that enable comparisons. Despite this, it is very rare to find any compelling empirical performance comparisons in the literature on learning methods [Prechelt 1996]. I will begin this chapter by defining generalisation, which is the measure of predictive performance, then discuss possible experimental designs, and finally give details of the two most promising designs for comparing learning methods, both of which have been implemented in the DELVE environment.

2.1 Generalisation

Usually, learning methods are trained with one of two goals: either to identify an interpretation of the data, or to make predictions about some unmeasured events. The present study is concerned only with accuracy of this latter use. In statistical terminology, this is sometimes called the expected out-of-sample predictive loss; in the neural network literature it is referred to as generalisation error. Informally, we can define this as the expected loss for a particular method trained on data from some particular distribution on a novel (test) case from that same distribution.

In the formalism alluded to above and used throughout this thesis the objective of learning will be to minimize this expected loss. Some commonly used loss functions are squared error loss for regression problems and 0/1-loss for classification; others will be considered as well. It should be noted that this formalism is not fully general, since it requires that losses can be evaluated on a case by case basis. We will also disallow methods that use the inputs of multiple test cases to make predictions. This confinement to fixed training sets and single test cases rules out scenarios which involve active data selection, incremental learning where the distribution of data drifts, and situations where more than one test case is needed to evaluate losses. However, a very broad class of learning problems can naturally be cast in the present framework.

In order to give a formal definition of generalisation we need to consider the sources of variation in the basic experimental unit, which consists of training a method on a particular set of training cases and measuring the loss on a test case. These sources of variation are

1. Random selection of test case.
2. Random selection of training set.
3. Random initialisation of learning method; e.g. random initial weights in neural networks.
4. Stochastic elements in the training algorithm used in the method; e.g. stochastic hill-climbing.
5. Stochastic elements in the predictions from a trained method; e.g. Monte Carlo estimates from the posterior predictive distribution.

Some of these sources are inherent to the experiments while others are specific to certain methods such as neural networks. Our definition of generalisation error involves the expectation over all these effects

$$G_F(n) = \int L\big[F_{r_i,r_t,r_p}(\mathcal{D}_n, x), t\big] p(x,t) p(\mathcal{D}_n) p(r_i) p(r_t) p(r_p)\, dx\, dt\, d\mathcal{D}_n\, dr_i\, dr_t\, dr_p. \qquad (2.1)$$

This is the generalisation error for a method that implements the function F, when trained on training sets of size n. The loss function L measures the loss of making the prediction $F_{r_i,r_t,r_p}(\mathcal{D}_n, x)$ using training set $\mathcal{D}_n$ of size n and test input x when the true target is t. The loss is averaged over the distribution of training sets $p(\mathcal{D}_n)$, test points $p(x, t)$ and random effects of initialisation $p(r_i)$, training $p(r_t)$ and prediction $p(t_p)$.

Here it has been assumed that the training examples and the test examples are drawn independently from the same (unknown) distribution. This is a simplifying assumption that holds well for many prediction tasks; one important exception is time series prediction, where the training cases are usually not drawn independently. Without this assumption, empirical evaluation of generalisation error becomes problematic.

The definition of generalisation error given here involves averaging over training sets of a particular size. It may be argued that this is unnecessary in applications where we have a particular training set at our disposal. However, in the current study, we do empirical evaluations in order to get an idea of how well methods will perform on other data sets with similar characteristics. It seems unreasonable to assume that these new tasks will contain the same peculiarities as particular training sets from the empirical study. Therefore, it seems essential to take the effects of this variation into account, especially when estimating confidence intervals for G.

Evaluation of G is difficult for several reasons. The function to be integrated is typically too complicated to allow analytical treatment, even if the data distribution were known. For real applications the distribution of the data is unknown and we only have a sample from the distribution available. Sometimes this sample is large compared to the n for which we wish to estimate $G(n)$, but for real datasets we often find ourselves in the more difficult situation of trying to estimate $G(n)$ for values of n not too far from the available sample size.

The goal of the discussion in the following sections is the design of experiments which allow the generalisation error to be estimated together with the uncertainties of this estimate, and which allow the performance of methods to be compared. The ability to estimate uncertainties is crucial in a comparative study, since it allows quantification of the probability that the observed differences in performance can be attributed to chance, and may thus not reflect any real difference in performance.

In addition to estimating the overall uncertainty associated with the estimated generalisation error it may sometimes be of interest to know the sizes of the individual effects giving rise to this uncertainty. As an example, it may be of interest to know how much variability there is in performance due to random initialisation of weights in a neural network method. However, there are potentially a large number of effects which could be estimated — and to estimate them all would be rather a lot of work. In the present study I will focus on one or two types of effects that are directly related to the sensitivity of the experiments. These effects will in general be combinations of the basic effects from eq. (2.1). The same general principles can be used in slightly modified experimental designs if one attempts to isolate other effects.

2.2 Previous approaches to experimental design

This section briefly describes some previous approaches to empirical evaluation in the neural network community. These have severe shortcomings, which the methodologies discussed in the remainder of this chapter will attempt to address.

Perhaps the most common approach is to use a single training set $\mathcal{D}_n$, where n is chosen to be some fraction of the total number of cases available. The remaining fraction of the cases are devoted to a test set. In some cases an additional *validation* set is also provided; this set is also used for fitting model parameters (such as weight-decay constants) and is therefore in the present discussion considered to be part of the training set. The empirical mean loss on the test set is reported, which is an unbiased and consistent estimate of the generalisation loss. It is possible (but not common practice) to estimate the uncertainty introduced by the finite test set. In particular, the standard error due to this uncertainty on the generalisation estimate falls with the number of test cases as $n_{\text{test}}^{-1/2}$. Unfortunately the uncertainty associated with variability in the training set cannot be estimated — a fact which is usually silently ignored.

The above simple approach is often extended using n-way cross-testing. Here the data is divided into n equally sized subsets, and the method is trained on n–1 of these and tested on the cases in the last subset. The procedure is repeated n times with each subset left out for testing. This procedure is frequently employed with $n = 10$ [Quinlan 1993]. The advantage that is won at the expense of having to train 10 methods is primarily that the number of test cases is now increased to be the size of the entire data set. We may also suspect that since we have now trained on 10 (slightly) differing training sets, we may be able to estimate the uncertainty in the estimated $G_F(n)$. However, this kind of analysis is complicated by the fact that the training sets are dependent (since several training sets include the same training examples). In particular, one would need to model how the overlapping training sets introduce correlations in the performance estimates, which seems very difficult.

Recently, a book on the StatLog project appeared [Michie et al. 1994]. This is a large study using many sources of data and evaluating 20 methods for classification. In this study, either single training and test sets or n-way cross-testing was used. The authors also discuss the possible use of bootstrapping for estimating performance. However, they do not attempt to evaluate uncertainties in their performance estimates, and ignore the statistical difficulties which their proposals entail.

In the ELENA project [Guérin-Dugué et al. 1995] simple (non-paired) analysis of categorical losses is considered. Although a scheme resembling 5-way cross-testing was used, the subsequent analysis failed to take the dependence between the training sets into account. In the conclusions it is remarked: "... [W]e evaluated this robustness by using a Holdout method on five trials and we considered the minimum and maximum error by computing confidence intervals on these extrema. We obtained large confidence intervals and this measure hasn't been so helpful for the comparisons."

In conclusion, these approaches do not seem applicable to addressing fundamental questions such as whether one method generalises better that another on data from a particular task, since they do not provide ways of estimating the relevant uncertainties. Occasionally, a t-test for significance of difference has been used [Larsen and Hansen 1995; Prechelt 1995], again using a particular training set and using pairing of losses of different methods on test examples.

2.3 General experimental design considerations

The essence of a good experimental design is finding a suitable tradeoff between practicality and statistical power. By practicality of the approach I am referring to the number of experiments required and the complexity of these in terms of both computation time and memory. By statistical power, I mean the ability of the tests to (correctly) identify trends of small magnitude in the experiments. It should be obvious that these two effects can be traded off against each other, since in general we may gain more confidence in conclusions with more repetitions, but this becomes progressively less practical.

The practicality of an approach can be subdivided into three issues: computational time complexity of the experiments, memory and data requirements, and computational requirements for the statistical test. Many learning algorithms require a large amount of computation time for training. In many cases there is a fairly strong (super-linear) dependency between the available number of training cases and the required amount of computation time. However, we are not free to determine the number of training cases in the experimental design, since this is regarded as being externally fixed according to our particular interests. Thus, the main objective is to keep the number of training sessions as low as possible. The time needed for making predictions from the method for test cases may occasionally be of concern, however this requirement will scale linearly with the number of test cases.

The data and memory considerations have different causes but give rise to similar restrictions in the tests. The data requirement is the total number of cases available for constructing training and test sets. For real datasets this will always be a limited number, and in many cases this limitation is of major concern. In cases where artificial data is generated from a simulator, one may be able to generate as many test cases as desired, but for very large sets it may become impractical to store all the individual losses from these tests (which will be necessary when performing paired tests, discussed later in this chapter).

Finally we may wish to limit ourselves to tests whose results are easily computed from the outcomes of the learning experiments. The analysis of some otherwise interesting experimental designs cannot be treated analytically, and approximate or stochastic computation may be needed in order to draw the desired conclusions. Such situations are probably undesirable for the present applications, since it is often difficult to ensure accuracy or convergence with such methods. In such cases people may find the required computational mechanics suspect, and the conclusions will not in general be convincing.

The statistical power of the tests depends on the details of the experimental design. In general, the more training sets and test cases, the smaller the effects that can be detected reliably. But also the distributional assumptions about the losses are of importance. These issues are most easily clarified through some examples. From a purely statistical point of view, the situation is simplest when one can assume independence between experimental observations. As an extreme case, we may consider an experimental design where a method is trained several times using disjoint training sets, and single independently drawn test cases. The analysis of the losses in this case is simple because the observed losses are independent and the Central Limit theorem guarantees that the empirical mean will follow an unbiased Gaussian distribution with a standard deviation scaling as $n^{-1/2}$. However, for most learning methods that we may wish to consider this approach will be computationally prohibitively expensive, and for real problems where the total amount of data is limited, such an approach is much too wasteful of data: the amount of information extracted from each case is far too small.

In order to attempt to overcome the impracticality of this previous design example, we may use the same training set for multiple test cases, thereby bringing down the total number of required training sessions. This corresponds to using (disjoint) test sets instead of individual test cases. Computationally, this is a lot more attractive, since many fewer training sessions are required. We extract more information per training run about the performance of the method by using several test cases. However, the losses are no longer independent, since the common training sets introduce dependencies, which must be accounted for in the analysis of the design. A persistent concern with this design is that it requires several disjoint training and test sets, which may be a problem when dealing with real data sets of limited size. For artificially generated (and very large real) datasets, this design may be the most attractive and its properties are discussed in the following section under the name "hierarchical ANOVA design".

To further increase the effectiveness of the use of data for real learning tasks, we can test all the trained methods on all the available testing data, instead of carving up the test data into several disjoint sets. By doing more testing, we are able to extract more information about the performance of the method. Again, this comes at an expense of having to deal with a more complicated analysis. Now the losses are not only dependent through common training sets but also through common test cases. This design will be discussed in a later section under the title "2-way ANOVA design". This will be the preferred design for real data sets.

The different requirement for disk-storage for the hierarchical and 2-way designs may also be of importance. When methods have been tested, we need to store all the individual losses in order to perform paired comparisons (discussed in detail in the next section). Although disk storage is cheap, this requirement does become a concern when testing numerous methods on large test sets. In this respect the hierarchical design is superior, since losses for more test cases can be stored with the same disk requirements.

Attempts can be made to further increase the effectiveness (in terms of data) of the tests. Instead of using disjoint training sets, one may reuse cases in several training and test sets. The widely used n-way cross-testing mentioned in the previous section is an example of such a design. There are no longer any independencies in these designs, and it becomes hard to find reasonable and justifiable assumptions about how the performance depends on the composition of the training sets. In traditional n-way cross-testing the data is split into n subsets, and one could attempt to model the effects of the subsets individually and neglecting their interactions, but this may not be a good approximation, since one may expect the training cases to interact quite strongly. These difficulties deterred us from

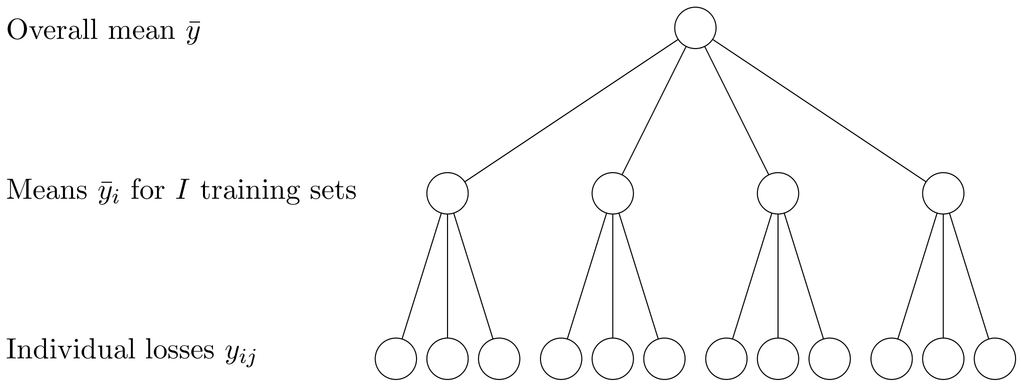

Overall mean $\bar{y}$
Means $\bar{y}_i$ for I training sets
Individual losses y_{ij}

Figure 2.1: Schematic diagram of the hierarchical design. In this case there are $I = 4$ disjoint training sets and $I = 4$ disjoint test sets each containing $J = 3$ cases. Since both training and test sets are disjoint, the average losses for each training set $\bar{y}_i$ are independent estimates of the expected loss μ.

using these designs. It is possible that there is some way of overcoming the difficulties and this would certainly be of importance if one hopes to be able to use small datasets for benchmarking. It should be noted that when n-way cross-testing is usually used in the literature, one does not attempt to estimate uncertainties associated with the performance estimates. In such cases it is not easy to justify the conclusions of the experiments.

2.4 Hierarchical ANOVA design

The simplest loss model that we will consider is the analysis of variance (ANOVA) in the hierarchical design. In this loss model, the learning algorithm is trained on I different training sets. These training sets are disjoint, i.e., a specific training case appears only in a single training set. Associated with each of the training sets there is a test set with J cases. These test sets are also disjoint from one another and disjoint from the training sets.

We train the method on each of the I training sets and for each training set we evaluate the loss on each case in the corresponding test set. A particular training set and the associated test cases will be referred to as an *instance* of the task in the following. We assume that the losses can be modeled by

$$y_{ij} = \mu + a_i + \varepsilon_{ij}. \tag{2.2}$$

Here y_{ij} is the loss on test case j from test set i when the method was trained on training set i. The a_i and ε_{ij} are assumed Normally and independently distributed with

$$a_i \sim \mathcal{N}(0, \sigma_a^2) \qquad\qquad \varepsilon_{ij} \sim \mathcal{N}(0, \sigma_e^2). \tag{2.3}$$

The μ parameter models the mean loss which we are interested in estimating. The a_i variables are called the *effects* due to training set, and can model the variability in the losses that is caused by varying the training set. Note, that the training set effects include all sources of variability between the different training sessions: the different training examples and stochastic effects in training, e.g., random initialisations. The ε_{ij} variables model the residuals; these include the effects of the test cases, interactions between training and test cases and stochastic elements in the prediction procedure. For some loss functions, these Normality assumptions may not seem appropriate; refer to section 2.6 for a further discussion. In the following analysis, we will not attempt to evaluate the individual contributions to the a_i and ε_{ij} effects.

Using eq. (2.2) and (2.3) we can obtain the estimated expected loss and one standard deviation error bars on this estimate

$$\hat{\mu} = \bar{y} \qquad\qquad \mathrm{SD}(\hat{\mu}) = \Big(\frac{\sigma_a^2}{I} + \frac{\sigma_\varepsilon^2}{IJ}\Big)^{1/2}, \tag{2.4}$$

where a hat indicates an estimated value, and a bar indicates an average. This estimated standard error is for fixed values of the σ's, which we can estimate from the losses. We introduce the following means

$$\bar{y}_i = \frac{1}{J}\sum_j y_{ij} \qquad\qquad \bar{y} = \frac{1}{IJ}\sum_i\sum_j y_{ij}, \tag{2.5}$$

and the "mean squared error" for a and ε and their expectations

$$\begin{aligned} \mathrm{MS}_a &= \frac{J}{I-1}\sum_i(\bar{y}_i - \bar{y})^2 & E[\mathrm{MS}_a] &= J\sigma_a^2 + \sigma_\varepsilon^2 \\ \mathrm{MS}_\varepsilon &= \frac{1}{I(J-1)}\sum_i\sum_j(y_{ij} - \bar{y}_i)^2 & E[\mathrm{MS}_\varepsilon] &= \sigma_\varepsilon^2. \end{aligned} \tag{2.6}$$

In ANOVA models it is common to use the following minimum variance unbiased estimators for the σ^2 values which follow directly from eq. (2.6)

$$\hat{\sigma}_\varepsilon^2 = \mathrm{MS}_\varepsilon \qquad\qquad \hat{\sigma}_a^2 = \frac{\mathrm{MS}_a - \mathrm{MS}_\varepsilon}{J}. \tag{2.7}$$

Unfortunately the estimate $\hat{\sigma}_a^2$ may sometimes be negative. This behaviour can be explained by referring to fig. 2.1. There are two sources of variation in $\bar{y}_i$; firstly the variation due to the differences in the training sets used and secondly the uncertainty due to the finitely many test cases evaluated for that training set. This second contribution may be much greater than the former, and empirically eq. (2.7) may produce negative estimates if the variation in $\bar{y}_i$ values is less than expected from the variation over test cases. It is customary to truncate negative estimates at zero (although this introduces bias).

In order to compare two learning algorithms the same model can be applied to the *differences* between the losses from two learning methods k and k'

$$y_{ij} = y_{ijk} - y_{ijk'} = \mu + a_i + \varepsilon_{ij}, \tag{2.8}$$

with similar Normal and independence assumptions as before, given in eq. (2.3). In this case μ is the expected difference in performance and a_i is the training set effect on the difference. Similarly, ε_{ij} are residuals for the difference loss model. It should be noted that the tests derived from this model are known as *paired* tests, since the losses have been paired according to training sets and test cases. Generally paired tests are more powerful than non-paired tests, since random variation which is irrelevant to the difference in performance is filtered out. Pairing requires that the same training and test sets are used for every method. Pairing is readily achieved in DELVE, since losses for methods are kept on disk.

A central objective in a comparative loss study is to get a measure of how confident we can be that the observed difference between the two methods reflects a real difference in performance rather than a random fluctuation. Two different approaches will be outlined to this problem: the standard t-test and a Bayesian analysis.

The idea underlying the t-test is to assume a *null hypothesis*, and compute how probable the observed data or more extreme data is under the sampling distribution given the hypothesis. In the current application, the null hypothesis is $H_0: \mu = 0$, that the two models have identical average performances. It may seem odd to focus on this null hypothesis, when it would seem more natural to draw our conclusions based on $p(\mu < 0|\{y_{ij}\})$ and $p(\mu > 0|\{y_{ij}\})$. The reasoning underlying the frequentist test of H_0 is the following: if we can show that we are unlikely to get the observed losses given the null hypothesis, then we can presumably have confidence in the sign of the difference. Technically, the treatment of composite hypothesis, such as H_0': $\mu < 0$ is much more complicated than a simple hypothesis. Thus, since H_0 can be treated as a simple hypothesis (through exact analytical treatment of the unknown σ_a^2 and σ_ε^2), this is often preferred although it may at first sight seem less appropriate.

Under the null hypothesis, H_0: $\mu = 0$, the distribution of the differences in losses and their partial means can be obtained from eq. (2.8), giving

$$y_{ij} \sim \mathcal{N}(0, \sigma_a^2 + \sigma_\varepsilon^2), \qquad \bar{y}_i \sim \mathcal{N}(0, \sigma_a^2 + \sigma_\varepsilon^2/J) \tag{2.9}$$

for which the variances are unknown in a practical application. The different partial means $\bar{y}_i$ are independent observations from the above Gaussian distribution. A standard result (dating back to Student and Fisher) from the theory of sampling distributions states if $\bar{y}_i$ is independently and Normally distributed with unknown variance, then the t-statistic

$$t = \bar{y}\Big(\frac{1}{I(I-1)}\sum_i (\bar{y}_i - \bar{y})^2\Big)^{-1/2} \tag{2.10}$$

has a sampling distribution given by the t-distribution with $I-1$ degrees of freedom

$$p(t) \propto \Big(1 + \frac{t^2}{I-1}\Big)^{-I/2}. \tag{2.11}$$

To perform a t-test, we compute the t-statistic, and measure how unlikely it would be (under the null hypothesis) to obtain the observed t-value or something more extreme. More precisely, the p-value is

$$p = 1 - \int_{-t}^{t} p(t')dt', \tag{2.12}$$

for which there does not exist a closed form expression; numerically it is easily evaluated via the incomplete beta distribution for which rapidly converging continued fractions are known, [Abramowitz and Stegun 1964]. Notice, that the t-test is two-sided, i.e., that the limits of the integral are $\pm t$, reflecting our prior uncertainty as to which method is actually the better. If in contrast it was apriori inconceivable that the true value of μ was negative we could use a one-sided test, extending the integral to $-\infty$ and getting a p-value which was only half as large.

Very low p-values thus indicate that we can have confidence that the observed difference is not due to chance. Notice that failure to obtain small p-values does not necessarily imply that the performance of the methods are equal, but merely that the observed data does not rule out this possibility, or the possibility that the sign of the actual difference differs from that of the observed difference.

Fig. 2.2 shows an example of the output from DELVE when comparing two methods. Here the estimates of performances $\bar{y}_k$ and $\bar{y}_{k'}$, their estimated difference $\hat{\mu}$ and the standard error on this estimate $\mathrm{SD}(\hat{\mu})$ are given and below the two effects $\hat{\sigma_a}$ and $\hat{\sigma_\varepsilon}$. Finally, the p-value for a t-test is given for the significance of the observed difference.

At this point it may be useful to note that the standard error for the difference estimate $\mathrm{SD}(\hat{\mu})$ is computed using fixed estimates for the standard deviations, given by eq. (2.7),

```
               Estimated expected loss for knn-cv-1:      357.909
                 Estimated expected loss for /lin-1:       397.82
                      Estimated expected difference:     -39.9114
             Standard error for difference estimate:      11.4546

        SD from training sets and stochastic training:      15.3883
     SD from test cases & stoch. pred. & interactions:      271.541

        Significance of difference (T-test), p = 0.0399302

        Based on 4 disjoint training sets, each containing 256 cases and
                 4 disjoint test sets, each containing 256 cases.
```

Figure 2.2: An example of applying this analysis to comparison of the two methods `lin-1` and `knn-cv-1` using the squared error loss function on the task `demo/age/std.256` in DELVE.

where the distribution of $\hat{\mu}$ is Gaussian. However, there is also uncertainty associated with the estimates for these standard deviations. This could potentially be used to obtain better estimates of the standard error for the difference (interpreted as a 68% confidence interval); computationally this may be cumbersome, since it requires evaluations of t from p in eq. (2.12) which is a little less convenient. For reasonably large values of I the differences will be small, and our primary interest is not in these intervals but rather in the p-values (which are computed correctly).

As an alternative to the frequentist hypothesis test, one can adopt a Bayesian viewpoint and attempt to compute the posterior distribution of μ from the observed data and a prior distribution. In the Bayesian setting the unknown parameters are the mean difference μ and the two variances σ_a^2 and σ_ε^2. Following Box and Tiao [1992] the likelihood is given by

$$p(\{y_{ij}\}|\mu,\sigma_a^2,\sigma_\varepsilon^2) \propto \\ (\sigma_\varepsilon^2)^{-I(J-1)/2}(\sigma_\varepsilon^2 + J\sigma_a^2)^{-I/2} \exp\Big(-\frac{J\sum_i(\bar{y}_i-\mu)^2}{2(\sigma_\varepsilon^2+J\sigma_a^2)} - \frac{\sum_i\sum_j(y_{ij}-\bar{y}_i)^2}{2\sigma_\varepsilon^2}\Big). \tag{2.13}$$

We obtain the posterior distribution by multiplying the likelihood by a prior. It may not in general be easy to specify suitable priors for the three parameters. In such circumstances it is sometimes possible to dodge the need to specify subjective priors by using improper non-informative priors. The simplest choices for improper priors are the standard non-informative

$$p(\mu) \propto 1 \qquad p(\sigma_a^2) \propto \sigma_a^{-2} \qquad p(\sigma_\varepsilon^2) \propto \sigma_\varepsilon^{-2}, \tag{2.14}$$

since the variances are positive scale parameters. In many cases the resulting posterior is still proper, despite the use of these priors. However, in the present setting these priors do not lead to proper posteriors, since there is a singularity at $\sigma_a^2 = 0$; the data can be explained (i.e., acquire non-vanishing likelihood) by the σ_ε^2 effect alone and the prior density for σ_a^2

will approach infinity as σ_a^2 goes to zero. This inability to use a non-informative improper prior reflects a real uncertainty in the analysis of the design. For small values of σ_a^2 the likelihood is almost independent of this parameter and the amount of mass placed in this region of the posterior is largely determined by the prior. In other words, the likelihood does not provide much information about σ_a^2 in this region. An alternative prior is proposed in Box and Tiao [1992], setting

$$p(\mu) \propto 1 \qquad p(\sigma_\varepsilon^2) \propto \sigma_\varepsilon^{-2} \qquad p(\sigma_\varepsilon^2 + J\sigma_a^2) \propto (\sigma_\varepsilon^2 + J\sigma_a^2)^{-1}. \tag{2.15}$$

This prior has the somewhat unsatisfactory property that the effective prior distribution depends on J, the number of test cases per training set, which is an unrelated arbitrary choice by the experimenter. On the positive side, the simple form of the posterior allows us to express the marginal posterior for μ in closed form

$$\begin{aligned} p(\mu|\{y_{ij}\}) &= \int_0^\infty \int_0^\infty p(\mu, \sigma_\varepsilon^2, \sigma_a^2) p(\{y_{ij}\}|\mu, \sigma_a^2, \sigma_\varepsilon^2)\, d\sigma_a^2\, d\sigma_\varepsilon^2 \\ &\propto a_2^{-p_2}\, \mathrm{betai}_{a_2/(a_1+a_2)}(p_2, p_1), \end{aligned} \tag{2.16}$$

where betai is the incomplete beta distribution and

$$a_1 = \frac{1}{2}\sum_i \sum_j (y_{ij} - \bar{y}_i)^2 \qquad a_2 = \frac{J}{2}\sum_i (\bar{y}_i - \mu)^2 \qquad p_1 = \frac{I(J-1)}{2} \qquad p_2 = \frac{I}{2}. \tag{2.17}$$

In fig. 2.3 the posterior distribution of μ is shown for a comparison between two learning methods. The p-value from the frequentist test in fig. 2.2 is $p = 0.040$ which is reasonably close to the posterior probability that μ has the opposite sign of the observed difference, which was calculated by numerical integration to be 2.3%. These two styles of analysis are making statements of a different nature, and there is no reason to suspect that they should produce identical values. Whereas the frequentist test assumes that $\mu = 0$ and makes a statement about the probability of the observed data or something more extreme, the Bayesian analysis treats μ as a random variable. However, it is reassuring that they do not differ to a great extent.

There are several reasons that I have not pursued the Bayesian analysis further. The most important reason is that my primary concern was to find a methodology which could be adopted in DELVE, for which the Bayesian method does not seem appropriate. Firstly, because the Bayesian viewpoint is often met with scepticism, and secondly because of analytical problems when attempting to use priors other than eq. (2.15). Perhaps the most promising approach would be to use proper priors and numerical integration to evaluate eq. (2.16) and then investigate how sensitive the conclusions are to a widening of the priors. Sampling approaches to the problem of estimating the posterior may be viable, and open

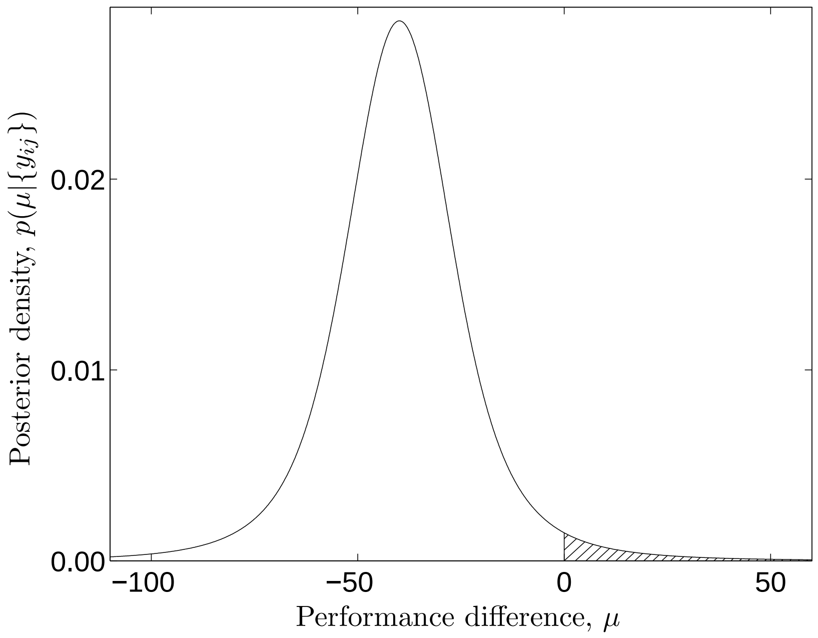


Figure 2.3: Posterior distribution of μ when comparing the `lin-1` and `knn-cv-1` methods on the `demo/age/std.256` data for squared error loss, using eq. (2.16). By numerical integration it is found that 2.3% of the mass lies at positive values for μ (indicated by hatched area).

up interesting possibilities of being able to relax some of the distributional assumptions underlying the frequentist t-test. However, extreme care must be taken when attempting to use sampling methods (such as simple Gibbs sampling) where it may be hard to ensure convergence, since this may leave the conclusions from experiments open to criticism.

2.5 The 2-way ANOVA design

The experimental setup for a 2-way design differs from the hierarchical design in that we use all the test cases for every training session thereby gaining more information about the performances, fig. 2.4. This is more efficient (in terms of data) which may be important if the number of available cases is small. However, the analysis of this model is more complicated. The loss model is:

$$y_{ij} = \mu + a_i + b_j + \varepsilon_{ij}, \tag{2.18}$$

with a_i being the effects for the training sets, b_j the effects for the test cases, and ε_{ij} their interactions and noise. As was the case for the hierarchical design, these effects may have several different components, but no attempt will be made to estimate these individually.

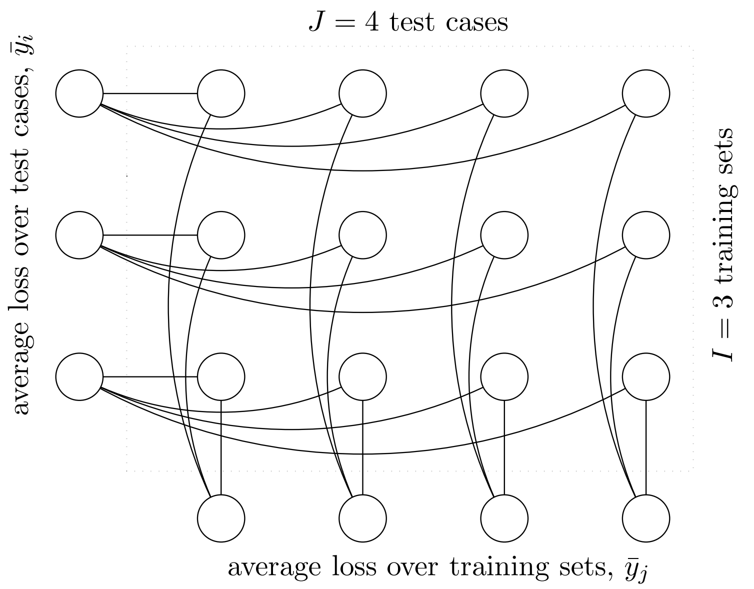

$J = 4$ test cases
average loss over test cases, $\bar{y}_i$
$I = 3$ training sets
average loss over training sets, $\bar{y}_j$

Figure 2.4: Schematic diagram of the 2-way design. There are $I = 3$ disjoint training sets and a common test set containing $J = 4$ cases giving a total of 12 losses. The partial average performances are not independent.

We make the same assumptions of independence and normality as previously

$$a_i \sim \mathcal{N}(0, \sigma_a^2) \qquad b_j \sim \mathcal{N}(0, \sigma_b^2) \qquad \varepsilon_{ij} \sim \mathcal{N}(0, \sigma_\varepsilon^2). \tag{2.19}$$

In analogy with the hierarchical design, these assumptions give rise to the following expectation and standard error

$$\hat{\mu} = \bar{y} \qquad \mathrm{SD}(\hat{\mu}) = \Big(\frac{\sigma_a^2}{I} + \frac{\sigma_b^2}{J} + \frac{\sigma_\varepsilon^2}{IJ}\Big)^{1/2}. \tag{2.20}$$

We introduce the following partial mean losses

$$\bar{y} = \frac{1}{IJ}\sum_i\sum_j y_{ij} \qquad \bar{y}_i = \frac{1}{J}\sum_j y_{ij} \qquad \bar{y}_j = \frac{1}{I}\sum_i y_{ij}, \tag{2.21}$$

and the "mean squared error" for a, b and ε and their expectations:

$$\begin{aligned}
\mathrm{MS}_a &= \frac{J}{I-1}\sum_i(\bar{y}_i - \bar{y})^2 & E[\mathrm{MS}_a] &= J\sigma_a^2 + \sigma_\varepsilon^2 \\
\mathrm{MS}_b &= \frac{I}{J-1}\sum_j(\bar{y}_j - \bar{y})^2 & E[\mathrm{MS}_b] &= I\sigma_b^2 + \sigma_\varepsilon^2 \\
\mathrm{MS}_\varepsilon &= \frac{1}{(I-1)(J-1)}\sum_i\sum_j\big((y_{ij}-\bar{y}) - (\bar{y}_i - \bar{y}) - (\bar{y}_j - \bar{y})\big)^2 & E[\mathrm{MS}_\varepsilon] &= \sigma_\varepsilon^2
\end{aligned} \tag{2.22}$$

Now we can use the empirical values of MS_a, MS_b and MS_ε to estimate values for the σ's:

$$\hat{\sigma_\varepsilon}^2 = \mathrm{MS}_\varepsilon \qquad \hat{\sigma_b}^2 = \frac{\mathrm{MS}_b - \mathrm{MS}_\varepsilon}{I} \qquad \hat{\sigma_a}^2 = \frac{\mathrm{MS}_a - \mathrm{MS}_\varepsilon}{J} \tag{2.23}$$

These estimators are uniform minimum variance unbiased estimators. As before, the estimates for σ_a^2 and σ_b^2 are not guaranteed to be positive, so we set them to zero if they are negative. We can then substitute these variance estimates into eq. (2.20) to get an estimate for the standard error for the estimated mean performance.

Note that the estimated standard error $\hat{\sigma}$ diverges if we only have a single training set (as is common practice!). This effect is caused by the hopeless task of estimating an uncertainty from a single observation. At least two training sets must be used and probably more if accurate estimates of uncertainty are to be achieved.

Another important question is whether the observed difference between two learning procedures can be shown to be significantly different from each other. To settle this question we again use the model from eq. (2.18), only this time we model the difference between the losses of the two models, k and k':

$$y_{ijk} - y_{ijk'} = \mu + a_i + b_j + \varepsilon_{ij}, \tag{2.24}$$

under the same assumptions as above. The question now is whether the estimated overall mean difference $\hat{\mu}$ is significantly different from zero. We can test this hypothesis using a quasi-F test [Lindman 1992], which uses the F statistic with degrees of freedom:

$$\begin{aligned} F_{\nu_1,\nu_2} &= (\mathrm{SS}_m + \mathrm{MS}_\varepsilon)/(\mathrm{MS}_a + \mathrm{MS}_b), \quad \text{where} \quad \mathrm{SS}_m = IJ\bar{y}^2 \\ \nu_1 &= (\mathrm{SS}_m + \mathrm{MS}_\varepsilon)^2/(\mathrm{SS}_m^2 + \mathrm{MS}_\varepsilon^2/((I-1)(J-1))) \\ \nu_2 &= (\mathrm{MS}_a + \mathrm{MS}_b)^2/(\mathrm{MS}_a^2/(I-1) + \mathrm{MS}_b^2/(J-1)). \end{aligned} \tag{2.25}$$

The result of the F-test is a p-value, which is the probability given the null-hypothesis ($\mu = 0$) is true, that we would get the observed data or something more extreme. In general, low p-values indicates a high confidence in the difference between the performance of the learning procedures.

Unfortunately this quasi-F test is only approximate even if the assumptions of independence and Normality are met. I have conducted a set of experiments to clarify how accurate the test may be. For our purposes, the most serious mistake that can be made is what is normally termed a type I error: concluding that the performances of two methods are different when in reality they are not. In our experiments, we would not normally anticipate that the performance of two different methods would be exactly the same, but if we ensure that the test only rarely strongly rejects the null hypothesis if it is really true, then presumably it will be even rarer for it to declare the observed difference significant if its sign is opposite to that of the true difference.

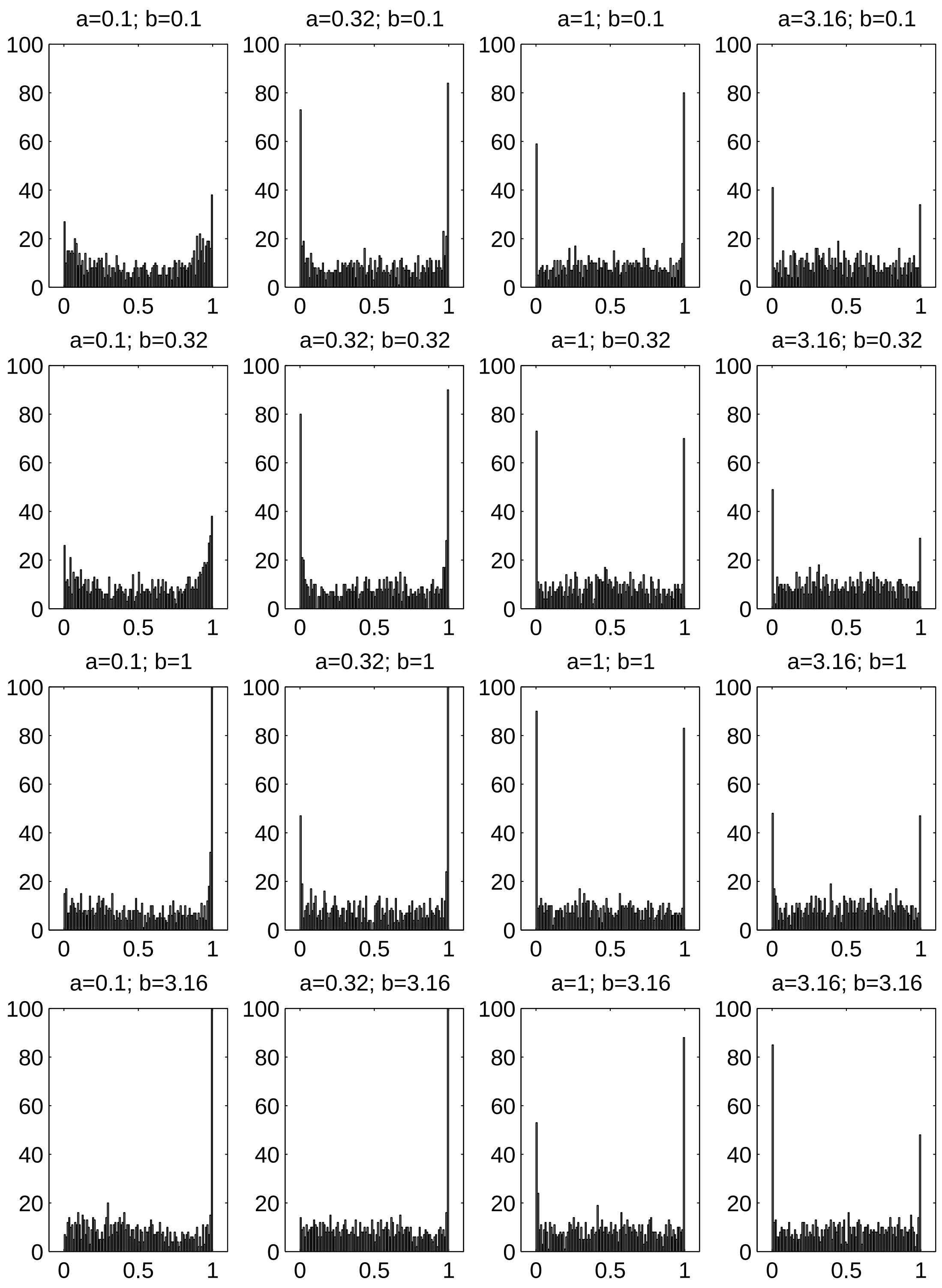


Figure 2.5: Experiments using 2 training instances, showing the empirical distribution of 1000 p-values in 100 bins obtained from fake observations from under the null hypothesis. Here a and b give the standard deviations for the training set effect and test case effect respectively.

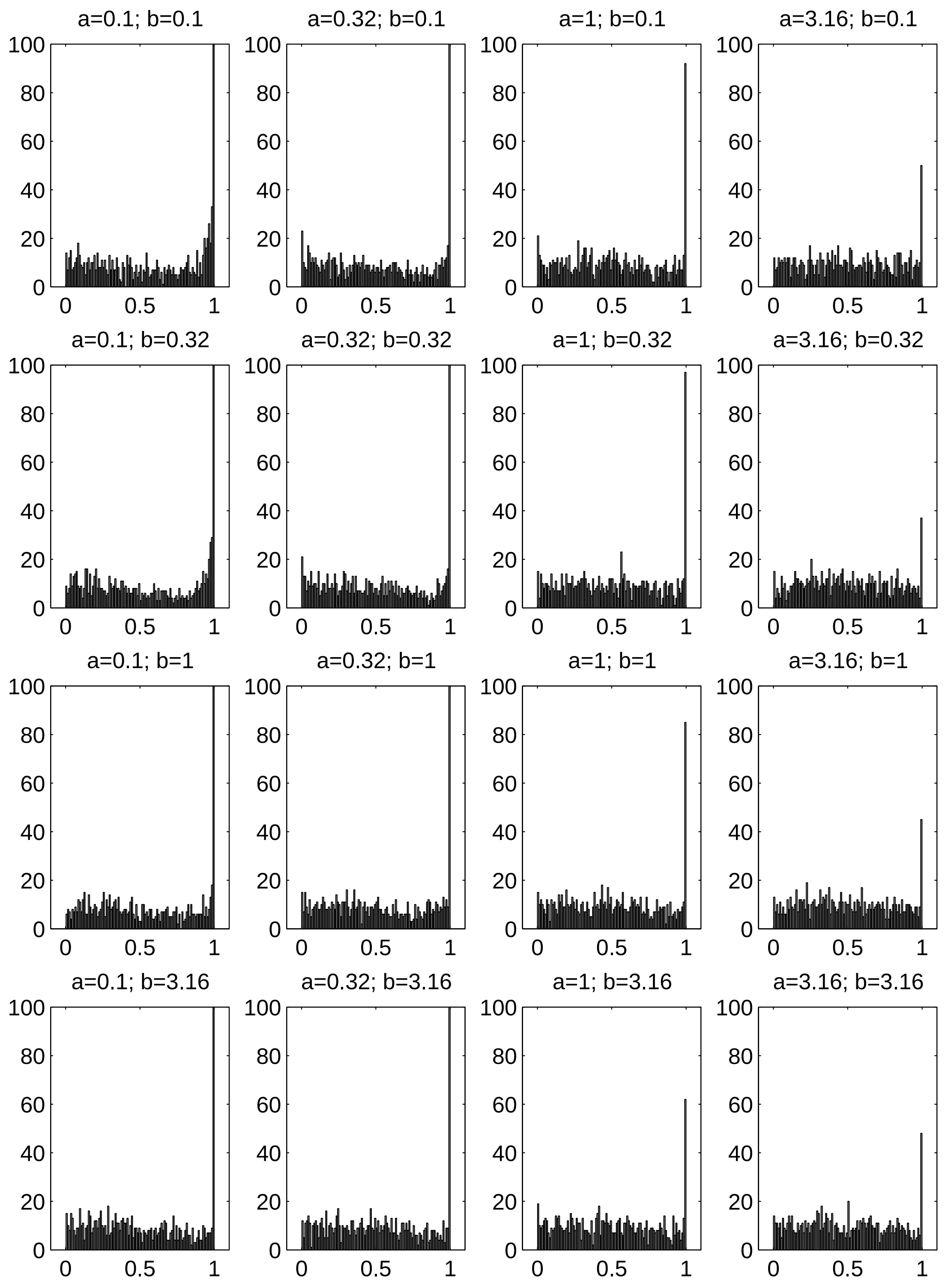


Figure 2.6: Experiments using 4 training instances, showing the empirical distribution of p-values obtained from fake observations from under the null hypothesis. Here a and b give the standard deviations for the training set effect and test case effect respectively.

I generated mock random losses under the null hypothesis, from eq. (2.18) and (2.19) with $\mu \equiv 0$ and $\sigma_\varepsilon = 1.0$ for various values of σ_a and σ_b. The unit σ_ε simply sets the overall scale without loss of generality. I then computed the p-value for the F-test for repeated trials. In fig. 2.5 and 2.6 are histograms of the resulting p-values. Ideally these histograms ought to show a uniform distribution — the reason why these do not (apart from finite sample effects) is due to the approximation in the F-test. The most prominent effects are the spikes in the histograms around $p = 0$ and $p = 1$. The spikes at $p = 1$ are not of great concern since the test here is strongly in favor of the (true) null hypothesis. This may lead to a reduced *power* of the test, but not to type I errors. The spikes that occur around $p = 0$ are directly of concern. Here the test is strongly rejecting the null hypothesis, leading us to infer that the methods have differing performance when in fact they do not. This effect is only strong in the case where there are only 2 instances and where the training set effect is large. With 4 instances (and 8, not shown) these problems have more or less vanished. To avoid interpretative mistakes whenever there are fewer than 4 instances and the computed p-value is less than 0.05, the result is reported by DELVE as $p < 0.05$.

2.6 Discussion

One may wonder what happens to the tests described in earlier sections when the assumptions upon which they rely are violated. The independence assumptions should be fairly safe, since we are carefully designing the training and test sets with independence in mind. The Normality assumptions however, may not be met very well. For example, it is well known that when using squared error loss, one often sees a few outliers accounting for a large fraction of the total loss over the test set. In such cases one may wonder whether squared error loss is really the most interesting loss measure. Given that we insist on pursuing this loss function, we need to consider violations of Normality.

The Normality assumptions of the experimental designs are obviously violated in the case of loss estimation for squared error loss functions, which are guaranteed positive. This objection disappears for the comparative designs where only the loss differences are assumed Normal. However, it is well known that extreme losses may occour — so Gaussian assumptions may be inappropriate. As a solution to this problem, Prechelt [1994] suggests using the log of the losses in t-tests after removing up to 10% “outliers”. I do not advocate this approach. The loss function should reflect the function that one is interested in minimising. If one isn’t concerned by outliers then one should choose a loss function that reflects this. Removing outlying losses does not appear defensible in a general application. Also, method

A having a smaller expected log loss than method B does not imply anything about the relation of their expected losses.

Generally, both the t-test and F-test are said to be fairly robust to small deviations from Normality. Large deviations in the form of huge losses from occasional outliers turn out to have interesting effects. For the comparative loss models described in the previous sections, the central figure determining the significance of an observed difference is the ratio of the mean difference to the uncertainty in this estimate $\bar{y}/\hat{\sigma}$, as in eq. (2.10). If this ratio is large, we can be confident that the observed difference in not due to chance. Now, imagine a situation where $\bar{y}/\hat{\sigma}$ is fairly large; we select a loss difference y' at random and perturb it by an amount ξ, and observe the behaviour of the ratio as we increase ξ

$$\frac{\bar{y}}{\hat{\sigma}} = \frac{\sum y_i + \xi}{\sqrt{n(\sum y_i^2 + \xi^2 + y'\xi) - (\sum y_i + \xi)^2}} \rightarrow \frac{1}{\sqrt{n-1}} \quad \text{when} \quad \xi \rightarrow \infty. \tag{2.26}$$

Thus, for large values of n the tests will tend to become less significant, as the magnitude of the outliers increase. Here we seem to be lucky that outliers will not tend to produce results that appear significant but merely reduce the power of the test. However, this tendency may in some cases have worrying proportions. In fact, let's say we are testing two methods against each other, and one seems to be doing significantly better than the other. Then the losing method can avoid losing face in terms of significance by increasing its loss on a single test case drastically. Because the impact on the mean is smaller than the impact on $\hat{\sigma}$ for such behaviour, the end result for large ξ is a slightly worse performance for the bad method, but insignificant test results. This scenario is not contrived; I have seen its effects on many occasions and we shall see it in Chapter 5.

This somewhat unsatisfactory behaviour arises from the symmetry assumptions in the loss model. If the losses for one model can have occasional huge values, and the distribution of loss differences is assumed symmetric, it could also happen (although it didn't) that the other model would have a huge loss, hence the insignificant result. Clearly, this is not exactly what we had in mind. There may be cases where these assumptions are reasonable, but there are situations where some methods may tend to make wild predictions while others are more conservative.

It is possible that these deficiencies could be overcome in a Bayesian setting that allowed for non-Gaussian and skew distributional assumptions. It seems obvious that great care must be taken when designing such a scheme, both with respect to its theoretical properties as well as provisions for a satisfactory implementation of the required computations.

Another idea as to how this situation could be remedied is to allow the “winning” method to perturb the losses of the “losing” method, subject to the constraint that losses of the losing method may only be lowered. This may in many cases alleviate the problems of insignificance in situations plagued by extreme losses in a competing method. Several questions remain open in respect to this approach. What is the sampling distribution for the obtained p-values under the null hypothesis? Is there a unique (and simple) way of figuring out which losses to perturb and by how much? I have not pursued these ideas further, but this may well be worthwhile. For the time being, it underlines that one should always consider both the mean difference in performance as well as the p-value for the test. This will also help reduce the importance of very small p-values when they are associated with negligible reductions in loss.

The loss models considered in this chapter have mainly been developed with continuous loss functions in mind. Continuous loss functions are used when the outputs are continuous, and tasks of classification can similarly be handled if one has access to the output class probabilities (which are continuous). However, it is also quite common to use the binary 0/1-loss function for classification. It is not quite obvious how well the present loss models will work for binary losses. Clearly, the assumptions about Normality are not appropriate — but they will probably not lead to ridiculous conclusions. It does not seem straightforward to design more appropriate models for discrete losses that allow for the necessary components of variability. An empirical study of tests of difference in performance of learning methods for binary classification has appeared in [Dietterich 1996].

Chapter 3

Learning Methods

3.1 Algorithms, heuristics and methods

A prerequisite of measuring the performance of a learning method is defining exactly what the method is. This may seem like a trivial statement, but a detailed investigation reveals that it is uncommon in the neural network literature to find a description of an algorithm that is detailed enough to allow replication of the experiments — see [Quinlan 1993; Thodberg 1996] for examples of unusually detailed descriptions. For example, an article may propose to use part of the training data for a neural network as a validation set to monitor performance while training and to stop training when a minimum in validation error is encountered (this is known as early stopping). I will refer to such a description as an *algorithm*. This algorithm must be accompanied by details of the implementation, which I will call *heuristics* in order to produce a *method* which is applicable to practical learning problems. In this example, the heuristics would include details such as the network architecture, the minimization procedure, the size of the validation set, rules for how to determine whether a minimum in validation error was reached, etc.

It is often appealing to think of performance comparisons in terms of algorithms and not heuristics. For example, one may wish to make statements like: "Linear models are superior to neural networks on data from this domain". In this case we are clearly talking about algorithms, but as I have argued above, the empirical assessments supporting such statements necessarily involve the methods — including heuristics. We hope that in most cases the exact details of the heuristics are not crucial to the performance of the method, so that it will be reasonable to generalise the results of the methods to the algorithm itself. It should be stressed that the experimental results involving methods are the objective basis of the more subjective (but more useful) generalisations about algorithms. A more principled approach of investigating several sets of heuristics for each algorithm would be extremely arduous and would still not address the central issue of attempting to project experimental results to novel applications.

I focus my attention on automatic methods, i.e., methods that can be applied without human intervention. The reason for this choice is primarily a concern about reproducibility. It may be argued that for practical problems one should allow a human expert to design special models that take the particular characteristics of the learning problem into account. This does not rule out the usefulness of automatic procedures as aids to an expert. Also, it may be possible to invent heuristics which embody some of the "common sense" of the expert — however, it turns out that this can be an extremely difficult endeavor. My approach is to try to develop methods with sufficiently elaborate heuristics that the method cannot be improved upon by a simple (well documented) modification. I require the methods to be automatic, but I monitor the progress of the algorithm and take note of the cases where the heuristics seem to break down, in order to be able to identify the reasons for poor performance.

The primary target of comparisons is the predictive performance of the methods. However, it does not seem reasonable to completely ignore computational issues, such as cpu time and memory requirements. For many algorithms one may expect there to be a tradeoff between predictive accuracy and cpu time — for example when training an ensemble of networks, we may expect the performance to improve as the ensemble gets larger. I wish to study algorithms that have a reasonably large amount of cpu time at their disposal. For many practical learning problems a few days of cpu time on a fast computer would typically not seem excessive. However, for practical reasons I will limit the computational resources to a few hours per task.

It turns out that it is convenient from a practical point of view to develop heuristics for a particular amount of cpu time, so that the algorithm itself can make choices based on the amount of time spent so far, etc. As an example, consider the case of training an ensemble of 10 networks. In general, reasonable heuristics for this problem are difficult to devise because it may be very hard to determine how long it is necessary to train the individual nets for. If we have a fixed time to run the algorithm, we may circumvent this problem by simply training all nets for an equal amount of time. Naturally, it may turn out that none of the nets were trained well in this time; indeed, it may turn out to have been better to train a single net for the entire allowed period of time, instead of trying an ensemble of 10 nets. I have used this convenient notion of a cpu time constraint for many of the methods, although this may not correspond well to realistic applications. In the experiments, the algorithms will be tested for different amounts of allowed time, and from these performance measures it is usually possible to judge whether the algorithm could perform better given more time.

3.2 The choice of methods

In this thesis, experiments are carried out using eight different methods. Six of these methods will be described in the remainder of this chapter and the two methods relying on Gaussian processes will be developed in the following chapter.

Two of the methods, a linear method called `lin-1` and a nearest neighbor method using cross-validation to choose the neighborhood size called `knn-cv-1`, rely on simple ideas that are often used for data modeling. These methods are included as a "base-line" of performance, to give a feel for how well simple methods can be expected to perform on the tasks.

Two versions of the MARS (Multivariate Regression Splines) method have been included. This method was developed by Friedman [1991], who has also supplied the software. This method is not described in detail in this thesis, since it has been published by Friedman. The primary goal of including these methods is to provide some insight into how neural network methods compare to methods developed in the statistics community with similar aims.

The `mlp-ese-1` method relies on ensembles of neural networks trained with early stopping. This method is included as an attempt at a thorough implementation of the commonly used early stopping paradigm. The intention of including this method is to get an impression of the accuracy that can be expected from this widely-used technique.

The `mlp-mc-1` method implements Bayesian learning in neural networks. The software for this method was developed by Neal [1996]. This method uses Monte Carlo techniques to fit the model and may be fairly computer intensive. Given enough time, one may expect this method to have very good predictive performance. It should thus be a strong competitor.

It would be of obvious interest to include many other promising methods in this study. Algorithms which seem of particular interest include the “Evidence” framework developed by MacKay [1992a] and methods relying on weight-decay and pruning following the ideas of [Le Cun et al. 1990]. However, it has turned out to be very difficult to design appropriate automating heuristics for these algorithms. The Evidence methods seem quite sensitive to initial values of the regularising constants and may not work well for large networks. For all the algorithms it is difficult to automatically select network sizes and reasonable numbers of training iterations.

One of the motivations behind the DELVE project was to enable the advocates of various algorithms themselves to present heuristics for their algorithms which could then be tested in the DELVE environment. This avoids the common problem of people comparing their highly tuned methods to “crummy versions” of competing methods. These considerations have motivated my decision not to implement a large number of methods. Instead I present results that constitute a challenge to the designers of those methods.

The following sections contain descriptions and discussions of the methods that were studied, except for the Gaussian process methods which are discussed in the next chapter. First a few general comments: None of the methods include detailed specifications of any kind of preprocessing. It is assumed that the default preprocessing in DELVE is applied. For the datasets considered in this thesis, binary values are encoded using the values 0 and 1 and real inputs are rescaled by a linear transformation such that that the training set has zero median and unit average absolute deviation from the median. This is intended as a robust way of centering and scaling attributes. For some problems, this scaling might be inappropriate, e.g., if the relative scales of inputs convey important information; such special cases are not considered further here. The performance measurements will focus on three loss functions: squared error loss, absolute error loss and negative log predictive loss. Evaluation of this last loss type requires that the method produces a predictive distribution, and the loss is the negative log density of the test targets under the predictive distribution. An interesting aspect of this loss type is that the method is forced to know about the size of uncertainties in its predictions in order to be able to do well — in contrast to the situation with the two other more commonly used loss functions.

3.3 The linear model: `lin-1`

The linear model is one of the most popular models used in data analysis. The reason for its popularity is that it is both conceptually and computationally simple to fit a linear model, and the resulting model is easily given an intuitive representation. The most prominent deficiency of the linear model is its strong assumption about the true relationship in the data; for data which do not conform well to a linear model, predictions may be inaccurate and over-confident.

My implementation of the linear model is called `lin-1`. Details of the implementation are given in appendix A. The algorithm for fitting a linear model is well known and the only heuristic necessary is a principled way of handling the situation where the linear system used to determine the parameters of the model is close to singular.

For simplicity, I will initially assume that the targets are scalar; a simple extension to handle multiple outputs will be given later. The linear model is defined as

$$f_w(x) = \sum_{i=1}^{m+1} x_i w_i, \tag{3.1}$$

where $w_i,\ i = 1, \ldots, m+1$ are the $m+1$ model parameters, w_{m+1} being the bias and $x_i,\ i = 1, \ldots, m$ are the inputs, augmented by $x_{m+1} \equiv 1$ to take care of the bias. The model is fit to the training data $\mathcal{D} = \{x^{(i)}, t^{(i)} | i = 1 \ldots n\}$ by maximum likelihood, assuming zero mean Gaussian noise with variance σ^2. The likelihood is

$$p\big(t^{(1)}, \ldots, t^{(n)} \big| x^{(1)}, \ldots, x^{(n)}, w, \sigma^2\big) \propto \prod_{c=1}^{n} \exp\Big(-\frac{\big(f_w(x^{(c)}) - t^{(c)}\big)^2}{2\sigma^2}\Big). \tag{3.2}$$

The maximum likelihood estimate for the weights w_{ML} is independent of σ^2 and is found by minimizing the cost function

$$E(w) = \sum_{c=1}^{n} \big(f_w(x^{(c)}) - t^{(c)}\big)^2, \tag{3.3}$$

with respect to the model parameters. Notice that the Gaussian noise assumption gives rise to a squared error cost function; this cost function will always be used regardless of whatever *loss function* we choose to evaluate the linear model. The solution to this optimization problem is well known from linear algebra; if the solution is unique, then

$$\frac{\partial E(w)}{\partial w_j} = \mathbf{0} \;\Rightarrow\; \sum_{c=1}^{n} \Big(\sum_{i=1}^{m+1} w_i x_i^{(c)} - t^{(c)} \Big) x_j^{(c)} = \mathbf{0} \;\Rightarrow\; w_{\text{ML}} = \mathbf{A}^{-1}\mathbf{b}$$
$$\text{where } \mathbf{A}_{ii'} = \sum_{c=1}^{n} x_i^{(c)} x_{i'}^{(c)}, \quad \mathbf{b}_i = \sum_{c=1}^{n} t^{(c)} x_i^{(c)}, \quad i, i' = 1, \ldots, m+1. \tag{3.4}$$

If $\mathbf{A}$ is ill-conditioned, numerical evaluation of eq. (3.4) may be troublesome, and even if fairly accurate solutions could be obtained, these would not necessarily lead to good model predictions. In an attempt to define a method which has a high degree of reproducibility (i.e., that would produce the same results on another machine) I propose choosing w such that directions in input space with insufficient variation in the training set are ignored by the model. This can conveniently be computed using singular value decomposition (SVD), see for example [Press et al. 1992]. The decomposition is $\mathbf{A} = \mathbf{U}\,\text{diag}(\lambda_i)\mathbf{V}^{\text{T}}$, where $\mathbf{U}$ and $\mathbf{V}$ are orthonormal matrices and λ is a vector of length $m+1$ containing the singular values of $\mathbf{A}$. A regularised $\tilde{\mathbf{A}}^{-1}$ can be computed by setting $1/\lambda_i = 0$ for those i whose λ_i are too small in

$$\tilde{\mathbf{A}}^{-1} = \mathbf{V}\,\text{diag}(1/\lambda_i)\mathbf{U}^{\text{T}}. \tag{3.5}$$

The exact criterion for regularisation is: set $1/\lambda_i = 0$ whenever $\lambda_i < 10^{-6}\max_j(\lambda_j)$, i.e., whenever $\mathbf{A}$ is close to singular. The constant of 10^{-6} is chosen as a rather conservative estimate of machine precision which will not interfere when $\mathbf{A}$ is well-conditioned. The condition number of $\mathbf{A}$ depends on the scale of the inputs, so the procedure should always be used in conjunction with the standard normalisations provided by DELVE. The (modified) maximum likelihood weights can then be computed as

$$\tilde{w}_{\text{ML}} = \tilde{\mathbf{A}}^{-1}\mathbf{b}. \tag{3.6}$$

We now derive the predictive distribution for the model. For simplicity we will derive the predictive distribution for a fixed estimate of the noise, and only account for uncertainty in the predictions arising from the estimated noise inherent in the data and from uncertainty in the estimate for the weights. The noise is estimated by the standard unbiased estimator

$$\hat{\sigma}^2 = \frac{1}{n-k} \sum_{c=1}^{n} \big(f_{\tilde{w}_{\text{ML}}}(x^{(c)}) - t^{(c)} \big)^2 \tag{3.7}$$

where k is the number of parameters in w_{ML} which were fit by the data; this is $m+1$ minus 1 for every singular value whose reciprocal was set to zero in eq. (3.5). This estimate may break down if there are too few training cases (if $n \leq m+1$), in which case we can't compute a predictive distribution. Assuming an improper uniform prior on the weights, the posterior for the weights is proportional to the likelihood. Thus, the predictive distribution for a test case with input $x^{(n+1)}$ is

$$\begin{aligned} &p\big(t^{(n+1)}\big|\mathcal{D}, x^{(n+1)}, \hat{\sigma}^2\big) \\ &\qquad\propto \int p\big(t^{(n+1)}\big|\mathcal{D}, x^{(n+1)}, \hat{\sigma}^2\big) p\big(w\big|\mathcal{D}, \hat{\sigma}^2\big) d^{m+1}w \\ &\qquad\propto \int p\big(t^{(n+1)}\big|x^{(n+1)}, w, \hat{\sigma}^2\big) p\big(t^{(1)}, \ldots, t^{(n)}\big|x^{(1)}, \ldots, x^{(n)}, w, \hat{\sigma}^2\big) d^{m+1}w \\ &\qquad\propto \int \exp\Big(-\frac{\big(f_w(x^{(n+1)}) - t^{(n+1)}\big)^2}{2\hat{\sigma}^2} - \frac{1}{2}(w - \tilde{w}_{\mathrm{ML}})^{\mathrm{T}} \tilde{\mathbf{A}} (w - \tilde{w}_{\mathrm{ML}})\Big) d^{m+1}w. \end{aligned} \tag{3.8}$$

This Gaussian integral can be solved exactly; we get a Gaussian predictive distribution with mean and variance

$$p\big(t^{(n+1)}\big|\mathcal{D}, x^{(n+1)}, \hat{\sigma}^2\big) \sim \mathcal{N}(\hat{\mu}, \hat{e}^2), \qquad \text{where} \qquad \begin{aligned} \hat{\mu} &= f_{\tilde{w}_{\mathrm{ML}}}(x^{(n+1)}) \\ \hat{e}^2 &= \hat{\sigma}^2 + (x^{(n+1)})^{\mathrm{T}} \tilde{\mathbf{A}}^{-1} x^{(} \end{aligned}$$

The optimal point prediction for any symmetric loss function is given by $f_{\tilde{w}_{\mathrm{ML}}}(x)$. Consequently, these predictions can be used for both absolute error and squared error loss functions. For the negative log density loss function, we compute the log of the density of the targets under the predictive Gaussian distribution

$$\log p\big(t^{(n+1)}\big|\mathcal{D}, x^{(n+1)}, \hat{\sigma}^2\big) = -\frac{1}{2}\log(2\pi\hat{e}^2) - \frac{(\hat{\mu} - t)^2}{2\hat{e}^2}. \tag{3.10}$$

For tasks that have multiple outputs, we can re-use the decomposition of $\mathbf{A}$ from eq. (3.5) for every set of weights. The maximum likelihood weights and inherent noise estimates can be found by using eq. (3.6) and (3.7) for each target attribute, and point predictions can be obtained from the maximum likelihood weights as before. For the log density predictions we assume that the joint density for the targets is Gaussian with a diagonal covariance matrix, such that the density of the targets can be obtained by summing (in the log domain) contributions of the form in eq. (3.10). This is equivalent to assuming that the residual errors in the outputs are independent.

This completes the definition of the linear model; following this recipe the model will always be uniquely defined by the data. Many elaborations to this basic linear model exist, but they will not be pursued further here.

The computational complexity involved in fitting `lin-1` is $O(nm^2 + m^3)$, for computing and decomposing the $\mathbf{A}$ matrix respectively. Even for fairly large tasks this can usually be considered trivial, since it scales only linearly with n.

3.4 Nearest neighbor models: `knn-cv-1`

Nearest neighbor models are popular non-parametric memory-based models. I will consider a simple k nearest neighbor model, which in the context of DELVE will be called `knn-cv-1`. I will attempt to define a nearest neighbor method that has reproducible results, that does not depend heavily on details of implementation, that has few "free" parameters, and that behaves reasonably under a broad variety of conditions. Also, I wish the method to be applicable to our three standard loss functions (absolute error, squared error and negative log density loss).

Simple nearest neighbor models do not require any training. The algorithm for making predictions involves searching through the training cases to find those whose inputs are closest to the inputs of the test case, and then using some kind of weighted average of the targets of these neighbors as a prediction. Neighborhoods can either be defined in terms of a kernel (e.g. Gaussian) which supplies weighting factors, or in terms of the number of neighbors, k, to use (or hybrids of these). Although methods using kernels have the intuitively appealing ability to weight neighbors according to distance, they will not be used here. Firstly, they involve the somewhat arbitrary choice of kernel shape, and secondly, the choice of width for the kernel (a continuous equivalent of the discrete choice of k) may be plagued by local minima, unless the width itself is expressed in terms of distances to neighbors. This makes the procedure unsuitable as a reproducible base-line nearest neighbor approach. Instead, I will use the k nearest neighbor approach with uniform weighting of neighbors for predictions. More sophisticated methods are certainly possible; for regression tasks it is common to fit local linear models to the neighbors as in LOESS [Cleveland 1979], but these will not be pursued further here.

We also need to define the metric in which to measure closeness; I will use the simple and common choice of Euclidean distance. Many extensions of nearest neighbor algorithms exist which attempt to adapt the distance metric [Lowe 1995; Hastie and Tibshirani 1996], but these more complicated algorithms will not be pursued here.

We need to resolve what to do if two or more cases have the same distance as the k'th nearest neighbor. In an attempt to make the algorithm less sensitive to round off effects on different floating point arithmetic implementations, I will further set some small distance tolerance, below which cases are deemed to have *tied* distances. Two cases are defined to be ties if their squared distances to the test point differ by less than 10^{-6}. I propose the following scheme for making a prediction for a test case:

1. Sort the cases by distances, placing ties in arbitrary order.
2. Find all the neighbors which have the same distance as the k'th case in the ordering (including both cases earlier and later in the list) and average their targets.
3. Include this average of ties in the final prediction with a weight equal to the number of ties that were at or earlier than k in the list.

This procedure together with the small tolerance on deciding whether cases are ties, should help in avoiding differences due to finite precision arithmetic.

We need to find an appropriate neighborhood size, k. In general, we expect the optimal choice of k to depend on the particular dataset and on the number of available training points. As the number of training cases increases, the nearest neighbor method will be consistent (a *consistent* estimator gets the right answer in the limit of an infinitely large sample size) only if k grows. An appealing idea is to find the best k by leave one out cross validation. Using this procedure, we leave out each of the training cases in turn, find its k' nearest neighbors (handling ties appropriately) and compute the loss associated with predictions using this neighborhood. This is repeated for all $k' = 1 \ldots n{-}1$, and k is selected to be the k' with the smallest average loss. The leave-one-out procedure gives an unbiased estimate for the situation where we had $n{-}1$ training cases, which in the case of fairly large n should be close to optimal for the current training set size.

The neighborhood size k is estimated using the desired loss function. For the squared error loss function, the average of the targets of the nearest neighbors are used as predictions. If the outputs are multidimensional, then averages are computed for each dimension. For absolute error loss we use the median along each dimension as a prediction.

Some special problems present themselves for negative log density loss. We have to form a predictive distribution based on the k nearest neighbors. The simplest idea is to fit a Gaussian to the targets of the k neighbors, again weighting the neighbors uniformly. The mean for the Gaussian is chosen simply to be the empirical mean of the k nearest neighbors. Two special cases arise when estimating the variance. Firstly, if the leave one out procedure estimates $k = 1$, then we cannot estimate both the mean and variance for the predictive distribution based on the single nearest neighbor. Secondly, if all k neighbors have exactly the same targets (or targets that differ only very slightly), the naive empirical variance estimate is undesirable since it leads to unbounded losses. Both of these problems can be addressed by regularising the variance estimate

$$v = \frac{1}{k}\Big(\sum_{k'=1}^{k}(y^{(k')} - \bar{y})^2 + \varsigma^2\Big), \tag{3.11}$$

where ς^2 is the regulariser, $y^{(k')}$ is the targets of the k''th nearest neighbor and $\bar{y}$ is the average of the targets for the k nearest neighbors. A convenient value to choose for ς^2 is the average squared difference between targets corresponding to first nearest neighbors, the average extending over all training cases. This can be interpreted as a global estimate of the variance which, whenever available, is modified by local information. Thus the regulariser will act much like a conjugate prior. In the case of $k = 1$, no local information about variance is available, and the sum over k' will vanish from eq. (3.11) leaving just the global term. As k grows, the amount of local information increases and the importance of the regulariser decreases. The problem of vanishing variance estimates is also addressed by this approach, since normalisation of the targets before learning guarantees that $\varsigma^2 > 0$ (except for the pathological case where all available training cases have identical targets).

The computational demands of the leave-one-out procedure for training the `knn-cv-1` model depend on how it is implemented. We need to leave out each of the n cases in turn; for each of these we compute the distance to all other neighbors, which takes a total time of $O(n^2m)$. Then we need to sort these distances, requiring $O(n \log n)$. Now for each of the $n-1$ possible values of k we need to compute the mean, variance and median of the neighbors. For simple implementations, computation of these values take $O(n)$ for each of these estimates. However, it should be noted that it is possible to re-use partial results obtained with other values of k; means and variances can be updated in constant time and the median can (through use of a heap data structure) be updated in $O(\log n)$. The total computational effort involved in finding k is thus $O\big(n^2(m+n+\log n)\big)$ for a simple implementation or $O\big(n^2(m+\log n)\big)$ for a more elaborate implementation. The current version of `knn-cv-1` uses the simple implementation.

Once the appropriate value of k is found, we can make predictions in time $O\big(mn+k+n\log n\big)$ for each test case, if the prediction algorithm sorts all training cases according to distance (the current implementation of `knn-cv-1` uses this approach). If instead of sorting we attempt to locate the k closest neighbors, we can replace $\log n$ in the previous expression by k, but it is not clear which is best in general, since we expect the optimal k to grow (at a sub-linear rate) as n grows.

3.5 MARS with and without Bagging

The Multivariate Adaptive Regression Splines (MARS) method of Friedman [1991] has also been tested. This is a fairly well known method for non-linear regression for high dimensional data from the statistics community. A detailed description of MARS will not be given here, see [Friedman 1991]. The following is a simplistic account of MARS which gives a flavor of the method. The input space is carved up into several (overlapping) regions in which splines are fit. The fit is built using first a constructive phase, which introduces input regions and splines, followed by a pruning phase. The final model has the form of a sum of products of univariate splines; it is a continuous function (with continuous derivatives) and is additive in the sets of variables allowed to interact.

Friedman has supplied his FORTRAN implementation of MARS (version 3.6). Two versions of the method have been tested. The original implementation is given the DELVE name `mars3.6-1`. Since MARS is not very computationally demanding, it can be used in conjunction with the Bagging procedure of Breiman [1994]. Using this method, one trains MARS on a number of bootstrap samples of the training set and averages the resulting predictions. The bootstrap samples are generated by sampling the original training set with replacement; samples of the same size as the original training set are used. Only one set of predictions is generated regardless of the loss function — the same predictions are used for the absolute error loss function and the squared error loss function. The negative log density loss function is not applicable to the current version of MARS. The bagged version of MARS is referred to as `mars3.6-bag-1`.

The following parameter settings have been used for MARS: 50 bootstrap repetitions were used in the bagging procedure, the maximum number of basis functions was 15, the maximum number of variables allowed to interact was 8. The computational cost of applying this method is fairly modest. Using a training set with 32 inputs and 1024 training cases, the 50 bootstrap replications take a total of 5 minutes on a 200MHz R4400/R4010 processor.

3.6 Neural networks trained with early stopping: `mlp-ese-1`

In this method, the predictions are the average of the outputs of between 3 and 50 multi-layer perceptron neural networks each trained on 2/3 of the training data for a time determined by validation on the remaining 1/3. All the networks have identical architectures with a single hidden layer of hyperbolic tangent units. Early stopping is a technique for avoiding over-fitting in models that use iterative learning procedures. One uses a model that is expected to have too large capacity and through early stopping one ensures that the model does not over-fit. Consequently this approach should be helpful in overcoming the bias/variance dilemma of frequentist methods [Geman et al. 1992]. Since early stopping will help to avoid overfitting, we do not necessarily need an accurate estimate of the required model capacity, but can use a network with a large number of hidden units.

A fraction of the training cases are held out for validation, and performance on this set is monitored while the iterative learning procedure is applied. Typically, the validation error will initially decrease as the model fits the data better and better, but later on, when the model begins to over-fit, the validation error starts rising. The idea is to stop learning as soon as this minimum in validation error is achieved. For neural networks, one of the great advantages of early stopping is that it simplifies the otherwise difficult question of how long to train the model.

A number of details have to be specified to make the method practically applicable. This section contains a discussion of the issues involved and the decisions that I have made for how to implement the method. I should stress that I am not claiming my choices to be optimal in any sense, rather I have tried to define an automatic method which cannot in any obvious (and well documented) way be improved upon.

I use a multi-layer perceptron neural network with a single hidden layer of hyperbolic tangent units and a linear output unit. All units have biases. The network is fully connected (including direct connections from inputs to outputs). A network with I inputs and H hidden units implements the function

$$f(x) = \sum_{h=1}^{H} g_h(x) v_h + \sum_{i=1}^{I} x_i w_i + b_0 \qquad g_h(x) = \tanh\big(\sum_{i=1}^{I} x_i u_{ih} + a_h\big), \tag{3.12}$$

where v_h are the hidden-output weights, b_0 the output-bias, w_i the direct input-output weights, a_h the hidden-biases, and u_{ih} the input-hidden weights.

The number of hidden units is chosen to be the smallest such that the number of weights is at least as large as the total number of training cases (after removal of the validation cases). There is experimental evidence that the number of hidden units is not important as long as it is large enough [Tetko et al. 1995]. We need to decide how large a fraction of the training cases to use for validation. There are two obvious conflicting interests; we want a large validation set to achieve a good estimate of performance and we want a large training set to be able to train a complex model, namely as complex a model as could be supported by the entire training data set. For non-linear models trained with finite amounts of data, there seem to be no helpful theoretical results about this trade-off. I chose to use one third of the training cases (rounded down if necessary) for validation and the rest for training.

Another question is the exact criterion for when the training should stop, since we do not want to stop merely when small fluctuations in the validation error occur. A number of more or less complicated stopping criteria have been proposed in the literature, but none has convincingly been shown to outperform others. Here I use the simple scheme of training until the iteration with smallest validation error lies 33% backwards in the run. In this way I will have trained for 50% more epochs than "necessary" and can thus be fairly confident that I have found a good minimum. I will do a validation after every line-search in the training run (training with conjugate gradients). Since the validation set is only half the size of the training set and validation only requires a forward pass, validation will typically slow down training by about 20% (since typically the line search involved in each iteration requires on average about 1.3 forward and backward passes), which is not alarming. More compute efficient strategies could be employed by validating only at longer intervals, at the risk of finding worse stopping times. If a simpler method of steepest descent with a fixed step-size was used as an optimization technique, it would probably be sufficient to validate at longer intervals, but the conjugate gradient technique I use involves line-searches and can potentially take very large steps.

The idea of including the validation data in the training set and continuing training is appealing, since it is expected that better performance can be achieved with larger training sets. However, the problem of when to stop this additional training presents itself. One suggestion could be to continue training on the combined set until the same training error per case as was found by early stopping is achieved. From a theoretical point of view, this idea seems reasonable in an average case analysis, since it is expected that validation cases will have a slightly larger error initially than the training cases. However, there are practical problems; what if the validation error is already lower (per case) than the training error? And what if it is impossible to achieve the same training error using all the training data? Even if scenarios as bad as these do not occur frequently, it is still possible that the generalisation performance will often be decreased by using this procedure.

Instead of including all the training data and retraining, I will effectively use all the data by training an ensemble of networks, each of which is trained using a different validation set (chosen at random). This is a convenient way of combining the benefits of averaging over ensembles with the ability to use all cases for training. When using different validation sets for each net, one cannot choose between networks and only include the ones with best validation error, since the variation in validation error might be due to the differing validation sets. Consequently I will include all trained nets in the final ensemble. A more complicated hybrid scheme with a few networks trained for each validation set could be contemplated, but for simplicity this will not be pursued further.

Additionally, we must take care of the cases where early stopping seems to stop too early or too late. In the initial phases of training it is conceivable that the validation error fluctuates quite wildly and this may cause the stopping criteria to be fulfilled (since the extra 50% epochs may be very few and thus not be a reliable indicator of local minima). To rule out this scenario, I require that each training run uses at least 2% of the total available training time and that at least 250 iterations have been done (unless the minimization algorithm converges). This will limit the number of possible members of the final ensemble to 50, which is probably adequate for getting most of the benefit from averaging. On the other hand, it may be that the validation error keeps decreasing and the stopping criterion is never satisfied. Recall that the *expected* behaviour of the validation error is that it should increase when over-fitting sets in, but there is no guarantee that this will happen in any particular run. On the other hand, it may be that it simply takes a large number of epochs to train the network. To resolve this dilemma, I will terminate a training if it uses more than 33% of the total computational resources available, such that the final ensemble will contain at least 3 networks. Technically, some of the above criteria can be conflicting; in the actual implementation the conditions for deciding to stop the training of a particular net in order of precedence are: if the network has converged; OR if more than 33% of the allowed cpu time has been used on training this net; OR if more than 250 iterations of training has been completed AND the minimum validation error was achieved at least 33% backwards in the run AND we've used at least 2% of the total available cpu time on this net.

This method can now be run automatically without any further specifications, except that a limit on the allowed amount of cpu time has to be specified. As a default I have chosen to let the allowed cpu time scale linearly with the number of training cases. The default allows 1.875 seconds per training case; this means that a net with 1024 training cases will be allowed to train for 32 minutes.

3.7 Bayesian neural network using Monte Carlo: `mlp-mc-1`

A Bayesian implementation of learning in neural networks using Monte Carlo sampling has been developed by Neal [1996]. This computation intensive method has shown encouraging performance in [Neal 1996] and in a study using several datasets in [Rasmussen 1996].

For a full description of the method the reader is referred to [Neal 1996]. Here a brief description of the algorithm will be given, along with the heuristics employed. A feed forward multi-layer perceptron neural network with a single hidden layer of hyperbolic tangent units is used; the network is fully connected, including direct connections from the input to the output layer. The output units are linear. All units have biases. The network is identical to the one used by the `mlp-ese-1` method, eq. (3.12). All the network parameters are given prior distributions specified in terms of hyperparameters. Predictions are made using Monte Carlo samples of the posterior distribution of weights.

The network weights, $\mathbf{w}$, together with the hyperparameters are collectively termed θ. The posterior distribution of θ is given by Bayes' rule

$$p(\theta|\mathcal{D}) \propto p(\theta)p(t^{(1)},\ldots,t^{(n)}|x^{(1)},\ldots,x^{(n)},\theta), \tag{3.13}$$

where $p(\theta)$ is the prior and $p(t^{(1)},\ldots,t^{(n)}|x^{(1)},\ldots,x^{(n)},\theta)$ is the likelihood. We assume that the noise is independent Gaussian with variance σ^2. We don't know the magnitude of the noise, so we will attempt to infer it from the data. We put a Gamma prior on the inverse variance $\tau = \sigma^{-2}$, known as the *precision*. The gamma density is given by

$$p(\tau) \sim \text{Gamma}(\mu,\alpha) \propto \tau^{\alpha/2-1}\exp(-\tau\alpha/2\mu), \tag{3.14}$$

where μ is the mean and α is known as the shape parameter. We use $\mu = 400$ (corresponding to $\sigma = 0.05$) and $\alpha = 0.5$, which is intended as a vague prior, allowing noise variances down to 10^{-4} and up past unity. The density for the resulting prior is depicted in fig. 3.1.

The network weights are assigned to groups according to the identity of the units that they connect. There are 5 groups of weights: output-biases b_o, hidden-biases b_h, input-hidden weights w_h, hidden-output weights w_o and direct input-output weights w_d. The priors for the network weights are different for each group of weights. They are given in terms of hierarchically specified distributions in which higher level hyperparameters are shared between all the weights in the group, thereby introducing dependencies between weights.

The output-biases are given zero-mean Gaussian priors $b_o \sim \mathcal{N}(0,\sigma^2)$ with a standard

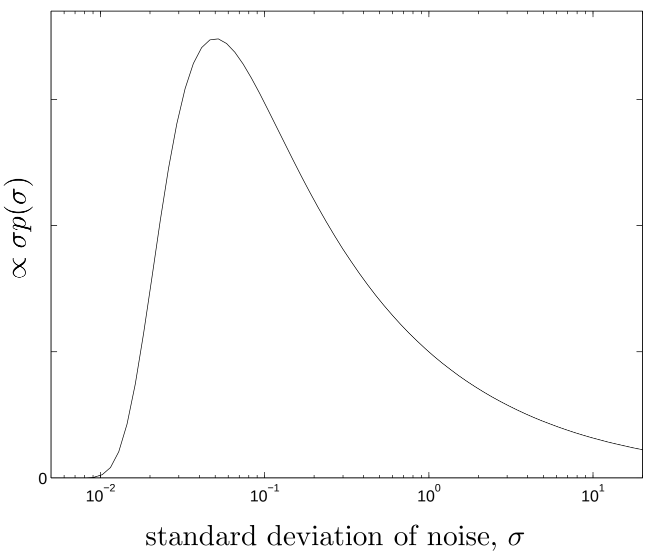


Figure 3.1: Non-normalised prior on the noise level σ. The function $\sigma p(\sigma)$ has been plotted against σ on a log scale to allow the usual interpretation of probabilities as being proportional to areas under the curve.

deviation of $\sigma = 1$. Since the targets in the training set will have been normalised by DELVE to have roughly zero mean and unit variance, this prior should accommodate typical values.

The group of hidden-unit biases is given a hierarchical prior consisting of two layers: $b_h \sim \mathcal{N}(0, \sigma^2)$; the distribution of σ is specified in terms of a precision $\tau = \sigma^{-2}$ and given a Gamma form as eq. (3.14) with parameters: $\mu = 100$ and $\alpha = 0.5$. I have not attempted to plot the resulting prior on the weights (obtained by integrating out τ) since the dependencies between weights through the common value of τ makes it difficult to give a faithful representation of the distribution.

The hidden-output weights are given a similar two layer prior, with parameters $\alpha = 0.5$ and $\mu = 100H^{-1}$, where H is the number of hidden units. This prior is scaled according to the number of hidden units with the variance for the weights inversely proportional to the number of hidden units. The scaling accomplishes an invariance of the prior on the magnitude of the output signal with respect to the number of hidden units. Such a decoupling of signal magnitude and model complexity is useful in setting priors and when considering networks with numbers of hidden units tending to infinity (see section 4.1).

The input-to-hidden weights are given a three layer prior: again each weight is given a zero-mean Gaussian prior $w \sim \mathcal{N}(0, \sigma^2)$; the corresponding precision for the weights out of input unit i is given a Gamma prior with a mean μ and a shape parameter $\alpha_1 = 1$: $\tau_i \sim \text{Gamma}(\mu, \alpha_1)$. The mean μ is determined on the top level by a Gamma distribution with mean $\mu_0 = 25I^2$, where I is the number of inputs, and shape parameter $\alpha_0 = 0.5 :$ $\mu_i \sim \text{Gamma}(25I^2, \alpha_0)$. The prior variance for the weights scales proportional to I^{-2}. This is done in accordance with the (subjective) expectation, that the more inputs there are, the more unlikely it is for any single one of them to be very important for the predictions. Under the scaled prior, a constant number of inputs will be "important" (i.e., have a large value of σ_i) as the number of inputs increase. The direct input-to-output connections are also given this prior.

The above-mentioned three layer prior incorporates the idea of Automatic Relevance Determination (ARD), due to MacKay and Neal, and discussed in [Neal 1996]. The hyperparameters, τ_i, associated with individual inputs can adapt according to the relevance of the input; for an unimportant input, τ_i can grow very large (governed by the top level prior), thus forcing σ_i and the associated weights to vanish.

Given the likelihood and the prior we can compute the posterior distribution for the network weights. In order to make predictions, we integrate over the posterior. The predictive distribution for the target $t^{(n+1)}$ corresponding to a novel input $x^{(n+1)}$ is given by

$$p\big(t^{(n+1)}|x^{(n+1)}, \mathcal{D}\big) = \int p\big(t^{(n+1)}|x^{(n+1)}, \theta\big)p(\theta|\mathcal{D})d\theta \simeq \frac{1}{T}\sum_{t=1}^{T} p\big(t^{(n+1)}|x^{(n+1)}, \theta^{(t)}\big), \quad (3.15)$$

where $\theta^{(t)}$ are samples drawn from the posterior distribution. For neural network models this integral cannot be handled analytically. Instead, we employ the *Hybrid Monte Carlo* algorithm [Duane et al. 1987] to obtain samples from the posterior with which we can approximate the predictive distribution. This method combines the Metropolis algorithm with dynamical simulation which helps to avoid the random walk behavior of simple forms of Metropolis; this is essential if we wish to explore weight space efficiently. The hyperparameters are updated using Gibbs sampling.

Sampling from the posterior weight distribution is performed by iteratively updating the values of the network weights and hyperparameters. Each iteration involves two components: weight updates and hyperparameter updates. A cursory description of these steps follows.

Weight updates are done using the Hybrid Monte Carlo algorithm. This algorithm is also used for the gp-mc-1 method in section 4.6, where a more extensive explanation is given. A fictitious dynamical system is generated by interpreting weights as positions, and augmenting the weights $\mathbf{w}$ with momentum variables $\mathbf{p}$. The purpose of the dynamical system is to give the weights "inertia" so that random walk behaviour can be avoided during exploration of weight space. The total energy, $\mathcal{H}$, of the system is the sum of the kinetic energy, $\mathcal{K}$, (a function of the momenta) and the potential energy, $\mathcal{E}$. The potential energy is defined such that $p(\mathbf{w}) \propto \exp(-\mathcal{E})$. We sample from the joint distribution for $\mathbf{w}$ and $\mathbf{p}$ given by $p(\mathbf{w}, \mathbf{p}) \propto \exp(-\mathcal{E} - \mathcal{K})$, under which the marginal distribution for $\mathbf{w}$ is given by the posterior. A sample of weights from the posterior can therefore be obtained by simply ignoring the momenta.

Sampling from the joint distribution is achieved by two steps: 1) finding new points in phase space with near-identical energies $\mathcal{H}$ by simulating the dynamical system using a discretised approximation to Hamiltonian dynamics, and 2) changing the energy $\mathcal{H}$ by doing Gibbs sampling for the momentum variables.

Hamilton's first order differential equations for $\mathcal{H}$ are approximated by a series of discrete first order steps (specifically by the *leapfrog* method). The first derivatives of the network error function enter through the derivative of the potential energy, and are computed using back-propagation. In the original version of the hybrid Monte Carlo method, the final position is then accepted or rejected depending on the final energy $\mathcal{H}^*$ (which is not necessarily equal to the initial energy $\mathcal{H}$ because of the discretisation). Here we use a modified version that uses an average (in the probability domain) over a window of states instead [Neal 1994]. The step size of the discrete dynamics should be as large as possible while keeping the rejection rate low. The step sizes are set individually using several heuristic approximations, and scaled by an overall parameter ε. We use $L = 100$ iterations, a window size of 10 and a step size of $\varepsilon = 0.15$ for all simulations.

The momentum variables are updated using a modified version of Gibbs sampling, allowing the energy $\mathcal{H}$ to change. A "persistence" of 0.95 is used; the new value of the momentum is a weighted sum of the previous value (weight 0.95) and the value obtained by Gibbs sampling (weight $(1-0.95^2)^{1/2}$) [Horowitz 1991]. With this form of persistence, the momenta change approximately 20 times more slowly, thus increasing the "inertia" of the weights, so as to further help in avoiding random walks. Larger values of the persistence will further increase the weight inertia, but reduce the rate of exploration of $\mathcal{H}$. The advantage of increasing the weight inertia in this way rather than by increasing L is that the hyperparameters are updated at shorter intervals, allowing them to adapt to the rapidly changing weights.

The hyperparameters are updated using Gibbs sampling. The conditional distributions for the hyperparameters given the weights are of the Gamma form, for which efficient generators exist, except for the top-level hyperparameter in the case of the 3 layer priors used for the weights from the inputs; in this case the conditional distribution is more complicated and the method of Adaptive Rejection Sampling [Gilks and Wild 1992] is employed.

The network training consists of two levels of initialisation before sampling for network weights which are used for prediction. At the first level of initialisation the hyperparameters (standard deviations for the Gaussians) are kept constant at 0.5 for hyperparameters controlling output weights and to 0.1 for all other hyperparameters, while the weights themselves are set to zero. Then the weights are allowed to grow during 1000 leapfrog iterations (with the hyperparameters remaining fixed). Neglecting this phase can cause the network to get caught for a long time in a state where weights and hyperparameters are both very small.

The Markov chain described above is then invoked and run for as long as desired, eventually producing networks from the posterior distribution. The initial 1/3 of these nets are discarded, since the algorithm may need time to reach regions of high posterior probability. Networks sampled during the remainder of the run are saved for making predictions according to eq. (3.15). I use 100 samples from the posterior to approximate the integral. Probably the predictions would not get vastly different if more samples were used in the approximation and this has been avoided because the disk requirements for storing these network samples become prohibitive.

Since the output unit is linear, the final prediction can be seen as coming from a huge (fully connected) ensemble net with appropriately scaled output weights. The size of the individual nets is given by the rule that we want at least as many network parameters as we have training cases (with a lower limit of 6 hidden units and an upper limit of 25). We hope thereby to be well out of the under-fitting region. Using even larger nets would probably not gain us much (in the face of the limited training data) and is avoided for computational reasons.

All runs used the parameter values given above. The only check that is necessary is that the rejection rate stays low, say below 5%; if not, the step size should be lowered. In all runs reported here, $\varepsilon = 0.15$ was adequate. The parameters concerning the Monte Carlo method and the network priors were all selected based on intuition and on experience with toy problems. Thus no parameters need to be set by the user, save the specification of the allowable cpu time.

Chapter 4

Regression with Gaussian Processes

This chapter presents a new method for regression which was inspired by Neal's work [Neal 1996] on priors for infinite networks and pursued in [Williams 1996; Williams and Rasmussen 1996]. Gaussian Process (GP) models are equivalent to a Bayesian treatment of a certain class of multi-layer perceptron networks in the limit of infinitely large networks. In the Gaussian process model, these large numbers of network weights are not represented explicitly and the difficult task of setting priors on network weights is replaced by a simpler task of setting priors for the GP's.

Close relatives to the GP model presented here have appeared in various guises in the literature. An essentially similar approach to the present was taken in [O'Hagan 1978; O'Hagan 1994], but surprisingly it has not spurred much general interest. This thesis extends the work of O'Hagan by adapting model parameters using their associated likelihood. Gaussian Process models are being used for analysis of computer experiements [Koehler and Owen 1996], presumably because they are applicable to modelling noise-free data. This seems to be an undue restriction however, and it is shown in the following chapters that GP models are also very attractive for modelling noisy data.

The approach of variational analysis to regularised function approximation has been taken in [Poggio and Girosi 1990; Girosi et al. 1995] and the related spline models have been studied by Wahba [1990]. In these approaches, cross validation or generalised cross validation (GCV) are used to estimate regularisation parameters. These approaches may not be viable with large numbers of regularisation constants, as required for Automatic Relevance Determination (ARD) based regularisation schemes. The GP models also have a close correspondence with variable metric kernel methods [Lowe 1995].

In the present exposition, the two versions of GP models are formulated in terms of a probabilistic model in a Bayesian setting. In the simplest approach, the parameters controlling the model are optimized in a maximum aposteriori (MAP) approach. In the second version, the Bayesian formalism is taken further, to allow integration over parameters.

4.1 Neighbors, large neural nets and covariance functions

The methods relying on Gaussian processes described in this chapter differ in several respects from other methods commonly used for data modeling and consequently it may not be easy to get an intuitive feel for how the model works. In order to help readers who are familiar with more conventional methods, I will start the discussion of the GP model via an analogy to kernel smoother models. This should help to make the subsequent formal account more digestible. Next, a simple calculation will show how neural networks with Gaussian priors on their weights imply Gaussian process priors over functions.

A kernel smoother consists of a kernel function and a local model. The kernel function defines a neighborhood by giving weights to training examples and the local model is fit to these weighted examples for predictions. The Gaussian and tri-cubic functions are common choices of kernel functions. The kernel function is a function of the model inputs and returns a *weighting* for each training example to be used when fitting the local model. Often kernel functions which depend only on (Euclidean) distance are used, e.g., the Gaussian kernel $K \propto \exp(-d^2/2\sigma^2)$, where d is the distance between the kernel center and the input of a training case. The properties of the smoother are controlled by the width of the kernel, often expressed in terms of numbers of neighbors and selected by cross-validation. More advanced methods with more flexible kernels, sometimes referred to as *variable metric* models, have also been explored [Lowe 1995]. Often computational arguments are used to favour kernels which put a non-zero weight on only a small fraction of the training cases. When predicting the output for a novel test input, the kernel is centered on the test input and weightings for the training cases are determined. A common choice of local model is the linear model fit by (weighted) least squares [Cleveland 1979].

In the GP methods, the role of the kernel function and local model are both integrated in the covariance function. Like the kernel function, the covariance function is a function of the model inputs, but rather than returning a weighting for a training case given a test input, it returns the covariance between the outputs corresponding to two inputs. As with the kernel method, the Gaussian, $C(x, x') \propto \exp(-d^2/2\sigma^2)$ where d is the Euclidean distance between x and x', would be a reasonable choice for a covariance function (note however, that the the name "Gaussian process" does not refer to the form of the covariance function). Assuming that the mean output is zero, the covariance between the outputs corresponding to inputs x and x' is defined as $E[y(x)y(x')]$. Thus, inputs that are judged to be close by the covariance function will have outputs that are highly correlated and thus are likely to be quite similar. In the Gaussian process model we assume that the outputs of any finite set of cases have a joint multivariate Gaussian distribution with covariances given by the covariance function. Predictions will be made by considering the covariances between the test case and all the training cases, which will enable us to compute the most likely output for the test case. In fact we can obtain the entire predictive distribution for the test output, which given the assumptions is also Gaussian.

The idea of using Gaussian processes directly was inspired by investigations by Neal [1996] into priors over weights for neural networks. Consider a neural network with I inputs, a single output unit and a single layer of H tanh hidden units. The hidden and output units have biases and the network is fully connected between consecutive layers

$$f(x) = \sum_{h=1}^{H} g_h(x) v_h + b_0 \qquad\qquad g_h(x) = \tanh\big(\sum_{i=1}^{I} x_i u_{ih} + a_h\big). \tag{4.1}$$

The weights are all given zero mean Gaussian priors; the standard deviations for the input-to-hidden weights, u_{ih}, and hidden-biases, a_h, are σ_u and σ_a respectively, for the hidden-to-output weights, v_h, and the output-bias, b, the standard deviations are σ_v and σ_b. This sort of prior was suggested by MacKay [1992b].

Consider the distribution of a network output under the prior distribution of weights, given a specific input $x^{(i)}$. The contribution of each hidden unit has a mean of zero: $E[v_h g_h(x^{(i)})] = E[v_h]E[g_h(x^{(i)})] = 0$, since v_h and $g_h(x^{(i)})$ are independent. The variance of the contribution from each hidden unit is finite $E[(v_h g_h(x^{(i)}))^2] = \sigma_v^2 E[(h_g(x^{(i)}))^2]$, since $h_g(x^{(i)})$ is bounded. Setting $V(x^{(i)}) = E[(h_g(x^{(i)}))^2]$, we can conclude by the Central Limit Theorem that as the number of hidden units H tends to infinity, the prior distribution of $f(x^{(i)})$ converges to a zero mean Gaussian with variance $\sigma_b^2 + H\sigma_v^2 V(x^{(i)})$. By selecting a value for σ_v which scales inversely with H we obtain a well defined prior in the limit of infinite numbers of hidden units.

Following a similar argument, the joint distribution for several inputs converges in the limit of infinite H to a multivariate Gaussian with means of zero and covariance of

$$\begin{aligned} E[f(x^{(i)})f(x^{(j)})] &= E\Big[\big(\sum_{h=1}^{H} g_h(x^{(i)})v_h + b_0\big)\big(\sum_{h=1}^{H} g_h(x^{(j)})v_h + b_0\big)\Big] \\ &= \sigma_b^2 + H\sigma_v^2 E[g_h(x^{(i)})g_h(x^{(j)})], \end{aligned} \tag{4.2}$$

which defines a Gaussian process; the crucial property being that the joint distribution of any finite set of function values is Gaussian. I will not attempt to further characterize the covariance functions implied by weight priors for neural networks, see [Neal 1996].

The preceding paragraphs are meant to motivate investigation of models relying on Gaussian processes. The correspondences to these other models will not play a crucial role in the following — rather, it will be shown how to model data purely in terms of Gaussian processes. This will require that the reader becomes accustomed to thinking about distributions over functions in terms of their covariances; this will hopefully be aided by illustrations of functions drawn from various Gaussian process priors.

4.2 Predicting with a Gaussian Process

Formally, a Gaussian process is a collection of random variables $\{Y_x\}$ indexed by a set $x \in X$, where any finite subset of Y's has a joint multivariate Gaussian distribution. A typical application of Gaussian processes is in the field of signal analysis, where the random variables are indexed by time. In contrast, in our case we index the random variables by X, the input space of dimensionality m. A Gaussian process is fully specified by its mean and its covariance function, $C(x, x')$. In the following I will consider only GP's with a mean of zero. The random variable Y_x will model the output of the GP model when the input is x. The covariance is a function of the inputs $C(x, x') = E\big[\big(Y_x - \mu(x)\big)\big(Y_{x'} - \mu(x')\big)\big]$, where $\mu \equiv 0$. Here, $C(x, x')$ is called the *covariance function* and the matrix of covariances between pairs of examples is referred to as the *covariance matrix*. In section 4.3 it will be shown how to parameterise the covariance function; for now, we consider the form of $C(x, x')$ as given. For presentational convenience, in the following I will index training and test cases by superscripts in brackets, such that $y^{(i)}$ is the output associated with input $x^{(i)}$.

Our goal is, as usual, to compute the distribution $p(y^{(n+1)}|\mathcal{D}, x^{(n+1)})$ of scalar output $y^{(n+1)}$ given a test input $x^{(n+1)}$ and a set of n training points $\mathcal{D} = \{(x^{(i)}, t^{(i)})|i = 1 \ldots n\}$. Note that we distinguish between the model outputs $y^{(i)}$, and the training set targets $t^{(i)}$; they are both random variables but differ in that the targets are noise corrupted versions of the outputs. We introduce $n+1$ stochastic variables $Y^{(1)}, \dots, Y^{(n)}, Y^{(n+1)}$, modeling the function at the corresponding inputs $x^{(1)}, \dots, x^{(n+1)}$. Note that this formulation is different from the common frequentist model where a *single* random variable is used to model the conditional distribution $p(t|x)$ of the targets given the inputs. An example will help to clarify the difference: consider the linear model with coefficients given by the vector β; the linear model and noise contributions are

$$y = \beta^{\mathrm{T}} x, \qquad t = y + \varepsilon, \tag{4.3}$$

where ε is the noise. In the frequentist framework the use of a single random variable t is possible because the model output y is a *deterministic* function of the inputs x, once the parameters β have been estimated (using for example maximum likelihood). In a Bayesian approach, on the contrary, the model parameters are treated as random variables and the model outputs are obtained through integration over model parameters, introducing stochasticity and dependencies between outputs, thus requiring separate random variables. In fact, in the GP model the *only* quantities being modeled are the covariances between outputs.

We proceed by assigning a multivariate Gaussian prior distribution to these variables

$$p\big(y^{(1)}, \dots, y^{(n)}, y^{(n+1)} \big| x^{(1)}, \dots, x^{(n+1)}\big) \propto \exp -\frac{1}{2}\mathbf{y}^{\mathrm{T}}\Sigma^{-1}\mathbf{y}, \quad \text{where} \quad \Sigma = C\big(x^{(p)}, x^{(q)}\big). \tag{4.4}$$

Note that this prior specifies the joint distribution of the function values (not the noisy targets) given the inputs. It is a *prior* in so far as the targets for the training cases have not yet been considered. A reasonable prior would specify that we expect a larger covariance between function values corresponding to nearby inputs than for function values corresponding to inputs that are further apart. Covariance functions are discussed in the next section.

It will often be convenient to partition the prior covariance matrix in the following manner

$$\Sigma = \begin{bmatrix} K & \mathbf{a} \\ \mathbf{a}^{\mathrm{T}} & b \end{bmatrix}, \tag{4.5}$$

where K contains the covariances between pairs of training cases and $\mathbf{a}$ is a vector of covariances between the test case and the training cases and b is the prior covariance between the test case and itself.

The likelihood relates the underlying function which is modeled by the y variables to the observed noisy targets $t^{(i)}$, $i = 1 \dots n$. The noise is assumed to be independent Gaussian with some (unknown) variance r^2

$$p\big(t^{(1)}, \dots, t^{(n)} \big| y^{(1)}, \dots, y^{(n)}, y^{(n+1)}\big) \propto \exp -\frac{1}{2}(\mathbf{y}-\mathbf{t})^{\mathrm{T}}\Omega^{-1}(\mathbf{y}-\mathbf{t}), \qquad \text{where} \quad \Omega^{-1} = \begin{bmatrix} r^{-2}I & \mathbf{0} \\ \mathbf{0}^{\mathrm{T}} & 0 \end{bmatrix}. \tag{4.6}$$

Here I is an $n \times n$ identity matrix. To gain some notational convenience in the following, I have written this joint distribution conditioning also on $y^{(n+1)}$, although the distribution does not depend on this variable. Formally the vector $\mathbf{t}$ in this equation has $n{+}1$ elements, whereas there are only n observed targets — the vector is simply augmented by an element (its value is inconsequential).

Using Bayes' rule, we combine the prior and the likelihood to obtain the posterior distribution

$$\begin{aligned} &p\big(y^{(1)}, \dots, y^{(n+1)} \big| \mathcal{D}, x^{(n+1)}\big) \\ &\quad \propto p\big(y^{(1)}, \dots, y^{(n+1)} \big| x^{(1)}, \dots, x^{(n+1)}\big) p\big(t^{(1)}, \dots, t^{(n)} \big| y^{(1)}, \dots, y^{(n+1)}\big) \\ &\quad = p\big(t^{(1)}, \dots, t^{(n)}, y^{(1)}, \dots, y^{(n+1)} \big| x^{(1)}, \dots, x^{(n+1)}\big) \end{aligned} \tag{4.7}$$

The constant of proportionality in this equation is $p(\mathcal{D})$, a normalisation constant which is independent of the $y^{(i)}$ variables. Note, that in general it is not necessary to compute the normalisation terms in any of the previous equations. The posterior is

$$\begin{aligned} p\big(y^{(1)}, \dots, y^{(n)}, y^{(n+1)} \big| \mathcal{D}, x^{(n+1)}\big) &\propto \exp\Big(-\frac{1}{2}\mathbf{y}^{\mathrm{T}}\Sigma^{-1}\mathbf{y} - \frac{1}{2}(\mathbf{y}-\mathbf{t})^{\mathrm{T}}\Omega^{-1}(\mathbf{y}-\mathbf{t})\Big) \\ &= \exp\Big(-\frac{1}{2}(\mathbf{y}-\mathbf{y}_m)^{\mathrm{T}}\big[\Sigma^{-1}+\Omega^{-1}\big](\mathbf{y}-\mathbf{y}_m)\Big), \end{aligned} \tag{4.8}$$

where $\mathbf{y}_m$ is a vector of posterior means. The posterior distribution is again Gaussian with an inverse covariance given by the sum of the inverse covariances of the prior and the likelihood. The posterior mean is where the product takes on its maximum value, and can consequently be found by differentiation

$$\frac{\partial}{\partial y^{(i)}} \log p\big(y^{(1)}, \dots, y^{(n)}, y^{(n+1)} \big| \mathcal{D}, x^{(n+1)}\big) = \mathbf{0} \implies \mathbf{y}_m = \big[\Sigma^{-1}+\Omega^{-1}\big]^{-1}\Omega^{-1}\mathbf{t}. \tag{4.9}$$

However, we are mostly interested in the distribution for $y^{(n+1)}$ and do not necessarily wish to attempt to evaluate this expression. Notice that the posterior means $\mathbf{y}_m$ do not necessarily coincide with the corresponding target values. To compute the distribution for $y^{(n+1)}$ we need to marginalise over $y^{(1)}, \dots, y^{(n)}$ from eq. (4.8) which yields a Gaussian distribution with mean and variance

$$p\big(y^{(n+1)} \big| \mathcal{D}, x^{(n+1)}\big) \sim \mathcal{N}(\mu_{y^{(n+1)}}, \sigma^2_{y^{(n+1)}}), \qquad \begin{aligned} \mu_{y^{(n+1)}} &= \mathbf{a}^{\mathrm{T}}Q^{-1}\mathbf{t} \\ \sigma^2_{y^{(n+1)}} &= b - \mathbf{a}^{\mathrm{T}}Q^{-1}\mathbf{a} \end{aligned}$$

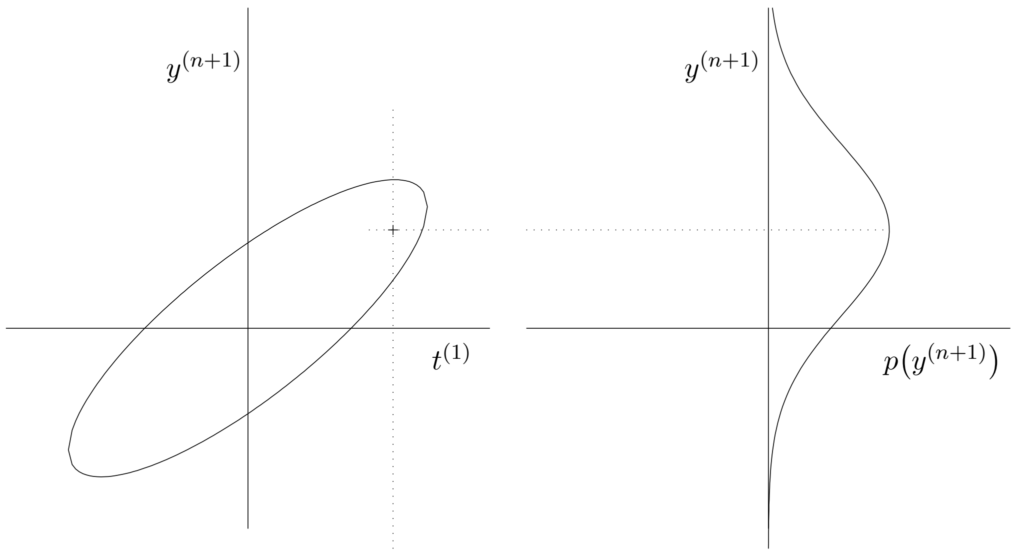


Figure 4.1: The left panel shows the one standard deviation contour of the joint Gaussian distribution of a single training case $t^{(1)}$ and the test point $y^{(n+1)}$ in the situation where we have a single training point, $n = 1$. When the target value for the training point is observed, we condition on this value indicated by the dotted vertical line. This gives rise the the predictive distribution on the right. The vertical axes of the two plots are to scale.

where $Q = K + r^2 I$. This completes the analysis of the model, since we now have access to the desired predictive distribution. Depending on the loss function of a particular application, we can make optimal predictions by making point predictions that minimize the expected loss. Notice, that in order to use this result we must invert the Q matrix which has size $n \times n$.

A slightly different view of the same formalism as above, may lead to a better intuitive understanding of what is going on. Starting from the middle line of eq. (4.7) we can directly marginalise over the $y^{(i)}$, $i = 1 \dots n$, which we are not interested in and obtain

$$\begin{aligned}\int \cdots \int p\big(t^{(1)}, \dots, t^{(n)}, y^{(1)}, \dots, y^{(n+1)} \big| x^{(1)}, \dots, x^{(n+1)}\big) dy^{(1)} \cdots dy^{(n)} = \\ p\big(t^{(1)}, \dots, t^{(n)}, y^{(n+1)} \big| x^{(1)}, \dots, x^{(n+1)}\big) \propto \exp -\frac{1}{2}\mathbf{t}_y^{\mathrm{T}} (\Sigma^{-1} + \Omega^{-1}) \mathbf{t}_y,\end{aligned} \tag{4.11}$$

where $\mathbf{t}_y$ indicates the vector of targets augmented by $y^{(n+1)}$. We can then condition on the observed values of the targets in the training set, and using the standard rule for conditioning Gaussians we recover the result from eq. (4.10). This conditioning is illustrated in fig. 4.1.

4.3 Parameterising the covariance function

In the previous section we have seen how to derive the predictive distribution for test cases when the covariance function was given. This section discusses various choices for covariance functions.

There are many possible choices of prior covariance functions. Formally, we are required to specify a function which will generate a non-negative definite covariance matrix for any set of input points. From a modeling point of view, we wish to specify prior covariances which contain our prior beliefs about the structure of the function we are modeling.

As is often the case in Bayesian modeling, it turns out that it is convenient to specify priors in terms of *hyperparameters*, whose values (or distributions) are not specified a priori, but will be adapted as the model is fit to the training data. An example will clarify this idea; the covariance function used extensively in this thesis sets the covariance between the points $x^{(p)}$ and $x^{(q)}$, $p, q = 1 \dots n$ to be

$$C\big(x^{(p)}, x^{(q)}\big) = a_0 + a_1 \sum_{i=1}^{m} x_i^{(p)} x_i^{(q)} + v_0 \exp\Big(-\frac{1}{2}\sum_{i=1}^{m} w_i \big(x_i^{(p)} - x_i^{(q)}\big)^2\Big), \tag{4.12}$$

where $\theta = \log(v_0, r^2, w_1, \dots, w_m, a_0, a_1)$ is a vector of parameters playing the role of hyperparameters (the log is applied elementwise). Note that the noise level r^2 is included in the hyperparameters although it doesn't appear in the covariance function, since it will be treated in the same way as the hyperparameters in the following analysis.

The covariance is conceptually made up of two components. The first part involving the a_0 and a_1 parameters controls the scale of the bias and linear contributions to the covariance. To understand the form of these contributions consider a linear function

$$y(x) = \alpha_0 + \sum_{i=1}^{m} \alpha_i (x_i - \bar{x}_i), \tag{4.13}$$

where the coefficients α_i, $i = 0, \dots, m$ are considered random variables and $\bar{x}_i$ is the mean of x_i. Giving the α variables independent zero-mean distributions with finite variance we can compute the covariances

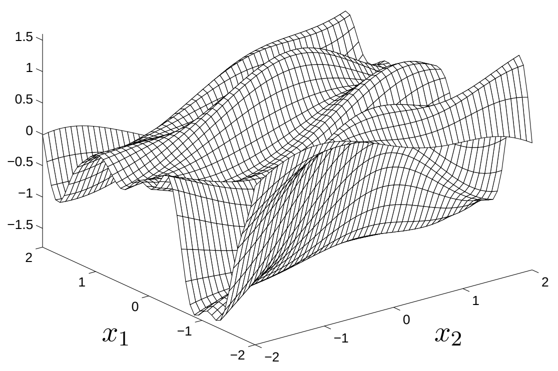
$$\begin{aligned} C(x, x') = E\big[y(x)y(x')\big] &= E\Big[\alpha_0^2 + \sum_{i=1}^{m} \alpha_i^2 (x_i - \bar{x}_i)(x'_i - \bar{x}_i)\Big] \\ &= a_0 + a_1 \sum_{i=1}^{m} (x_i - \bar{x}_i)(x'_i - \bar{x}_i), \end{aligned} \tag{4.14}$$

where a_0 is the variance of α_0 and we have assumed that all the remaining coefficients have a common variance a_1. The pre-processing of the data used for tests normalises the inputs to have a mean (or median) of zero, so we can (approximately) drop the $\bar{x}_i$ term and recover the form for linear contribution to the covariance in eq. (4.12). One may worry that the use of a common variance a_1 for all inputs may be dimensionally inconsistent if inputs are measured in different units. Naturally, we may group the variables of common variance differently; in particular, one choice would be to allow a separate hyperparameter for each input dimension.

The contributions from the linear terms in the covariance functions may become large for inputs which are quite distant from the bulk of the training examples. This is probably a useful mechanism for extrapolating to data points which are at the boundary of the training examples; without this term the covariance from the local interactions alone would become close to zero, and the prediction at large distances would be close to zero.

The last part of the covariance function in eq. (4.12) expresses the idea that cases with nearby inputs should have highly correlated outputs. The w_i parameters are multiplied by the coordinate-wise distances in input space and thus allow for different distance measures for each input dimension. For irrelevant inputs, the corresponding w_i should be small in order for the model to ignore these inputs. The "characteristic lengths" for input directions are given by $w_i^{-1/2}$. When a w_i parameter gets large, the resulting function will have a short characteristic length and the function will vary rapidly along the corresponding axis, indicating that this input is of high "importance". This idea is closely related the Automatic Relevance Determination (ARD) idea of MacKay and Neal. The associated v_0 variable gives the overall scale of the local correlations. In the following discussion, this term will be referred to as the ARD-term.

The covariance in eq. (4.12) will be used extensively for the experiments in this thesis. Many other choices may also be reasonable. Some other choices will be discussed briefly at the end of this chapter. As an example, two functions drawn at random from the ARD-part of the prior covariance function are plotted in fig. 4.2. Each of the two 50×50 mesh plots represent a single sample drawn at random from the 2500 dimensional joint Gaussian distribution. These random samples were obtained by multiplying a vector of zero-mean unit-variance Gaussian random numbers by the Cholesky factor of the covariance matrix, requiring 48Mb of memory and 8 minutes of cpu time.



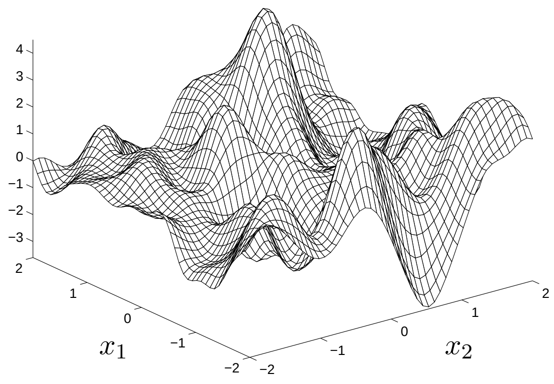


Figure 4.2: Functions drawn at random from ARD prior covariance functions. There are two input dimensions x_1 and x_2. Only the ARD-part of the covariance in eq. (4.12) was considered. In the left plot, the ARD hyperparameters were $w_1 = 10$ and $w_2 = 0.5$, corresponding to characteristic lengths of 0.32 and 1.41 respectively. In the plot on the right, both ARD hyperparameters were set to 10. As expected the functions generated from this prior are smooth. In both plots $v_0 = 1$.

4.4 Adapting the covariance function

So far, we have only considered the properties of the model for fixed values of the hyperparameters. In the following I will discuss how to adapt the hyperparameters in the light of training data. The log likelihood of the hyperparameters at this higher level is given by

$$\begin{aligned}\log p(\mathcal{D}|\theta) &= \log p\big(t^{(1)}, \ldots, t^{(n)} | x^{(1)}, \ldots, x^{(n)}, \theta\big) \\ &= -\frac{1}{2}\log\det Q - \frac{1}{2}\mathbf{t}^{\mathrm{T}} Q^{-1}\mathbf{t} - \frac{n}{2}\log 2\pi.\end{aligned} \tag{4.15}$$

It is possible to express analytically the partial derivatives of the log likelihood, which can form the basis of an efficient learning scheme. These derivatives are

$$\frac{\partial}{\partial\theta_i}\log p\big(t^{(1)}, \ldots, t^{(n)} | x^{(1)}, \ldots, x^{(n)}, \theta\big) = -\frac{1}{2}\,\mathrm{trace}\left(Q^{-1}\frac{\partial Q}{\partial\theta_i}\right) + \frac{1}{2}\mathbf{t}^{\mathrm{T}} Q^{-1}\frac{\partial Q}{\partial\theta_i}Q^{-1}\mathbf{t}. \tag{4.16}$$

In order to compute the parameters of the predictive distribution in eq. (4.10) it was necessary to invert the matrix Q; we also need it for the partial derivatives of the likelihood. Since the covariance matrix is guaranteed to be positive definite we can invert it using Cholesky decomposition [Golub and van Loan 1989] which requires $\frac{5}{6}n^3$ multiply and accumulate operations. Cholesky decomposition also produces the determinant of the matrix which is needed to evaluate the likelihood. The remaining computations involved in evaluating the likelihood and its partial derivatives (vector by matrix multiply and evaluation of the trace of a product) are of order $O(n^2)$. Thus, the main computational obstacle is computation of Q^{-1} which is certainly feasible on workstation computers with n of several hundred. For larger values of n the computations can be demanding; as an example, the inversion for $n = 1024$ takes about 5 minutes of CPU time on our 200MHz R4400/R4010 processor.

Several learning schemes are possible given the availability of partial derivatives of the likelihood. Maximum likelihood can be implemented by employing gradient descent. One may also specify a prior on the hyperparameters, a *hyperprior* and compute the posterior for the hyperparameters. Two possible implementations are covered in the following sections: a conjugate gradient approach to maximum aposteriori (MAP) estimation and a Monte Carlo method for integrating over hyperparameters. First I will discuss the hyperprior.

The same priors are used for the MAP method and the Monte Carlo approach. The prior assumes that the training data has been normalised to roughly zero mean and unit variance. The priors on the log of a_0, a_1 and r^2 are all Gaussian with mean -3 and standard deviation 3, corresponding to fairly vague priors. The prior on log v_0 is Gaussian with mean -1 and standard deviation 1. This prior seems reasonable, since v_0 gives the variance of the signal predicted by the ARD-term of the covariance function; since the targets are assumed to be normalized to roughly unit variance, we expect the hyperparameter (which is in the log domain) to be in the vicinity of 0 for tasks with low noise and lower for tasks with very high noise.

The priors for the ARD hyperparameters are more complicated. I wish to introduce a scaling of this prior with the number of inputs m. If m is small e.g. 8, then it seems reasonable that say 4 of these parameters are highly relevant; however if m is larger e.g. 32, then probably somewhat less than 16 inputs are highly relevant. Consequently, we expect the prior on the importance of the ARD hyperparameters to be lower with increasing numbers of inputs. Following the derivations in [Neal 1996] we employ a gamma prior whose mean scales with the number of inputs m; the inverse hyperparameters are given gamma priors

$$p(w^{-1}) = \frac{(\alpha/2\mu)^{\alpha/2}}{\Gamma(\alpha/2)}(w^{-1})^{\alpha/2-1}\exp(-w^{-1}\alpha/2\mu), \qquad \text{with} \quad \mu = \mu_0 m^{2/\alpha}. \tag{4.17}$$

Here μ^{-1} is the mean of w^{-1}. I chose $\alpha = 1$ and $\mu_0 = 1$ from looking a plots of the distributions, see fig 4.3. It may be difficult to decide how many inputs should be thought relevant, so one might attempt to make μ_0 a top-level hyperparameter, and set some vague prior on this. This has not been attempted in the current implementation. The resulting prior for the θ parameters, which are given by $\theta = \log(w)$ is

$$p(\theta) = m^{-1}(2\pi)^{-1/2}\exp\Big(-\frac{\theta}{2} - \frac{\exp(-\theta)}{2m^2}\Big). \tag{4.18}$$

Naturally, this prior may not be appropriate for all tasks — indeed there may be some tasks where a large number of inputs are all relevant. However, the prior is still vague enough

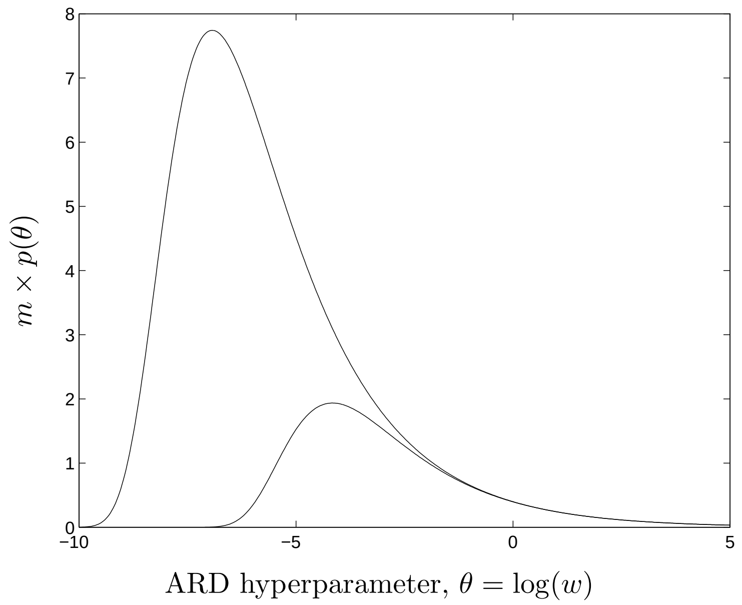


Figure 4.3: The scaling gamma prior used for the ARD hyperparameters. Here $m \times p(\theta)$ is plotted against $\theta = \log(w)$ for $m = 8$ and $m = 32$ from eq. (4.18). Note, that the expected number of hyperparameters corresponding to important inputs is the same in the two cases; for larger values of m the mass of the prior moves toward lower values indicating that we expect a larger proportion of the inputs to be less relevant.

that many hyperparameters may grow to large values if the likelihood strongly suggests this. Another benefit of putting pressure on the hyperparameters to remain small is that this may help the learning procedures in locating areas of high posterior probability. Consider the situation early in learning before the hyperparameters have had time to adapt; the hyperparameters will be more or less random and if any irrelevant ones take on large values then the exponential of eq. (4.12) may take on very small values. This will tend to attenuate the partial derivatives of the likelihood which in turn may make learning slow. Another way of saying this is that the initial search for regions of high posterior probability may be slow if the prior and the likelihood are very different.

4.5 Maximum aposteriori estimates

The maximum aposteriori (MAP) estimates are found using a conjugate gradient optimization technique (discussed in detail in Appendix B) to locate a (local) maximum of the posterior. This approach has the advantage that a reasonable approximation to a (local) maximum can be found with relatively few function and gradient evaluations. Hence, this would be the preferred approach when there is a large number of training cases, when integration via Monte Carlo is computationally infeasible. When there is a large number of training cases we would generally expect the posterior to be fairly narrow, such that the predictions for a MAP method may not differ much from the results of integrating over hyperparameters if it were feasible.

The risk using a conjugate gradient optimization technique is that one may get stuck in bad local maxima. Since the algorithm is "greedy" it can get stuck in even very shallow local maxima. It is difficult to say whether this is a big problem in the current context. One could attempt to clarify this by trying multiple random restarts for the optimization, but I have not pursued this. Rather, I just do a single run allowing about 150 function and gradient evaluations, by which time the likelihood is changing very slowly in cases with, for example, 36 hyperparameters.

It should be noted that the resulting MAP estimates for the hyperparameters may be very useful for interpretation of the data. The linear and non-linear parts of the function are separated out in the different sets of hyperparameters (although small ARD hyperparameters may also carry linear contributions) and the magnitude of the different ARD hyperparameters may convey the relative importance of the input attributes through their relation to the characteristic lengths.

The initialisation of the hyperparameters is fairly important, since it may be that very inappropriate initial values will make the partial derivatives of the likelihood very small, thus creating problems for the optimization algorithm. I found that it works well to set $w = 1/m$, $v_0 = 1$ and all of a_0, a_1 and r^2 to $\exp(-2)$.

4.6 Hybrid Monte Carlo

According to Bayesian formalism we should multiply the prior $p(\theta)$ by the likelihood $p(\mathcal{D}|\theta)$ from eq. (4.15) and integrate over the resulting posterior. Unfortunately, the likelihood has a complex form, so analytical integration is infeasible. Instead we approximate the integral using a Markov chain to sample from the posterior

$$\begin{aligned} p\big(y^{(n+1)}|\mathcal{D}, x^{(n+1)}\big) &= \int p\big(y^{(n+1)}|\mathcal{D}, x^{(n+1)}, \theta\big) p(\theta|\mathcal{D}) d\theta && (4.19) \\ &\simeq \frac{1}{T}\sum_{t=1}^{T} p\big(y^{(n+1)}|\mathcal{D}, x^{(n+1)}, \theta^{(t)}\big), && (4.20) \end{aligned}$$

where $\theta^{(t)}$ are samples from the posterior distribution for θ. Note that the terms in the sum are Gaussian distributions which means that the approximate predictive distribution is a mixture of Gaussians with identical mixing proportions. As the number of samples T grows, the approximate predictive distribution will tend to the true predictive distribution. Note, that the predictive distribution may have a complex form, e.g. multi-modal.

The Hybrid Monte Carlo (HMC) method [Duane et al. 1987] seems promising for this application. When attempting to sample from complicated multidimensional distributions it is often advantageous to use gradient information to seek regions of high probability in cases where these gradients can be computed. One can then use a Markov chain that takes a series of small steps while exploring the probability distribution (while ensuring that regions are visited according to their probability). One of the problems with simple implementations of this idea is that the distributions are explored using random walks, which have poor exploratory properties.

The Hybrid Monte Carlo method avoids this random walk behaviour by creating a fictitious dynamical system where θ plays the role of position variables, which are augmented by a set of momentum variables ϕ. The combined system tends to avoid random walks since the momentum will introduce "inertia" in the hyperparameters and tend to keep the system moving in the same direction on successive steps.

Formally, the total energy $\mathcal{H}$ of the system is the sum of the kinetic energy $\mathcal{K}$ and the potential energy $\mathcal{E}$. There are a total of $m + 4$ hyperparameters, and the kinetic energy is a function of the associated momenta $\mathcal{K}(\phi) = \frac{1}{2}\sum_{i=1}^{m+4} \phi_i^2/\lambda$, where λ is the particle mass. The potential energy $\mathcal{E}$ is defined in such a way that $p(\theta|\mathcal{D}) \propto \exp(-\mathcal{E})$. We sample from the joint distribution for θ and ϕ given by $p(\theta, \phi|\mathcal{D}) \propto \exp(-\mathcal{E} - \mathcal{K})$; the marginal of this distribution for θ is the required posterior. A sample of the hyperparameters from the posterior can therefore be obtained by simply ignoring the momenta. Sampling from the joint distribution is achieved in two steps: (i) finding new points in phase space with near identical energies $\mathcal{H}$ by simulating the dynamical system using a discretised approximation to Hamiltonian dynamics, and (ii) changing the energy $\mathcal{H}$ by doing Gibbs sampling for the momentum variables.

I will use a variation of HMC due to Horowitz [1991]. In this approach, having defined the parameters L, ϵ and α, transitions of the Markov chain take place according to the following scheme

1. Starting from the current state (θ, ϕ), perform L *leapfrog* steps with step size ϵ, re-

sulting in the state (θ^*, ϕ^*).

2. With probability $\min(1, \exp[\mathcal{H}(\theta, \phi) - \mathcal{H}(\theta^*, \phi^*)])$, accept the new state, $(\theta, \phi) := (\theta^*, \phi^*)$; otherwise reject the new state, and retain the old state with negated momenta, $(\theta, \phi) := (\theta, -\phi)$.

3. Update the total energy of the system by perturbing the momenta according to $\phi_i := \alpha\phi_i + \nu_i\sqrt{1-\alpha^2}$ for all i, where ν_i are drawn at random from a zero-mean unit-variance Gaussian.

Hamilton's differential equations govern the evolution of the dynamical system through fictitious time τ

$$\frac{d\theta_i}{d\tau} = \frac{\partial\mathcal{H}}{\partial\phi_i} = \frac{\phi_i}{\lambda} \qquad \frac{d\phi_i}{d\tau} = -\frac{\partial\mathcal{H}}{\partial\theta_i} = -\frac{\partial\mathcal{E}}{\partial\theta_i}. \tag{4.21}$$

In practice we cannot simulate these equations exactly, since the partial derivative of $\mathcal{E}$ with respect to θ is a complicated function. The leapfrog iterations are used to approximate the dynamics

$$\begin{aligned}
\phi_i(\tau + \tfrac{\epsilon}{2}) &= \phi_i(\tau) - \frac{\epsilon}{2}\frac{\partial\mathcal{E}}{\partial\theta_i}\big(\theta(\tau)\big) \\
\theta_i(\tau + \epsilon) &= \theta_i(\tau) + \epsilon\phi_i(\tau + \tfrac{\epsilon}{2})/\lambda \\
\phi_i(\tau + \epsilon) &= \phi_i(\tau + \tfrac{\epsilon}{2}) - \frac{\epsilon}{2}\frac{\partial\mathcal{E}}{\partial\theta_i}\big(\theta(\tau + \epsilon)\big)
\end{aligned} \tag{4.22}$$

The formal proof of the correctness of this approach to sampling from the posterior in the context of neural networks can be found in [Neal 1996; Neal 1993]. In the current implementation I use $L = 1$, a single leapfrog iteration to find proposal states. It is not entirely clear whether longer trajectories with the standard algorithm would be better than the use of "persistence", see discussion in [Neal 1993].

The step sizes ϵ are set to the same value for all hyperparameters. This may work fairly well, since the hyperparameters are given in the log domain, such that the step sizes implicitly scale with the magnitude of the underlying parameter. The step size is chosen to scale as $\epsilon \propto n^{-1/2}$, since the magnitude of the gradients at a "typical" point under the posterior are expected to be scale roughly as $n^{1/2}$ when the prior is vague. The particle mass λ is arbitrarily set to 1 and the constant of proportionality for the step sizes is chosen to give a low rejection rate; I have found that $\epsilon = 0.5n^{-1/2}$ typically gives rejection rates of around 1%.

The momenta are updated with a "persistence", $\alpha = 0.95$. This will make the total energy change approximately 20 times more slowly than without persistence. The advantage of persistence is that consecutive steps in hyperparameter space will tend to be in the same direction (because of the momenta) thus avoiding random walks. This is particularly important if the posterior distribution is highly correlated in which case exploration by random walks can be extremely slow. Whether or not the posterior distributions for hyperparameters in the current context are generally highly correlated is an open question. However, it seems reasonable in such cases to introduce a little bit of persistence, since this will help when the distributions are correlated and slow down progress only if the distributions are fairly spherical — situations where sampling is a fairly easy task.

To make predictions, we let the Markov chain run for as long as desired and save the states at regular intervals for predictions. I discard the initial 1/3 of the run, since the chain has to converge on the posterior distribution. I then use 26 samples[1] evenly distributed in the remainder of the run for predictions. It is probably reasonable to expect that fewer samples will suffice for predictions with `gp-mc-1` than are needed for `mlp-mc-1`, since the former model effectively has an implicit integration over "weights". The predictions are made by averaging the predictive distributions for each of the samples of the posterior. Thus, the predictive distribution is a mixture of Gaussians. It should be noted that making predictions may itself be computationally intensive. As well as finding the inverse covariance matrix (for each of the 26 posterior samples) one has to evaluate eq. (4.10) taking $O(n^2)$ for each test point. If there are many test points this may be a considerable effort.

4.7 Future directions

We have seen that the priors implied by large neural networks with Gaussian weight priors can be used directly for modeling without implementing the neural network. It is dubious whether it is useful to attempt to approximate Gaussian processes via Bayesian neural networks with large numbers of hidden units. However, there may be situations where the neural network parameterisation can provide some insight. Finally, it should be noted that neural networks with non-Gaussian priors do not necessarily define Gaussian processes, and useful models may be found in this class of networks.

Commonly, neural networks with fairly small numbers of hidden units are used for modeling purposes. In the case of Bayesian learning, small numbers of hidden units are usually

[1]I originally intended to use 52 of samples for predictions, but due to a misunderstanding of my own program interface I only got 26. Because of the long run-times for the programs, I redefined the method instead of re-running it.

chosen for computational reasons, and in non-Bayesian settings sometimes for fear of over-fitting. It is conceivable that for some modeling problems a network with 3 hidden units can approximate the true function very well, but I do not suspect this to be the general case. The form of the covariance for the neural network is determined by the shape of the activation function of the hidden units. However, there do not seem to be any reasons why tanh units should be particularly advantageous. Therefore, from a statistical point of view there is no clear reason why one should expect Gaussian processes or Bayesian neural networks to perform very differently. This issue may be clarified though empirical tests in Chapter 5. However, very different implementations of these two schemes have been suggested, and it may be that whichever model has the most effective implementation should be preferred.

A number of possible extensions to the basic ideas may be of interest. These include the use of more complicated covariance functions to implement additive models, extensions of computational nature to allow Gaussian processes to be applied to large datasets and extensions to classification tasks. These possibilities are discussed briefly below.

One possibility is to attempt to model additive functions with Gaussian processes. Additive functions involve multiple components that each depend only on a subset of the inputs. These components interact only additively. Such additive models are popular in statistics [Hastie and Tibshirani 1990] both for reasons of computational convenience and to enable discovery of interpretable structure in the data. Additive functions in the GP-framework can be attained by simply adding in several ARD-terms of the form of the exponential in eq. (4.12), each governed by a separate set of hyperparameters but otherwise identical. These multiple sets of ARD hyperparameters may be able to discover additive structure in the data, such that only a certain number of inputs are considered relevant in each of the covariance contributions. See [Neal 1996] for a discussion of additive models in the context of neural networks. The computational overhead for these additive models is fairly modest; at each iteration one needs to calculate the partial derivatives of the likelihood, and perhaps the search for good hyperparameter values may be more demanding, since the space is larger.

As an example, two functions drawn at random from additive covariance functions are shown in fig. 4.4. Each of the two additive components only depends on one of the inputs. The ARD hyperparameters (for the active inputs) have the same values as in fig. 4.2.

One of the severe limitations of Gaussian processes is the computational requirements when the training sets get large. I have been able to handle up to about 1000 training cases,

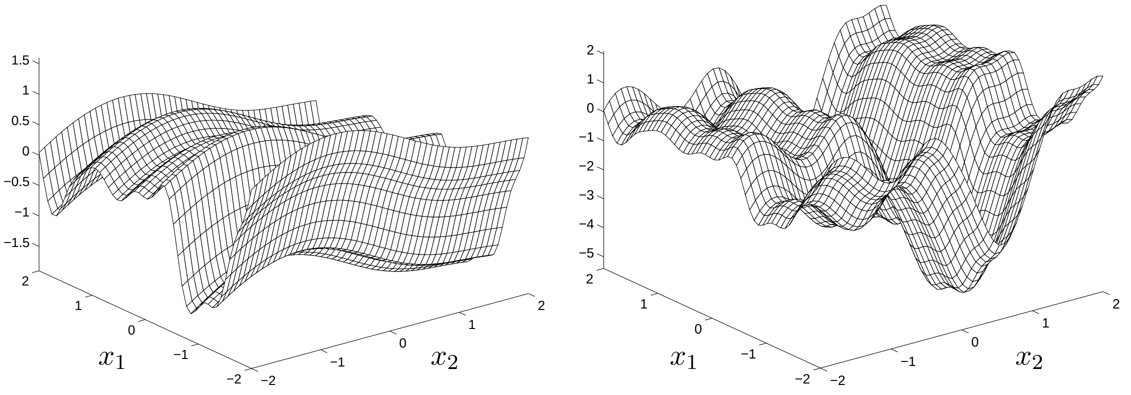


Figure 4.4: Functions drawn at random from additive ARD prior covariance functions. There are two input dimensions x_1 and x_2. There were two ARD-terms in the covariance function each of which depends only on a single input variable. In the left-hand plot the ARD parameters for the "active" inputs were $w_1 = 10$ and $w_2 = 0.5$; in the right-hand plot $w_1 = 10$ and $w_2 = 10$. In both plots $v_0 = 1$.

requiring 24Mb memory (my implementation stores 3 $n \times n$ matrices for time efficiency) and 8 hours of training time; since these requirements scale quadratically and cubically with the number of training cases it becomes difficult to handle significantly larger training sets. It should be noted that algorithms exist for matrix inversion in less than $O(n^3)$ time; these algorithms are not easily implemented, and for modest values of n (such as a few thousands), the computational benefits are not overwhelming.

Another possibility is to consider approximate techniques for matrix inversion. Two approaches seem promising here. One could attempt to make the covariance sparse by zeroing terms of small magnitude; this might work best in conjunction with the removal of the linear term from the covariance matrix (one might fit a standard linear model and apply the GP to the residuals from a linear fit). The use of a sparse covariance matrix corresponds to only considering points that actually have high covariance, and would probably yield good approximations, say considering the 20 points with highest covariance to each of the training points. However, it may be difficult to find algorithms that can take advantage of the form of the covariance matrix. This approximate scheme could either be employed for long trajectories of leapfrog steps, with a final acceptance or rejection based on an exact calculation; or one could simply use the approximate computations in the current scheme (thus abandoning the guarantee of eventual convergence to the exact posterior).

Secondly, it should be noted that we do not need the entire inverse covariance matrix in our computations, but the three quantities xQ^{-1}, $\det(Q)$ and $\mathrm{trace}(Q^{-1}X)$, when x is a vector and X a matrix. Such terms can be approximated efficiently using conjugate gradient and

Monte Carlo methods [Skilling 1989; Skilling 1993], and this approximation has recently been implemented for Gaussian processes [Gibbs and MacKay 1996].

Gaussian processes for classification have recently been explored by Barber and Williams [1996].

Chapter 5

Experimental Results

This chapter contains the results and discussions of empirical tests of learning methods. The tests are focused on measuring and comparing predictive performance for several real and simulated datasets under varying computational constraints. All the experiments are carried out using the DELVE environment; in particular the statistical framework developed in Chapter 2 will be used extensively. The methods which are tested are all described in detail in Chapters 3 and 4.

Three sets of experiments are presented. The first experiment addresses the question whether application of bagging to the MARS procedure is advantageous. Such studies comparing two variants of the same method are a common and important tool for model development.

The second set of experiments attempts to rigorously compare seven different methods on a real dataset, from the well known "Boston Housing" study. This data has been widely used for benchmarking purposes and has also been incorporated in DELVE. These experiments are aimed at clarifying how large the uncertainties in the performance measurements may be when using real datasets of limited size.

Finally, a number of simulated datasets will be used in an extensive comparison of the seven methods. The aim is to test whether the complicated Bayesian neural networks and Gaussian Processes are able to outperform the simpler and more traditional methods. In particular, attention will be paid to computational considerations, and we will attempt to clarify how much computation time is needed to gain possible advantages of the complicated methods.

5.1 The datasets in DELVE

The experiments in this chapter will use data from two different sources. The "Boston Housing" dataset is a well known real dataset that has previously been used for benchmarking purposes. The other data source is the `kin` and `pumadyn` data families, which have been generated from a realistic robot arm simulator specifically for benchmarking purposes.

In the DELVE environment the learning problems are specified using a hierarchical scheme, whose naming convention will be used in the following. The *datasets* are at the top level; these contain a list of cases (sometimes called examples) with their attribute values as well as some rudimentary information about the ranges of possible values. At the top level there is no mention of what should be learnt. At the next level, called the *prototask* level, a list of attributes to be used as inputs and outputs (in the case of supervised learning) are given and the cases to be used are specified. At the *task* level, the sizes of training and test sets are given and the desired loss function is specified. At the task level, enough information about the problem has been specified such that the expected loss for a method is well defined. At the bottom of the hierarchy are the *task-instances*, containing the actual training and test sets on which the methods can be run. Prototasks with similar characteristics are loosely categorised into *families*.

The "Boston Housing" data was collected in connection with a study of how air quality affects housing prices [Harrison and Rubenfeld 1978]. This data set is publicly available at the UCI database [Murphy and Aha 1994] and in DELVE. The dataset has been used in several benchmarking studies, e.g. [Neal 1996; Quinlan 1993]. The dataset has a total of 506 cases, containing values for 14 attributes (see fig. 5.1).

The object is to predict the median house value (MEDV) from the other 13 attributes. In DELVE, this dataset is called `boston` and the associated prototask is called `boston/price`; the data is randomly divided into 256 cases for training and a single common test set of 250 cases according to the 2-way ANOVA model discussed in section 2.5. The 2-way model for the analysis of losses was chosen since the total number of cases is fairly limited. Three sizes of tasks are generated containing 8 instances of 32 training cases, 4 instances of 64 training cases and 2 instances of 128 training cases.

CRIM	per capita crime rate by town
ZN	proportion of residential land zoned for lots over 25,000 sq. ft.
NDUS	proportion of non-retail business acres per town
CHAS	Charles River dummy variable (1 if tract bounds river; 0 otherwise)
NOX	nitric oxides concentration (parts per 10 million)
RM	average number of rooms per dwelling
AGE	proportion of owner-occupied units built prior to 1940
DIS	weighted distances to five Boston employment centres
RAD	index of accessibility to radial highways
TAX	full-value property-tax rate per $10,000
PTRATIO	pupil-teacher ratio by town
B	$1000(Bk - 0.63)^2$ where Bk is the proportion of blacks by town
LSTAT	lower status of the population
MEDV	Median value of owner-occupied homes in $1000's

Figure 5.1: The attributes of the `boston` dataset

The `kin/dist` and `pumadyn/accel` families each contain 8 datasets that are synthetically generated from realistic simulations of a robot arm. Both families were generated by Zoubin Ghahramani using a simulator developed by Corke [1996]. Below are short descriptions of these datasets — for more details refer to the DELVE web-site.

The static state of the robot arm is defined in terms of the lengths, a_i, the twists, α_i, offset distances, d_i, and joint angles, θ_i for each of the links in the arm. The dynamic state of the arm can be described by including forces and the first and second order time derivatives of the variables, here denoted by the "dot" notation, e.g. $\dot{\theta}_i$ for the angular velocity at joint i.

The tasks in the `kin/dist` family is to predict the distance of the end-point of an 8-link robot arm from a fixed point as a function of various state parameters. Thus, these tasks are static and of a geometric nature. The tasks in this family have either 8 or 32 inputs. For the tasks with 8 inputs, the inputs are the joint angles, θ_i, and all other parameters are set to fixed values. Uniform noise was added to the inputs, so the noise modeled at the outputs is not expected to be Gaussian. For the `kin/dist` tasks with 32 inputs, the inputs were a_i, α_i, d_i and θ_i. The joint angles, θ_i, were corrupted by additive uniform noise, and α_i, a_i, d_i and target distance were corrupted by multiplicative uniform noise, meant to emulate noisy measurements.

The `pumadyn/accel` family contains 8 datasets synthetically generated from a realistic simulation of the dynamics of a "Puma 560" robot arm. The tasks associated with these datasets consist of predicting the angular acceleration of the end-link of the robot arm, given six joint angles, θ_i, six joint velocities, $\dot{\theta}_i$, five joint torques, τ_i, five multiplicative mass changes, Δm_i, five multiplicative length changes, Δl_i and five multiplicative perturbations in viscous friction of joints, Δb_i. Gaussian noise was added to θ_i, $\dot{\theta}_i$ and τ_i. For the 8-dimensional tasks, the inputs were the joint angles and velocities of links 1, 2 and 3 and the torques at joint 1 and 2 — all other joint angles, velocities and torques were set to zero and there were no fluctuations in masses, lengths or viscosities. For tasks with 32 inputs, all parameters were allowed to vary. The flavour of these task is to learn the dynamical properties of the arm.

Both data-families give rise to 8 different tasks of various characteristics. The tasks have names (e.g. `pumadyn-32fh`) which encode some aspects of the tasks. The number in the name of the prototask (8 or 32) denotes the input dimensionality. The following '`f`' or '`n`' denotes "fairly linear" tasks, and "non-linear" tasks. The trailing '`m`' or '`h`' denotes medium or high levels of unpredictability (noise). The definitions of these features are given in terms of indexes. The index of non-linearity is

$$I_{nl} = \frac{E_L}{E_C}, \tag{5.1}$$

where E_L is the squared error loss of the best linear fit on noise-free data and E_C is the squared error loss for the corresponding constant fit. Note that E_C is the variance of the target, and therefore I_{nl} is one minus the proportion of the variance explained by a linear model. Datasets with $I_{nl} < 0.05$ were classified as being *fairly linear*, and $I_{nl} > 0.2$ were classified as being *non-linear*. The index of unpredictability is defined by

$$I_u = \frac{E_N}{E_C}, \tag{5.2}$$

where E_N is the expected squared error loss on noisy data of the best non-linear fit, i.e., the expected value of the function. In practice, E_N was estimated as follows: for each case in the dataset, several targets were generated for fixed inputs and the average variance (over cases) of these targets was taken as a measure of E_N. Data with $0.01 > I_u > 0.05$ are classified as having medium noise, and tasks where $I_u > 0.2$ as high noise.

Each dataset contains 8192 cases which are divided evenly between training and test set. Since there are a large number of cases, the tasks are set up according to the hierarchical model described in section 2.4. Tasks of 5 different sizes are created, with training sets containing 64, 128, 256, 512 and 1024 cases respectively. For each of the four smallest tasks there are 8 instances and the 8 corresponding test sets contain 512 cases each; for the largest task there are only 4 instances and the corresponding test sets comprise 1024 cases.

The advantages of the data in these two families are their large test sets as well as the various controlled attributes: input size, degree of non-linearity and degree of non-predictability. This may enable us to pinpoint the characteristics for which certain learning methods excel or perform poorly.

All the experiments are conducted using the DELVE environment. Three different loss functions will be used: squared error loss, absolute error loss and negative log density loss. The last of these is evaluated by measuring the negative log of the density of the targets in the test set, under the predictive distribution from the model. The `mars3.6-bag-1` and `mlp-ese-1` methods do not produce predictive distributions and will therefore not be evaluated with this error measure. The losses reported here have been *standardized* by DELVE, such that we can make intuitive sense of the losses without knowing the domain of the data. For squared error loss, standardization is obtained by dividing the loss by the variance of the targets of the test cases; this causes the trivial method that guesses the mean of the data to have a loss of approximately one. For absolute error loss, standardization is done by division by the absolute deviation from the median. For negative log probability, standardization is done by subtracting the loss that would be obtained if one used a Gaussian predictive distribution with mean and standard deviation from the test set. Thus, standardized losses for the negative log density loss function are approximately zero for simple methods and negative for better methods.

5.2 Applying bagging to MARS

The DELVE framework can be used both to compare methods which differ in fundamental principles as well as testing small refinements of methods. As an example of this latter sort, I will attempt to clarify whether bagging [Breiman 1994] improves the performance of the MARS method [Friedman 1991] by comparing the methods `mars3.6-1` and `mars3.6-bag-1`. Since MARS is not very computationally demanding, it may often be possible to use bagging. One possible disadvantage of using bagging may be increased difficulty associated with interpretation of the composite model.

The two methods `mars3.6-1` and `mars3.6-bag-1` as described in section 3.5 are tested on the `kin` and `pumadyn` data families. The results for the squared error loss function

training set sizes	64		128		256		512		1024	
	r	p	r	p	r	p	r	p	r	p
`kin-32fh`	36.4	1	27.2	2	8.9	16	3.7	13	-3.8	22
`kin-32fm`	26.3	4	24.1	2	10.8	30	-8.1	28	-5.7	47
`kin-32nh`	40.1	1	28.7	1	23.6	1	12.3	1	6.8	1
`kin-32nm`	25.8	1	28.8	1	21.7	1	13.0	1	6.1	1
`kin-8fh`	37.3	1	16.0	4	16.8	1	13.2	2	3.8	4
`kin-8fm`	37.0	7	17.0	5	13.6	20	13.0	2	9.8	7
`kin-8nh`	20.7	5	17.6	1	11.6	1	8.3	1	9.4	4
`kin-8nm`	31.3	2	19.2	1	16.0	1	11.5	1	7.7	1
`pumadyn-32fh`	21.3	2	32.7	13	11.4	1	9.0	1	2.6	4
`pumadyn-32fm`	4.3	52	24.2	2	22.4	1	3.8	1	3.8	3
`pumadyn-32nh`	25.0	21	22.2	24	9.3	2	10.4	1	3.3	17
`pumadyn-32nm`	28.3	43	-23.9	44	22.5	1	16.8	1	5.3	7
`pumadyn-8fh`	12.9	11	3.6	59	4.5	16	5.5	1	3.6	7
`pumadyn-8fm`	20.2	2	15.1	2	8.4	3	4.4	2	5.9	4
`pumadyn-8nh`	12.1	41	19.0	2	10.5	1	10.1	1	4.0	7
`pumadyn-8nm`	22.7	8	30.3	1	17.0	3	13.0	3	7.5	25
`averages`	25.1		18.9		14.3		8.7		4.3	

Figure 5.2: Comparison of plain `mars3.6-1` and bagged `mars3.6-bag-1` on the `kin` and `pumadyn` data families. For each task, the expected decrease in squared error introduced by bagging is given in percent $r = 100\% \times (L_p - L_b)/L_p$, where L_p is the expected squared error loss for plain mars and L_b is the expected squared error loss for bagged mars. The columns marked p give the p-values in percent rounded up to the nearest integer for a pairwise comparison between the two methods.

are tabulated in fig. 5.2. The table shows that bagging leads to an improved expected performance on all except 4 tasks out of a total of 80. Further, for the tasks where a decrease was seen in performance, the difference was not statistically significant — the lowest p-value being 0.22. For tasks with small training sets the relative improvement is substantial — on average 25% for training sets containing 64 cases. For larger training sets the benefits of bagging decline. This is not surprising, since given enough training examples plain MARS can probably get a performance close to the theoretical limit (the inherent noise-level in the data) which does not leave much scope for improvement by bagging. It is conceivable however, that bagging may degrade performance for very large training sets.

In conclusion, the experiments strongly favour bagging MARS. No example was found of statistically significant decrease in performance by using bagging, for any size or dataset, although this does not rule out the possibility of the existence of such tasks. It should be noted that it would not be easy to show theoretically that bagging always improves certain methods, such as is possible for example for ensembles [Hansen and Salamon 1990]. This is because the individual members in the bag do not have the same expected performance as a model trained with plain MARS. The empirical evidence suggests that the benefits of averaging outweigh the expected decline in performance of the individual models.

Similar experiments could provide insights into the possible utility of more elaborate versions of the other methods under investigation. For example, weight-decay selected by leave-one-out cross-validation can be implemented efficiently for the linear model, and may under broad conditions be helpful. Another possibility is the extension of `knn-cv-1` to incorporate a running line smoother, [Cleveland 1979].

5.3 Experiments on the `boston/price` prototask

The performance of seven different learning methods was evaluated using three different loss functions on `boston/price`. In all cases, the analysis of losses was done using the 2-way ANOVA model described in section 2.5.

The results are displayed using a special plot, which conveys information about average losses with standard errors as well as p-values for pairwise comparisons, fig. 5.3. In the upper half of the plots the estimated expected losses are indicated by small horizontal lines intersected by vertical lines indicating the standard error. The y-axis gives the standardized losses. The losses for different task-sizes are grouped together in rectangles with the training set sizes indicated above the rectangle. Inside the rectangles, losses for several methods are given; the horizontal ordering of the methods is the same as the vertical ordering given below the y-axis. The ∅ symbol indicates that the method has not been run. An upward arrow indicates that the loss is off the graph.

Below the x-axis is a matrix of p-values for pairwise comparisons, obtained from paired t-tests or F-tests. The p-values have been rounded up to the nearest whole number in percent, and are reported in one digit in percent. Thus the strongest p-value is 1% and the weakest reported value is 9%. If the value was larger than 9% it is not considered significant and is not reported. The p-values are reported *in the column of the winning method.* Thus, looking column-wise at the matrix tells you which methods the current method significantly out-performs. Looking row-wise, you see which methods significantly out-performed the method of that row. The dots in the matrix are simply included to help guide the eye. These matrices are most easily read row-wise. Note that there are two possible causes of the absence of an entry in the row: either the row method performed worse, or it did not perform statistically significantly better. However, the presence of an entry in a row always means that the row method was significantly outperformed.

The results of tests for the `boston/price` data is given in fig. 5.3 – 5.5, for squared error

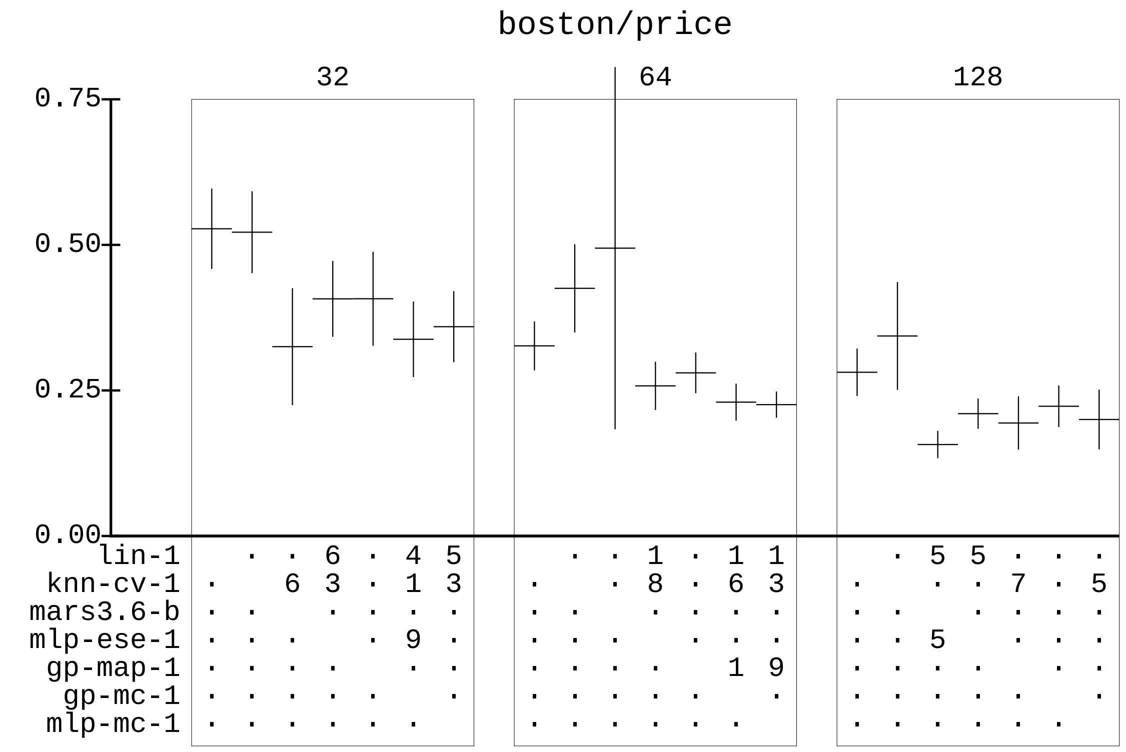


Figure 5.3: Performances on the `boston/price` data using squared error loss. The horizontal ordering of the methods is the same as the vertical ordering. The numerical values in the matrices under the x-axis are (rounded up) p-values for pairwise comparisons given in percent. The p-values appear in the column of the winning method. See section 5.3 for further details.

loss, absolute error loss and negative log density loss respectively. The most striking feature of these plots is the large error bars and consequently small number of significant results.

Several aspects are important when interpreting the plots. Recall, that the error bars are computed separately for each method and therefore do not reflect the effects of pairing. In fig. 5.3 for squared error loss and 64 training cases, the `gp-map-1` and `gp-mc-1` methods have significantly overlapping error-bars; nevertheless, the p-value for the pairwise comparison is $p < 1\%$ in favour of `gp-mc-1`. Here the pairing of the two methods leads to a cancellation of noise, thus enabling a stronger statement about the difference in performance than would be anticipated from the error-bars alone.

A different effect works in the opposite direction. The error-bars are computed from single estimates of the variability due to training sets and test-cases, which are themselves prone to uncertainty which is not accounted for in the plots, but which the F-tests producing the p-values do take into account. As an example, compare `knn-cv-1` and `mars3.6-bag-1` for

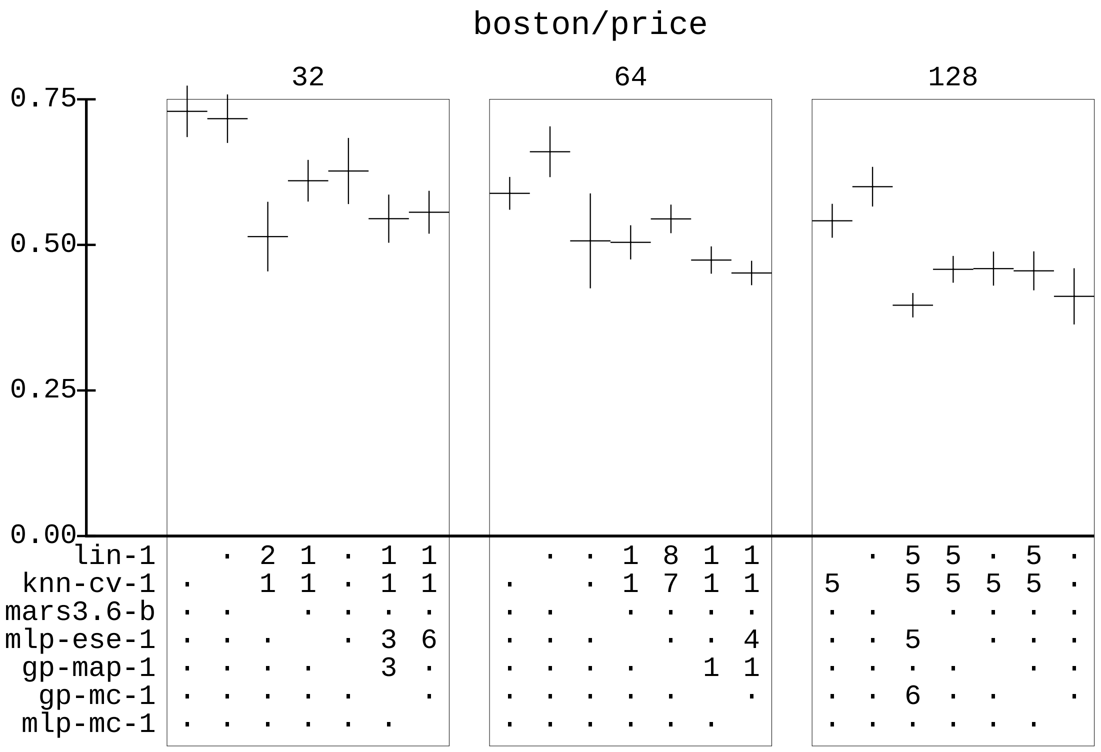


Figure 5.4: Performances on the `boston/price` data using absolute error loss.

128 training cases in fig. 5.3. Here the error-bars are clearly separated on the graph, but the pairwise test comes out insignificant. This is due to the uncertainty in the estimated effects of training sets; this uncertainty is large because only 2 instances were available for this size of training set. Surprisingly, the difference between `gp-map-1` and `knn-cv-1` is (marginally) significant in this example, although `gp-map-1` has larger error-bars and worse expected performance than `mars3.6-bag-1` — again this different outcome is due to the effects of pairing. Remember that the strongest possible p-value for tests with the 2-way ANOVA and only two instances is $p = 5\%$ (as discussed in section 2.5).

All in all, the test results are not very strong. The `lin-1` and `knn-cv-1` never have lower expected loss than the other methods, except for `gp-map-1` for the two smallest tasks for negative log density loss in which case the differences are not significant. In general, the differences between `lin-1`, `knn-cv-1` and the remaining methods are only sometimes considered significant. Comparisons within the better methods remain inconclusive, except that `mlp-ese-1` and `gp-map-1` never win significantly over the others. We also note that the results for different loss functions show similar tendencies.

It is clear that the usefulness of the `boston/price` data is severely limited by the large

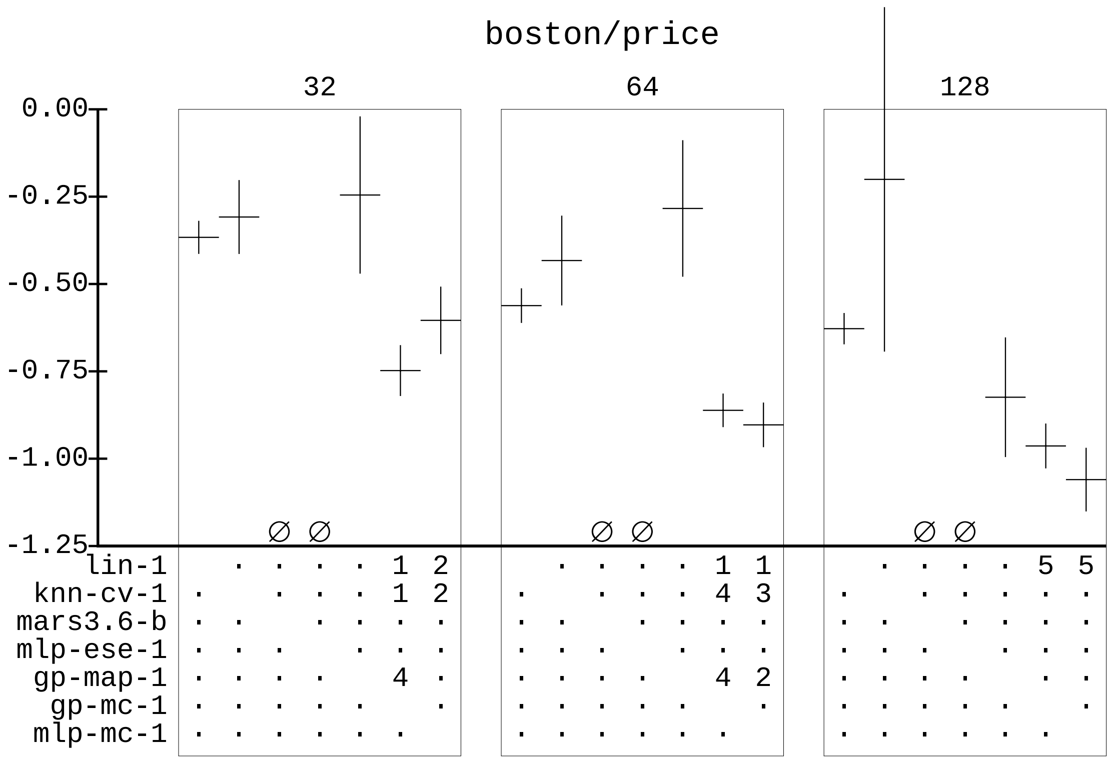


Figure 5.5: Performances on the `boston/price` data using negative log density loss; note that the `mars3.6-bag-1` and `mlp-ese-1` methods do not produce a predictive distribution and this loss type cannot be applied to them.

uncertainties associated with the loss estimates. Differences in expected losses as large as $10 - 20\%$ are often not declared significant. However, differences of this magnitude would often be important for practical purposes.

It may be argued that the failure to obtain statistically significant results is caused by a sub-optimal split of data into training and test sets. Since the cases were split roughly half/half into training and test partitions, it is not possible to increase these sets by more than a factor of two; leading to an expected reduction in error-bars of the order $\sqrt{2}$. In conclusion it seems unlikely that experiments with a much larger sensitivity can be designed using the `boston` data if one insists on using disjoint training sets. As discussed in section 2.3, the analysis of experiments with overlapping training sets would require assumptions to be made about the dependencies introduced by overlapping training sets, which seems difficult. However, this route could be pursued in future, to enable meaningful assessment to be made with real datasets not exceeding 500 cases.

In the literature on learning methods, it is not rare that results on even smaller datasets than boston (with 506 cases) are reported — and sometimes losses differing by only a few percent are reported. The present results suggest that it may be wise to exercise caution when evaluating results of tests on such small datasets. In general, one should be wary of results that do not consider the uncertainties in the observed performances.

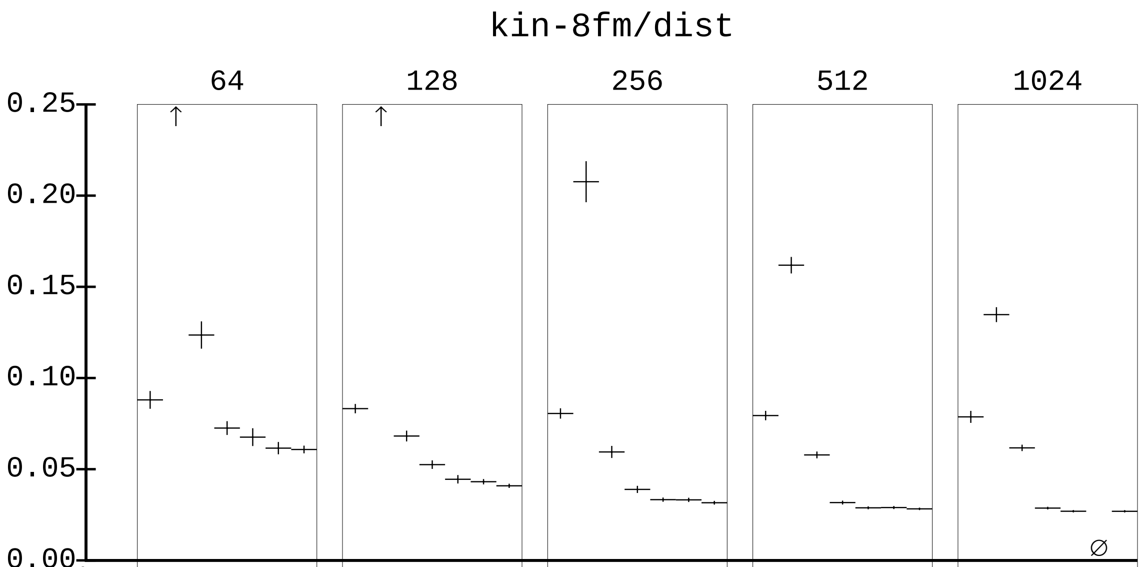
5.4 Results on the kin and pumadyn datasets

Extensive tests have been carried out on the kin and pumadyn families. These datasets are much larger than the boston set, allowing reliable detection of much smaller differences in performance. The same seven methods are tested as in the experiments on the boston data. Results on each of the 16 prototasks on the 5 different task sizes are given in the plots in fig. 5.6 – 5.13 on the following pages. For reasons of brevity only results for squared error loss function are given.

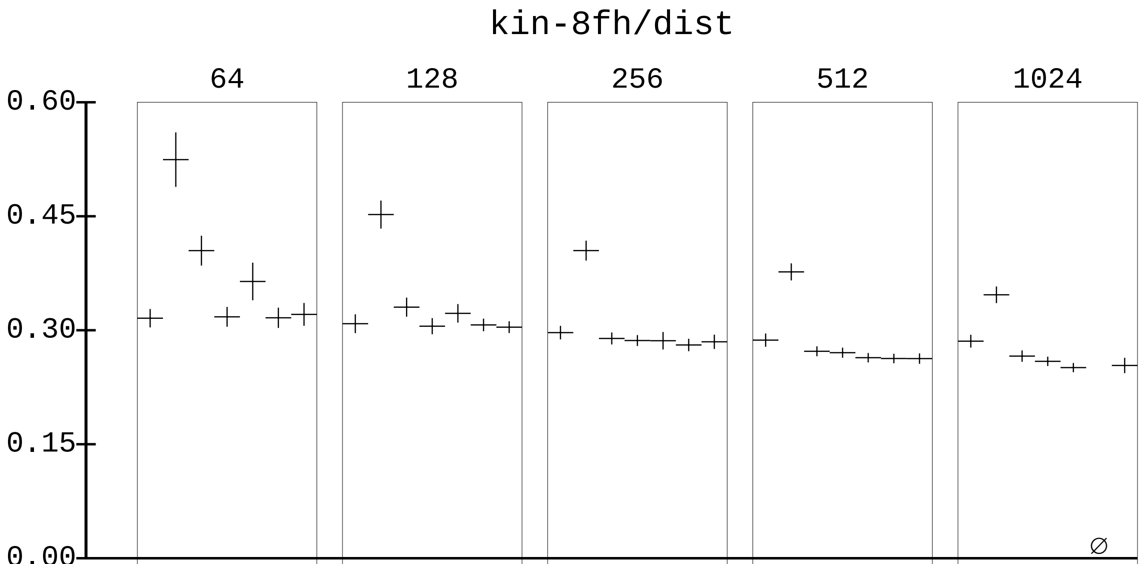
The error-bars in these plots are mostly quite small, although there is a tendency for the tasks with small training sets to have slightly larger error-bars than the tasks with large training sets. This tendency probably reflects a larger training set effect — for such small training sets, the performance of the methods is sensitive to the details of the training sets. However, in general the error-bars are small and a large number of differences are significant.

Notice that the mlp-mc-1 method has larger error-bars than the other methods for tasks with small training sets for the kin-8nm, pumadyn-32nm and pumadyn-32nh datasets. This may be caused by convergence problems for that method. Sometimes it can be hard for this method to find areas of high posterior probability if the posterior is very peaked; the Markov chain responsible for the integration over the posterior may take a long time to "discover" a posterior peak and spend much time sampling from big volumes under the prior. This effect can often be seen if the estimated noise level is plotted as a function of the iteration number. At first this noise estimate is relatively high and then suddenly jumps to a much lower value, and seldom returns to the high values. I have tried to reduce this effect by initially training the network with fixed values for the hyperparameters (as described in section 3.7) in the hope that regions of high posterior probability could be located rapidly under these conditions. Perhaps a more elaborate initialisation scheme would be advisable.

The different tasks exhibit very different behaviours. The performance on the kin-8nm task improves strongly as the number of training cases increases. In contrast, the training set size seems of much less importance for kin-8fh, except for the knn-cv-1 method which is

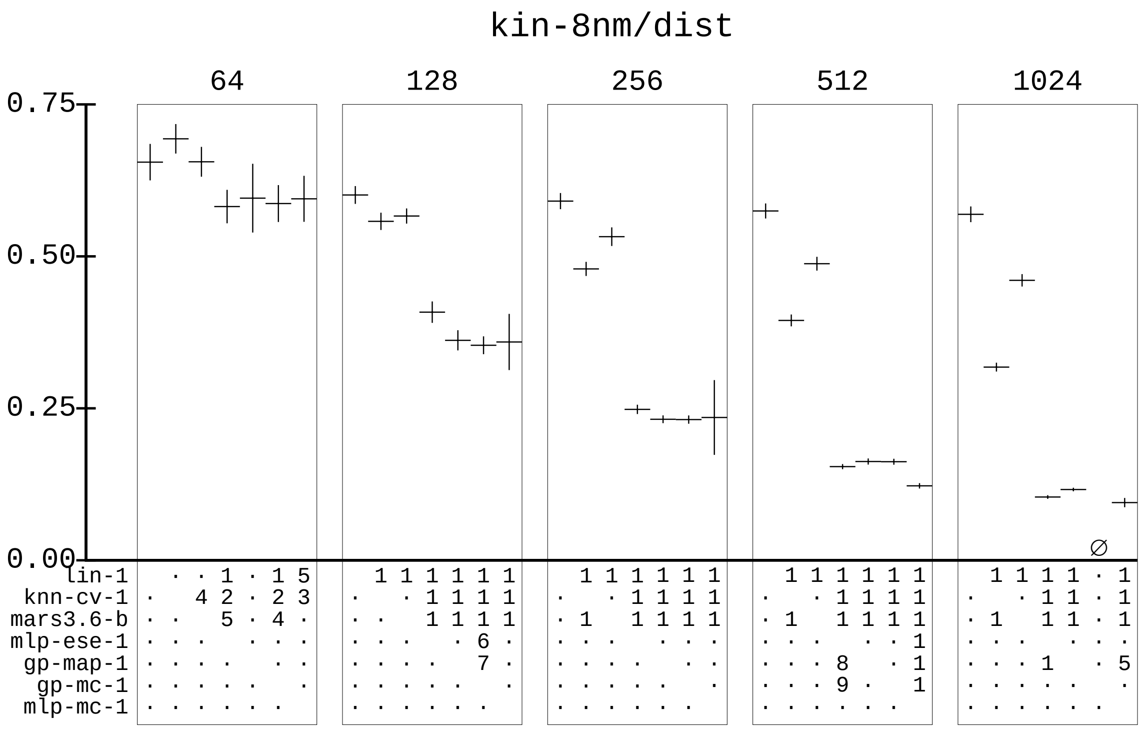


	64	128	256	512	1024
lin-1	· · 1 1 1 1	· 1 1 1 1 1	· 1 1 1 1 1	· 1 1 1 1 1	· 1 1 1 · 1
knn-cv-1	1 1 1 1 1 1	1 1 1 1 1 1	1 1 1 1 1 1	1 1 1 1 1 1	1 1 1 1 · 1
mars3.6-b	1 · 1 1 1 1	· · 1 1 1 1	· · 1 1 1 1	· · 1 1 1 1	· · 1 1 · 1
mlp-ese-1	· · · · 5 2	· · · 1 1 1	· · · 1 1 1	· · · 1 1 1	· · · 4 · 2
gp-map-1	· · · · 3 6	· · · · · ·	· · · · · 1	· · · · · 2	· · · · · ·
gp-mc-1	· · · · · ·	· · · · · ·	· · · · · 1	· · · · 7 2	· · · · · ·
mlp-mc-1	· · · · · ·	· · · · · ·	· · · · · ·	· · · · · ·	· · · · · ·



	64	128	256	512	1024
lin-1	· · · · · ·	· · · · · ·	· · 2 9 3 7	· 1 1 1 1 1	· 1 1 1 · 1
knn-cv-1	1 1 1 1 1 1	1 1 1 1 1 1	1 1 1 1 1 1	1 1 1 1 1 1	2 1 1 1 · 1
mars3.6-b	1 · 1 · 1 1	· · 7 · 5 6	· · · · · ·	· · · 4 1 1	· · · 2 · ·
mlp-ese-1	· · · · · ·	· · · · · ·	· · · · · ·	· · · 9 3 6	· · · 1 · ·
gp-map-1	2 · · 2 2 2	· · · · 7 ·	· · · · · ·	· · · · · ·	· · · · · ·
gp-mc-1	· · · · · ·	· · · · · ·	· · · · · ·	· · · · · ·	· · · · · ·
mlp-mc-1	· · · · · ·	· · · · · ·	· · · · · ·	· · · · · ·	· · · · · ·

Figure 5.6: Experimental performance comparisons using squared error loss function for the `kin` tasks with 8 inputs, fairly linear relationships, and medium and high noise levels.



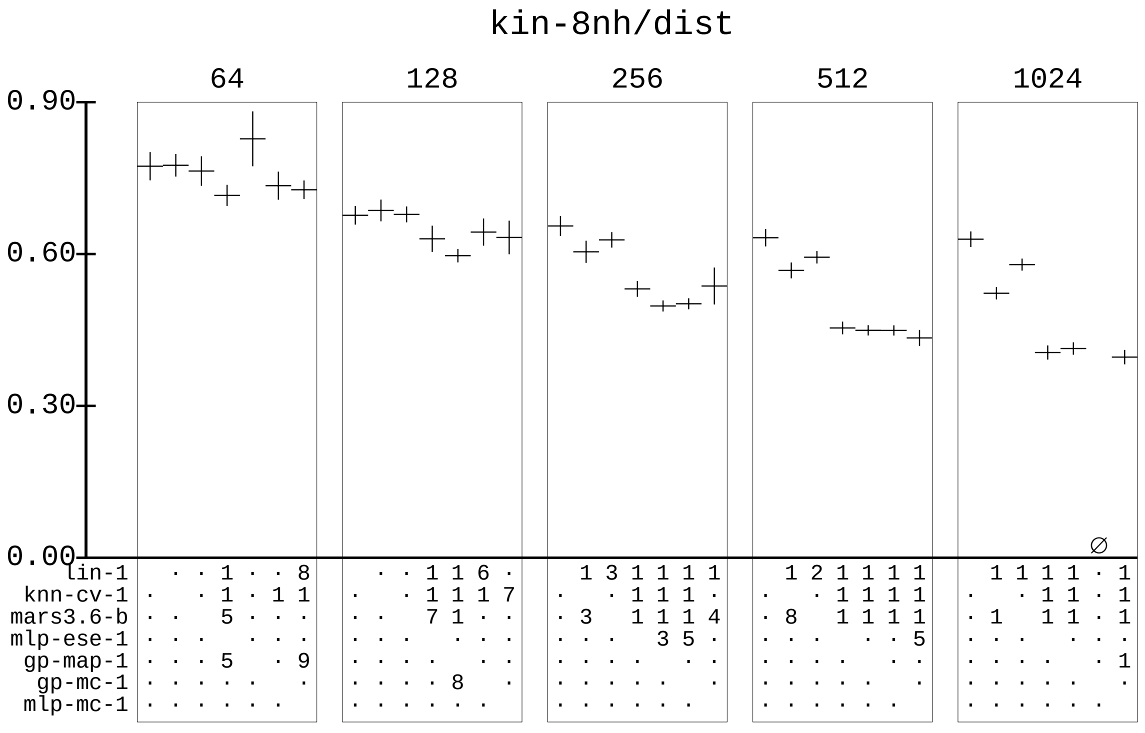
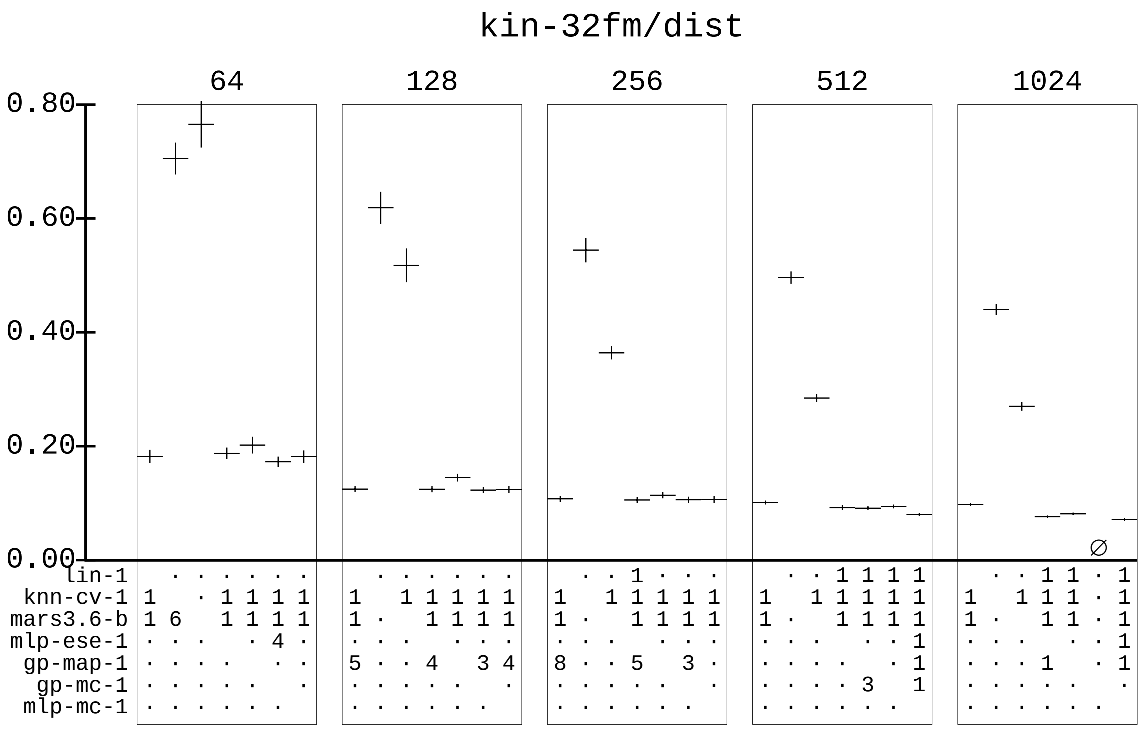


Figure 5.7: Experimental performance comparisons using squared error loss function for the `kin` tasks with 8 inputs, non-linear relationships, and medium and high noise-levels.



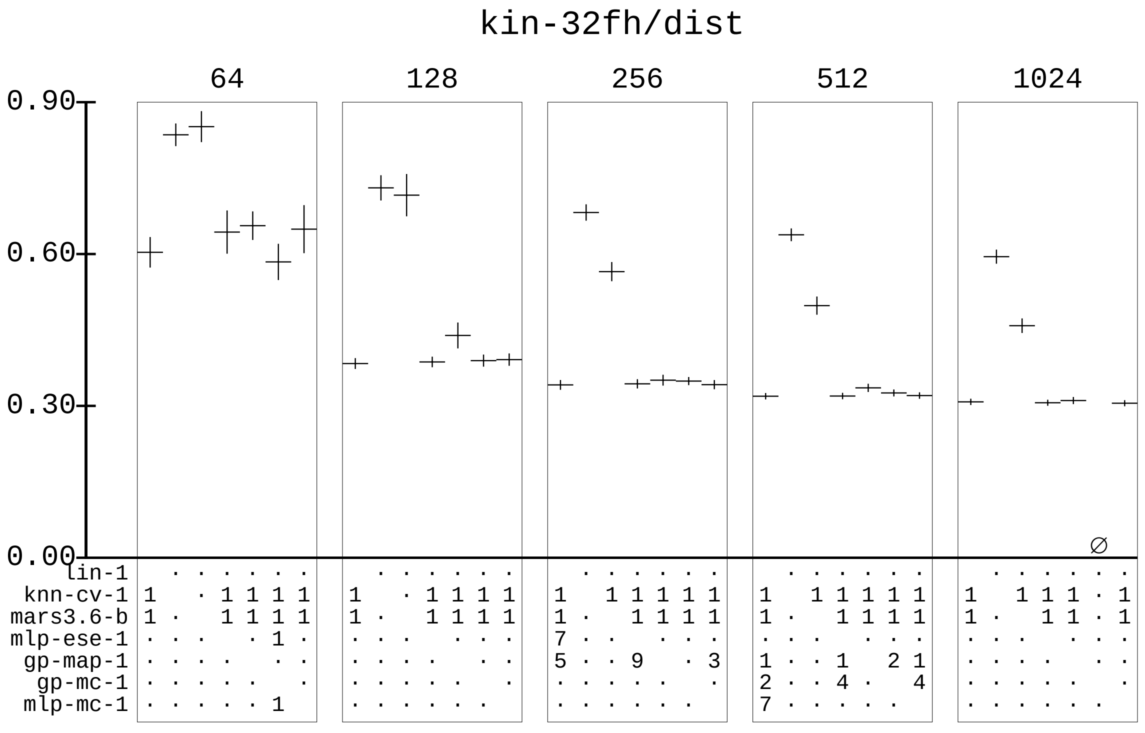
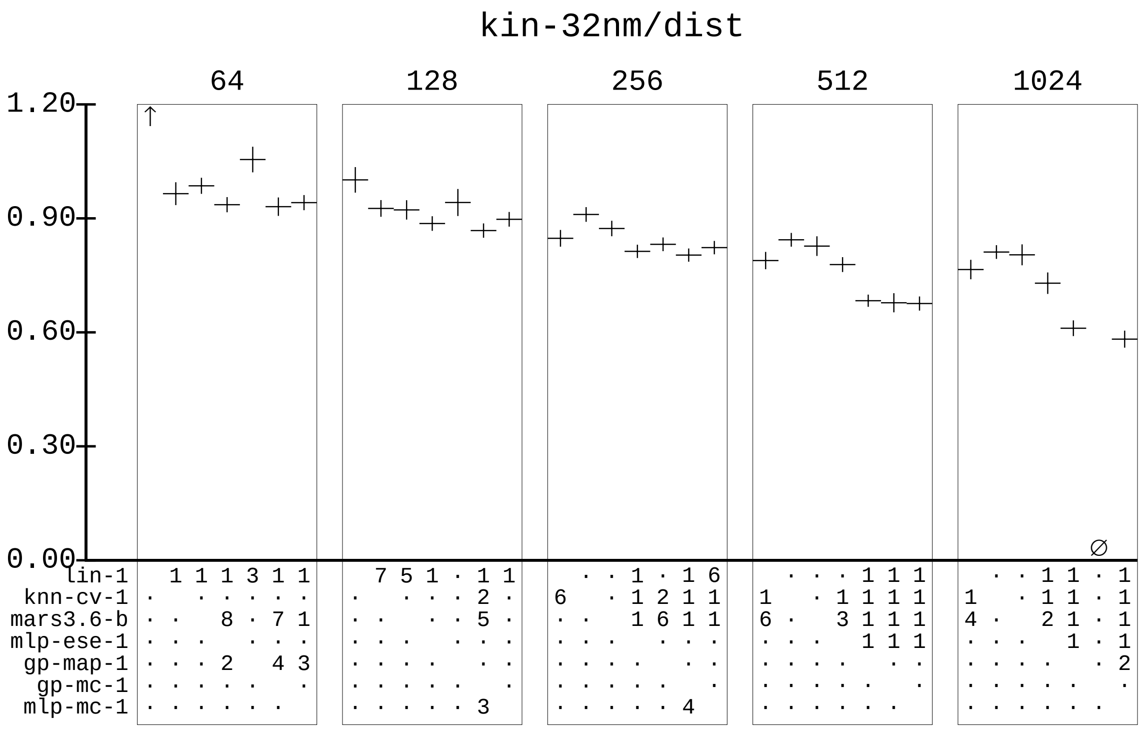


Figure 5.8: Experimental performance comparisons using squared error loss function for the `kin` tasks with 32 inputs, fairly linear relationships, and medium and high noise-levels.



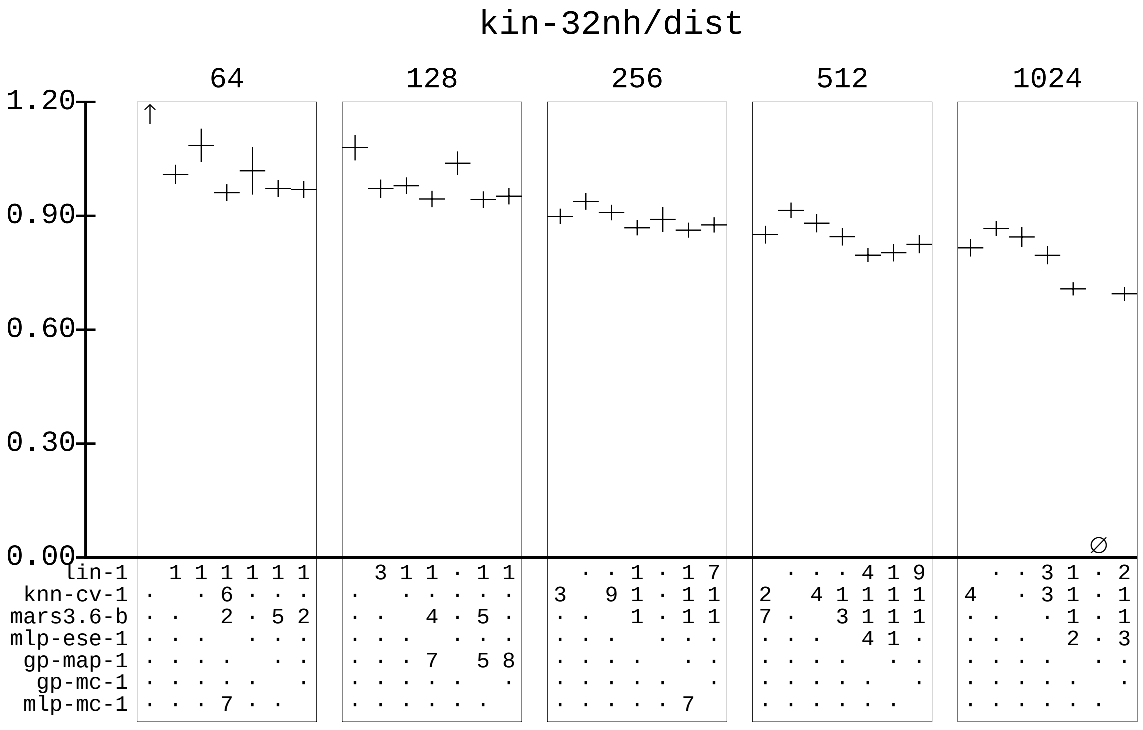
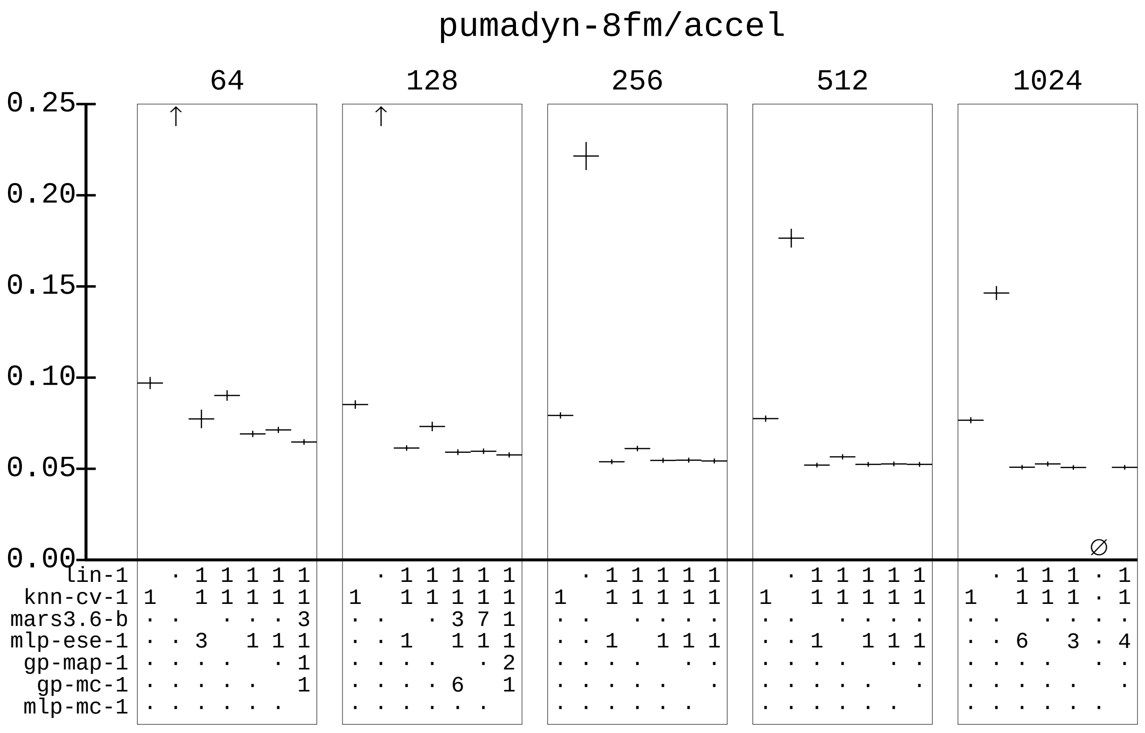


Figure 5.9: Experimental performance comparisons using squared error loss function for the kin tasks with 32 inputs, non-linear relationships, and medium and high noise-levels.



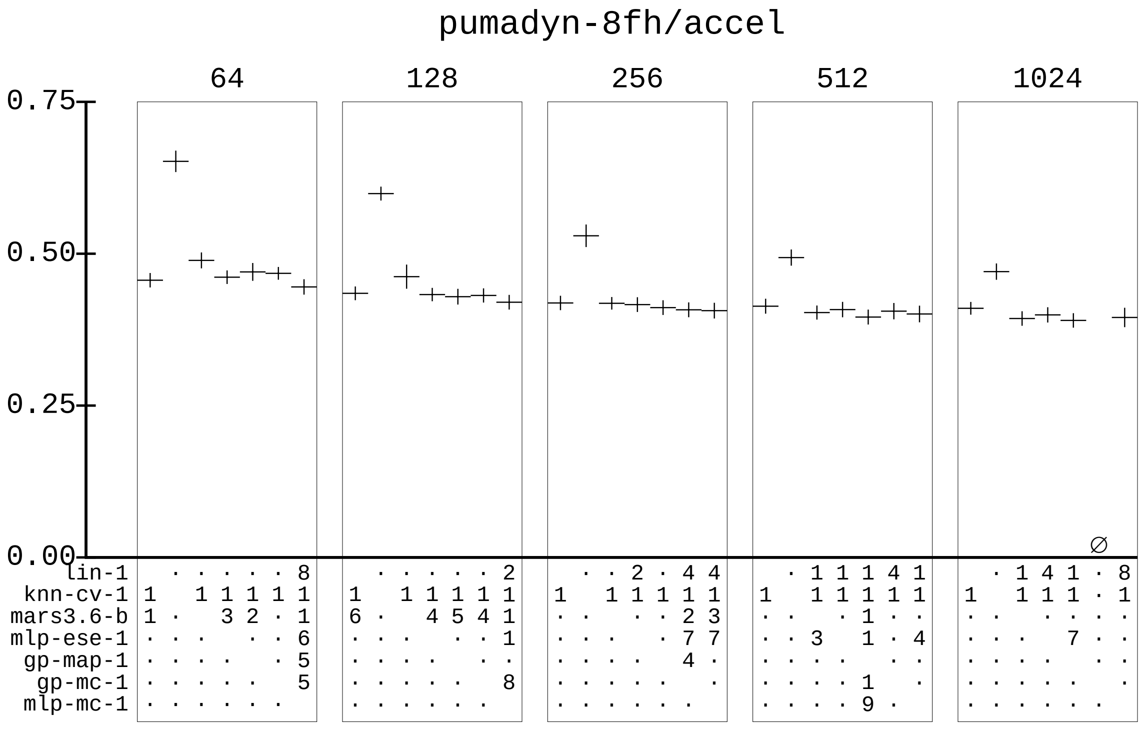
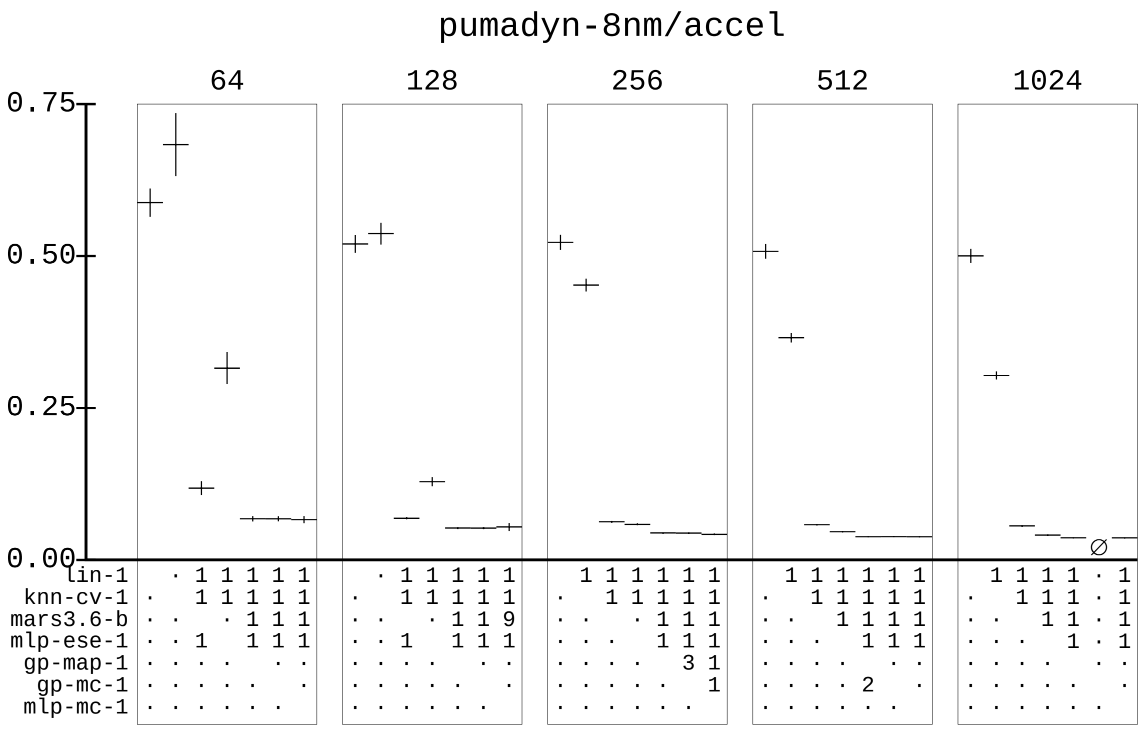


Figure 5.10: Experimental performance comparisons using squared error loss function for the `pumadyn` tasks with 8 inputs, fairly linear relationships, and medium and high noise-levels.



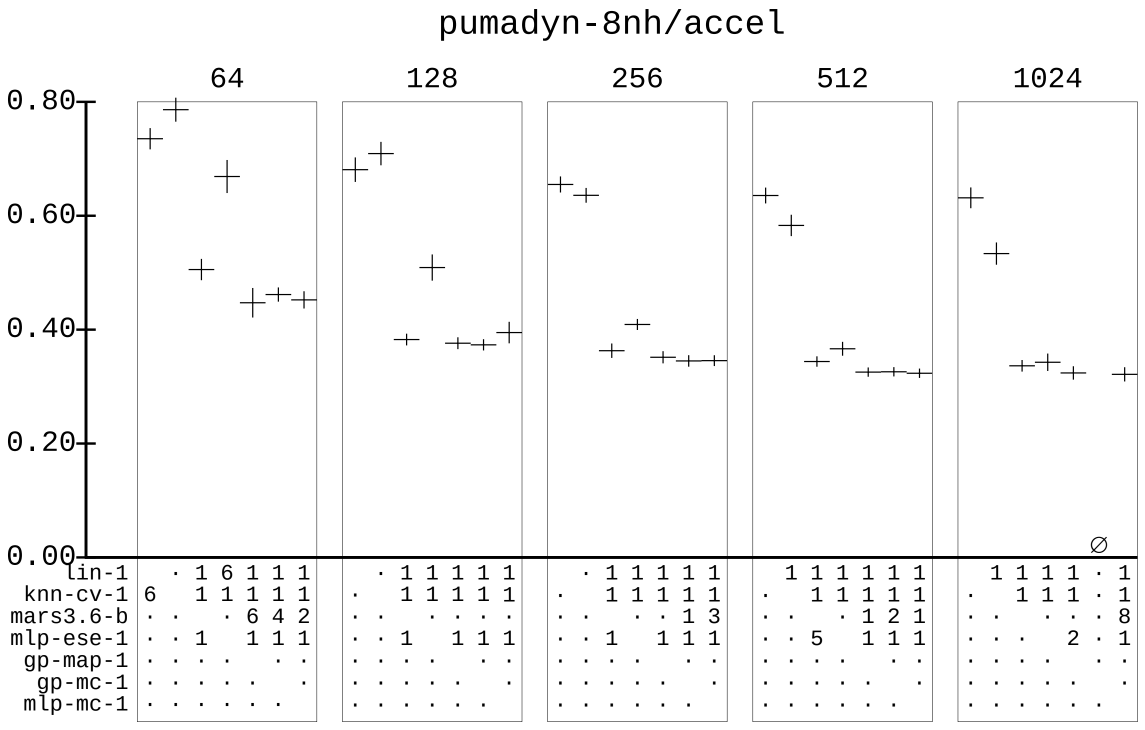
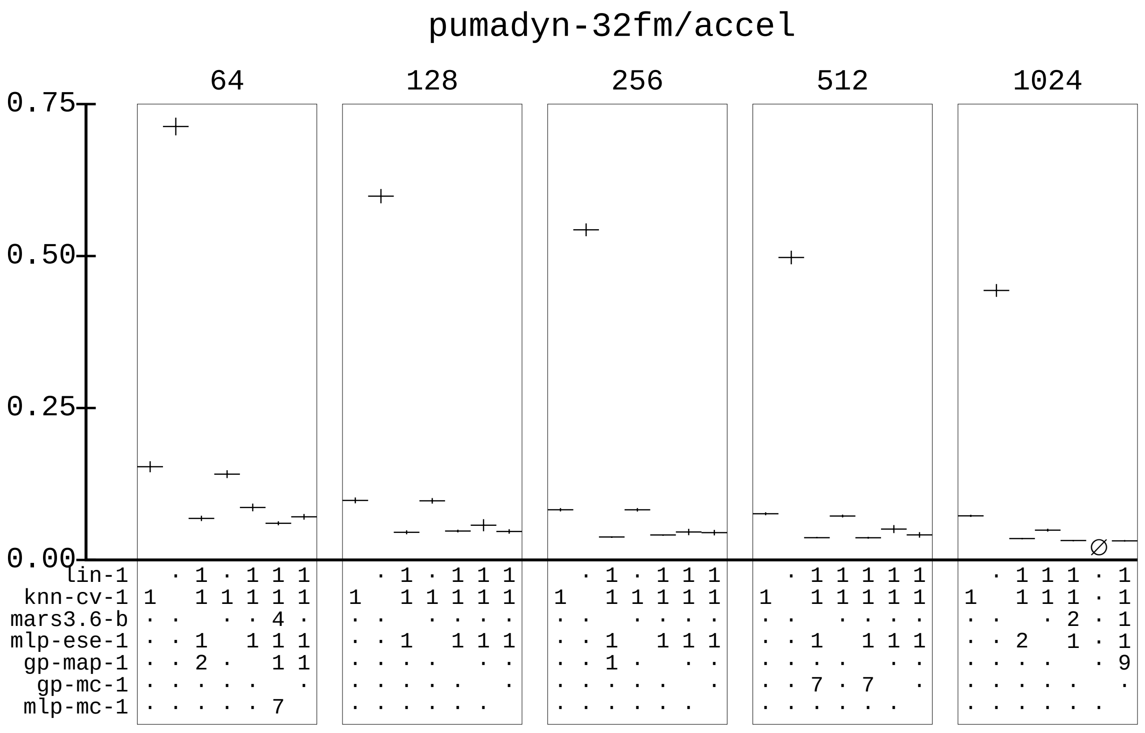


Figure 5.11: Experimental performance comparisons using squared error loss function for the pumadyn tasks with 8 inputs, non-linear relationships, and medium and high noise-levels.



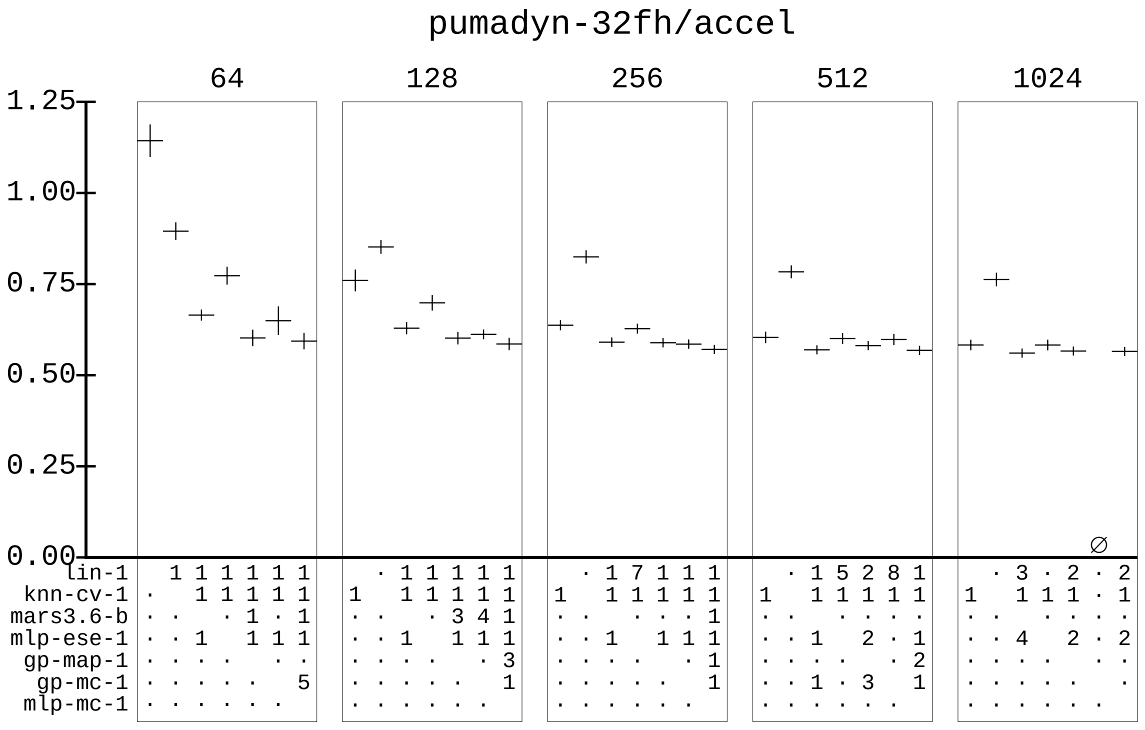
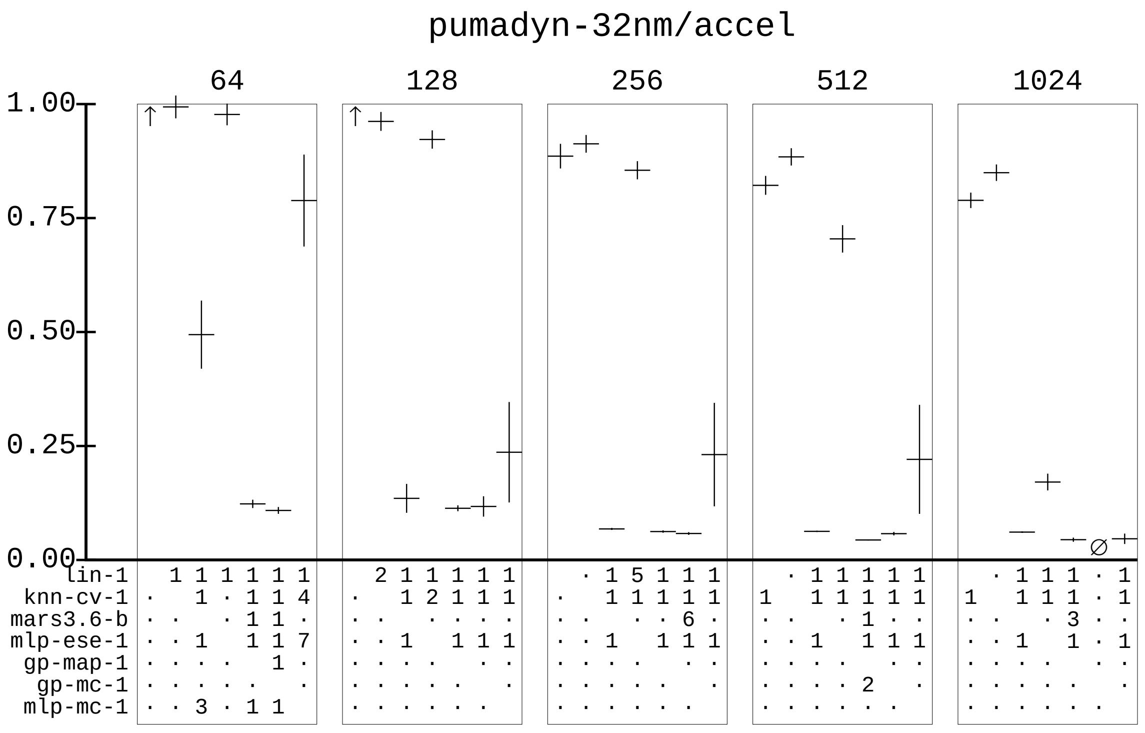


Figure 5.12: Experimental performance comparisons using squared error loss function for the `pumadyn` tasks with 32 inputs, fairly linear relationships, and medium and high noise-levels.



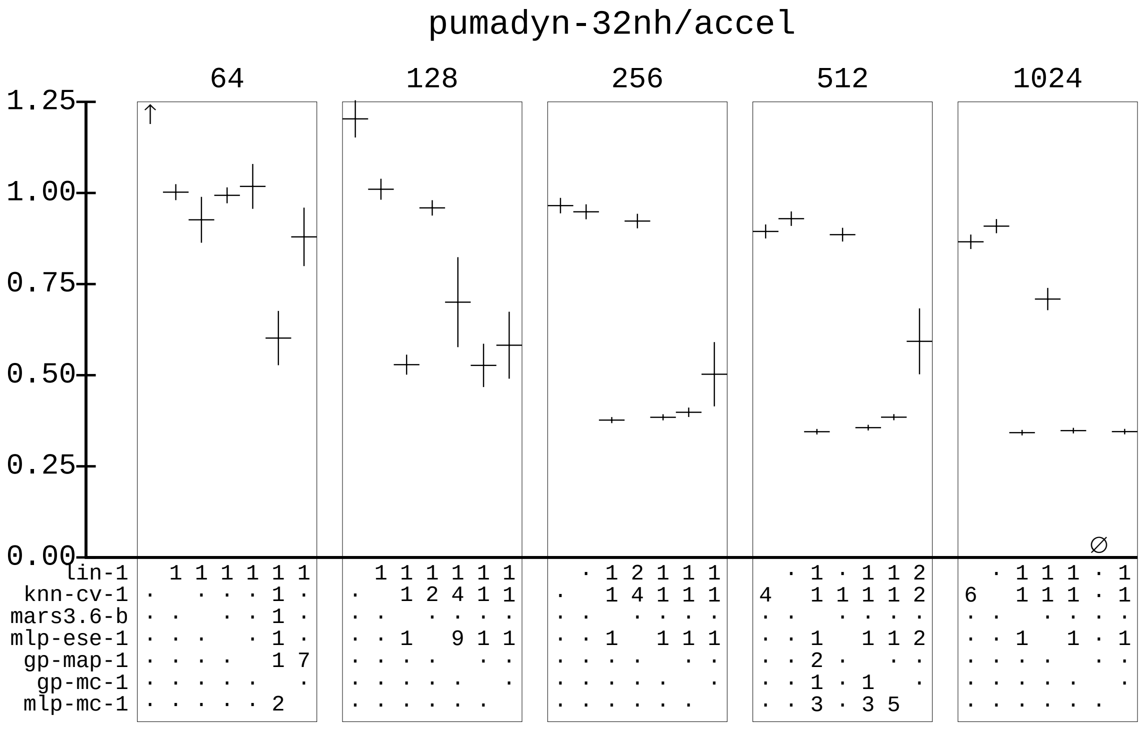


Figure 5.13: Experimental performance comparisons using squared error loss function for the pumadyn tasks with 32 inputs, non-linear relationships, and medium and high noise-levels.

doing rather poorly. The non-linear high-dimensional version of `kin` seems very hard to learn and the errors remain high until the very largest training sets, where the `gp-map-1` and `mlp-mc-1` methods are making some improvements.

The linear method usually does very well on the "fairly linear" tasks, although it only beats the sophisticated methods `gp-map-1`, `gp-mc-1` and `mlp-mc-1` on a single occasion (the `kin-32fh` task). Also, `lin-1` has pronounced difficulties for the smallest instances of the fairly linear high dimensional tasks. I suspect we are seeing effects of over-fitting — this might be alleviated by introducing regularisation in the linear method. As expected, `lin-1` also does badly on the non-linear tasks, often having a performance close to that of `knn-cv-1`.

The `knn-cv-1` mostly does a very poor job. This is most noticeable for the fairly linear tasks where the other methods perform well. This may well be the reason why nearest neighbor methods are rarely used for regression (they are almost exclusively used for classification tasks). However, a simple extension to the method by using local linear models in the neighborhood, such as LOESS [Cleveland 1979] may help.

A spectacular difference in performances is seen for `pumadyn-32nm` (and to a lesser degree in `pumadyn-32nh`), where the good methods perform very well with only small amounts of training data, whereas `lin-1` and `knn-cv-1` never seem to catch on. The `mlp-ese-1` method seems to be somewhat intermediate, improving when given 1024 training cases, although still short of the performance of the GP methods given only 1/16 of the training data. This enormous difference in performance coincides with which methods do some kind of variable selection. The Automatic Relevance Determination (ARD) scheme of the `gp-map-1`, `gp-mc-1` and `mlp-mc-1` is helping to focus on the relevant subset of the large number of inputs. Examination of the hyperparameters for the GP methods reveal that only 3–4 inputs are considered significant. The `mlp-ese-1` method does not have any form of variable selection, the result being that large amounts of training data are needed for this method. MARS is also capable of ignoring inputs and this method does very well on these two tasks.

In order to get a clear overview of the performance tendencies as a function of the characteristics of the datasets, I have produced plots of average performances in fig. 5.14 – 5.17. The data from the two families can be characterised by four binary characteristics: domain, input-dimensionality, linearity and noise-level. In these figures, I separate tasks according to one characteristic and average together performances over the other three characteristics. Thus, each figure has two graphs displaying averages over eight tasks for the two values of

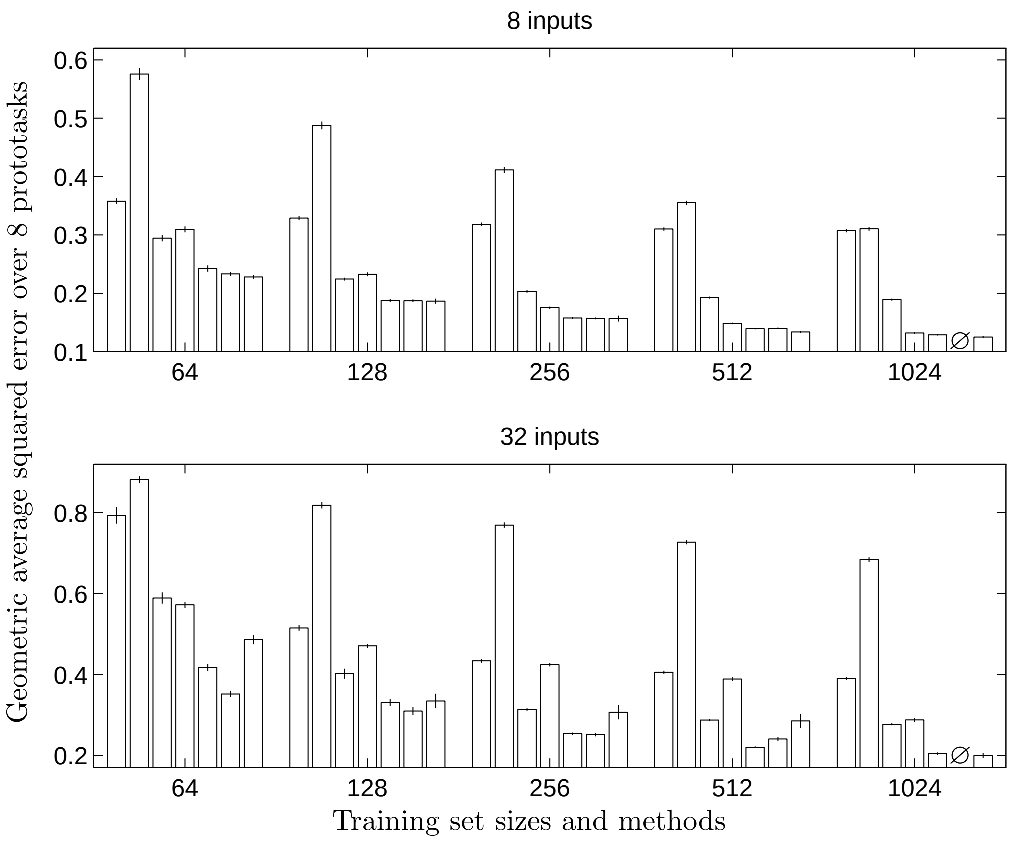


Figure 5.14: Geometric average performances conditional on number of inputs. Approximate one standard error error-bars are indicated by vertical line segments. The methods are grouped horizontally according to training set size. Within a group, the order is: `lin-1`, `knn-cv-1`, `mars3.6-bag-1`, `mlp-ese-1`, `gp-map-1`, `gp-mc-1` and `mlp-mc-1`.

the chosen characteristic. The methods are grouped into five groups, one for each training set size. Within the groups, the ordering of the methods is the same as in the earlier performance plots: `lin-1`, `knn-cv-1`, `mars3.6-bag-1` and in the middle `mlp-ese-1`; then follows `gp-map-1`, `gp-mc-1` and `mlp-mc-1`.

In these plots, the geometric average was thought more sensible than the usual mean, since the performances on different tasks differ substantially. The geometric average focuses attention on relative differences rather than absolute ones. When looking at these plots, one should not pay attention to the values of the performances, but rather focus on how the performance of a particular method compares to "the field" in the two plots. The plots also show approximate one standard error error-bars. The geometric average can be expressed as a usual average in the log domain, $\exp(\frac{1}{n}\sum \log y_i)$. Assuming the performance estimates for individual tasks, y_i, to be Gaussian, and that their standard deviations are much smaller

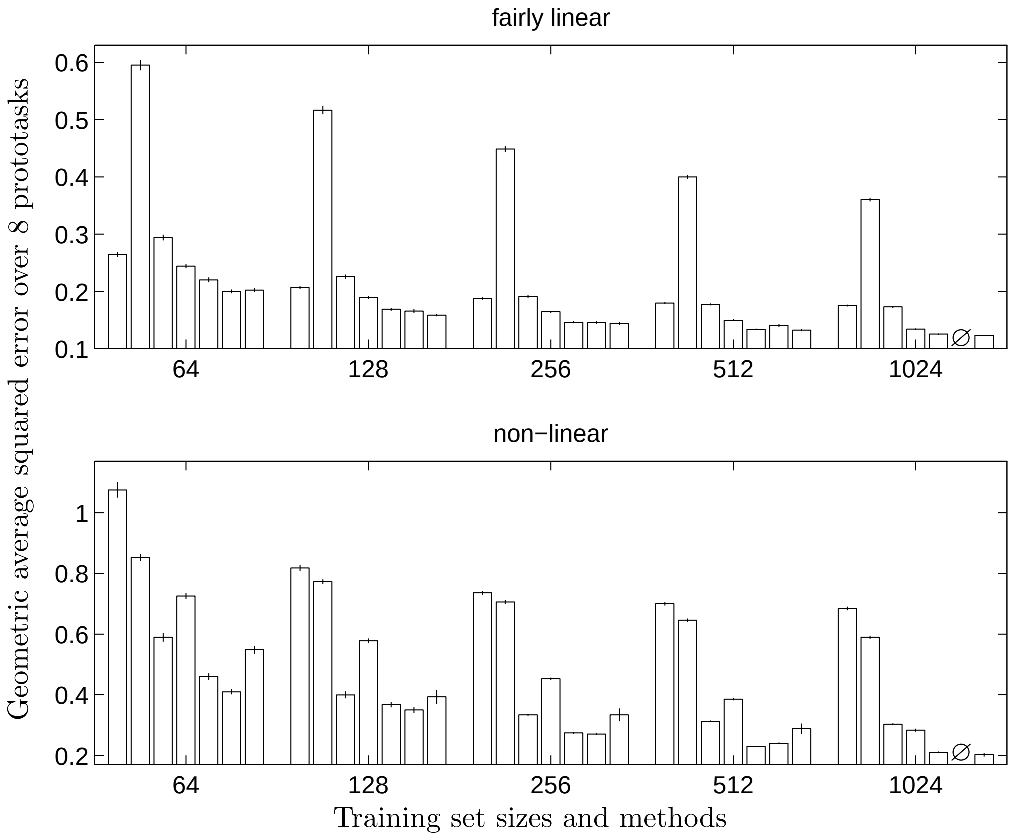


Figure 5.15: Geometric average performances conditional on degree of linearity.

than their means, then the distribution of $\log y_i$ is also approximately Gaussian. We can thus use standard tools for averaging the independent performance measures in the log domain and use the same approximations to transform back to the original domain.

In fig 5.14 we note that `mlp-ese-1` has difficulties for tasks with high-dimensional input. It also appears that `mlp-mc-1` has some difficulty for high dimensional inputs and small training sets. In fig. 5.15 we similarly see that `mlp-ese-1` does poorly on non-linear tasks. This figure also confirms our expectations that `lin-1` does not do well on non-linear tasks.

The plot in fig. 5.16 is interesting, because it may give an indication of how different the two data families are. Results seem fairly similar, with two exceptions: `mars3.6-bag-1` does badly on the `kin` data and `mlp-ese-1` does badly on `pumadyn`. Naturally, the extension of DELVE to incorporate more data families is essential, such that these tendencies which will bias our conclusions with respect to other effects can be reduced.

Finally, fig. 5.17 shows the difference between tasks with medium and high noise. The

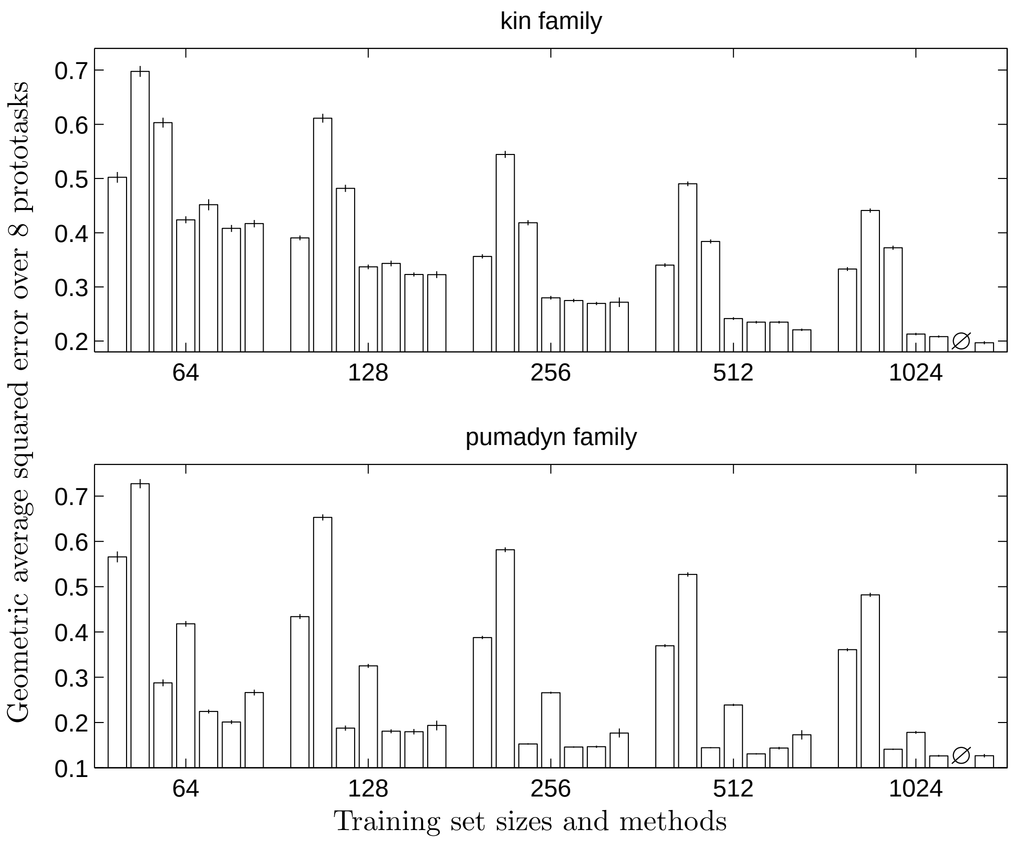


Figure 5.16: Geometric average performances conditional on data family.

strongest effect here is that `knn-cv-1` seems to do much worse in cases where there is smaller noise — I expect however, that in fact we are seeing the other methods doing comparatively better in the case of less noise.

It is a fairly surprising finding in fig. 5.14 and 5.15 that the conventional neural network method `mlp-ese-1` seems less desirable for non-linear tasks and for high-dimensional tasks than e.g. `mars3.6-bag-1`. The prevailing folklore about neural networks has for a long time been that neural networks are especially suitable for high-dimensional tasks and non-linear tasks. My experiments do not support this view. However, it should be noted that the performance difference between `mlp-ese-1` and `mars3.6-bag-1` also correlates highly with the identity of the data-family in fig. 5.16. This forces us to interpret the results with caution — tests on more data-families are needed in order to enable more firm conclusions.

The cpu time consumed by the different methods are given in the table in fig. 5.18. The time needed for the `lin-1`, `knn-cv-1` and `mars3.6-bag-1` methods is mainly determined by the

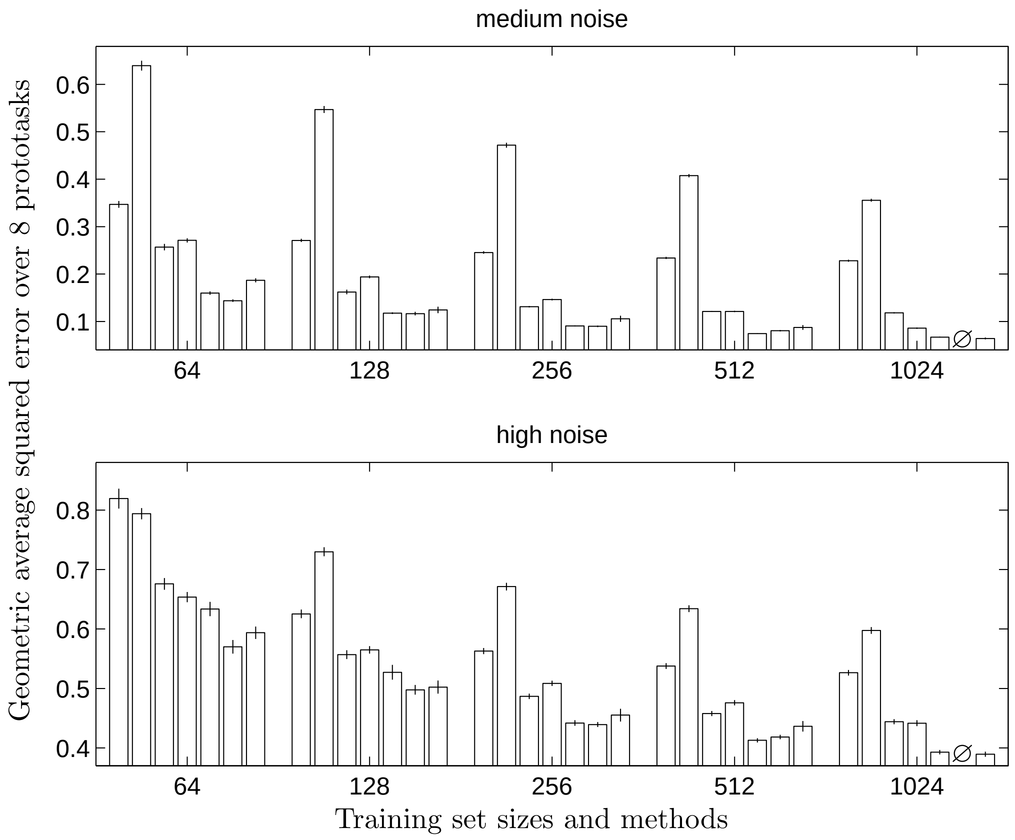


Figure 5.17: Geometric average performances conditional on noise level.

size of training sets and depends only slightly on the input dimensionality. The remaining four methods all have an iterative nature, and a limit on computation time must be given. For the `mlp-ese-1` method I used a linear scaling of time with the number of training cases — allowing 1.875 seconds per training case. This linear scaling is chosen as a compromise between the assumed requirements and practical limitations on compute time. Probably the computational requirements grow roughly quadratically for `mlp-ese-1`; the time per iteration scales quadratically, since the number of training cases and the number of network weights grow. It is more difficult to guess the scaling behaviour of the required number of iterations until early stopping sets in — possibly an increasing number of iterations is required for larger training sets. For the times given here, the method always had enough time (even for the largest tasks) to train at least 25 members for the predictive ensemble. Consequently, it seems reasonable to suppose that we have reached the point of diminishing returns in terms of ensemble size. For the smaller tasks, the linear scaling ensures that the method has even more plentiful time.

	64	128	256	512	1024
`lin-1`	< 0.01	< 0.01	< 0.01	0.01	0.02
`knn-cv-1`	< 0.01	0.04	0.15	1	6
`mars3.6-bag-1`	0.2	0.3	0.7	2	4
`mlp-ese-1`	2	4	8	16	32
`gp-map-1`	0.25	2	16	128	512
`gp-mc-1`	32	64	128	256	-
`mlp-mc-1`	32	64	128	256	512

Figure 5.18: Approximate total (training and prediction) cpu time in minutes for combinations of learning methods and training set sizes that were used for the experiments on the `kin` and `pumadyn` data families.

For the methods which require large amounts of computation time, it is of interest to investigate how the trade-off between time and accuracy behaves for the different methods as a function of training set sizes. These results are presented in fig. 5.19 for the four most computationally demanding methods. The approximate error-bars are computed using the same approximation as in fig. 5.14 – 5.17.

For any particular method, performance gets better with more training time. However, for a fixed time (proportional to the number of training cases) it is occasionally better to use fewer training cases; for example the `gp-mc-1` method for short run-times does worse with 512 cases than with 256 cases. This is due to the super-linear time requirement of the method, and the resulting lack of convergence for the large instances. For the longer runs, the larger training sets again become advantageous. However, `gp-mc-1` was not applied to the largest instances, since it was fairly obvious that it would fail given the time constraints considered here.

Note that the results for the `mlp-mc-1` and `gp-mc-1` methods reported earlier in this section were all for 30 seconds per training case. For `mlp-ese-1` only 1.875 seconds per training case were allowed, but this is probably adequate to get most of the benefit from this method. The time for the `gp-map-1` method is dictated by the cubic time dependency of the training set size, and is very limited for small training sets.

It is interesting to compare the times used by `mlp-mc-1` with the times used for this method by its inventor [Neal 1996]. In that study the run-times were selected on the basis of plots of various quantities showing the possibility of equilibrium. The main simulations involved a 10-way cross-test using the “Boston Housing” data, and used about 210 seconds per

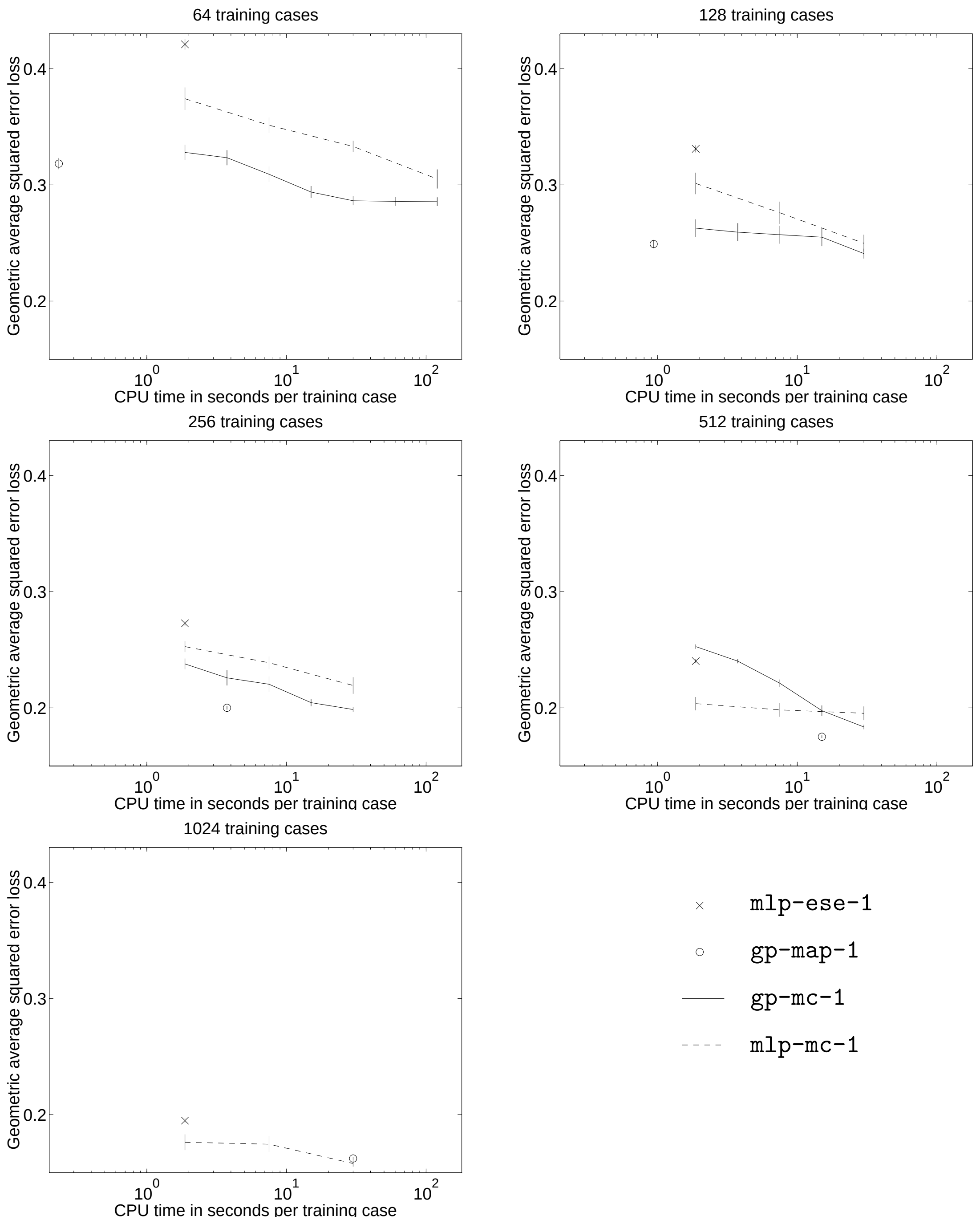


Figure 5.19: The geometric average squared test errors over the 16 prototasks in the `kin` and `pumadyn` data families for different training set sizes as a function of the training time for four computation intensive methods. The vertical line segments indicate approximate one standard error error-bars. The `mlp-ese-1` and `gp-map-1` methods have only been run for a single amount of time; for the `mlp-mc-1` and `gp-mc-1` methods predictions were made after several intervals. The `gp-mc-1` method has not been run for the largest training set size. All the plots are to the same scale. Note the log time-scale.

training case for training sets of about 456 cases (on a marginally slower machine). The current results use one or two orders of magnitude less computation time, but still show that `mlp-mc-1` is better than `mlp-ese-1` even for very short times. This is a strong result; if you have time enough to train `mlp-ese-1`, you would be better off using `mlp-mc-1` even though this method would not yet have converged. Obviously, if even shorter times are required, other methods are more appropriate.

For `mlp-mc-1` there seems to be a tendency for the smaller instances to benefit more from longer runs than the larger instances, which may appear surprising, since one may reasonably assume that more than linear time is required for good results. There may be a variety of reasons for this behaviour. Firstly, this observation may be an artifact of the limited time scale, and the performance for the larger instances may improve for longer runs than were allowed in the present study. Secondly, the bad results for the short runs for the small datasets may be caused by failure to reach areas of high posterior probability; for small training sets the likelihood is fairly weak, and the Markov chain may spend a lot of time sampling from the vast areas under the vague prior before discovering the peaks in the posterior. An indication of this behaviour can be found in fig. 5.13 where poor performance and large error bars for all but the largest instances probably indicate a failure on single instances. For large training sets the likelihood becomes strong enough to avoid this problem. A third reason may be that the posterior for the hyperparameters becomes broad or multi-modal for small amounts of data, which can severely affect the time required to get a good sample from the posterior.

The runs of `gp-mc-1` and `mlp-mc-1` for the smallest training set sizes were extended a factor of 4 further than for the other sizes. Interestingly, the `gp-mc-1` method seems to have converged at about 30 seconds per training case whereas `mlp-mc-1` improves further given more time. One may wonder if it is possible to achieve convergence in `gp-mc-1` for even shorter times. At 30 seconds per training case, the Markov chain can perform roughly 20000 iterations for the smallest training sets. Given this large number of iterations, one may suspect the "persistence" of 20 iterations to be much too low. Perhaps a scheme of gradually increasing the persistence throughout the run may further decrease the required run-time. If the long convergence times are due to high correlations in the posterior an increase in the persistence should certainly help; if on the other hand the problems are caused by the Markov chain only infrequently switching between multiple modes in the posterior, such simple measures may not help.

From a theoretical point of view, one may conjecture that `gp-mc-1` and `mlp-mc-1` would perform nearly equally well in the limit of large amounts of computation time, since they both use (at least approximately) Gaussian process priors and integrate out all the nuisance parameters. There are of course differences in the form of the covariance matrix and the exact shape of the hyper-priors, but these differences may not be of great practical importance. This conjecture cannot be proven by the present result, since it is not clear that we have reached convergence in most cases. However, all the experimental evidence is consistent with the conjecture. If we agree with this conjecture, then choosing between the `gp-mc-1` and `mlp-mc-1` methods can be done solely on the basis of which method has the most efficient implementation. For small datasets, the `gp-mc-1` implementation seems more efficient; above roughly 500 training examples, the standard implementation of `gp-mc-1` becomes hard to manage.

The simple `gp-map-1` method seems to have very desirable properties. For the large instances it performs very well using reasonable amounts of computation time. For the smallest instances it is very fast, but can be beaten by sampling methods that are given 2 or 3 orders of magnitude more computation time.

What are the overall conclusions from this performance study? Naturally, the trade-off between accuracy and computational requirements depends on details of the individual application. However, the current experiments allow for very strong conclusions. Within the characteristics of data explored in these experiments, the methods implementing Gaussian processes do very well regardless of the characteristics of the data; the choice of method is determined primarily by the training set size. If the training set is very small it may be worthwhile trying `gp-mc-1` otherwise `gp-map-1` is nearly as good and much faster. If your dataset is large (around 1000 cases or more) you may be better off using `mlp-mc-1`, although more efficient implementations of GP methods may become available.

Chapter 6

Conclusions

This thesis has provided a rigorous statistical framework for analysis of empirical performance evaluations. In the neural network and statistics communities, empirical evaluations of methods play a key role. It is therefore essential to base these experiments on solid statistical theory. Hitherto, the field has been swamped by proposals for novel learning methods without convincing empirical evidence. The future success of the field depends heavily on improving the standards of empirical tests.

Two experimental designs were developed. The “2-way” design is suitable for datasets with a limited number of cases, since only a single test set is needed. The “hierarchical” design is simpler and enables the use of exact tests for method comparisons. This design requires multiple test sets, and is therefore most suitable for simulated data where large numbers of test cases may be generated.

Unfortunately, it is fairly cumbersome to make good comparisons. The development of the publicly available DELVE software environment should ease this burden. DELVE includes results for previously tested methods and should provide a good basis for testing new methods. The performance tests using DELVE are still somewhat laborious, since several training instances must be tried before firm conclusions can be drawn. This is an inherent property of the task. It does pose problems for testing compute-intensive methods. In this thesis I have allowed a maximum of 8 hours of computation time per task instance, which by many standards would be regarded as fairly modest. The total run-time for the experiments presented in this thesis is however fairly large at about 6 months.

Two methods relying on direct manipulation of Gaussian process priors over functions have been defined. Although several authors have previously touched upon such ideas, they have remained largely unknown. This is puzzling given their highly desirable properties. The `gp-map-1` method relies on simple optimisation and is very fast for small datasets. Methods for small datasets have numerous applications in statistics. The direct applicability of Automatic Relevance Determination (ARD) allows for easy interpretation of the trained models. The more complicated `gp-mc-1` implementation is very attractive from the Bayesian point of view, since we are able to integrate out all the parameters of the model. For small datasets this can be achieved in reasonable amounts of time.

Perhaps the reason for the neglect of these methods is that people have failed to notice that modern computers can easily invert large matrices. Or maybe inherent skepticism towards Bayesian methodology has played a role. I hope that thorough experimental evidence will succeed in bringing these methods into wide use.

The experimental results on the `boston` dataset show very large error-bars on the performance estimates. It is argued that it may be very difficult to design more sensitive experiments on datasets of moderate size. If this trend carries to other datasets, this implies that great caution should be exercised when interpreting much of the experimental evidence given in the literature. It also presents us with the problem that many real datasets that have been used for benchmarking may not really be suitable.

The solution to this problem may be to use data from realistic simulators for benchmarking purposes. These datasets can be made adequately large, and as a further benefit, datasets with controlled characteristics may be produced, enabling determination of how the strengths and weaknesses of methods depend on the characteristics of the data. Such experiments have failed to confirm the widely held beliefs in the community that traditional neural network methods such as `mlp-ese-1` are especially well-suited for tasks with a high degree of non-linearity and tasks with high input dimensionality. Additional experiments on other data-families would be helpful to corroborate these findings.

I also found that although a Bayesian treatment of neural networks `mlp-mc-1` may take a long time to converge, they perform better than the conventional method `mlp-ese-1` even when allowed fairly short times, i.e., run-time comparable to that of `mlp-ese-1` corresponding to 2 orders of magnitude less time than in the first published results for this method. Again, this contrasts with the commonly held belief that Bayesian methods are “too slow” to be of any practical importance. For many applications involving small amounts of data, such methods are definitely tractable.

It may be argued that my use of the single `mlp-ese-1` method as representing "conventional neural network methods" is too limited. It is certainly true that many other variants are in common use including training with weight-decay, networks with multiple hidden layers, networks with radial basis functions, networks trained using constructive or pruning-based approaches, etc. Evaluation of methods relying on these principles would obviously be of interest. Because of the existence of such a large number of algorithms and the lack of specification of heuristics, it is impossible to make a benchmarking study that can fend off attacks regarding the choice of methods and details of heuristics. Therefore it is essential that authors of methods themselves precisely define their methods and test them against other methods on publicly available datasets. DELVE has been designed to meet these requirements. In conclusion, researchers should not be required to test everybody else's methods. Researchers whose methods have been rigorously tested will be making constructive and convincing contributions.

From a theoretical point of view there may be reasons to expect the three methods `gp-map-1`, `gp-mc-1` and `mlp-mc-1` to perform fairly similarly, since they all (at least approximately) implement modeling with Gaussian process priors over functions. And indeed the performance of these three methods is often quite close, and consistently better than the other methods. The strength of these conclusions is somewhat limited by the use of only two data-families. However, it should be noted that the methods implementing GP's do well regardless of the input dimensionality, the degree of non-linearity, the noise-level and the identity of the data-family. Which method to choose depends mostly on the efficiency of the methods for that particular training set size. For small datasets `gp-mc-1` performs best, for intermediate sizes `gp-map-1` becomes more desirable because it is faster, and for large sets `mlp-mc-1` performs best.

The experimental results presented in this thesis allow for very strong conclusions about which methods are preferable. This is not typical for earlier benchmarking studies [Prechelt 1995; Michie et al. 1994; Guérin-Dugué et al. 1995]. For example, Prechelt [1995] is not able to draw general conclusions about the 6 methods he tried on 14 different problems. There may be several reasons for this. Probably two of the most important reasons is that I have tested 7 quite different methods, and that I have used large datasets.

The biggest weakness in the experimental study is the small number of data families. In the near future, DELVE will be extended with many more datasets, thereby allowing more confidence in the experimental conclusions.

The results of the empirical tests performed in this thesis are available through DELVE.

These methods and results represent an important source for people interested in the performance of learning methods. Hopefully, people will report their experimental results to DELVE, forming a valuable source of experimental evidence. Time will show whether Gaussian processes will be able to stand up to the competition.

Appendix A

Implementations

This appendix contains source listings of important parts of the implementations of each of the methods described in the thesis, except the MARS and `mlp-mc-1` methods. For the sake of brevity, less interesting details, such as reading example files etc., are not included but may be found at the DELVE web-site.

The source (in C) for the `mlp-mc-1` has been made available by Radford Neal and can be retrieved through his homepage at `http://www.cs.toronto.edu/~radford`.

A.1 The linear model `lin-1`

The source for a straight forward implementation of the `lin-1` method is given on the following pages. The usage of the program is

```
lin-1 instance-number
```

The program reads training examples from the file `train.`*n*, test inputs from `test.`*n* and test targets from `targets.`*n*, where n is the instance number. Point predictions are written to `cguess.`*n* files. If there are an adequate number of training cases (if the number of training cases exceeds the input dimension) then losses according to the negative log density loss function are written to `cldens.L.`*n*.

```
/* lin-1.c: Robust linear method for regression.
 *
 * Reads training examples from "train.n", test inputs from "test.n" and
 * targets from "targets.n".  Produces point predictions in "cguess.n" and
 * densities of targets under a predictive distribution in "cldens.L.n". Here
 * "n" is the instance number, supplied as a command argument.  Handles badly
 * conditioned cases where inputs are (close to) linearly dependent.
 *
 * (c) Copyright 1996 by Carl Edward Rasmussen. */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "util.h"

#define MAX_SV_RATIO 1.0e6       /* Maximum allowed ratio of singular values */
#define two_pi 6.28318530717959

extern void svd(real **A, real *S2, int n);   /* sigular value decomposition */
static real f(real *x, real *w, int length);    /* linear function with bias */

main(int argc, char **argv)
{
  int  i, j, k, no_inp, no_tar, num_fit = 0;
  char df[10], df2[10];
  real **A, **w, **b, *S2, *c, tmp, *sigma2, sig2, mu;
  FILE *fp;
  struct exampleset train, test;

  if (argc != 2) {
    fprintf(stderr, "Usage: %s instance-number\n", argv[0]); exit(-1);
  }

  train.num = test.num = no_inp = no_tar = -1;      /* default for "unknown" */
  sprintf(df, "test.%s", argv[1]);                    /* name of test inputs */
  sprintf(df2, "targets.%s", argv[1]);               /* name of test targets */
  loadExamples(&test, &no_inp, &no_tar, df, df2);
  sprintf(df, "train.%s", argv[1]);                 /* name of training file */
  loadExamples(&train, &no_inp, &no_tar, df, NULL);

  A  = createMatrix(2*(no_inp+1), no_inp+1);   /* double size for svd() call */
  b  = createMatrix(no_tar, no_inp+1);
  w  = createMatrix(no_tar, no_inp+1);
  S2 = (real *) malloc((size_t) (no_inp+1)*sizeof(real));
  c  = (real *) malloc((size_t) (no_inp+1)*sizeof(real));
  sigma2 = (real *) malloc((size_t) no_tar*sizeof(real));

/* Construct the A matrix; since A is known to be symmetric, we only compute
 * the upper triangular matrix elements and place them symmetrically. Don't
 * forget the bias inputs. */

  for (i=0; i<no_inp; i++) {
    for (j=i; j<no_inp; j++) {
      for (tmp=0.0, k=0; k<train.num; k++)
        tmp += train.inp[k][i]*train.inp[k][j];
      A[i][j] = A[j][i] = tmp;
    }
    for (tmp=0.0, k=0; k<train.num; k++)    /* contribution from bias inputs */
      tmp += train.inp[k][i];
    A[i][j] = A[j][i] = tmp;
  }
  A[i][j] = train.num;                                      /* corner element */

/* Construct b matrix. If there is only a single target, then b is a vector,
 * but is implemented as a matrix with a single row. */

  for (j=0; j<no_tar; j++) {
    for (i=0; i<no_inp; i++) {
      for (tmp=0.0, k=0; k<train.num; k++)
        tmp += train.inp[k][i]*train.tar[k][j];
      b[j][i] = tmp;
    }
    for (tmp=0.0, k=0; k<train.num; k++)    /* contribution from bias inputs */
      tmp += train.tar[k][j];
    b[j][i] = tmp;
  }

/* Do singular value decomposition of A = USV'; on return, the first no_inp+1
 * rows of A contain the product US and the remaining rows contain V (not V');
 * S2 contains the square of the singular values ordered with the largest
 * first. We "invert" S2, zeroing when S2[i] < sqrt(S[0])/MAX_SV_RATIO; Then we
 * compute invA = V*invS2*(US)' one row at a time and store them in the lower
 * half of A. Lastly, compute w = invA*b.' */

  svd(A, S2, no_inp+1);
  for (i=no_inp; i>=0; i--)                                    /* "invert" S2 */
    if (S2[i]*sq(MAX_SV_RATIO) > S2[0]) {                 /* SV large enough? */
      num_fit++;
      S2[i] = 1.0/S2[i];
    }
    else
      S2[i] = 0.0;                                        /* delete direction */

  for (i=0; i<=no_inp; i++) {                /* compute invA = V*invS2*(US)' */
    for (j=0; j<=no_inp; j++) {
      for (tmp=0.0, k=0; k<=no_inp; k++)
        tmp += A[i+no_inp+1][k]*A[j][k]*S2[k];
      c[j] = tmp;
    }
    for (j=0; j<=no_inp; j++) A[i+no_inp+1][j] = c[j];  /* copy "c" into "A" */
  }

  for (k=0; k<no_tar; k++)                             /* compute w = invA*b */
    for (i=0; i<=no_inp; i++) {
      for (tmp=0.0, j=0; j<=no_inp; j++)
        tmp += A[i+no_inp+1][j]*b[k][j];
      w[k][i] = tmp;
    }

/* Produce point predictions for the test cases and write them to the "cguess"
 * file; one line per example and no_tar predictions per line. */

  fp = openPredFile("cguess.%s", argv[1]);     /* file for point predictions */
  for (k=0; k<test.num; k++) {        /* make predictions for all test cases */
    for (j=0; j<no_tar; j++)                               /* for each output */
      fprintf(fp, "%f ", f(test.inp[k], w[j], no_inp));
    fprintf(fp,"\n");
  }
  fclose(fp);

/* If train.num <= no_inp+1 we cannot produce a reasonable predictive
 * distribution; otherwise, the log density of the targets under the Gaussian
 * predictive distribution are are written to "cldens" files. */

  if (train.num <= no_inp+1)            /* are there too few training cases? */
    fprintf(stderr, "Warning: No \"cldens.L\" files produced",
                    " - too few training examples...\n");
  else {
    fp = openPredFile("cldens.L.%s", argv[1]);
    for (j=0; j<no_tar; j++) {        /* estimate noise level for each target */
      for (tmp=0.0, k=0; k<train.num; k++)
        tmp += sq(f(train.inp[k], w[j], no_inp)-train.tar[k][j]);
      sigma2[j] = tmp/(train.num-num_fit);
    }

    for (k=0; k<test.num; k++) {       /* make predictions for all test cases */
      for (i=0; i<=no_inp; i++)
        c[i] = f(test.inp[k], A[i+no_inp+1], no_inp);
      sig2 = f(test.inp[k], c, no_inp);        /* noise from uncertainty in w */
      for (tmp=0.0, i=0; i<no_tar; i++)
        tmp -= log(two_pi*(sig2+sigma2[i]))+sq(f(test.inp[k], w[i], no_inp)-
                                          test.tar[k][i])/(sig2+sigma2[i]);
      fprintf(fp, "%f\n", 0.5*tmp);
    }
    fclose(fp);
  }
  free(A[0]); free(A); free(w[0]); free(w); free(b[0]); free(b);
  free(S2); free(c); free(sigma2);
}

static real f(real *x, real *w, int length)     /* linear function with bias */
{
  int  i;
  real tmp = 0.0;

  for (i=0; i<length; i++) tmp += x[i]*w[i];
  return tmp+w[i];                                             /* add the bias */
}
```

```
/* svd.c: Perform a singular value decomposition A = USV' of square matrix.
 *
 * This routine has been adapted with permission from a Pascal implementation
 * (c) 1988 J. C. Nash, "Compact numerical methods for computers", Hilger 1990.
 * The A matrix must be pre-allocated with 2n rows and n columns. On calling
 * the matrix to be decomposed is contained in the first n rows of A. On return
 * the n first rows of A contain the product US and the lower n rows contain V
 * (not V'). The S2 vector returns the square of the singular values.
 *
 * (c) Copyright 1996 by Carl Edward Rasmussen. */

#include <stdio.h>
#include <math.h>
#include "util.h"

void svd(real **A, real *S2, int n)
{
  int  i, j, k, EstColRank = n, RotCount = n, SweepCount = 0,
       slimit = (n<120) ? 30 : n/4;
  real eps = 1e-15, e2 = 10.0*n*eps*eps, tol = 0.1*eps, vt, p, x0,
       y0, q, r, c0, s0, c2, d1, d2;

  for (i=0; i<n; i++) { for (j=0; j<n; j++) A[n+i][j] = 0.0; A[n+i][i] = 1.0; }
  while (RotCount != 0 && SweepCount++ <= slimit) {
    RotCount = EstColRank*(EstColRank-1)/2;
    for (j=0; j<EstColRank-1; j++)
      for (k=j+1; k<EstColRank; k++) {
        p = q = r = 0.0;
        for (i=0; i<n; i++) {
          x0 = A[i][j]; y0 = A[i][k];
          p += x0*y0; q += x0*x0; r += y0*y0;
        }
        S2[j] = q; S2[k] = r;
        if (q >= r) {
          if (q<=e2*S2[0] || fabs(p)<=tol*q)
            RotCount--;
          else {
            p /= q; r = 1.0-r/q; vt = sqrt(4.0*p*p+r*r);
            c0 = sqrt(0.5*(1.0+r/vt)); s0 = p/(vt*c0);
            for (i=0; i<2*n; i++) {
              d1 = A[i][j]; d2 = A[i][k];
              A[i][j] = d1*c0+d2*s0; A[i][k] = -d1*s0+d2*c0;
            }
          }
        } else {
          p /= r; q = q/r-1.0; vt = sqrt(4.0*p*p+q*q);
          s0 = sqrt(0.5*(1.0-q/vt));
          if (p<0.0) s0 = -s0;
          c0 = p/(vt*s0);
          for (i=0; i<2*n; i++) {
            d1 = A[i][j]; d2 = A[i][k];
            A[i][j] = d1*c0+d2*s0; A[i][k] = -d1*s0+d2*c0;
          }
        }
      }
    while (EstColRank>2 && S2[EstColRank-1]<=S2[0]*tol+tol*tol) EstColRank--;
  }
  if (SweepCount > slimit)
    fprintf(stderr,
          "Warning: Reached maximum number of sweeps (%d) in SVD routine...\n",
           slimit);
}
```

A.2 k nearest neighbors for regression `knn-cv-1`

The source for the `knn-cv-1` method is included in the following pages. The usage is

```
knn-cv-1 instance-number
```

The program reads training examples from the file `train.`*n*, test inputs from `test.`*n* and test targets from `targets.`*n*, where n is the instance number. The outer loop of the program does the "leave one out" for each training cases in turn. Inside the loop the remaining cases are sorted according to distance from the input of the left out case, and the losses for the three standard loss functions (squared error loss, absolute error loss and negative log density loss) are accumulated for each value of k (the neighborhood size).

The optimal value of k for each loss is echoed to `stderr` and predictions are made using these neighborhood sizes. The predictions are written to files named `cguess.A.`*n*, `cguess.S.`*n* and `cldens.L.`*n* according to the loss type.

```
/* knn-cv-1.c - k nearest neighbors for regression.
 *
 * Reads training cases from "train.n", test inputs from "test.n" and test
 * targets from "targets.n". Writes point predictions to "cguess.A.n",
 * "cguess.S.n" and densities under a predictive distribution to
 * "cldens.L.n". Here "n" is the instance number supplied as a command
 * argument. For each loss type, "k" is selected by leave one out cross
 * validation.
 *
 * Copyright (c) 1996 by Carl Edward Rasmussen. */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <values.h>
#include "util.h"

#define tolerance 1.0e-6
#define two_pi 6.28318530717959
#define SWAP(a, b) temp = (a); (a) = (b); (b) = temp;

extern real median(real *a, int k);  /* find median of first k elements in a */
static int  comp(const void *x, const void *y);    /* function used by qsort */
static void sort_dist(int k, real *inp);      /* sort k examples by distance */
static void find_neighbors(int k);            /* find targets of k neighbors */
static void var_est_1nn(real *glob_var);       /* estimate variance from 1nn */

int no_inp, no_tar;
real **targets,    /* matrix containing the targets of the nearest neighbors */
     **loo_est,       /* estimates for 3 loss functions for every value of k */
     **dist_tar;              /* array of distances and targets of neighbors */
struct exampleset train, test;

main(int argc, char **argv)
{
  int  i, j, k, l, top, k0, k1, k2;
  char df2[10], df[10];                     /* strings containing file names */
  real *m, *v, *glob_var, tmp, *temp;
  FILE *fp0, *fp1, *fp2;                     /* pointers to prediction files */

  if (argc != 2) {
    fprintf(stderr, "Usage: %s instance-number\n", argv[0]); exit(-1);
  }

  train.num = test.num = no_inp = no_tar = -1;      /* default for "unknown" */
  sprintf(df, "test.%s", argv[1]);                       /* name of test file */
  sprintf(df2, "targets.%s", argv[1]);                /* name of targets file */
  loadExamples(&test, &no_inp, &no_tar, df, df2);
  sprintf(df, "train.%s", argv[1]);                  /* name of training file */
  loadExamples(&train, &no_inp, &no_tar, df, NULL);
  top = train.num-1;

  dist_tar = createMatrix(train.num+2,1+no_tar);
  dist_tar++; dist_tar[-1][0] = -MAXFLOAT;  /* place a sentinel before array */
  dist_tar[top][0] = MAXFLOAT;  /* and two after; useful in find_neighbors() */
  dist_tar[train.num][0] = MAXFLOAT;  /* avoiding check for array boundaries */
                                                      /* when looking for ties */
  loo_est = createMatrix(train.num, 3);
  for (k=0; k<3; k++) for (i=0; i<train.num; i++) loo_est[i][k] = 0.0;
  targets = createMatrix(train.num, no_tar);
  m = (real *) malloc((size_t) no_tar*sizeof(real));
  v = (real *) malloc((size_t) no_tar*sizeof(real));
  glob_var = (real *) malloc((size_t) no_tar*sizeof(real));

/* Do "leave one out" by swapping cases with the last example, and doing knn on
 * the first "top" cases, for each value of k. Accumulate the "leave one out"
 * estimates for each loss type in the columns of "loo_est". */

  var_est_1nn(glob_var);                   /* estimate variance based on 1nn */
  for (i=top; i>=0; i--) {                              /* leave out example i */
    SWAP(train.inp[i], train.inp[top]); SWAP(train.tar[i], train.tar[top]);
    sort_dist(top, train.inp[top]);
    for (k=1; k<train.num; k++) {      /* now do "leave one out" for every k */
      find_neighbors(k);
      for (l=0; l<no_tar; l++) {   /* find mean and variance for each target */
        for (m[l]=0.0, v[l]=glob_var[l], j=0; j<k; j++)
          { m[l] += targets[l][j]; v[l] += sq(targets[l][j]); }
        m[l] /= k; v[l] = v[l]/k-sq(m[l]); m[l] = sq(m[l]-train.tar[top][l]);
      }
      for (tmp=0.0, l=0; l<no_tar; l++) tmp += m[l];
      loo_est[k][0] += tmp;
      for (tmp=0.0, l=0; l<no_tar; l++)
        tmp += fabs(median(targets[l], k)-train.tar[top][l]);
      loo_est[k][1] += tmp;
      for (tmp=0.0, l=0; l<no_tar; l++) tmp += log(two_pi*v[l])+m[l]/v[l];
      loo_est[k][2] += tmp;
    }
  }
  for (k0=k1=k2=i=1; i<train.num; i++) {       /* find k’s with minimum loss */
    if (loo_est[i][0] < loo_est[k0][0]) k0 = i;
    if (loo_est[i][1] < loo_est[k1][1]) k1 = i;
    if (loo_est[i][2] < loo_est[k2][2]) k2 = i;
  }
  printf("Loss types and loo k-values: S: %d, A: %d and L: %d\n", k0, k1, k2);

/* Use the estimated k values to make predictions, and write them to the
 * apropriate files. */

  fp0 = openPredFile("cguess.S.%s", argv[1]);
  fp1 = openPredFile("cguess.A.%s", argv[1]);
  fp2 = openPredFile("cldens.L.%s", argv[1]);

  for (i=0; i<test.num; i++) {        /* make predictions for all test cases */
    sort_dist(train.num, test.inp[i]);
    find_neighbors(k0);                      /* first for squared error loss */
    for (l=0; l<no_tar; l++) {
      for (tmp=0.0, j=0; j<k0; j++) tmp += targets[l][j];
      fprintf(fp0, "%f ", tmp/k0);
    }
    fprintf(fp0, "\n");
    find_neighbors(k1);                          /* then absolute error loss */
    for (l=0; l<no_tar; l++)
      fprintf(fp1, "%f ", median(targets[l], k1));
    fprintf(fp1, "\n");
    find_neighbors(k2);              /* and lastly negative log density loss */
    for (tmp=0.0, l=0; l<no_tar; l++) {
      for (m[l]=0.0, v[l]=glob_var[l], j=0; j<k2; j++)
        { m[l] += targets[l][j]; v[l] += sq(targets[l][j]); }
      m[l] /= k2; v[l] = v[l]/k2-sq(m[l]); m[l] = sq(m[l]-test.tar[i][l]);
      tmp += log(two_pi*v[l])+m[l]/v[l];
    }
    fprintf(fp2, "%f\n", -0.5*tmp);
  }

  fclose(fp0); fclose(fp1); fclose(fp2);
  free(dist_tar[-1]); free(--dist_tar); free(loo_est[0]); free(loo_est);
  free(targets[0]); free(targets); free(m); free(v); free(glob_var);
}
```

```
/* Fill in the dist_tar array and sort it by the first column. The first column
 * contains the sqared Euclidian distance in input space between the loo case
 * and the remaining training cases; the remaining "no_tar" columns contain the
 * targets for those cases. */

static void sort_dist(int k, real *inp)
{
  int  i, j;
  real dist;

  for (i=0; i<k; i++) {
    for (dist=0.0, j=0; j<no_inp; j++) dist += sq(train.inp[i][j]-inp[j]);
    dist_tar[i][0] = dist;
    for (j=0; j<no_tar; j++) dist_tar[i][j+1] = train.tar[i][j];
  }
  qsort(dist_tar, k, sizeof(real *), comp);
}

/* Find neighbors and write their targets into the "targets" array. Mostly
 * this can be done by copying from the sorted "dist_tar" array, but we need
 * to take care when there are ties; in this case we use the average of the
 * targets of the tied cases. */

static void find_neighbors(int k)
{
  static int  i, j, k1 = 2, k2 = 0;
  static real tmp;

  for (j=0; j<no_tar; j++) for (i=0; i<k; i++)               /* copy targets */
    targets[j][i] = dist_tar[i][j+1];
  while (dist_tar[k-1][0]-dist_tar[k-k1][0] < tolerance) k1++;
  while (dist_tar[k+k2][0]-dist_tar[k-1][0] < tolerance) k2++;
  if (k1+k2 > 2) {                 /* if there were any ties, then fix them */
    for (j=0; j<no_tar; j++) {
      for (tmp=0.0, i=k-k1+1; i<k+k2; i++) tmp += dist_tar[i][j+1];
      tmp /= k1+k2-2;
      for (i=k-k1+1; i<k; i++) targets[j][i] = tmp;
    }
    k1 = 2; k2 = 0;
  }
}

/* Estimate variance based on 1nn; this number is needed for evaluation of "log
 * probability" losses. Return an array of variances - one for each target. */

static void var_est_1nn(real *var)
{
  int  i, j, top = train.num-1;
  real *temp;

  for (j=0; j<no_tar; j++) {
    for (var[j]=0.0, i=top; i>=0; i--) {              /* leave out example i */
      SWAP(train.inp[i], train.inp[top]); SWAP(train.tar[i], train.tar[top]);
      sort_dist(top, train.inp[top]);
      find_neighbors(1);
      var[j] += sq(targets[j][0]-train.tar[top][j]);
    }
    var[j] /= train.num;
  }
}

static int comp(const void *x, const void *y)      /* function used by qsort */
{
  return (*((const real **)x)[0] > *((const real **)y)[0]) ? 1 : -1;
}
```

A.3 Neural networks trained with early stopping `mlp-ese-1`

The main portions of the source for the `mlp-ese-1` appears on the following pages. The usage of the program has the form

```
mlp-ese-1 instance-number [[-] cpu-time] [seed]]
```

The program will read training and test sets from the files `train.`*n* and `test.`*n* and write predictions to `cpred.`*n* where n is the instance-number. The allowed cpu-time may optionally be specified (in seconds); negative values are interpreted as times *per training case.* The default value is -1.875, i.e., 1.875 seconds per training case. A seed for the random number generator may be specified — by default it is chosen according to time and date.

Each member of the ensemble is training using a different random partition into training and validation sets. As the members of the ensemble are trained, their weights are dumped to a file called `wgts.`*n*. When training terminates all the networks in this file are re-read in order to produce the ensemble predictions. Running diagnostics is printed to `stderr`.

The `main` program uses the conjugate gradient minimization procedure described in appendix B. Functions implementing the neural network function and a function computing partial derivatives of the cost function with respect to the weights of the network are also given.

Finally, the `randomizeNet` function randomly initialises the network weights. The random weights are drawn from a Gaussian distribution of zero mean and a standard deviation of $0.1\sqrt{f_{\rm in}}$, where $f_{\rm in}$ is the fan-in of the unit which the weight projects to.

```
/* mlp-ese-1.c: Neural network ensembles trained with early stopping.
 *
 * An ensemble of neural networks is trained using early stopping. Each member
 * of the ensemble is validated on a validation set picked at random. Weights
 * of the trained nets are dumped to a log-file, which is re-read at the end of
 * training to produce predictions from the ensemble. The default running time
 * is 1.875 seconds per training example. Running diagnostics is printed to
 * stderr.
 *
 * (c) Copyright 1996 by Carl Edward Rasmusen. */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include "util.h"

#define MinEnsembleSize   3
#define MaxEnsembleSize  50
#define ValidateFraction  3
#define EarlyStopMargin   1.5
#define MinIteration    250
#define min(a,b) (((a)<(b))?(a):(b))

void createNet();
void destroyNet();
void randomizeNet(real magnitude);
real prior(real *dw) { return 0.0; }        /* we do not have a weight prior */
extern void initTimer(struct itimerval *timer, int time);
extern long elapsedTime(struct itimerval *timer);
real cost(struct exampleset ex, int wc);
void net(real *inp, real *out);
extern int conj(int *iter, int *epoch, int *restart, real *costvalue);

struct itimerval timer;
struct exampleset train;                            /* the training examples */

real *w,                           /* array [no_wts] of all network weights */
     **wi,   /* array [no_hid] of arrays [no_inp] of input-to-hidden weights */
     **wd,                        /* array of direct input-to-output weights */
     **wo,                              /* array of hidden-to-output weights */
     *output;                                    /* array of network outputs */
int  no_hid, no_inp, no_tar,        /* no. of hidden, input and target units */
     no_wts;                                         /* total no. of weights */


main(int argc, char **argv)
{

  char df[50];                                                  /* file name */
  long seed,
       length, used_time = 0;                   /* times are in milliseconds */
  int  restart,    /* used with conj(); if 1 then conj uses steepest descent */
       i, j, k, l,
       iter, best_iter,     /* current and best iteration for early stopping */
       ensembleSize,                 /* counts number of nets trained so far */
       succ;
  real *tmp,
       junk,
       perf, best_perf,   /* current and best performance for early stopping */
       **pred,                                       /* array of predictions */
       *best_w;            /* array of weights of best net in early stopping */
  struct exampleset valid, test;

extern int TIMEOUT;  /* set to 1 when SIGVTALRM is caught; when time is up */
FILE *wgtf,                /* file for storing ensemble of network weights */
     *pred_file;                    /* file to which predictions are dumped */


length = -1875;          /* default length is 1.875 secs per training case */
time(&seed);                                  /* chose a seed based on time */
no_inp = no_tar = -1;    /* defaults for "unknown" in case of no arguments */
if (argc < 2 || argc > 4 ||
    argc > 2 && ((length = 1000*atof(argv[2])) == 0.0) ||  /* length in ms */
    argc==4 && ((seed = atoi(argv[3])) <= 0)) {
  fprintf(stderr, "Usage: %s instance-number [[-]cpu-time [seed]]\n", argv[0]);
  exit(-1);
}
srand48(seed);

sprintf(df, "test.%s", argv[1]);                        /* read test inputs */
test.num = train.num = no_inp = no_tar = -1;      /* default for "unknown" */
loadExamples(&test, &no_inp, (i=0, &i), df, NULL);
sprintf(df, "train.%s", argv[1]);
loadExamples(&train, &no_inp, &no_tar, df, NULL);
if (length < 0) length *= -train.num;  /* training time was given per case */
valid.num = train.num/ValidateFraction;
train.num -= valid.num;         /* now train on only the training fraction */
valid.inp = &train.inp[train.num]; valid.tar = &train.tar[train.num];


sprintf(df, "wgts.%s", argv[1]);
if ((wgtf = fopen(df, "w+b")) == NULL) {
  fprintf(stderr, "Could not open weight file %s for writing ...bye!\n", df);
  exit(-1);
}

no_hid = ceil((train.num-(no_inp+1.0)*no_tar)/(no_inp+no_tar+1.0));
no_wts = (no_inp+1)*no_hid+(no_inp+no_hid+1)*no_tar;
fprintf(stderr, "Using nets with %3d hidden units and %5d weights.\n",
        no_hid, no_wts);
createNet();
best_w = (real *) malloc((size_t) no_wts*sizeof(real));
ensembleSize = 0;
fprintf(stderr, "net  train-err valid-err  best-iter  max-iter\n");
while (used_time < length && ensembleSize < MaxEnsembleSize) {

  for (i=0; i<train.num+valid.num; i++) {
    j = drand48()*train.num;
      tmp = train.inp[j]; train.inp[j] = train.inp[i]; train.inp[i] = tmp;
      tmp = train.tar[j]; train.tar[j] = train.tar[i]; train.tar[i] = tmp;
    }

  randomizeNet(0.1);
  initTimer(&timer, min(length/MinEnsembleSize, length-used_time));
  best_perf = cost(valid, 0);               /* initialize best_perf and... */
  for (i=0; i<no_wts; i++) best_w[i] = w[i];   /* best_w to the random net */
  iter = best_iter = 0; restart = 1;   /* conj() should restart its search */
  do {
    succ = conj((i=1, &i), (j=0, &j), &restart, &junk); iter++;
    if ((perf=cost(valid, 0)) < best_perf) {
      best_iter = iter; best_perf = perf;          /* update "best" values */
      for (i=0; i<no_wts; i++) best_w[i] = w[i];          /* save this net */
    }
  } while ((succ) && (!TIMEOUT) && ((iter < EarlyStopMargin*best_iter) ||
   (iter < MinIteration) || (MaxEnsembleSize*elapsedTime(&timer) < length)));
  used_time += elapsedTime(&timer);
  fwrite(best_w, no_wts*sizeof(real), 1, wgtf);      /* append net to wgtf */
```

```
    if (used_time<length || ensembleSize<MinEnsembleSize) {   /* add if time */
      for (i=0; i<no_wts; i++) w[i] = best_w[i];
      fprintf(stderr, "%3d %9.4f %9.4f %7d    %7d\n", ++ensembleSize,
                                 cost(train, 0), best_perf, best_iter, iter);
    }
  }

  rewind(wgtf);                              /* compute ensemble predictions */
  pred = createMatrix(no_tar, test.num);
  for (k=0; k<no_tar; k++) for (i=0; i<test.num; i++) pred[k][i] = 0.0;

  for (j=0; j<ensembleSize; j++) {
    fread(w, (long) no_wts*sizeof(real), 1, wgtf);
    for (i=0; i<test.num; i++) {
      net(test.inp[i], output);
      for (k=0; k<no_tar; k++) pred[k][i] += output[k];
    }
  }

  for (k=0; k<no_tar; k++)                               /* use mean prediction */
    for (i=0; i<test.num; i++) pred[k][i] /= ensembleSize;
  fclose(wgtf);

  pred_file = openPredFile("cguess.%s", argv[1]);   /* point prediction file */
  for (i=0; i<test.num; i++) {                         /* write predictions */
    for (k=0; k<no_tar; k++) fprintf(pred_file, "%f ", pred[k][i]);
    fprintf(pred_file, "\n");
  }
  fclose(pred_file); free(pred[0]); free(pred);

}
```

```
/* feval.c: Output and cost evaluation evaluation for neural net.
 *
 * The function "net" evaluates the outputs given the inputs for a fully
 * connected feed forward net with a single hidden layer of tanh units. The
 * function "cost" returns the squared error cost for all examples in a set
 * with or without weight panelaty.
 *
 * (c) Copyright 1996 by Carl Edward Rasmusen. */

#include <math.h>
#include "util.h"
#include <stdlib.h>

extern int no_hid, no_inp, no_tar, no_wts;
extern real *w, **wi, **wd, **wo, *hidden, *output;
extern real prior(real *dw);

void net(real *in, real *out)          /* compute network output given input */
{
  int  i, j;
  real x, *tmp;

  for (i=0; i<no_hid; i++) {             /* compute activity of hidden units */
    tmp = wi[i];
    x = tmp[no_inp];                                          /* hidden bias */
    for (j=0; j<no_inp; j++) x += in[j]*tmp[j];
    hidden[i] = tanh(x);
  }

  for (i=0; i<no_tar; i++) {           /* compute activities of output units */
    tmp = wo[i];
    x = tmp[no_hid];                                          /* output bias */
    for (j=0; j<no_hid; j++) x += hidden[j]*tmp[j];
    tmp = wd[i];
    for (j=0; j<no_inp; j++) x += in[j]*tmp[j];            /* direct weights */
    out[i] = x;
  }
}

real cost(ex, wc)     /* calc. error on examples, including penalty iff wc=1 */
  struct exampleset ex;
  int    wc;
{
  int  i, j;
  real x, *tmp, e = 0.0;

  for (i=0; i<ex.num; i++) {
    net(ex.inp[i], output);
    tmp = ex.tar[i];
    for (j=0; j<no_tar; j++) {
      x = output[j] - tmp[j];
      e += x*x;
    }
  }
  x = 0.0; if (wc == 1) x = prior(NULL);
  return (e+x)/ex.num;
}
```

```
/* fgeval.c: Evaluate cost and gradients over training set for neural net.
 *
 * The function returns the (batch) cost over the trainng set, and returns an
 * array of partial derivatives with repect to all the weights in a fully
 * connected feed-forward neural net with a single hidden layer of tanh units.
 *
 * (c) Copyright 1996 by Carl Edward Rasmusen. */

#include <math.h>
#include "util.h"

extern int no_inp, no_hid, no_tar, no_wts, nfgeval;
extern real *w, **wi, **wd, **wo, *output;
extern struct exampleset train;

real *hidden;                                    /* activity of hidden units */

extern real prior(real *dw);

real fgeval(dw)
  real *dw;
{
  int  i, j, k, l, m;
  real e = 0.0, x, y, *tmp, *pat;

  for (i=0; i<no_wts; i++) dw[i] = 0.0;                  /* zero derivatives */
  for (k=0; k<train.num; k++) {
    pat=train.inp[k];
    for (i=0; i<no_hid; i++) {                  /* compute hidden activities */
      tmp = wi[i];
      x = tmp[no_inp];                                        /* hidden bias */
      for (j=0; j<no_inp; j++)
        x += pat[j]*tmp[j];
      hidden[i] = tanh(x);
    }
    m = 0;
    for (i=0; i<no_tar; i++) {                  /* compute output activities */
      tmp = wo[i];
      x = tmp[no_hid]-train.tar[k][i];           /* output bias minus target */
      for (j=0; j<no_hid; j++) x += hidden[j]*tmp[j];
      tmp = wd[i];
      for (j=0; j<no_inp; j++) x += pat[j]*tmp[j];         /* direct weights */
      output[i] = x;
      e += x*x;                                  /* accumulate squared error */
      for (j=0; j<=no_hid; j++) dw[m++] += x*hidden[j];
      for (j=0; j<no_inp; j++) dw[m++] += x*pat[j];
    }
    for (j=0; j<no_hid; j++) {
      y=hidden[j];
      x = 0.0; for (i=0; i<no_tar; i++) x += output[i]*wo[i][j];
      y=x*(1.0-y*y);
      for (l=0; l<no_inp; l++)
        dw[m++] += y*pat[l];
      dw[m++] += y;
    }
  }
  nfgeval++;
  return 0.5*(e+prior(dw));
}
```

```
/* init.c: Functions to create, destroy and randomly initialize neural nets.
 *
 * Functions allocate and free the memory for fully connected neural networks
 * with a single hidden layer of tanh units, whose size are given by global
 * variables. The "randomizeNet" function sets the weights to random values
 * drawn from a zero mean Gaussian with s standard deviation given by the
 * "magnitude" parameter, scaled down by the sqare root of the fan-in fo the
 * units they project to.
 *
 * (c) Copyright 1996 by Carl Edward Rasmusen. */

#include <math.h>
#include <stdlib.h>
#include "util.h"

extern int  no_inp, no_tar, no_hid, no_wts;
extern real *w, **wi, **wd, **wo, *dw1, *dw2, *s, *hidden, *output;

void createNet()
{
  int i;

  w = (real *) malloc((size_t) no_wts*4*sizeof(real));
  wo = (real **) malloc((size_t) no_tar*sizeof(real*));
  wd = (real **) malloc((size_t) no_tar*sizeof(real*));
  wo[0] = &w[0]; wd[0] = &w[no_hid+1];
  for (i=1; i<no_tar; i++) { wo[i] = wo[i-1]+no_hid+1; wd[i] = wo[i]+no_inp; }
  wi = (real **) malloc((size_t) no_hid*sizeof(real*));
  wi[0] = &w[(no_hid+no_inp+1)*no_tar];
  for (i=1; i<no_hid; i++) wi[i] = wi[i-1]+no_inp+1;
  dw1 = &w[no_wts]; dw2 = &w[no_wts*2]; s = &w[no_wts*3];
  hidden = (real *) malloc((size_t) (no_hid+no_inp+1)*sizeof(real));
  hidden[no_hid] = 1.0;                               /* set the output bias */
  output = (real *) malloc((size_t) no_tar*sizeof(real));
}

void destroyNet()
{
  free(w); free(wi); free(hidden); free(output); free(wo); free(wd);
}

#define NORM cos(6.283185*drand48())*sqrt(-2.0*log(drand48()))

void randomizeNet(real magnitude)
{
  int i;

  for (i=0; i<(no_hid+no_inp+1)*no_tar; i++)
    w[i] = magnitude*NORM/sqrt(no_hid+no_inp+1.0);
  for (; i<no_wts; i++)
    w[i] = magnitude*NORM/sqrt(no_inp+1.0);
}
```

A.4 Bayesian neural networks `mlp-mc-1`

The source for this method was written by Radford Neal and is publicly available on the Web through `http://www.cs.toronto.edu/~radford`. The software used for the experiments in this thesis is the release dated 26 August 1996. Since it is quite long I will not reproduce it here. Instead I will supply the script (written for the `csh` shell) which I used to run the simulations for this method. For a more detailed description of the options available refer to the documentation of the software.

```
net-spec log.$1 8 6 1 / - x0.2:0.5:1 0.1:0.5 - x0.05:0.5 x0.2:0.5:1 1
model-spec log.$1 real 0.05:0.5
data-spec log.$1 8 1 / train.$1 .

rand-seed log.$1 $1
net-gen log.$1 fix 0.1 0.5
mc-spec log.$1 repeat 10 sample-noise heatbath hybrid 100:10 0.15
net-mc log.$1 1
mc-spec log.$1 sample-sigmas heatbath 0.95 hybrid 100:10 0.15 negate
net-mc log.$1 @32 58

net-pred bn log.$1 @10.7:32+100 / test.$1 > cguess.S.$1
net-pred bd log.$1 @10.7:32+100 / test.$1 > cguess.A.$1
net-pred bp log.$1 @10.7:32+100 / test.$1 targets.$1 > clptarg.L.$1
```

The argument to the script `$1` contains the instance number. In this example a network with 8 inputs and a single layer of 6 hidden units is trained for 32 minutes of cpu-time. The number of hidden units should be selected according to the heuristic rules given in section 3.7. The net-spec command specifies the architecture of the network and the weight priors as discussed in section 3.7. The following lines specify the noise model and tells the program where to get the data files. After setting the seed for the random number generator the hyperparameters are initialized. Then the Markov chain is set up to perform 10 iterations each of 100 leapfrog steps with a step size of 0.15 for fixed values of the hyperparameters. These iterations are performed in the following call to `net-mc` and should allow the weights to attain somewhat reasonable values.

In the following line the Markov chain is set up for the remainder of the run. The `sample-sigmas` command updates the hyperparameters and the `heatbath 0.95` command updates momenta with a persistance of 0.95; the remainder of the line specifies Hybrid Monte Carlo for the weights with leaprog trajectories of length 100, window size 10 and step size 0.15. The call to `net-mc` allows for 32 minutes of cpu time and saves every 58th iteration to the log file — an interval which by experimentation was determined appropriate for the desired number of samples in 32 minutes. The last three lines of the script generate predictions from 100 samples from the last 2/3 of the run for the three standard loss types.

A.5 Gaussian Processes

This section contains the source code of the porgrams used for the Gaussian Process methods in the thesis. However, I am currently rewriting the software to be more portable and more flexible by allowing specification of the desired covariance function and noise models via the command line. This new software will be made available to the public domain via DELVE.

The source for the two methods `gp-map-1` and `gp-mc-1` is given in the following. The code that is specific to the covariance function is located in the `cov-1.c` file. Then follows the generic programs for the two methods in `gp-map.c` and `gp-mc.c`. The optimization for the MAP method is done using the conjugate gradient method described in appendix B. The leapfrog method for the Hybrid Monte Carlo algorithm follows in the `leapfrog.c` file. Lastly the generic prediction program `gp-pred.c` is listed followed by the matrix inversion code in `invspd.c` and a routine to find medians in `median.c`.

The programs `gp-map-1` and `gp-mc-1` store the values of the hyperparameters (and momentum variables) at regular intervals in so-called log files. The `gp-pred-1` program makes predictions for test cases based on the information in the log files. For problems containing many training cases, this design is wasteful, since the prediction program has to invert the same matrices as have already been inverted by the training program.

The script (using the `csh` shell) used to run the `gp-map-1` method on the instance given by `$1` using log files named `log` for a total cpu time of 120 seconds consists of the two following lines:

```
gp-map-1 log $1 @120
gp-pred-1 log $1 `grep -c ^ log.$1`
```

Here the `grep` command simply returns the number of records written to the log file, causing only the last record to be used for the predictions.

Similarly, the script used for the `gp-mc-1` method on instance number `$1` for 32 minutes of cpu time (1920 seconds), saving 39 log records and using the 26 records of the last 2/3 of the run for predictions is:

```
gp-mc-1 log $1 @1920+39
gp-pred-1 log $1 @640:1920+26
```

Note that the implementation supports continuation of a run on a specific log file by simply re-calling `gp-mc-1` with a longer time specification; this feature was used when generating data for fig. 5.19.

```
/* cov-1.c: Contains the code for a specific covariance function.
 *
 * This file contains the code for the training (gp-mc) and prediction
 * (gp-pred) programs for regression with Gaussian processes, which is specific
 * to particular covariance functions. In this file (cov1.c) the covariance is
 * of the form c(x^i,x^j)=u_0 + e*\delta(i,j) + u_1*sum_k (x^i)_k*x(^j)_k +
 * v*exp-0.5*sum_k w_k((x^i)_k-(x^j)_k)^2, where u_0 is the variance controling
 * the bias, u_1 is a variance controling the parameters of the linear
 * component, w_k are the (inverse) length scales, and e is the noise
 * variance. The actual hyperparameters used in the program are exp(u_0),
 * exp(u_1), exp(w_k) and exp(e), which is a hack to ensure that the variances
 * stay positive. The hyperparameters are ordered in w=[w_0,...,w_k-1, v, u_0,
 * u_1, e].
 *
 * (c) Copyright 1996 Carl Edward Rasmussen */

#include <stdlib.h>
#include <stddef.h>
#include <math.h>
#include "util.h"

real *ew;                               /* exp of some of w, for convenience */

extern int  no_inp, no_wts, nfgeval;
extern real *w, *q, **K, **K1, **K2;
extern struct exampleset train;

extern real invspd(real **, int n);

void init()                      /* set no_wts and create and initialise w[] */
{
  int i;
  no_wts = no_inp+4;
  w = (real*) malloc((size_t) no_wts*sizeof(real));
  for (i=0; i<no_inp; i++) w[i] = -log((real) no_inp);      /* ARD style w's */
  w[i++] = 0.0;                                          /* signal variance v */
  for (; i<no_wts; i++) w[i] = -2.0;
  ew = (real*) malloc((size_t) no_wts*sizeof(real));
}

/* This function returns -log prior for the hyperparameters and augments the
 * array of derivatives by the effect of the prior. The prior on w is Gaussian.
 */

real prior(real *dw)
{
  int  i;
  real r = 0.0,
       mean = -3.0, sigma = 3.0, mu = 1.0;            /* prior specification */

  for (i=0; i<no_inp; i++) {
    r += log(6.2831953*no_inp*no_inp/mu)+w[i]+mu/(no_inp*no_inp*exp(w[i]));
    dw[i] += 0.5*(1.0-mu*exp(-w[i])/(no_inp*no_inp));
  }
  r += sq((w[i]+1.0)/1.0);
  dw[i++] += (w[i]+1.0)/(1.0*1.0);
  for (; i<no_wts; i++) {
    r += sq((w[i]-mean)/sigma);
    dw[i] += (w[i]-mean)/(sigma*sigma);
  }
  return 0.5*r;
}

static real trace_prod(real **a, real **b)
{
  static real r;
  static int i, j;

  for (r=0.0, i=0; i<train.num; i++) {
    for (j=0; j<i; j++) r += a[j][i]*b[j][i];
    for (; j<train.num; j++) r += a[i][j]*b[i][j];
  }
  return r;
}

/* This function returns "-log posterior for w" (plus a constant), and the
 * derivatives of this with respect to w. A pointer to a function which
 * augments the function value and the derivatives according to the prior must
 * be supplied; if this is NULL then the likelihood is used instead of the
 * posterior.
 */

real fgeval(real *dw)
{
  int  i, j, k;            /* i and j index training cases, k indexes inputs */
  real r, rr;                                              /* miscellaneous */

  for (i=0; i<no_wts; i++) ew[i] = exp(w[i]);                 /* compute ew */

  for (i=0; i<train.num; i++) { /* set upper triangle of K[][] to covariance */
    for (j=i; j<train.num; j++) {
      for (r=rr=0.0, k=0; k<no_inp; k++) {
        r += ew[k]*sq(train.inp[i][k]-train.inp[j][k]);
        rr += train.inp[i][k]*train.inp[j][k];
      }
      r = ew[no_inp]*exp(-0.5*r);
      K1[i][j] = r; K[i][j] = r+ew[no_inp+2]*rr+ew[no_inp+1];
    }
    K[i][i] += ew[no_inp+3];
  }

  for (i=0; i<train.num; i++)
    for (j=i; j<train.num; j++) K2[i][j] = K[i][j];                 /* copy */
  r = invspd(K2, train.num);                   /* r = log det K; K2 = inv K */

  for (i=0; i<train.num; i++) {                         /* compute q[] and r */
    for (rr=0.0, j=0; j<i; j++) rr += train.tar[j][0]*K2[j][i];
    for (; j<train.num; j++) rr += train.tar[j][0]*K2[i][j];
    q[i] = rr;                                           /* q = t * inv(K) */
    r += train.tar[i][0]*rr;                          /* r = t * inv(K) * t */
  }
  r *= 0.5;                   /* r = 0.5 * log det K + 0.5 * t * inv(K) * t */

/* Work out derivatives of -log posterior with respect to the hp. First for the
 * scales parameters w[0,...,no_inp-1], the signal scale w[no_inp], the scale
 * for the bias w[no_inp+1], the scale for the linear part of the model
 * w[no_inp+2] and lastly the noise scale w[no_inp+3].
 */

  dw[no_inp] = trace_prod(K1, K2);                    /* Tr[inv(K)*dK/dv] */
  for (i=0; i<train.num; i++) {
    for (rr=0.0, j=0; j<i; j++) rr += q[j]*K1[j][i];
    for (; j<train.num; j++) rr += q[j]*K1[i][j];
    dw[no_inp] -= rr*q[i];
  }
  dw[no_inp] *= 0.5;

  for (k=0; k<no_inp; k++) {                                 /* input scales */
```

```
    for (i=0; i<train.num; i++) for (j=i; j<train.num; j++)
      K[i][j] = -K1[i][j]*0.5*ew[k]*sq(train.inp[i][k]-train.inp[j][k]);
    dw[k] = trace_prod(K, K2);
    for (i=0; i<train.num; i++) {
      for (rr=0.0, j=0; j<i; j++) rr += q[j]*K[j][i];
      for (; j<train.num; j++) rr += q[j]*K[i][j];
      dw[k] -= rr*q[i];
    }
    dw[k] *= 0.5;
  }

  for (i=0; i<train.num; i++)                            /* set K1 = dK/du_1 */
    for (j=i; j<train.num; j++) {
      for (rr=0.0, k=0; k<no_inp; k++) rr += train.inp[i][k]*train.inp[j][k];
      K1[i][j] = ew[no_inp+2]*rr;
    }
  dw[no_inp+2] = trace_prod(K1, K2);
  for (i=0; i<train.num; i++) {
    for (rr=0.0, j=0; j<i; j++) rr += q[j]*K1[j][i];
    for (; j<train.num; j++) rr += q[j]*K1[i][j];
    dw[no_inp+2] -= rr*q[i];
  }
  dw[no_inp+2] *= 0.5;

  for (rr=0.0, i=0; i<train.num; i++)                                /* bias */
    for (j=i+1; j<train.num; j++) rr += K2[i][j];
  rr *= 2.0; for (i=0; i<train.num; i++) rr += K2[i][i]; dw[no_inp+1] = rr;
  for (rr=0.0, i=0; i<train.num; i++) rr += q[i];
  dw[no_inp+1] -= rr*rr; dw[no_inp+1] *= 0.5*ew[no_inp+1];

  for (rr=0.0, i=0; i<train.num; i++) rr += K2[i][i]-q[i]*q[i];     /* noise */
  dw[no_inp+3] = 0.5*rr*ew[no_inp+3];

  r += prior(dw);                                        /* augment by prior */
  nfgeval++;

  return r;
}

/* This function returns mean and variance for predictions for a set of test
 * cases, given values fo the hyperparameters. It uses globals: w, ew, no_wts,
 * t, train and K.
 */

void pred(real *y, real *s2, struct exampleset test)
{
  int  i, j, k;
  real r, rr, *g, *h;

  h = (real *) malloc((size_t) train.num*sizeof(real));
  g = (real *) malloc((size_t) train.num*sizeof(real));

  for (i=0; i<no_wts; i++) ew[i] = exp(w[i]);                  /* compute ew */

  for (i=0; i<train.num; i++) { /* set upper triangle of K[][] to covariance */
    for (j=i; j<train.num; j++) {
      for (r=rr=0.0, k=0; k<no_inp; k++) {
        r += ew[k]*sq(train.inp[i][k]-train.inp[j][k]);
        rr += train.inp[i][k]*train.inp[j][k];
      }
      K[i][j] = ew[no_inp]*exp(-0.5*r)+ew[no_inp+2]*rr+ew[no_inp+1];
    }
    K[i][i] += ew[no_inp+3];
  }
  invspd(K, train.num);                                  /* invert covariance */
  for (i=0; i<test.num; i++) {
    for (j=0; j<train.num; j++) {
      for (r=rr=0.0, k=0; k<no_inp; k++) {
        r += ew[k]*sq(test.inp[i][k]-train.inp[j][k]);
        rr += test.inp[i][k]*train.inp[j][k];
      }
      g[j] = ew[no_inp]*exp(-0.5*r)+ew[no_inp+1]+ew[no_inp+2]*rr;
    }
    for (j=0; j<train.num; j++) {
      for (r=0.0, k=0; k<j; k++) r += g[k]*K[k][j];
      for (; k<train.num; k++) r += g[k]*K[j][k];
      h[j] = r;
    }
    r = 0.0; for (k=0; k<train.num; k++) r += h[k]*train.tar[k][0]; y[i] = r;
    r = 0.0; for (k=0; k<train.num; k++) r += h[k]*g[k];
    rr = 0.0; for (k=0; k<no_inp; k++) rr += test.inp[i][k]*test.inp[i][k];
    s2[i] = ew[no_inp]+ew[no_inp+1]+ew[no_inp+2]*rr+ew[no_inp+3]-r;
  }
  free(g); free(h);
}
```

```
/* gp-ml.c: Generic program for doing maximum likelihood with Gaussian
 * Processes
 *
 * (c) Copyright 1996 by Carl Edward Rasmussen. */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include "util.h"

real *w,                                                /* hyperparameters */
     *q,                                 /* help variable - used by fgeval() */
     **K, **K1, **K2;                                      /* main matrices */
int  no_wts,                                   /* number of hyperparameters */
     no_inp,                                             /* input dimension */
     no_tar;                                           /* number of targets */
struct itimerval timer;
struct exampleset train;

extern real *dw1, *dw2, *s;                /* arrays for the conj() function */

extern void init();
extern int conj(int *iter, int *epoch, int *restart, real *costvalue);
extern void initTimer(struct itimerval *timer, int time);
extern long elapsedTime(struct itimerval *timer);

main(argc, argv)
  int  argc;
  char **argv;
{
  int  j, k, restart, succ, length, mod, iter;
  real costvalue;
  extern int TIMEOUT;
  long nexttime;
  char   trainfile[50], logfile[50];
  FILE   *logf;

  if (argc<3 || argc>4) {
    fprintf(stderr,
    "Usage: %s log-file instance-number [@]length[{%%|+}logInterval]\n",
          argv[0]); exit(-1); }

  parse_length(argv[3], &length, &mod);
  if (length<0)
    initTimer(&timer, -length*1000);                /* limit by compute time */
  else
    initTimer(&timer, 604800);                 /* this is a week of cpu time */

  sprintf(logfile, "touch %s.%s", argv[1], argv[2]); system(logfile);
  sprintf(logfile, "%s.%s", argv[1], argv[2]);
  if ((logf = fopen(logfile, "r+")) == NULL) {
    fprintf(stderr, "Could not open log-file %s for writing ...bye!\n",
                     logfile); exit(-1); }

  sprintf(trainfile, "train.%s", argv[2]);
  train.num = -1; no_inp = -1;                   /* default for "don't know" */
  loadExamples(&train, &no_inp, (no_tar = 1, &no_tar), trainfile, NULL);
  init();

  K  = createMatrix(train.num, train.num);
  K1 = createMatrix(train.num, train.num);
  K2 = createMatrix(train.num, train.num);
  dw1 = (real*) malloc((size_t) no_wts*sizeof(real));
  dw2 = (real*) malloc((size_t) no_wts*sizeof(real));
  s = (real*) malloc((size_t) no_wts*sizeof(real));
  q = (real*) malloc((size_t) train.num*sizeof(real));

  restart = 1; iter = 0; nexttime = elapsedTime(&timer);
  do {
    succ = conj((j=1, &j), (k=0, &k), &restart, &costvalue); iter++;
    if (((mod>0) && !(iter % mod)) ||
        ((mod<0) && (elapsedTime(&timer)>nexttime))) {
      fprintf(logf, "%6d %8d %10.6f %10.6f %10.6f", iter, elapsedTime(&timer),
                                                      costvalue, 0.0, 0.0);
      for (j=0; j<no_wts; j++) fprintf(logf, " %10.6f", w[j]);
      for (j=0; j<no_wts; j++) fprintf(logf, " %10.6f", 0.0);
      fprintf(logf, "\n"); fflush(logf);
      nexttime -= 1000*abs(length)/mod;             /* set next time to save */
    }
  } while ((succ) && !(TIMEOUT) && (iter<length || length<0));
}
```

```c
/* gp-mc.c: Generic program for doing Monte Carlo with Gaussian Processes
 *
 * (c) Copyright 1996 by Carl Edward Rasmussen. */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include "rand.h"
#include "util.h"

real *w,                                                  /* hyperparameters */
     *dw1,                                     /* hyperparameter derivatives */
     *z,                                                  /* momentum terms */
     *q,                                 /* help variable - used by fgeval() */
     **K, **K1, **K2;                                       /* main matrices */
int  nfgeval,  /* number of func/grad evals. so far; incremented by fgeval() */
     no_wts,                                    /* number of hyperparameters */
     no_inp,                                              /* input dimension */
     no_tar;                                            /* number of targets */
struct itimerval timer;
struct exampleset train;

extern void init();
extern void leapfrog(FILE *df, real RHO, real EPSILON,
                                                  int num, int mod, int start);

setTime(struct itimerval *timer, long current)
{
  timer->it_value.tv_sec -= current/1000;
  setitimer(ITIMER_VIRTUAL, timer, timer);
}

main(int argc, char **argv)
{
  double r;
  char   trainfile[50], logfile[50];
  int    i, length, mod, current = 1;
  long   seed, tm = 0;
  FILE   *logf;

  time(&seed);           /* get a seed based on the time if none is provided */
  if (argc<4 || argc>5 || argc==5 && ((seed = atoi(argv[4])) <= 0)) {
    fprintf(stderr,
    "Usage: %s log-file instance-number [@]length[{%%|+}logInterval] [seed]\n",
          argv[0]); exit(-1); }
  rand_seed((int) seed);

  parse_length(argv[3], &length, &mod);
  if (length<0)
    initTimer(&timer, -length*1000);                /* limit by compute time */
  else
    initTimer(&timer, 604800000);              /* this is a week of cpu time */

  sprintf(logfile, "touch %s.%s", argv[1], argv[2]); system(logfile);
  sprintf(logfile, "%s.%s", argv[1], argv[2]);
  if ((logf = fopen(logfile, "r+")) == NULL) {
    fprintf(stderr, "Could not open log-file %s for writing ...bye!\n",
                     logfile); exit(-1); }

  sprintf(trainfile,"train.%s", argv[2]);
  train.num = -1; no_inp = -1;                   /* default for "don't know" */
  loadExamples(&train, &no_inp, (no_tar = 1, &no_tar), trainfile, NULL);

  init();

  K  = createMatrix(train.num, train.num);
  K1 = createMatrix(train.num, train.num);
  K2 = createMatrix(train.num, train.num);
  z = (real*) malloc((size_t) no_wts*sizeof(real));
  for (i=0; i<no_wts; i++) z[i] = 0.0;
  dw1 = (real*) malloc((size_t) no_wts*sizeof(real));
  q = (real*) malloc((size_t) train.num*sizeof(real));

  while (fscanf(logf, "%d %ld %lf %lf %lf", &current, &tm, &r, &r, &r) == 5) {
    for (i=0; i<no_wts; i++) { fscanf(logf, "%lf", &r); w[i] = r; }
    for (i=0; i<no_wts; i++) { fscanf(logf, "%lf", &r); z[i] = r; }
  }
  fseek(logf, (long) 0, SEEK_END);            /* prepare logfile for writing */
  if (tm<-1000*length) {
    setTime(&timer, tm);
    leapfrog(logf, 0.95, 0.5, length, mod, current);
  }
}
```

```
/* leapfrog.c: Perform leapfrog iterations for a set of differential eq.
 *
 * Do 'num' leapfrog iterations and save every 'mod' one to the log-file 'df'.
 * If 'num' is negative then just keep going until TIMEOUT is set.
 *
 * (c) Copyright 1996 Carl Edward Rasmussen */

#include <math.h>
#include <stdio.h>
#include "util.h"
#include "rand.h"

extern int  no_wts, TIMEOUT;
extern real *dw1, *z, *w;
extern struct itimerval timer;
extern struct exampleset train;

extern real prior(real *dw);
extern real fgeval(real *dw, real (*prior)(real *dw));

void leapfrog(FILE *df, real RHO, real EPSILON, int num, int mod, int start)
{
  int  i, j, k = 0, reject = 0;
  long nexttime = elapsedTime(&timer);
  real E_old, E_kin, E_pot, E_pot_old, *oz, *ow;                /* old state */

  EPSILON /= sqrt((real) train.num);                       /* scale stepsize */

  oz = (real*) malloc((size_t) no_wts*sizeof(real));
  ow = (real*) malloc((size_t) no_wts*sizeof(real));

  E_pot = fgeval(dw1, prior);

  E_kin = 0; for (j=0; j<no_wts; j++) E_kin += z[j]*z[j]; E_kin *= 0.5;
  E_old = E_pot+E_kin;

  for (i=start; i<=num || num<0; i++) {
    for (j=0; j<no_wts; j++)
      { oz[j] = z[j]; ow[j] = w[j]; } E_pot_old = E_pot;   /* save old state */

    for (j=0; j<no_wts; j++)                    /* initial 2/3 leapfrog step */
      { z[j] -= 0.5*EPSILON*dw1[j]; w[j] += EPSILON*z[j]; }

    E_pot = fgeval(dw1, prior);
    E_kin = 0.0; for (j=0; j<no_wts; j++)     /* remaining 1/3 leapfrog step */
      { z[j] -= 0.5*EPSILON*dw1[j]; E_kin += z[j]*z[j]; }
    E_kin *= 0.5;

    if (exp(E_old-E_pot-E_kin) < rand_uniform()) {                /* reject */
      reject++;
      for (j=0; j<no_wts; j++)       /* restore old state and negate momenta */
       { z[j] = -oz[j]; w[j] = ow[j]; } E_pot = E_pot_old;
    }

    for (j=0; j<no_wts; j++)                             /* replace momenta */
      z[j] = RHO*z[j]+sqrt(1.0-RHO*RHO)*rand_gaussian();
    E_kin = 0.0; for (j=0; j<no_wts; j++) E_kin += z[j]*z[j]; E_kin *= 0.5;
    E_old = E_pot+E_kin;
    k++;
    if (((mod>0) && !(i % mod)) ||
        ((mod<0) && (elapsedTime(&timer)>nexttime))) {
      fprintf(df, "%6d %8d %10.6f %10.6f %10.6f", i,
                    elapsedTime(&timer),
                    E_pot, E_kin, (real) reject/k);
      for (j=0; j<no_wts; j++) fprintf(df, " %10.6f", w[j]);
      for (j=0; j<no_wts; j++) fprintf(df, " %10.6f", z[j]);
      fprintf(df, "\n"); fflush(df);
      nexttime -= 1000*abs(num)/mod;                 /* set next time to save */
      k = reject = 0;                  /* reset iteration and reject counter */
    }
    if (TIMEOUT) exit(0);  /* TIMEOUT will be set when a SIGVTALRM is caught */
  }
  free(oz); free(ow);
}
```

```
/* gp-pred.c: Generic program for doing predictions with Gaussian Processes.
 *
 * Log record from a specified log file within a specified interval are used to
 * make predictions for squared error loss, absolute error loss and negative
 * log probability loss. For squared error loss, the mean of the predictive
 * distributon is used. For absolute error loss, the meadian is approximated by
 * MEDIANSSIZE samples from each record. For negative log predictive loss the
 * width of the predictive distribution is enlarged if the sum of the widths
 * of the predictive Gaussians is less than the spread of the central 2/3 of
 * of the means.
 *
 * (c) Copyright 1996 by Carl Edward Rasmussen. */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "util.h"
#include "rand.h"

#define MEDIANSSIZE 11                 /* sample size for median predictions */
#define two_pi 6.28318530717959

real *w,                                                  /* hyperparameters */
     **K,                                                   /* main matrices */
     **K1, **K2, *q;                                               /* unused */
int  no_wts,                                    /* number of hyperparameters */
     no_inp,                                              /* input dimension */
     no_tar,                                            /* number of targets */
     nfgeval;                                                      /* unused */
struct exampleset train, test;

extern void pred(real *y, real *s2, struct exampleset test);
extern void init();
extern real median(real *, int);
extern real select(real *, int, int);

main(argc, argv)
  int  argc;
  char **argv;
{
  double r;
  real   **s2, **means, **meds, *hlp, tmp;
  int    i, j, k, l, h, mm, low, high, mod;
  long   tm;                                                         /* time */
  char   trainfile[50], testfile[50], targetfile[50], logfile[50], outfile[50];
  FILE   *fp;

  if (argc != 4) {
    fprintf(stderr,
        "Usage: %s log-file instance-number [@][min]:[max][{%%|+}Interval]\n",
         argv[0]); exit(-1); }

  parse_range(argv[3], &low, &high, &mod);
  sprintf(trainfile, "train.%s", argv[2]);
  train.num = test.num = no_inp = -1;               /* default for "unknown" */
  loadExamples(&train, &no_inp, (no_tar=1, &no_tar), trainfile, NULL);
  sprintf(testfile, "test.%s", argv[2]);
  sprintf(targetfile, "targets.%s", argv[2]);
  loadExamples(&test, &no_inp, &no_tar, testfile, targetfile);

  sprintf(logfile, "%s.%s", argv[1], argv[2]);
  if ((fp=fopen(logfile, "r")) == NULL) {
    fprintf(stderr, "Could not open log file %s for reading... bye!\n",
            logfile); exit(-1); }
```

```
init();

if (low<0 || high==-1) {       /* range is given in time or no upper limit */
  while (fscanf(fp, "%d %ld %lf %lf %lf", &j, &tm, &r, &r, &r) == 5) {
    for (i=0; i<no_wts; i++) { fscanf(fp, "%lf", &r); w[i] = r; }
    for (i=0; i<no_wts; i++) fscanf(fp, "%lf", &r);
    if (low<0 && tm/1000>=-low) low = j;
    if (tm/1000<high || high==-1) k = j;
  }
  if (high != -2) high = k;
  rewind(fp);
}
if (high==-2) high = low;
if (mod==0) mod = high-low+1;
fprintf(stderr,
 "Using up to %d samples with indexes between %d and %d for predicting...\n",
  mod, low, high); fflush(stderr);

K = createMatrix(train.num, train.num);
s2 = createMatrix(mod, test.num);
means   = createMatrix(mod, test.num);
for (k=j=l=0; l<mod; l++) {
  if (mod==1) h=low; else h=low+(l*(high-low))/(mod-1);
  do {
    fscanf(fp, "%d %ld %lf %lf %lf", &j, &tm, &r, &r, &r);
    for (i=0; i<no_wts; i++) { fscanf(fp, "%lf", &r); w[i] = r; }
    for (i=0; i<no_wts; i++) fscanf(fp, "%lf", &r);
  } while (h>j);
  if (j>high) break;                          /* there are no more samples */
  pred(means[l], s2[l], test);
  k++;
  if (j==high) break;                         /* there are no more samples */
}
fclose(fp);

fp = openPredFile("cguess.S.%s", argv[2]); /* write predictions for S loss */
for (i=0; i<test.num; i++) {
  for (tmp=0.0,j=0; j<k; j++) tmp += means[j][i];
  fprintf(fp, "%10.6f\n", tmp/k);
}
fclose(fp);

fp = openPredFile("cguess.A.%s", argv[2]); /* write predictions for A loss */
meds = createMatrix(k, MEDIANSSIZE);
for (i=0; i<test.num; i++) {
  for (j=0; j<k; j++)
    for (l=0; l<MEDIANSSIZE; l++)
      meds[j][l] = means[j][i]+sqrt(s2[j][i])*rand_gaussian();
  fprintf(fp, "%10.6f\n", median(meds[0], MEDIANSSIZE*k));
}
fclose(fp);

fp = openPredFile("clptarg.L.%s", argv[2]);      /* write preds for L loss */
for (i=0; i<test.num; i++) {
  for (tmp=0.0,j=0; j<k; j++) {
    tmp += sqrt(s2[j][i]);
    meds[0][j] = means[j][i];
  }
  tmp /= select(meds[0], k, k*5/6) - select(meds[0], k, k/6);
  if (tmp < 1.0) for (tmp=sq(tmp),j=0; j<k; j++) s2[j][i] /= tmp;
  for (tmp=0.0,j=0; j<k; j++)
    tmp += 1/sqrt(two_pi*s2[j][i])*
          exp(-0.5*sq(test.tar[i][0]-means[j][i])/s2[j][i]);
  fprintf(fp, "%10.6f\n", log(tmp/k));
  }
  fclose(fp);

  free(meds[0]); free(meds); free(means[0]); free(means); free(s2[0]);
  free(s2);
}
```

```
/* invspd.c: Do an "in place" inversion of a real square symmetric positive
 * definite matrix "a" of size "n" by "n" and return log of it’s determinant.
 *
 * The function only looks at elements on or above the diagonal. Computes and
 * stores the lower triangular Cholesky factor in "a" (1/6n^3 flops) and does
 * inversion using forward (1/2n^3 flops) and backward (1/6n^3 flops)
 * substitution. On return, the upper diagonal matrix contains the inverse of
 * "a". See: Golub and Van Loan, "Matrix computations", 2nd edition, Johns
 * Hopkins University Press, 1989.
 *
 * (c) Copyright 1996 Carl Edward Rasmussen */

#include <math.h>
#include <stdio.h>
#define real double

real invspd(real **a, int n)
{
  int  i, j ,k;
  real s, *d, *x;

  d = (real *) malloc((size_t) n*sizeof(real));
  x = (real *) malloc((size_t) n*sizeof(real));

  for (i=0; i<n; i++)                          /* do Cholesky decomposition */
    for (j=i; j<n; j++) {
      s = a[i][j];
      for (k=i-1; k>=0; k--) s -= a[i][k]*a[j][k];
      if (i == j) {
        if (s <= 0.0) {
          fprintf(stderr,
              "Error: Matrix for inversion is not positive definite... bye!\n");
          exit(-1);
        }
        d[i] = sqrt(s);
      } else a[j][i] = s/d[i];
    }

  for (i=0; i<n; i++) {                                    /* for each colum */
    for (j=0; j<n; j++) {                         /* do forward substitution */
      s = (i == j) ? 1.0 : 0.0;                            /* of unit matrix */
      for (k=j-1; k>=0; k--) s -= a[j][k]*x[k];
      x[j] = s/d[j];
    }
    for (j=n-1; j>=i; j--) {                     /* do backward substitution */
      s = x[j];
      for (k=j+1; k<n; k++) s -= a[k][j]*x[k];
      a[i][j] = x[j] = s/d[j];
    }
  }

  s = 0.0; for (i=0; i<n; i++) s += log(d[i]);          /* compute log det a */
  free(x); free(d);
  return 2.0*s;
}
```

```
/* median.c: Return the median from an array.
 *
 * The following 3 routines are used to find the median of an array "a" of "k"
 * numbers in expected linear time. The value of k must be at least 1. The
 * elements in "a" are reordered by the function. The algorithm uses selection
 * and randomized partitioning, see for example Cormen, Leiserson and Rivest,
 * "Introduction to Algorithms", MIT Press, 1993.
 *
 * Copyright (c) 1996 Carl Edward Rasmussen */

#include <stdlib.h>
#include "util.h"

static real select(real *, int, int);                   /* private functions */
static int  partition(real *, int);

real median(real *a, int k)               /* return the median from an array */
{
  if (k/2 != (k+1)/2)                                         /* if k is odd */
    return select(a, k, k/2+1);          /* then the median is in the middle */
  else                               /* otherwise the mean of the two middle */
    return 0.5*(select(a, k, k/2) + select(a, k, k/2+1));         /* numbers */
}

/* Recursive function that returns the i'th smallest element from an array,
 * i=1..k. The elements are rearranged by the function. */

static real select(real *a, int k, int i)
{
  static int q;

  if (k == 1) return a[0];
  q = partition(a, k)+1;
  if (i <= q)
    return select(a, q, i);
  else
    return select(&a[q], k-q, i-q);
}

/* Partition an array around an element chosen at random and retun the index of
 * the partiton element. Upon returning all the elements with indexes smaller
 * than or equal to the index of the partition element have values that are
 * less than or equal to the partition element; the rest of the array have
 * values larger than or equal to the partition element. */

static int partition(real *a, int k)
{
  static real x, temp;
  static int  i, j;

  x = a[k*rand()/(RAND_MAX+1)]; i = -1; j = k;
  for (;;) {
    do j--; while (a[j] > x);
    do i++; while (a[i] < x);
    if (i < j) { temp = a[i]; a[i] = a[j]; a[j] = temp; } else return j;
  }
}
```

Appendix B

Conjugate gradients

This appendix describes the utility function `conj` which optimizes network weights using a conjugate gradient method. The function has a private function `lns` for doing line-searches.

The conjugate gradient method for minimizing a function of many variables works by iteratively computing search directions, along which a line search procedure minimizes the function, producing a new approximation to the (local) minimum of the objective function.

An *iteration* is defined as a computation of a search direction and the following line search. An *epoch* is defined as the computation of the function value and gradient, $f(\mathbf{x})$ and $\mathbf{g}(\mathbf{x}) = \nabla f(\mathbf{x})$. The function and gradients are always computed as a pair. An iteration might involve many epochs (because of the line search). The number of epochs is thus indicative of the computational effort.

B.1 Conjugate Gradients

At iteration number i the position in weight space is $\mathbf{x}_i$, the value of the objective function is $f(\mathbf{x}_i)$ and the gradient (vector of partial derivatives) is $\mathbf{g}_i = \nabla f(\mathbf{x}_i)$. I use what is known as the Polack-Ribiere version of conjugate gradients [Fletcher 1987], which computes the new search direction $\mathbf{s}_i$ from the old direction $\mathbf{s}_{i-1}$ and the current and old gradients $\mathbf{g}_i$ and $\mathbf{g}_{i-1}$:

$$\mathbf{s}_i = -\mathbf{g}_i + \frac{(\mathbf{g}_i - \mathbf{g}_{i-1})^{\mathrm{T}}\mathbf{g}_i}{(\mathbf{g}_{i-1})^{\mathrm{T}}\mathbf{g}_{i-1}}\mathbf{s}_{i-1}, \qquad i = 1, 2, 3, \ldots \quad \text{with} \;\; \mathbf{s}_0 = \mathbf{g}_0 = \mathbf{0}. \tag{B.1}$$

The slope of $f(\mathbf{x})$ in the search direction is denoted by $f'(x)$. It may happen that the function has a positive slope (i.e. there is no guarantee that the slope is negative), in which case I use the direction of steepest descent (i.e. $\mathbf{s}_i = -\mathbf{g}_i$) for that single iteration. The gradient of the cost function is computed using the back-propagation rule.

The conjugate gradient method can be shown to have quadratic convergence properties, which is much better than the originally proposed method of steepest descent. Also, a quadratic optimisation problem can be solved in n (the dimensionality of the problem) iterations if the line searches are exact. However, the cost function of a neural network is typically not quadratic in the weights, so the procedure will not converge in n steps. Rather, it will be used in an iterative manner. Sometimes, the algorithm is used with *restarts* (i.e. starting again with steepest descent) every n iterations, however this is not done in the present implementation, since typically n is very large and the Polack-Ribiere version which is used here, has a natural tendency to automatically restart whenever progress is slow, since in that case $\mathbf{g}_{i-1} \approx \mathbf{g}_i$ and consequently $\mathbf{s}_i \approx -\mathbf{g}_i$ in eq. (B.1).

B.2 Line search

The object of the line search is to minimize the objective function along the direction $\mathbf{s}$. One dimensional minimization problems are more tractable than multidimensional ones, but it is still not in general possible to find the global minimum and many epochs may be necessary to achieve close approximations to the minimum. Since in our case the optimisation problem is usually non-quadratic, it is not sensible to expend huge computational resources in doing very accurate line searches (remember that one function evaluation requires a full sweep through the entire training set). Consequently, the goal of the line search will merely be: to get significantly closer to a local minimum.

I use the two sided Wolfe-Powell conditions [Fletcher 1987] to determine whether a new point is significantly better than the current point. The Wolfe-Powell conditions consists of 2 inequalities. The first inequality requires that the absolute slope at the new point is smaller (in a absolute sense) than some fraction of the magnitude of the slope at the current point. This will in general guarantee that we have moved closer to a local extremum by a non-vanishing amount. The condition is $|f'(\mathbf{x}_i)| \leq \sigma |f'(\mathbf{x}_{i-1})|$, where I use $\sigma = 0.5$. In fig. B.1 this condition implies that acceptable points must lie between the intersections of lines $a1$ and $a2$ with f. The second inequality requires that a substantial fraction of the decrease expected from the current slope is being achieved, thus guaranteeing a decrease in

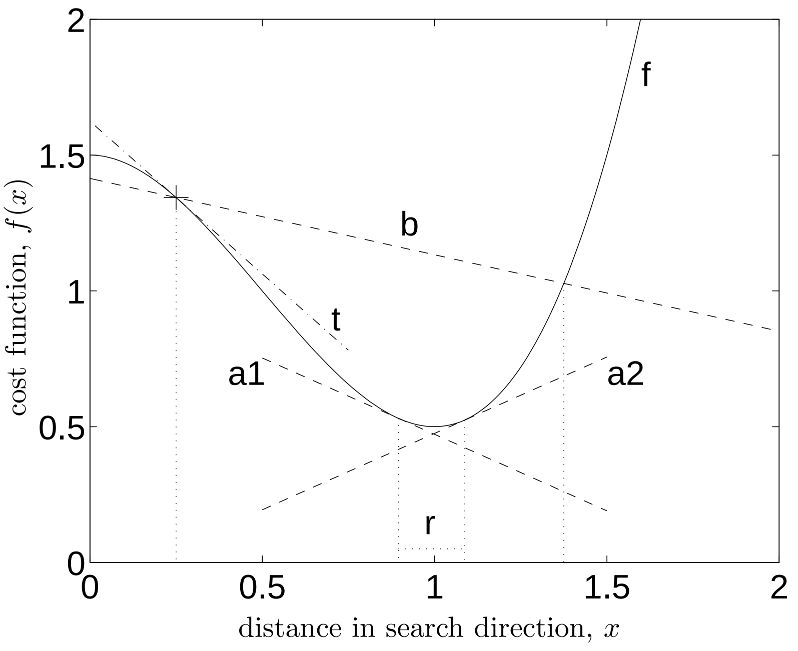


Figure B.1: Illustration of the Wolfe-Powell conditions. The current point is marked by '+' at $x = 0.25$. The cost function is f and a tangent t is drawn at the current point. The intersections of the lines $a1$, $a2$ and b with the function f defines the region r of acceptable points.

the function value and avoiding huge steps with negligible function decrease. The condition is $f(\mathbf{x}_i) \leq f(\mathbf{x}_{i-1}) + \rho(\mathbf{x}_i - \mathbf{x}_{i-1})f'(\mathbf{x}_{i-1})$, where I use $\rho = 0.25$. In fig. B.1, only smaller values of x than where the line labeled b intersects with f are acceptable. In the figure, the final region of acceptable points is marked by r. It can be shown that acceptable points always exist for a continuous f, which is bounded from below, when $0 < \rho < \sigma < 1$.

The line search iteratively improves on the guesses for the minimum until an acceptable point is found. The pre-conditions (which are guaranteed to be satisfied by the conjugate gradient routine) are: the current point is `x0`, `f(x0)` and `g(x0)` are computed, `f'(x0)` $\leq$ `0` is guaranteed, and `max` is undefined. Fig. B.2 represents this algorithm in pseudo-code.

An additional constraint is added to the code in fig. B.2, that a maximum of 20 epochs are allowed, to avoid infinite loops because of numerical inaccuracies; in this case the line search fails and the best approximation so far is returned. The notation `a2(x2)`, refers to the condition implied by the line $a2$ in fig. B.1, at the point `x2`, and it returns either `violated` or `satisfied`. The solution is always bracketed between `x1` and `max` (whenever `max` is defined).

The initial step-size guess for the current iteration is computed in the step size from the

```
x1 := x0 and x2 := initial guess
compute f(x2) and g(x2)
loop {
    while { a2(x2) or b(x2) are violated } do {
        max := x2
        x2 := interpolate(x1,x2)
        compute f(x2) and g(x2)
    }
    if { a1(x2) is satisfied } do { return x2 }              [SUCCESS!]
    x3 := extrapolate(x1, x2, max)
    x1 := x2 and x2 := x3
    compute f(x2) and g(x2)
}
```

Figure B.2: Pseudo-code for line search algorithm

previous iteration using the *slope ratio* method. The length of the previous step, δ_{i-1}, is multiplied by the ratio of previous and current slopes (but a maximum of 100):

$$\delta_i = \min(\frac{f'(x_{i-1})}{f'(x_i)}, 100)\delta_{i-1} \quad \texttt{x2} = \texttt{x0} + \delta_i \quad \text{with} \quad \delta_0 = 1/(1 + |f'(x_0)|). \tag{B.2}$$

The limit of a factor of 100 for the relative step size is introduced to avoid errors when $f'(x_i)$ is numerically close to zero. This simple heuristic slope ratio method seems to work very well in practice. Typically only about 1.3 epochs per iteration are needed, indicating that the initial guess often lies in the region of acceptable points.

Interpolation between `x1` and `x2` is done by finding the minimum of a quadratic or cubic polynomial, fit at `f(x1)`, `f(x2)`, `f'(x1)` and `f'(x2)`. A cubic is used unless `f(x2)` > `f(x0)`, in which case `f'(x2)` is ignored and a quadratic is used, since in this case `x2` can be very far from the minimizer, and it is feared that the derivative out there may be misleading. If the interpolating polynomial does not have a (local) minimum inside the interval, bisection is used. In addition, if the point is within 10% of the interval length from an interval endpoint, then the new guess is moved to this distance in order to ensure exploration and avoid stagnation by repeated evaluation at essentially identical points.

Extrapolation from the `x1` to `x2` interval is done using a cubic, fit at `f(x1)`, `f(x2)`, `f'(x1)` and `f'(x2)`. If the cubic does not have a minimum, if the minimum does not correspond to an extrapolation or if the extrapolation is further than 3 times the interval length, then this point at 3 times the interval length is used is order to prevent uncontrolled extrapolation. If the new point is too close to the current point (within 10% of the interval length), this 10% length is used, to ensure exploration.

The conjugate gradient algorithm calls the above line search procedure for each of the directions that are computed. The entire minimization procedure terminates when a prespecified number of function evaluations have been performed, or when two subsequent line searches have failed (this will usually happen for reasons of numerical inaccuracies when we are very close to a local minimum). All computations are done using double precision arithmetic.

B.3 Discussion

Other implementations of conjugate gradient methods have been proposed. Unfortunately it is quite rare that implementational details are given or discussed. It is also extremely rare to see good empirical evidence of the performance of optimisation procedures.

One paper which does address some of these issues is [Møller 1990]. He presents an implementation called Scaled Conjugate Gradients (SCG), which avoids the line search by using a finite differences approximation to the Hessian and computing a step size with a Newton like method. The SCG always uses 2 epochs per iteration, since the function and gradients are needed at the current point, and the gradient (which cannot be computed without first computing the function value) somewhere in the vicinity of the current point is needed for the finite differences approximation to the Hessian. The SCG method is thus very like a version of the present method modified to require at least one interpolation. The advantages of the present method are:

- a good guess (by the slope ratio algorithm) is used for the extra point needing evaluation, instead of a very local point for the finite differences method, thus giving more relevant information about the function.
- the SCG method ignores half of the function values which are computed, thus discarding useful information.
- the initial guess may be accepted, thus avoiding extra epochs (although it is conceivable that SCG might benefit from perhaps more accurate new points). Typically only 1.3 epochs per iteration are needed as opposed to CGS's 2.0.
- interpolation is (typically) done using a cubic (rather than SCG's quadratic) polynomial.
- rigorous criteria are used to ensure progress at each iteration.

For these reasons I prefer the present implementation, but it is conceivable that there are no significant differences between the methods in practice.

```c
#include <math.h>
#include <stdlib.h>
#include "util.h"

extern int  no_wts;                                 /* length of weight vector */
extern real *w;                                                /* weight vector */
extern real fgeval(real *dw);   /* evaluate function and partial derivatives */

real *dw1, *dw2,                /* two arrays of derivatives for all weights */
     *s;                           /* search direction used for linesearches */
int  nfgeval;  /* number of func/grad evals. so far; incremented by fgeval() */

int lns(f1, z, d1)
  real *f1,                                       /* current function value */
       *z,                                        /* guess for initial step */
       d1;                                                         /* slope */
{
  real RHO = 0.25, SIG = 0.5, INT = 0.1, EXT = 3.0;
  int  MAX = 20;
  real d2, d3, f2, f3, z2, z3, A, B, max = -1.0;
  int  i, k;

  for (i=0; i<no_wts; i++) w[i] += *z*s[i];                /* update weights */
  f2 = fgeval(dw2);
  d2 = 0.0; for (i=0; i<no_wts; i++) d2 += dw2[i]*s[i];
  f3=*f1; d3=d1; z3=-*z;              /* initialize point 3 equal to point 1 */
  k = nfgeval + MAX; while (nfgeval < k) { /* allow limited amount of search */
    while (((f2 > *f1+*z*RHO*d1) || (d2 > -SIG*d1)) && (nfgeval < k)) {
      max=*z;                                         /* tighten the bracket */
      if (f2 > *f1) z2=z3-(0.5*d3*z3*z3)/(d3*z3+f2-f3);     /* quadratic fit */
      else {                                                    /* cubic fit */
        A = 6.0*(f2-f3)/z3+3.0*(d2+d3);
        B = 3.0*(f3-f2)-z3*(d3+2.0*d2);
        z2 = (sqrt(B*B-A*d2*z3*z3)-B)/A;   /* numerical error possible - ok! */
      }
      if (z2 != z2) z2 = z3/2;               /* if z2 is NaN then bisection */
      if (z2 > INT*z3) z2 = INT*z3;      /* bound solution away from current */
      if (z2 < (1.0-INT)*z3) z2 = (1.0-INT)*z3;        /* bound away from z3 */
      *z += z2;                                  /* update absolute stepsize */
      for (i=0; i<no_wts; i++) w[i] += z2*s[i];            /* update weights */
      f2 = fgeval(dw2);
      d2 = 0.0; for (i=0; i<no_wts; i++) d2 += dw2[i]*s[i];
      z3 -= z2;                 /* z3 is now relative to the location of z2 */
    }
    if (d2 > SIG*d1) { *f1 = f2; return 1; }                      /* SUCCESS */
    A = 6.0*(f2-f3)/z3+3.0*(d2+d3);              /* make cubic extrapolation */
    B = 3.0*(f3-f2)-z3*(d3+2.0*d2);
    z2 = -d2*z3*z3/(B+sqrt(B*B-A*d2*z3*z3));    /* num. error possible - ok! */
    if (z2 != z2)                                             /* z2 is NaN? */
      z2 = (max < -0.5) ? *z*(EXT-1.0) : 0.5*(max-*z);          /* bisection */
    else if (z2 < 0.0)                /* minimum is to the left of current? */
      z2 = (max < -0.5) ? *z*(EXT-1.0) : 0.5*(max-*z);          /* bisection */
    else if ((max > -0.5) && (z2+*z > max))           /* extrap. beyond max? */
      z2 = 0.5*(max-*z);                                        /* bisection */
    else if ((max < -0.5) && (z2+*z > *z*EXT))        /* extrap. beyond EXT? */
      z2 = *z*(EXT-1.0);                                /* set to extrap. limit */
    else if (z2<-z3*INT)                       /* too close to current point? */
      z2 = -z3*INT;
    else if ((max > -0.5) && (z2 < (max-*z)*(1.0-INT))) /* too close to max? */
      z2 = (max-*z)*(1.0-INT);
    f3=f2; d3=d2; z3=-z2;                              /* swap point 2 and 3 */
    *z += z2;
    for (i=0; i<no_wts; i++) w[i] += z2*s[i];              /* update weights */
    f2 = fgeval(dw2);
    d2 = 0.0; for (i=0; i<no_wts; i++) d2 += dw2[i]*s[i];
  }
  *f1 = f2;
  return 0;                                               /* linesearch failed */
}

int conj(iter, epoch, restart, costvalue)
  int  *iter,       /* "iter" and "epoch" indicates the maximum number of... */
       *epoch,  /* iterations or epochs allowed. Actual numbers are returned */
       *restart;      /* if (*restart) then restart CG with steepest descent */
  real *costvalue;                    /* return the value of the costfunction */
{
  static int  ls_failed = 0;               /* set to 1 if line search failed */
  int         j,                                   /* miscellaneous counter */
              cur_iter = 0;                     /* counts current iteration */
  static real fun, slope1, step;
  real        *tmp, y, z, q, slope2;
  extern int  TIMEOUT;             /* is set to one when SIGVTALRM is caught */

  nfgeval = 0;   /* global int "number of function and gradient evaluations" */
  if (*restart) {            /* start by using direction of steepest descent */
    fun = fgeval(dw1);
    slope1 = 0.0;
    for (j=0; j<no_wts; j++) { s[j] = -dw1[j]; slope1 -= s[j]*s[j]; }
    step = -1.0/(slope1-1.0);          /* set initial step-size to 1/(|s|+1) */
    *restart = 0;          /* probably we won't want to restart on next call */
  }
  while ((!TIMEOUT) && ((*epoch == 0) || (nfgeval < *epoch)) &&
         ((*iter == 0) || (cur_iter < *iter))) {
    cur_iter++;
    if (lns(&fun, &step, slope1)) {              /* if line search succeeded */
      y = z = q = 0.0; for (j=0; j<no_wts; j++)
        { y += dw2[j] * dw2[j]; z += dw1[j] * dw2[j]; q += dw1[j] * dw1[j]; }
      y = (y-z)/q;
      for (j=0; j<no_wts; j++) s[j] = y*s[j]-dw2[j];        /* new direction */
      tmp = dw2; dw2 = dw1; dw1 = tmp;                   /* swap derivatives */
      slope2 = 0.0; for (j=0; j<no_wts; j++) slope2 += dw1[j]*s[j];
      if (slope2 > 0.0) {     /* must be negative, else use steepest descent */
        slope2 = 0.0;
        for (j=0; j<no_wts; j++) { s[j] = -dw1[j]; slope2 -= s[j]*s[j]; }
      }
      step *= (slope1/slope2 > 100.0) ? 100.0 : slope1/slope2; slope1 = slope2;
      ls_failed = 0;
    } else {                                              /* line search failed */
      if (ls_failed)          /* break if previous failed, else try steepest */
        { *epoch = nfgeval; *iter = cur_iter; *costvalue = fun; return 0; }
      dw1 = dw2;                                         /* swap derivatives */
      slope1 = 0.0;
      for (j=0; j<no_wts; j++) { s[j] = -dw1[j]; slope1 -= s[j]*s[j]; }
      step = -1.0/(slope1-1.0);     /* set new step-size guess to 1/(|s|+1) */
      ls_failed = 1;
    }
  }
  *epoch = nfgeval; *iter = cur_iter; *costvalue = fun; return 1;
}
```

Bibliography

Abramowitz, M. and Stegun, I. A. (1964). *Handbook of mathematical functions*, volume 55 of *Applied Mathematics Series.* National bureau of Standards, Washington.

Barber, D. and Williams, C. K. I. (1996). Gaussian processes for Bayesian classification via hybrid Monte Carlo. Submitted to NIPS 9.

Box, G. E. P. and Tiao, G. C. (1992). *Bayesian Inference in Statistical Analysis.* John Wiley and Sons inc.

Breiman, L. (1994). Bagging predictors. Technical Report 421, Department of Statistics, University of California, Berkeley, California 94720.

Cleveland, W. S. (1979). Robust locally-weighted regression and smoothing scatterplots. *J. Am. Statist. Assoc.*, 80:580–619.

Corke, P. I. (1996). A robotics toolbox for MATLAB. *IEEE Robotics and Automation Magazine*, 3(1):24–32.

Dietterich, T. (1996). Proper statistical tests for comparing supervised classification learning algorithms. Unpublished.

Duane, S., Kennedy, A. D., and Roweth, D. (1987). Hybrid Monte Carlo. *Physics Letters B*, 195:216–222.

Fletcher, R. (1987). *Practical methods for optimization.* John Wiley and Sons inc., second edition.

Friedman, J. H. (1991). Multivariate adaptive regression splines (with discussion). *Annals of Statistics*, 19:1–141. Source code: `http://lib.stat.cmu.edu/general/mars3.5`.

Geman, S., Bienenstock, E., and Doursat, R. (1992). Neural networks and the bias/variance dillemma. *Neural Computation*, 4(1):1–58.

Gibbs, M. and MacKay, D. J. (1996). Efficient implementation of gaussian processes. Available through `http://wol.ra.phy.cam.ac.uk/mackay/`.

Gilks, W. R. and Wild, P. (1992). Adaptive rejection sampling for Gibbs sampling. *Applied Statistics*, 41:337–348.

Girosi, F., Jones, M., and Poggio, T. (1995). Regularization theory and neural networks architechtures. *Neural Computation*, 7(2):219–269.

Golub, G. H. and van Loan, C. F. (1989). *Matrix computations.* Johns Hopkins University Press, second edition.

Guérin-Dugué, A. et al. (1995). Deliverable R3-B4-P - Task B4: Benchmarks. Technical report, Elena-NervesII: Enhanced Learning for Evolutive Neural Architecture, ESPRIT-Basic Research Project Number 6891. `ftp://ftp.dice.ucl.ac.be/pub/-neural-nets/ELENA/Benchmarks.ps.Z`.

Hansen, L. K. and Salamon, P. (1990). Neural netork ensembles. *IEEE transactions on Pattern recognition and Machine Intelligence*, 12:993–1001.

Harrison, Jr., D. and Rubenfeld, D. L. (1978). Hedonic housing prices and the demand for clean air. *Journal of Environmental Economics and Management*, 5:81–102.

Hastie, T. and Tibshirani, R. (1990). *Generalized Additive Models.* Number 43 in Monographs on Statistics and Applied Probability. Chapman and Hall.

Hastie, T. and Tibshirani, R. (1996). Discriminant adaptive nearest neighbor classification and regression. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems*, volume 8, pages 290–297. MIT Press. `ftp://playfair.stanford.edu/pub/hastie/nips95.ps.Z`.

Horowitz, A. M. (1991). A generalized guided Monte Carlo algorithm. *Physics Letters B*, 268:247–252.

Koehler, J. R. and Owen, A. B. (1996). Computer experiments.

Larsen, J. and Hansen, L. K. (1995). Empirical generalization assessment of neural network models. In *IEEE Workshop on Neural Networks for Signal Processing*, Boston.

Le Cun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In Touretzky, D., editor, *Advances in Neural Information Processing Systems*, volume 2, pages 598–605, San Mateo. (Denver 1989), Morgan Kaufmann.

Lindman, H. R. (1992). *Analysis of Variance in Experimental Design.* Springer-Verlag.

Lowe, D. G. (1995). Similarity metric learning for a varable–kernel classifier. *Neural Computation*, 7(1):72–85.

MacKay, D. J. C. (1992a). Bayesian interpolation. *Neural Computation*, 4(3):415–447.

MacKay, D. J. C. (1992b). A practical Bayesian framework for backpropagation networks. *Neural Computation*, 4(3):448–472.

Michie, D., Spiegelhalter, D., and Taylor, C., editors (1994). *Machine Learning, Neural and Statistical Classification.* Ellis Horwood Limited.

Møller, M. F. (1990). A scaled conjugent gradient algorithm for fast supervised learning. *Neural Networks*, 6:525–533.

Murphy, P. M. and Aha, D. W. (1994). UCI Repository of machine learning databases. Technical report, Department of Information and Computer Science, University of California, Irvine, CA. `http://www.ics.uci/~mlearn/MLRepository.html`.

Neal, R. M. (1993). Probabilistic inference using Markov chain Monte Carlo methods. Technical Report CRG-TR-93-1, Department of Computer Science, University of Toronto.

Neal, R. M. (1994). An improved acceptance procedure for the hybrid Monte Carlo algorithm. *Journal of Computational Physics*, 111:194–203.

Neal, R. M. (1996). *Bayesian Learning for Neural Networks.* Springer Verlag, New York. Revised version of Ph.D. thesis from Graduate Department of Computer Science, University of Toronto.

O'Hagan, A. (1978). On curve fitting and optimal design for regression. *Journal of the Royal Statistical Society, B*, 40:1–42. (with discussion).

O'Hagan, A. (1994). *Bayesian inference*, volume 2B of *Kendall's advanced theory of Statistics.* Edward Arnold, first edition.

Poggio, T. and Girosi, F. (1990). Networks for approximation and learning. *Proceedings of IEEE*, 78(9):1481–1497.

Prechelt, L. (1994). PROBEN1 – A set of neural network benchmark problems and benchmarking rules. Technical report, Fakultät für Informatik, Universität Karlsruhe. Doc: `pub/papers/techreports/1994/1994-21.ps.Z`, Data: `/pub/neuron/-Proben1.tar.gz` from `ftp.ira.uka.de`.

Prechelt, L. (1995). *Konstruktive neurale Lernverfahren auf Parallelrechnern.* PhD thesis, Fakultät für Informatik, Universität Karlsruhe. German language.

Prechelt, L. (1996). A study of experimental evaluations of neural network learning algorithms: Current research practice. *Neural Networks*, 9(3):457–462.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992). *Numerical Recipes in C.* Cambridge University Press, second edition.

Quinlan, J. R. (1993). Combining instance-based and model-based learning. In *Machine Learning: Proceedings of the Tenth International Conference*, Amherst, Massachusetts. Morgan Kaufmann.

Rasmussen, C. E. (1996). A practical Monte Carlo implementation of Bayesian learning. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems*, volume 8, pages 598–604. MIT Press. `ftp://ftp.cs.toronto.edu/pub/carl/practical_mc_nips.ps.gz`.

Rasmussen, C. E., Neal, R. M., Hinton, G. E., van Camp, D., Revow, M., Ghahramani, Z., Kustra, R., and Tibshirani, R. (1996). *The DELVE Manual.* University of Toronto. `http://www.cs.utoronto.ca/~delve`.

Skilling, J. (1989). The eigenvalues of mega–dimensional matrices. In Skilling, J., editor, *Maximum Entropy and Bayesian Methods, Cambridge 1988*, pages 455–466, Dordrecht. Kluwer.

Skilling, J. (1993). Bayesian numerical analysis. In Grandy, Jr., W. T. and Milonni, P., editors, *Physics and Probability*, Cambridge. C.U.P.

Tetko, I. V., Livingstone, D. J., and Luik, A. I. (1995). Neural network studies. 1. comparison of overfitting and overtraining. *J. Chem. Info. Comp. Sci.*, 35:826–833.

Thodberg, H. H. (1996). A review of Bayesian neural networks with an application to near infrared spectroscopy. *IEEE transactions on Neural Networks*, 7:56–72.

Wahba, G. (1990). *Spline models for Observational Data*, volume 59 of *Series in Applied Mathematics*. SIAM, Philadelphia.

Williams, C. K. I. (1996). Regression with Gaussian processes. *Annals of Mathematics and Artificial Intelligence.* to appear, `ftp://sparc-server.aston.ac.uk/neural/willicki/manna.ps.Z`.

Williams, C. K. I. and Rasmussen, C. E. (1996). Gaussian processes for regression. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems*, volume 8, pages 290–297. MIT Press. `ftp://ftp.cs.toronto.edu/pub/carl/gauss_process_nips.ps.gz`.