MCMC Methods for MLP-network and Gaussian Process and Stuff– A documentation for Matlab Toolbox MCMCstuff

Jarno Vanhatalo and Aki Vehtari
Laboratory of Computational Engineering,
Helsinki University of Technology,
P.O.Box 9203, FIN-02015 TKK, Espoo, Finland
{Jarno.Vanhatalo,Aki.Vehtari}@tkk.fi

May 22, 2006

Version 2.1

Abstract

MCMCstuff toolbox is a collection of Matlab functions for Bayesian inference with Markov chain Monte Carlo (MCMC) methods. This documentation introduces some of the features available in the toolbox. Introduction includes demonstrations of using Bayesian Multilayer Perceptron (MLP) network and Gaussian process in simple regression and classification problems with a hierarchical automatic relevance determination (ARD) prior for covariate related parameters. The regression problems demonstrate the use of Gaussian and Student's t-distribution residual models and classification is demonstrated for two and three class classification problems. The use of Reversible jump Markov chain Monte Carlo (RJMCMC) method and ARD prior are demonstrated for input variable selection.

Contents

Preface

Most of the code in the toolbox has been written by Aki Vehtari in the Laboratory of Computational Engineering at Helsinki University of Technology (TKK). The toolbox is based on Netlab software package[1] by Ian T. Nabney and Flexible Bayesian Modeling (FBM) software package[2] by Radford M. Neal. Currently there is code written by (in alphabetical order) Toni Auranen, Christopher M Bishop, James P. LeSage, Ian T Nabney, Radford Neal, Carl Edward Rasmussen, Simo Särkkä and Jarno Vanhatalo. For publication of the code as a toolbox, help texts of the functions were checked and improved and this document was written by Jarno Vanhatalo with help from Aki Vehtari. Special thanks are directed to Prof. Jouko Lampinen who helped compiling some Windows mex-files.

Changes for the second version

The documentation has been improved from the first version. Number of new features are discussed and demonstrated and the chapter structure has been modified. All the same information is included as in the first version, but the order in which they are discussed may have changed. The major changes for the version 2.0 of this documentation are:

- Discussion on selecting starting values for MCMC sampling
- Discussion about Reversible Jump MCMC method and its implementation in the software.
- Discussion on residual models in regression problems. Here we handle Gaussian and Student's t-distribution models.
- Demonstration program for 2-class classification problem for MLP.
- Demonstration program for 3-class classification problem for MLP.
- Demonstration program for input variable selection with RJMCMC for MLP.
- Demonstration program for regression problem in MLP with Student's t-distribution model for residual.
- Demonstration program for 2-class classification problem with Gaussian process.
- Demonstration program for input variable selection with RJMCMC for Gaussian process.
- Demonstration program for regression problem in GP with Student's t-distribution model for residual.

[1] http://www.ncrg.aston.ac.uk/netlab/

[2] http://www.cs.toronto.edu/~radford/fbm.software.html

Changes for the version 2.1

The major change for the version 2.1 is the construction of Student's t distribution residual model for Gaussian process regression. The residual model is different and new section 4.1.3, where the new model is explained, is added to the documentation. The demonstration program `demo_tgp` is changed to correspond the new model. Also few bugs in the code have been fixed. See "Changelog" in the toolbox for more detailed information.

1 Introduction

The purpose of this document is to introduce the software package and help people interested in the topic to use the software in their own work and possibly modify or extend the features.

The MCMC methods for MLP and GP software package has been written using Matlab and C programming languages and works with Matlab versions 6.* and 7.* as a toolbox. The functions and code used in the software implement and follow the approach used in Netlab software package[3] by Ian T. Nabney and Flexible Bayesian Modeling (FBM) software package[4] by Radford M. Neal. A more detailed treatment of FBM and Netlab can be found in References (Neal, 1993, 1996; Nabney, 2001). The software was used, for example, in the works by Lampinen and Vehtari (2001), and Vehtari and Lampinen (2002).

Basic design of this toolbox is based on Netlab. However, this toolbox is not compatible with Netlab, because the option handling has been changed to use structures similar to current default in Mathworks' toolboxes. Furthermore, the code in this toolbox has been streamlined and optimized for faster computation, code has been extended to include some of the features present in FBM and some other features. Some of the most computationally critical parts have been coded in C. For easier introduction to MLP and GP's Netlab is better suited, especially because of the accompanying text book (Nabney, 2001). Furthermore, since code was originally written mainly for Aki Vehtari's research purposes it is not as easy to use as it could be.

All the features of the software package are not (yet) treated in this documentation version. This is mainly an overview how to use the toolbox in simple regression and classification problems. The features that are discussed here are:

- Bayesian learning for MLP in a regression and classification problems.
- Bayesian learning for a Gaussian process in a regression and 2-class classification problem.
- Gaussian, Student's t- and Inverse-Gamma hierarchical prior structures.

[3] http://www.ncrg.aston.ac.uk/netlab/

[4] http://www.cs.toronto.edu/~radford/fbm.software.html

- A Gaussian hierarchical prior structure with Automatic Relevance Determination (ARD).
- Residual model in regression problem with Gaussian and Student's t-distribution.
- Metropolis-Hastings, hybrid Monte Carlo, Gibbs sampling and Reversible jump Markov chain Monte Carlo (RJMCMC) sampling methods.
- Input variable selection in MLP and GP using RJMCMC sampling.

The features implemented in the software but not discussed in this document version are:

- Residual models with Laplace distribution
- Covariate dependent grouped noise model

After the main functions and features of the software have been introduced, the software is demonstrated using simple examples illustrating the use of package in regression and classification problems. After the demonstration programs, Markov Chain Monte Carlo method is reviewed in more detail, discussing Metropolis-Hastings, hybrid Monte Carlo, Gibbs and Reversible jump Markov chain Monte Carlo sampling. The regression and 2-class classification with MLP network are treated for comparison with Gaussian process. All the optional features of functions discussed are not treated here. These optional features are mentioned in the context of functions they are related, but the use of all features is not discussed in this document version. The demonstration problems are treated completely in the section 3.3.

In order to diagnose the convergence of distribution samples, users of MCMCstuff toolbox should also download a Markov Chain Monte Carlo diagnostics toolbox available at http://www.lce.hut.fi/research/compinf/mcmcdiag/

2 A Bayesian approach

The key principle of Bayesian approach is to construct the posterior probability distribution for the unknown entities in a model given the data sample. To use the model, marginal distributions are constructed for all those entities that we are interested in, that is, the end variables of the study. These can be parameters in parametric models, or predictions in (non-parametric) regression or classification tasks.

Use of the posterior probabilities requires explicit definition of the prior probabilities for the parameters. The posterior probability for the parameters in a model M given data D is, according to Bayes' rule,

$$p(\theta|D, M) = \frac{p(D \mid \theta, M)p(\theta \mid M)}{p(D \mid M)}, \tag{1}$$

where $p(D \mid \theta, M)$ is the likelihood of the parameters θ, $p(\theta \mid M)$ is the prior probability of θ, and $p(D \mid M)$ is a normalizing constant, called evidence of the model M. The term M denotes all the hypotheses and assumptions that are made in defining the model, like

a choice of MLP network, specific residual model etc. All the results are conditioned on these assumptions, and to make this clear we prefer to have the term M explicitly in the equations. In this notation the normalization term $p(D \mid M)$ is directly understandable as the marginal probability of the data, conditioned on M. Integrating over everything, the chosen assumptions M and prior $p(\theta \mid M)$, comprise

$$p(D \mid M) = \int_{\theta} p(D \mid \theta, M) p(\theta \mid M) d\theta. \tag{2}$$

When having several models, $p(D \mid M_l)$ is the marginal likelihood of the model l, which can be used in comprising the probabilities of the models, hence the term evidence of the model.

The result of Bayesian modeling is the conditional probability distribution of unobserved variables of interest, given the observed data. In Bayesian MLP the natural end variables are the predictions of the model for new inputs, while the posterior distribution of weights is rarely of much interest. The posterior predictive distribution of output y^{new} for the new input x^{new} given the training data $D = \{(x^{\mathbf{1}}, y^{\mathbf{1}}), ..., (x^n, x^n)\}$, is obtained by integrating the predictions of the model with respect to the posterior distribution of the model

$$p(y^{\text{new}} \mid x^{\text{new}}, D, M) = \int p(y^{\text{new}} \mid x^{\text{new}}, \theta) p(\theta \mid D, M) d\theta, \tag{3}$$

where θ denotes all the model parameters and hyperparameters of the prior structures.

The probability model for the measurements, $p(y \mid x, \theta, M)$, contains the chosen approximation functions and residual models. It defines also the likelihood part in the posterior probability term, $p(\theta \mid D, M) \propto p(D \mid \theta) p(\theta \mid M)$. The probability model in a regression problem with additive noise is

$$y = f(x; \theta_w) + e, \tag{4}$$

where $f()$ is, for example, the MLP function

$$f(x, \theta_w) = b^2 + w^2 \tanh(b^1 + w^1 x). \tag{5}$$

The θ_w denotes all the parameters $w^{(1)}, b^{(1)}, w^{(2)}, b^{(2)}$, which are the hidden layer weights and biases, and the output weights and biases, respectively. The random variable e is the residual.

In *Markov chain Monte Carlo* (MCMC) the complex integrals of the expectation,

$$\hat{y}^{\text{new}} = E y^{\text{new}} \mid x^{\text{new}}, D, M = \int f(x^{\text{new}}, \theta) p(\theta \mid D, M) d\theta, \tag{6}$$

are approximated via drawing samples from the joint probability distribution of all the model parameters and hyperparameters. The integral in the equation (6) can be approximated using a sample of values $\theta^{(t)}$ drawn from the posterior distribution of parameters

$$\hat{y}^{\text{new}} \approx \frac{1}{N} \sum_{t=1}^{N} f(x^{\text{new}} | \theta^{(t)}, M). \tag{7}$$

The determination of output for new input data is done by forward propagating the data through N different networks, with parameters drawn from the posterior distribution, and taking the mean of all N outputs.

3 MLP-networks in MCMCstuff

3.1 Network architecture and prior structures

The software supports MLP networks with single hidden layer. A network with linear output function, corresponding to the MLP function in equation (5), can be used for regression problems. In two class classification problems the probability that a binary-valued target, y, has value 1 can be computed with the logistic output function

$$p(y = 1|x, \theta_\omega) = \frac{1}{1 + \exp(-f(x, \theta_\omega)}, \tag{8}$$

and in many class classification problem the probability that a class target, y, has value j can be computed with the softmax output function

$$p(y = j|x, \theta_\omega) = \frac{\exp(f_j(x, \theta_\omega))}{\sum_{l=1}^{M} \exp(f_l(x, \theta_\omega))}. \tag{9}$$

The prior system for all the network parameters is discussed in the section 3.1.2. In the case of regression problem a prior structure can be constructed also for the residual, equation (4), in the same manner as for the network parameters.

3.1.1 Constructing an MLP-network

An MLP network is created and its weights initialized to zero using the function `mlp2`, which has the syntax

```
net = mlp2(type, nin, nhid, nout)
```

Here `nin` is the number of input vectors, `nhid` is the number of hidden layers and `nout` number of outputs. `type` is the problem model for which the network is used. Currently it has the following options

`mlp2r`	for regression problems using linear output function
`mlp2b`	for classification problems using logistic output function
`mlp2c`	for classification problems with more than 2 classes using softmax output function

The function `mlp2` returns a structure containing the following fields:

`type`	string describing the network type
`nin`	number of inputs
`nhid`	number of hidden units
`nout`	number of outputs
`nwts`	total number of weights and biases
`w1`	first layer weight matrix, with dimensions nin× nhid
`b1`	first layer bias vector, with dimensions 1× nhid
`w2`	second layer weight matrix, with dimensions nhid× nout
`b2`	second layer bias vector, with dimensions 1× nout

Although the `mlp2` sets all the weights and biases initially to zero they can be initialized also afterward. The function `mlp2pak` combines all the components of weight matrices and bias vectors into a single row vector. The `mlp2unpak` function recombines the row vector to corresponding weight matrices and bias vectors. The facility to switch between these two representations for the network parameters is useful, for example, in training the network by error function minimization. The syntax's for these two functions are:

```
w = mlp2pak(net)
net = mlp2unpak(net,w)
```

where w is a row vector containing weights and biases.

3.1.2 The prior structure for network parameters

The prior structure of networks used in the software is discussed in detail by Lampinen and Vehtari (2001). The main idea is to construct a hierarchical prior system, where one can define the parameters of weight distribution as hyperparameters having their own distribution. The hierarchy can be constructed for every weight and bias as well as for every hyperparameter. The result is a hierarchical hyperparameter structure where upper and lower level parameters have influence on the posterior distribution of the parameter under inspection. A commonly used prior distribution for network parameters is Gaussian.

$$\omega_k \sim N(0, \alpha_k^2), \tag{10}$$

where ω_k represents the weights and biases of network and α_k^2 is the variance hyperparameter for given weight (or bias). The hyperparameter α^2 is given, for example, a conjugate inverse gamma hyperprior

$$\alpha_k^2 \sim \text{Inv-gamma}(\alpha_{\text{ave}}^2, \nu_\alpha). \tag{11}$$

Automatic Relevance Determination (ARD) prior is an automatic method for determining the relevance of inputs in MLP. The key idea of ARD is to create prior structure in which each group of weights connected to same input has its own hyperparameters. With separate hyperparameters, the weights from irrelevant inputs can have tighter priors than the ones connected to relevant inputs. A small prior variance for weights from irrelevant inputs reduces them towards zero more effectively than if having common larger variance for all the weights.

In *ARD* each group of weights ω_{kj} connected to an input x_k has common variance hyperparameter α_k^2. Variances α_k^2 are given an inverse Gamma hyperprior, $\alpha_k^2 \sim$ Inv-gamma$(\alpha_{\text{ave}}^2, \nu_\alpha)$, where the next level hyperparameter ν_α has to be given when creating priors and hyperparameter α_{ave}^2 either has to be given or sampled from the third level hyperparameters. This way there are three levels of hyperparameters, with distributions (Lampinen and Vehtari, 2001)

$$\omega_{kj} \sim N(0, \alpha_k^2) \tag{12}$$

$$\alpha_k^2 \sim \text{Inv-gamma}(\alpha_{\text{ave}}^2, \nu_\alpha) \tag{13}$$

$$\alpha_{\text{ave}}^2 \sim \text{Inv-gamma}(\alpha_0^2, \nu_{\alpha,\text{ave}}) \tag{14}$$

When using ARD the relevance measure of an input is related to the size of the weigths connected to that input. In linear models these weights define the partial derivatives of the output with respect to the inputs, which is equal to the predictive importance of the input (Lampinen and Vehtari, 2001), in the case of nonlinear MLP network the situation is, however, more complicated. Lampinen and Vehtari (2001) have discussed the ARD prior more carefully and it can be shown that the non-linearity of the input has larger effect on the relevance score of ARD than the predictive importance. The benefits of ARD are shown in demonstration program `demo_3class` in section 3.3.3 and in the section 3.2.3 the relevance of inputs is discussed with the help of reversible jump Markov chain Monte Carlo method.

3.1.3 Residual model

Usually we do not know the distribution of noise in the measured data, but as well as the network parameters can be modeled with hierarchical prior structure we can construct a hierarchical model also for the residual. In the predictions we can then integrate over the posterior distribution of noise parameters given the data. A commonly used Gaussian noise model is

$$e \sim N(0, \sigma^2), \tag{15}$$

where the variance σ^2 can be given a hyperprior similar to one discussed in the section 3.1.2. In this conjugate prior, σ^2 is sampled from inverse Gamma distribution $\sigma^2 \sim$ Inv-gamma(σ_0^2, ν_σ).

In the noise model in equation (15), the same noise variance is assumed for each sample. However, in practice this is not always a good assumption because the target distribution may contain error sources of non-Gaussian density. In heteroscedastic regression problems each sample $(\mathbf{x}^i, \mathbf{y}^i)$ can have different noise variance $(\sigma^2)^i$ governed by a common prior (Lampinen and Vehtari, 2001). This corresponds to prior structure similar to ARD

prior for network weights (compare to equations (12)-(14))

$$e^i \sim N(0, (\sigma^2)^i) \tag{16}$$
$$(\sigma^2)^i \sim \text{Inv-gamma}(\alpha^2_{\text{ave}}, \nu_\alpha) \tag{17}$$
$$\alpha^2_{\text{ave}} \sim \text{Inv-gamma}(\alpha^2_0, \nu_{\alpha,\text{ave}}). \tag{18}$$

In this parametrization the residual model is asymptotically same as Student's t-distribution with fixed degrees of freedom (Geweke, 1993; Lampinen and Vehtari, 2001; Gelman et al., 2004).

Student's t-distribution is more robust residual model than Gaussian. It is also possible to make degrees of freedom ν a hyperparameter with hierarchical prior. The conditional posterior of ν is not available in easy form and thus sampling from it can be done in several alternative ways. In demonstration we have used Gibbs sampling for discretized values of ν due it's simplicity. Residual model is then (Lampinen and Vehtari, 2001)

$$e \sim t_\nu(0, \sigma^2) \tag{19}$$
$$\nu = V j \tag{20}$$
$$j \sim U_d(1, J) \tag{21}$$
$$V1 \quad J = v_1, v_2, ..., v_J \tag{22}$$
$$\sigma^2 \sim \text{Inv-gamma}(\sigma_0, \nu_\sigma), \tag{23}$$

where $U_d(a, b)$ is a uniform distribution of integer values between a and b. The Student's t-distribution for residual model is demonstrated in the program `demo_tmlp` (section 3.3.5).

3.1.4 Creating a prior structure for network and residual

The prior structures supported in the software are

- `norm_p` for Gaussian prior
- `laplace_p` for prior of Laplace's distribution
- `t_p` for prior of Student's t-distribution
- `invgam_p` for inverse-gamma prior.

All the prior structures are constructed similarly and the same functions can be used for constructing prior for network parameters and residual model. As an example we discuss construction of Gaussian prior made with `norm_p` for network weights and `t_p` for

residual. The `invgam_p` is discussed in the section 4.1.4.

Creating a Gaussian hierarchical prior for network weights. The prior structure discussed in the section 3.1.2 can be constructed with the following function. The use of function for MLP weight prior and residual model is demonstrated in the section 3.3.1.

`net.p.w(i) = norm_p(pw),`

> Here `pw` is an array containing the hyperparameter values of prior distributions. The hyperparameter array has the form
>
> pw= $\{\alpha_k\ \alpha_{\text{ave}}\ \nu_\alpha\ \alpha_0\ \nu_{\alpha,\text{ave}}\ \}$.
>
> The parameters given in `pw` correspond the parameters in equations (12)-(14). If parameters are left empty the prior hierarchy is constructed only so far as there is parameters. If parameter α_k is a vector of length `net.nin` an ARD prior is constructed. Note that the hyperparameter given for `norm_p` is standard deviation α and not the variance α^2.

Function `norm_p` specifies the distribution for each hyperparameter and creates function handles to evaluate error and gradient for them; it returns a structure `q`, which has following fields:

first level	second level	third level	
`.f`	`.p.s.f`	`.p.s.p.s.f`	type of the distribution
`.fe`	`.p.s.fe`	`.p.s.p.s.fe`	fctn handle for error function
`.fg`	`.p.s.fg`	`.p.s.p.s.fg`	fctn handle for gradient function
`.a.s`	`.p.s.a.s`	`.p.s.p.s.a.s`	hyper-param. α_k α_{ave} or α_0
	`.p.s.a.nu`	`.p.s.p.s.a.nu`	hyper-param. ν_α or $\nu_{\alpha,\text{ave}}$

Here each column represents fields in `net.p.w{i}` (for weights and biases) or `net.p.r` (for residual). `p` represents always *prior information* for the parameters just below it, for example `net.p.w{1}.p` contains prior information for the first level hyperparameters of input weights. Usually in ARD there are hyperparameters only for α's, but hyperpriors for ν's are also possible, although, it seems that often there is not enough information in the data to update their posterior from prior. In *prior information* for `s` (or for `nu` also if we want) `a` represents *hyperparameter container* for a given parameter, where `s` and `nu` as α and ν respectively are contained. In this software the hyperparameter structure is continued until third level hyperparameters as seen in equations (12)-(14) and discussed by Lampinen and Vehtari (2001). The information of used distributions of a parameter and the information of error and gradient functions for that parameter are stored in the fields `f`, `fe` and `fg`. These fields contain strings that are used, for example, to form the function names for the full conditional distributions in Gibbs sampling (see page 63). The other prior structures are similar to the Gaussian with exception that the hierarchy does not go as high in level as in Gaussian. Lampinen and Vehtari (2001) have discussed the Student's t-distribution residual model in more detail.

A Gaussian hierarchical prior structure can easily be constructed also with a function `mlp2normp`. It is a utility to construct the Gaussian hierarchical prior structure for all

weights and biases. It calls function `norm_p` which creates the actual prior for each weight group.

```
net = mlp2normp(net,  pw),
```

where `net` is the network for which the priors are constructed and `pw` is an array containing the hyperparameters for each level. The hyperparameter array has the form

pw=	$\{\{\alpha_k$	α_{ave}	ν_α	α_0	$\nu_{\alpha,\text{ave}}$ }...	prior for the first layer weights
	$\{\alpha_k$	α_{ave}	ν_α	}...		prior for the first layer biases
	$\{\alpha_k$	α_{ave}	ν_α	α_0	$\nu_{\alpha,\text{ave}}$ }...	prior for the second layer weights
	$\{\alpha_k$	α_{ave}	ν_α	}}		prior for the second layer biases

where α_k denotes the initial value(s) for the hyperparameter(s). This array is based on the syntax in FBM.

The function also takes care of network indexes. The index fields `net.p.w(i).ii` for the network are created by calling function `mlp2index`. Indexes are needed when, for example, error functions are evaluated or Gibbs sampling is done (see page 63).

Scaling the hyperparameters. In hierarchical Gaussian prior it is also possible to do a scaling for input and hidden unit weight hyperparameters α_{ave} and α_0 according to the number of inputs or number of hidden units. This can be done by giving hyperparameters a value less than zero when calling function `mlp2normp`, that is, if hyperparameter $\alpha_{\text{ave}} < 0$, it is scaled to $\alpha_{\text{ave}} = \frac{-\alpha_{\text{ave}}}{K^{1/\nu_\alpha}}$, where K is the number of inputs if input weights hyperparameters are scaled and number of hidden units if hidden unit weights hyperparameters are scaled. Similarly if hyperparameter α_0 is given value less than zero it is scaled to $\alpha_0 = \frac{-\alpha_0}{K^{1/\nu_{\alpha,\text{ave}}}}$. The scaling of hyperparameters is treated in more detail by Lampinen and Vehtari (2001) and Neal (1996).

Creating a Student's t-distribution model for residual. The Student's t-distribution model is discussed in the section 3.1.3 and the structure discussed there can be constructed with following function. The use of this function is demonstrated in the section 3.3.5.

```
net.p.r = t_p(pw),
```

Here `pw` is an array containing the hyperparameter values of prior distributions. The hyperparameter array has the form

pw= $\{\sigma\ \nu\ \sigma_0\ \nu_\sigma\ \upsilon_1, \upsilon_2, ..., \upsilon_J\}$.

The parameters given in `pw` correspond the parameters in equations (19)-(23). If parameters are left empty the prior hierarchy is constructed only so far as there is parameters.

3.2 MCMC method for network parameters

The Markov chain Monte Carlo sampling for the hyperparameters is done from a full conditional distributions of the parameters given all the other parameters and data. As an example the distribution of first layer weight hyperparameters, α_{k^1}, above are sampled from

$$P(\alpha_{k^1} \mid D, \theta) = P(\alpha_{k^1} \mid \omega^1) \propto P(\omega^1 \mid \alpha_{k^1}) P(\alpha_{k^1}), \tag{24}$$

where ω^1 denotes the first layer weights. The actual sampling methods which are used to reach the desired distributions are Gibbs sampling (section 5.3) for hyperparameters, Hybrid Monte Carlo method (section 5.2) for weights and biases and RJMCMC for input variables (section 5.4). The methods are discussed shortly in section 5. A more complete treatment is given, for example, by Gilks et al. (1996), Nabney (2001), Neal (1993), Neal (1996), Green (1995), Liu (2001) and Robert and Casella (2004).

As discussed in the section 3.1.1 there are three different models in the software. The sampling is done similarly for all of them. The functions to do MCMC sampling are `mlp2r_mc`, `mlp2b_mc` and `mlp2c_mc` for models `mlp2r`, `mlp2b` and `mlp2c`, respectively .

The syntax for Monte Carlo sampling is as following:

```
[rec, net0, rstate] = mlp2r_mc(...
    opt, net, p, t, pp, tt, rec, rstate)
```

Here the input and output variables are as follows:

`opt`	an options structure for sampler.
`net`	a network for which the sampling is done.
`p`	a matrix containing the training input vectors in its rows.
`t`	a matrix containing the training target vectors on rows respective to the input vector rows in `p`.
`pp`	a matrix containing a test input data (optional).
`tt`	a matrix containing a test target data (optional).
`rec`	an old record from which to continue the sampling (optional).
`state`	the old state of random number generators used in samplers (optional).

The returned values are,

`rec`	the record structure containing samples and information from the sampling process.
`net0`	network with weights and hyperparameters from the last iteration.
`rstate`	the current state of the random number generators of samplers.

The main loop for sampling is carried out as below.

```
for k = 1:opt.nsamples
   for i1=1:opt.repeat
      sample inputs with RJMCMC,
      sample weights with HMC,
```

```
        sample hyperparameters with Gibbs,
      end
      save sample in record structure
    end
```

The sampled network parameters are stored in a record structure after every round of sampling. The record structure is constructed with an internal function in `mlp2*_mc` which is following:

```
rec = recappend(rec, ri, net, p, t, pp, tt, rejs).
```

The function takes old record `rec`, record index `ri`, training data `p`, target data `t`, test data `pp`, test target `tt` and rejections `rejs`. It returns a structure `rec` containing following record fields:

type	- type of network 'mlp2';
numInputs	- number of inputs
numLayers	- number of layers
numHidden	- number of hidden unit weights
numOutputs	- number of outputs
inputHiddenHyper	- σ for the hyperparameters of the weights from input to hidden layer
inputHiddenHyperNus	- ν for the hyperparameters of the weights from input to hidden layer
inputHiddenSigma	- σ's for the weights from input to hidden layer
inputHiddenNus	- ν's for the weights from input to hidden layer
hiddenOutputHyper	- σ for the hyperparameters of the weights from hidden layer to output
hiddenOutputSigma	- σ's for the weights from hidden layer to output
hiddenOutputNus	- ν's for the weights from hiddenlayer to output
hiddenBiasSigma	- σ's for the hidden layer biases
outputBiasSigma	- σ's for the output biases
inputii	- inputii's for covariate (input variable) selection (not treated here)
noiseHyper	- σ for the hyperparameters of noise distribution
noiseSigmas	- σ's for the noise distributions
noiseNus	- ν for the noise distributions
inputWeights	- input weights
inputBiases	- input biases
layerWeights	- layer weights
layerBiases	- layer biases
rejects	- rejection rate
etr	- training error
etst	- test error

The predictions of the networks represented in the record can be done by forward propagating a new input data through them. The forward propagation is done with a function

```
Y = mlp2fwds(fbmmlp,X).
```

This returns responses of the MCMC-simulated networks `fbmmlp` in matrix `Y`. Input values are given as rows of `X`.

3.2.1 Setting up the sampling options

General sampling options. MCMC sampling options are given in options structure, where all the desired values are saved. Options structure can be created with functions `mlp2b_mcopt`, `mlp2c_mcopt` and `mlp2r_mcopt`. For regression problem the options are set with the function

```
opt = mlp2r_mcopt(opt)
```

The non existing fields of options structure `opt` are set to default. These default options are

`nsamples`	= 1
`repeat`	= 1
`display`	= 1
`plot`	= 1
`gibbs`	= 0
`hmc_opt`	= hmc2_opt
`hmc_opt.stepsf`	= 'mlp2r_steps'
`persistence_reset`	= 0
`sample_inputs`	= 0

The option values represent the following characteristics.

nsamples The number of samples saved.

repeat How many times RJMCMC, HMC and Gibbs sampling are repeated between two saved samples.

display Information about the sampling process is printed on the screen with value 1 and not printed with 0.

hmc2_opt The name of the function used to initialize the hybrid Monte Carlo sampling options.

opt.stepsf A function to determine the heuristic step sizes for HMC.

persistence_reset Resets the persistence after every `repeat` iteration. The persistence is reset when the value is 1. This can be used to reduce the excess energy in the early phase of sampling.

sample_inputs Sample the inputs when 1 or 2. To use input sampling see section 3.2.3.

Hybrid Monte Carlo sampling options. The options for HMC are stored in the same options structure defined when calling `mlp2r_mcopt` (see page (16)) and they can be set to default by calling a function `hmc2_opt`.

```
opt = hmc2_opt(opt)
```

Here the empty fields of options structure are set to default which are as follows:

```
display      = 0
checkgrad    = 0
steps        = 1
nsamples     = 1
nomit        = 0
persistence  = 0
decay        = 0.9
stepadj      = 0.2
stepsf       = []
window       = 1
```

The option values represent the following characteristics.

display The information about HMC sampling is not printed on the screen with value 0. With value 1 the energy values and rejection threshold are printed at each step of the Markov chain. With value also position vectors are printed at each step. [checkgrad] Checks the user defined gradient function using finite differences method with value 1.

steps Defines the trajectory length (i.e., the number of leapfrog steps at each iteration).

nsamples The number of samples retained from the Markov chain.

nomit the number of samples omitted from the start of the chain.

persistence 0 for complete replacement of momentum variables 1 if momentum persistence is used.

decay Defines the decay used when a persistent update of (leap-frog) momentum is used. Bounded to the interval [0, 1).

stepadj The step adjustment used in leap-frogs.

stepsf The step size function.

window The size of the acceptance window.

3.2.2 Thinning the sample chain

The sample chain of the weights and biases obtained with function `mlp2r_mc` is an approximation of the true posterior distribution. The very first part of sample chain is usually not from the desired distribution. It takes for the sampler so called *burn in* samples to converge to the desired distribution. Based on the convergence diagnostics first samples up to approximate convergence has to be discarded. The samples of the chain are also correlated. These inflict that the sample chains usually have to be long. It is often useful to thin the sample chain as described in the appendix A. The posterior distributions of weights and biases are stored in the record structure that can be given as input argument for a thinning function, which can be used for easy removing of burn-in samples and thinning for all quantities stored in the record structure. The function is

```
rr = thin(r,nburn,nthin,nlast).
```

> This function returns a chain containing only every `nthin`:th simulation sample starting from sample number `nburn+1` and continuing to sample number `nlast`. The input parameter `r` is a record structure and the return value `rr` is also a record structure.

The convergence of the sample chain should be checked before and after thinning. Before thinning, convergence diagnostics such as PSRF test (Brooks and Gelman, 1998) should be used to determine burn-in time. After burn in samples have been removed it is useful to estimate the autocorrelation time using, for example, Geyer's initial monotone sequence estimator (Geyer, 1992). After thinning, the samples are assumed to be approximately independent and Kolmogorov-Smirnov test (Robert and Casella, 2004, p. 466) can be used as an additional convergence test.

Neal (1993, p. 104) discuss about *batching* a sample chain instead of thinning. The sample chain is divided evenly into batches of same size and the mean or median of each batch is evaluated. These can be handled as (quasi-) independent samples for which, for example, Kolmogorov-Smirnov test can be used to approximate the convergence of sample chain. In the software there is a function to evaluate the batch means or medians of each sample chain in the record structure. The function is

```
rr = batch(r,nburn,batchsize,nlast,fun)
```

> This function returns a chain containing batch means or medians of simulation sample chain starting from sample number `nburn+1` and continuing to sample number `nlast`. The input parameter `r` is a record structure and the return value `rr` is also a record structure. `fun` is an optional function handle to handle the batches, for example `@mean`, which returns a new chain `rr` of mean values of each batch. User can give any function for batch as a function handle and the default function that is used is mean.

3.2.3 Input variable selection with RJMCMC

In practical problems there are often many variables, but it is not necessarily known which of them are needed to solve the problem. The predictive importance of each input can

be approximated by comparing models with different input variables with, for example, cross-validation approach. The comparison of all the possible models, however, becomes quickly computationally prohibitive as the number of input variables increases. In section 3.1.2 we discussed ARD prior structure for MLP in the case of irrelevant inputs. However, the relevance score of an ARD prior for MLP, as discussed by Lampinen and Vehtari (2001), is affected more by the non-linearity of inputs than by the predictive importance. Here we discuss *reversible jump Markov chain Monte Carlo* (RJMCMC) (Green, 1995) method in predicting the relevance of input variable.

RJMCMC method is an extension of Metropolis-Hastings algorithm that allows jumps between models with different dimensional parameter spaces. In the case of input selection the dimension of networks parameter space changes with respect to the number of inputs chosen in the model. RJMCMC visits the models according to their posterior probability which allows it to be used for model selection. After sampling a chain of convenient length the goodness of model (or marginal posterior probability of input in our case) can be approximated from the number of visits to that model.

The posterior sample chain obtained with RJMCMC can be used also to keep in the uncertainty of model in final predictions. In this case the expectation of output given new input and data (equation (6)) is gotten by integrating over all the models M_l and parameters θ_l

$$\hat{y}^{\text{new}} = E y^{\text{new}} \mid x^{\text{new}}, D = \int f(x^{\text{new}}, k, \theta_k) p(k, \theta_k \mid D) d\theta dk. \tag{25}$$

The method is discussed shortly in section 5.4, for more complete treatment refer Green (1995) and Green et al. (2003). A demonstration program `demo_rjmcmc` (section 3.3.4) demonstrates the use of input variable sampling in the software.

The RJMCMC jumps can be done in couple of ways according to the value of `opt.sample_inputs`. The possible jumps are:

1. Pick random input and change its state. With probability of `opt.rj_opt.pswitch` change its state with an other input that has different state (the dimension of parameter space does not change) or with probability of `1-opt.rj_opt.pswitch` change its state (the dimension of parameter space changes) .
2. Randomly add or remove one input or switch the state of two random inputs. With probability of `opt.rj_opt.pswitch` change the states of two random variables. With probability `1-opt.rj_opt.pswitch` choose randomly whether to remove or add one input, with probabilities `opt.rj_opt.pdeath`, `1-opt.rj_opt.pdeath`, respectively.

The input variables are sampled with RJMCMC by setting following fields in the options structure `opt` (see chapter 3.2.1) and in the network structure `net` (see chapter 3.1.1):

opt.sample_inputs 0) Do not sample 1) pick random input and change its state 2) randomly add or remove one input.

opt.psample_inputs The rejection rate in input variable sampling can be quite high. With this value it can be adjusted how many times RJMCMC sampling is done in one loop of `opt.repeat` (see section 3.2). Often good jump rate for RJMCMC is approximately 1/opt.repeat.

opt.rj_opt RJMCMC options that has following fields:

pswitch When pswitch is different from 0 there are two possible actions. With probability 1-pswitch a state of random input is changed, This way the total number of input variables in the model changes. With probability `pswith` two random inputs are picked such a way that they are in different states and the state of both of them is changed. This way the total number of input variables in the model does not change.

lpk Logarithm of prior probabilities for number of input variables. `lpk` is a vector containing the prior probability that nth input is in the model.

pdeath The option is used when opt.sample_inputs = 2. The value of pdeath tells the probability of dropping one input variable from the model. If a variable is not dropped then one variable more is taken into the model.

net.inputii Index vector to store the information which inputs are used in the model.

The RJMCMC options can be set for example as below.

```
opt.sample_inputs=1;
opt.rj_opt.pswitch = 0.4;
opt.rj_opt.lpk = log(ones(1,net.nin)/net.nin);
net.inputii = logical(ones(1,net.nin));
```

The logarithm of prior probabilities for the number of input variables is given in the vector `opt.rj_opt.lpk` and `net.inputii` defines that the first model in RJMCMC sampling contains all the input variables. Currently probability of 0 inputs has to be zero and the vector of probabilities contains only the probabilities for $1...K$ inputs. Choosing the prior for input variables is discussed in detail by Vehtari and Lampinen (2001).

3.2.4 Defining the starting values for sampling

In theory, if the chain is irreducible, the choice of the starting values will not affect the stationary distribution. In practice the starting values can affect greatly the time the chain needs to reach an equilibrium. If the chain is for example slow-mixing and the starting values are chosen from areas of very low posterior probability, it may require long burn-in to reach areas of high probability. Because of high number of parameters which correlate in posterior distribution, sampling in Bayesian MLP is slow-mixing. Therefore it is worth of little effort to find good starting values.

Neal (1996) discussed of using different MCMC algorithm parameters in initial sampling phase and in actual sampling phase. In the first sampling phase only the weights are updated while the hyperparameters are fixed at certain value. This prevents hyperparameters from taking strange values before the weights have reached reasonable values.

After starting values for weights have been found next sampling phase is used to determine the starting values for hyperparameters. This method is used in the demonstration program `demo_2input` (chapter 3.3.1). In the software the sampling is made only for weights if the Gibbs sampler (section 3.2.1) is set off. The weights are sampled from $\omega_{kj} \sim N(0, \alpha_k^2)$, where α_k^2 stays fixed if Gibbs sampling is not used . The fixed value of α_k^2 is the one given to it when prior structure is created (chapter 3.1.2).

Vehtari et al. (2000) proposed choosing the starting values based on early stopping. In early stopping the weights are at first initialized to very small values. The training data is divided in two parts, from which the other part is used to train the MLP and the other to monitor the validation error. MLP is trained with optimization algorithm for minimizing the training error until the validation error begins to increase. The weights with minimum validation error are selected as starting values for MCMC sampling.

After early stopping we can approximate the variance of each weight group and this variance can be used as starting value for hyperparameters α_k^2. For example, in the case of ARD, for input to hidden weights there are as many weights leaving from one input as there are hidden units. The variance can then be approximated to be

$$\hat{\alpha}_k^2 \approx \frac{1}{K} \sum_{j=1}^{K} (\omega_{kj})^2, \tag{26}$$

where ω_{kj} is a weight from kth input to jth hidden unit, K is the number of hidden units and the mean of weights from kth input is 0. The MCMC sampling is always done first for weights and after that for hyperparameters. Now that the values of α_k^2 are set as above the first samples of weights will not run away from the good starting region defined with early stop.

This method is used in the demonstration programs `demo_2class` and `demo_3class` (chapters 3.3.2 and 3.3.3). The early stop starting values can be found with function `scges` which uses scaled conjugate gradient algorithm for training (Bishop, 1995).

```
[x, fs, vs] = scges(f, x, opt, gradf, varargin).
```

> The function uses a scaled conjugate gradient algorithm to find a local minimum of the function `f(x,p1,p2,...)` whose gradient is given by `gradf(x,p1,p2,...)`. Search is early stopped if value of `f(x,p1',p2',...)` does not decrease. Here `x` is a row vector and `f` returns a scalar value. The point at which `f` has a local minimum is returned as `x`.

The options for early stopped scaled conjugate gradient algorithm are set with the following function

```
opt = scges_opt(opt).
```

> Sets the empty options to default in `opt` structure. If the function is called without parameter an options structure with default options is returned. The defaults are.

`display`	= 0 -1 to not display anything, 0 to display just diagnostic messages n positive integer to show also the function values and validation values every nth iteration
`checkgrad`	= 0 1 to check the user defined gradient function
`maxiter`	= 1000 Maximum number of iterations
`tolfun`	= 1e-6 termination tolerance on the function value
`tolx`	= 1e-6 termination tolerance on x
`maxfail`	= 20 maximum number of iterations validation is allowed to fail if negative do not use early stopping

3.2.5 Finding the optimal sampling parameters

Finding the optimal sampling options is not usually easy. Most of the time it is more efficient to start sampling with less optimal parameters and sample longer chain than use lot of time to find the best sampling parameters. Diagnosing the convergence and validity of sample chain for predictions is discussed in the appendix A.

When sampling, the option `display` can be set to 1 in order to see, for example, the rejection rate of sampling process (see HMC sampling in section 5.2 and Gibbs sampling in section 5.3). The rejection rate should be in the range of 5-15% of all samples. If it is too small the sample chain does not move fast enough in the distribution phase space. Also the smaller the rejection rate is the higher autocorrelation time gets. If the rejection rate is too large the doubt that the sample chain obtained is not from the right distribution increases. When persistence is used the rejection rate should be closer to 5%, because the rejection reverses the momentum and thus high rejection rate would increase random walk. The rejection rate can be adjusted with HMC sampling options. Neal (1996) has discussed about choosing the parameters in more detail.

Recently, Neal (2005) proposed Short-Cut Metropolis algorithm which could be used to improve sampling by using several step size values. This could help sampling especially when the shape of the posterior is such that different step sizes would be optimal in different parts of the parameter space. Typical example of such distribution is a funnel like distribution. It is probable that posterior of MLP weights has funnell like shapes although problem is partly allenated by heuristic step size determination 5.2.1. This has not yet been implemented in this toolbox.

3.3 Demonstration programs

3.3.1 MLP network in regression problem with Gaussian noise

The demonstration program `demo_2input` is based on the data used in Paciorek and Schervish (2004). The data is from a two input one output function with Gaussian noise with mean zero and standard deviation 0.25, $N(0, 0.25^2)$.

The data is stored in a file in the matrix form, where each row contains 3-dimensional data vector. The first thing to do is to load the data from the file and separate the input variables from the output variable. The first two columns are the inputs and the third column is the output.

```
1  % Load the data.
2  data=load('./dat.1');
3  x = [data(:,1) data(:,2)];
4  y = data(:,3);
5
6  % Draw the data.
7  figure
8  title({'The noisy training data'});
9  [xi,yi,zi]=griddata(data(:,1),data(:,2),...
10   data(:,3),-1.8:0.01:1.8,[-1.8:0.01:1.8]');
11 mesh(xi,yi,zi)
```

Next the network is created

```
14 nin=size(x,2);
15 nhid=10;
16 nout=1;
17
18 net = mlp2('mlp2r', nin, nhid, nout);
```

Then the structures for prior and residual model are created

```
19 net=mlp2normp(net,...
20              {{repmat(0.1,1,net.nin) 0.05 0.5 -0.05 1}...
21               {0.1 0.05 0.5} ...
22               {0.1 -0.05 0.5} ...
23               {1}});
24
25 net.p.r = norm_p({0.05 0.05 0.5});
```

For weights we construct a prior structure with ARD. On the line 20 an own hierarchy for both inputs is created. `repmat` creates a row vector, where there is initial value 0.1 for α_k of both components of input vector. Because of negative input value α_0 for the first layer and α_{ave} for the hidden layer weights, the values of α_0 and α_{ave} are scaled as described in section 3.1.2. Now that there is not parameters α_0 and $\nu_{\alpha,\text{ave}}$ for the hidden layer, the hidden layer hyperparameters are determined using only the first and second level sampling. For the residual `norm_p` function is used to create a prior. Residual has a

normal prior distribution, $r \sim N(0, \sigma^2)$, where the hyperparameter σ^2 has inverse-gamma distribution, $\sigma^2 \sim$ Inv-gamma(α^2, ν), with hyperparameter values $\alpha = 0.05$ and $\nu = 0.5$.

After the network has been constructed and the prior structure has been created for it, the actual sampling can start. First the sampling options have to be initialized, which is done on the line 26. The default options are changed on specific ones on the lines 27-30 and on the line 31 the state of HMC sampler is initialized.

The sampling parameters used in this example were tested with Markov Chain Monte Carlo diagnostic tools to determine the approximate burn-in time and autocorrelation time.

```
26 opt=mlp2r_mcopt;
27 opt.repeat=50;
28 opt.plot=0;
29 opt.hmc_opt.steps=40;
30 opt.hmc_opt.stepadj=0.1;
31 hmc2('state', sum(100*clock));
```

The sampling is done in four parts as can be seen below. The first sampling round samples only for weights, which are first initialized to zero. During this round the "starting guess" for the weights is found. The sampling has to be done only for weights because if hyperparameters were also sampled they would correct themselves near to zero and the sampling would take much longer time to reach the equilibrium (see chapter 3.2.4). After each of the first three rounds only one sample is saved. The sampling of 2500 samples required approximately 5 hours CPU time on a 2400MHz Intel Pentium 4 workstation.

```
32 net = mlp2unpak(net, mlp2pak(net)*0);
33 [r1,net1]=mlp2r_mc(opt, net, x, y);
34
35 opt.hmc_opt.stepadj=0.2;
36 opt.hmc_opt.steps=60;
37 opt.repeat=70;
38 opt.gibbs=1;
39 [r2,net2]=mlp2r_mc(opt, net1, x, y, [], [], r1);
40
41 opt.hmc_opt.stepadj=0.3;
42 opt.hmc_opt.steps=100;
43 opt.hmc_opt.window=5;
44 opt.hmc_opt.persistence=1;
45 [r3,net3]=mlp2r_mc(opt, net2, x, y, [], [], r2);
46
47 opt.repeat=60;
48 opt.hmc_opt.steps=100;
49 opt.hmc_opt.stepadj=0.5;
50 opt.nsamples=2500;
51 [r,net]=mlp2r_mc(opt, net3, x, y, [], [], r3);
```

After sampling there are 2504 different weight and bias samples and as many networks. The burn-in samples have to be omitted and thinning is used to save computational resources. The length of burn-in has to be estimated, for example, with PSRF test. As described in the appendix A, the autocorrelation time can be estimated after burn-in has been omitted with function `geyer_imse`. The thinning is done on the line 52 with function `thin`, here the burn-in is 150 and only every 25th sample is accepted. This way 95 samples are saved for forward propagation. After thinning an additional test was done for two independent sample chains with Kolmogorov-Smirnov test. The convergence tests indicated approximate convergence for network weights and biases. On the lines 53-54 new data is constructed (it is done with `meshgrid` in order to be able to plot it with `mesh`). Lines 55-56 evaluates the outcome of network and lines 59-65 plots the data on 3D-surface. The output and the training data can be seen in figures 1(a) and 1(b).

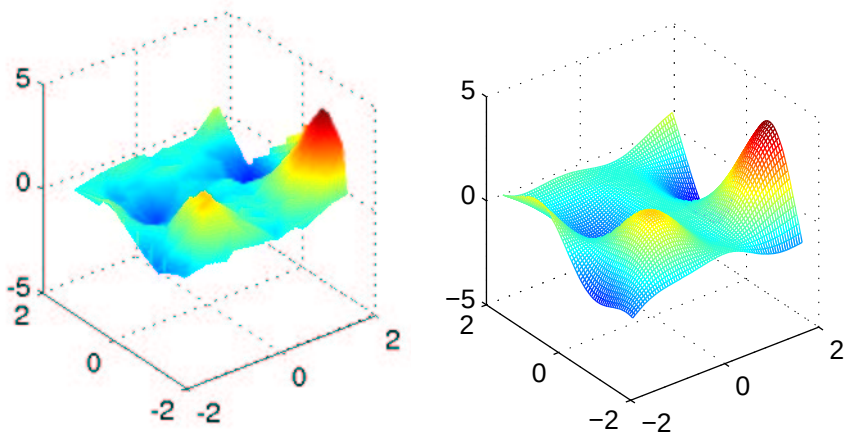
```
52 mlp=thin(r,150,25);
53 [p1,p2]=meshgrid(-1.8:0.05:1.8,-1.8:0.05:1.8);
54 p=[p1(:) p2(:)];
55 tms=squeeze(mlp2fwds(mlp,p))';
56 g=mean(tms);
57
58 %Plot the new data
59 gp=zeros(size(p1));
60 gp(:)=g;
61 figure
62 mesh(p1,p2,gp);
63 axis on;
64 rot;
65 pop;
```

3.3.2 MLP network in a 2-class classification problem

The demonstration program `demo_2class` is based on synthetic two class data used by Ripley (1996). The data consists of 2-dimensional vectors that are divided into to classes, labeled 0 or 1. Each class has a bimodal distribution generated from equal mixtures of Gaussian distributions with identical covariance matrices.

Training data is stored in matrix where each row contains the vector and its label. First we load the data and create the network structure. The network is given Gaussian multivariate hierarchical prior with ARD. The prior is similar to the one in regression problem demo in the section 3.3.1.

```
1  % Load the data
2  x=load('demos/synth.tr');
3  y=x(:,end);
4  x(:,end)=[];
5
6  nin=size(x,2);
7  nhid=10;
```



(a) The output of training data (b) The prediction

Figure 1: **The training data and the prediction of trained MLP in two dimensional regression problem**:

```
8  nout=size(y,2);
9  % create MLP with logistic output function ('mlp2b')
10 net = mlp2('mlp2b', nin, nhid, nout);
11
12 % Create a Gaussian multivariate hierarchical prior with ARD
13 % for network...
14 net=mlp2normp(net, {{repmat(10,1,net.nin) 0.05 0.5 -0.05 1}...
15                     {10 0.05 0.5} ...
16                     {10 -0.05 0.5} ...
17                     {1}})
```

In section 3.2.4 we discussed how to select starting values for MCMC sampling with early stopping. Here we find the early stop values for weights with scaled conjugate gradient optimization algorithm. After early stopping for weights the starting value for each hyper-parameter α_k is approximated by the variance of early stopped weights $\omega_{k.}$. The starting values for other hyperparameters are defined on the rows 52-57. This is done with one sample round without persistence for weights (page 62). After the starting values for all the sample chains are found we set the sampling options for main sampling on the lines 60-68 and start the main sampling.

```
18 % Intialize weights to zero and set the optimization parameters...
19 w=randn(size(mlp2pak(net)))*0.01;
20
21 fe=str2fun('mlp2b_e');
22 fg=str2fun('mlp2b_g');
23 n=length(y);
```

```
itr=1:floor(0.5*n);     % training set of data for early stop
its=floor(0.5*n)+1:n;   % test set of data for early stop
optes=scges_opt;
optes.display=1;
optes.tolfun=1e-1;
optes.tolx=1e-1;

% scaled conjugate gradient optimization with early stopping.
[w,fs,vs]=scges(fe, w, optes, fg, net, x(itr,:),y(itr,:), ...
          net,x(its,:),y(its,:));
net=mlp2unpak(net,w);

shypw1 = std(net.w1,0,2)';
shypb1 = std(net.b1);
shypw2 = std(net.w2(:));

net=mlp2normp(net, {{shypw1  0.5 0.05 -0.05 1}...
                    {shypb1  0.5 0.05} ...
                    {shypw2 -0.5 0.05} ...
                    {1}})

% First initialize random seed for Monte
% Carlo sampling and set the sampling options to default.
hmc2('state', sum(100*clock));
opt=mlp2b_mcopt;
opt.sample_inputs=0; % do not use RJMCMC for input variables

% sample without persistence.
opt.repeat=70;
opt.hmc_opt.steps=10;
opt.hmc_opt.stepadj=0.2;
opt.gibbs=1;
opt.nsamples=1;
[r,net,rstate]=mlp2b_mc(opt, net, x, y);

% starting values are found. start the main sampling.
opt.hmc_opt.stepadj=0.5;
opt.hmc_opt.persistence=1;
opt.hmc_opt.steps=40;
opt.hmc_opt.decay=0.95;
opt.repeat=50;
opt.nsamples=650;
opt.hmc_opt.window=5;

[r,net,rstate]=mlp2b_mc(opt, net, x, y, [], [], r, rstate);
```

After sampling, a convergence diagnostics has to be done for the sample chains. We sampled three independent sample chains which were compared to each others with PSRF

test (see appendix A) in order to find the length of burn-in. The autocorrelation time was approximated with Geyer's initial monotone sequence estimator. After these convergence tests the sample chains are thinned on the line 60. As additional convergence diagnostic Kolmogorov-Smirnov test was used to compare the three thinned sample chains.

After thinning we can use the sampled networks to find the decision line. The decision line together with the training data is plotted on the lines 71-85. On the lines 88-95 test data is still forward propagated through the network to test how well they are classified. Predictive accuracy based on the test data is estimated to be 90($\pm$3)%.

```
69 r=thin(r,200,6);
70
71 % Draw the decision line and training points in the same plot
72 [p1,p2]=meshgrid(-1.3:0.05:1.1,-1.3:0.05:1.1);
73 p=[p1(:) p2(:)];
74 tms=mean((logsig(mlp2fwds(r,p))),3);
75
76 %Plot the decision line
77 gp=zeros(size(p1));
78 gp(:)=tms;
79 contour(p1,p2,gp,[0.5 0.5],'k');
80
81 hold on;
82 % plot the train data o=0, x=1
83 plot(x(y==0,1),x(y==0,2),'o');
84 plot(x(y==1,1),x(y==1,2),'x');
85 hold off;
86
87 % test how well the network works for test data.
88 tx=load('demos/synth.ts');
89 ty=tx(:,end);
90 tx(:,end)=[];
91
92 tga=mean(logsig(mlp2fwds(r,tx)),3);
93
94 % calculate the percentage of misclassified points
95 missed = sum(abs(round(tga)-ty))/size(ty,1)*100;
```

3.3.3 MLP network in a 3-class classification problem

The program `demo_3class` demonstrates the benefits of ARD prior (section 3.1.2), when some of the inputs are irrelevant. ARD prior gives smaller values for the weigths connected to irrelevant inputs and we will later see how the posterior distribution of weights standard deviation α_k (in equation (12)) is concentrated near zero in case of irrelevant input.

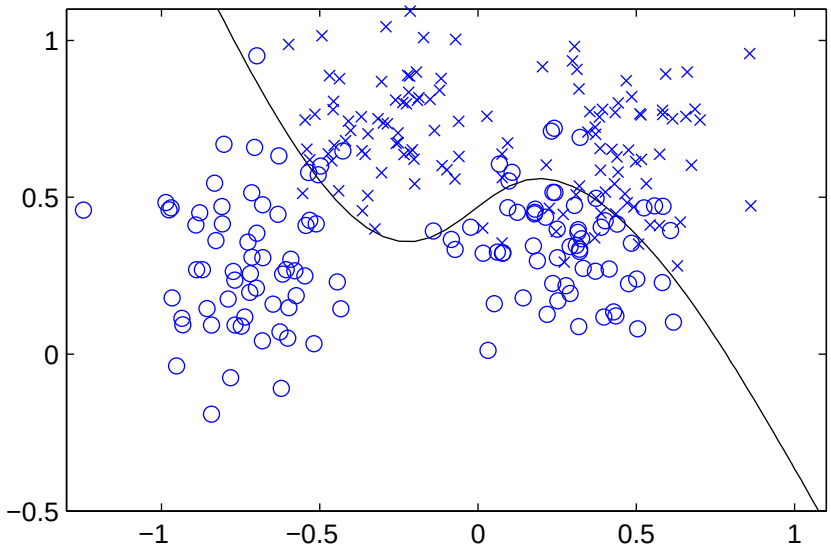


Figure 2: **Classification problem with two classes. The training data and decision line**

The data used in the demonstration program is the same used by Radford M. Neal in his three-way classification example in Software for Flexible Bayesian Modeling[5]. The data consists of 1000 4-D vectors which are classified into three classes. The data is generated by drawing the components of vector, x_1, x_2, x_3 and x_4, uniformly from (0, 1). The class of each vector is selected according to the first two components of the vector, x_1 and x_2. After this a Gaussian noise with standard deviation of 0.1 has been added to every component of the vector. The training data is plotted with respect to x_1 and x_2 in the figure 3.

Training data is stored in matrix where each row contains the vector and it's label. After loading the data we have to create target vectors. In the data matrix labeling of classes is done with values 0, 1 and 2, which we have to transform into vectors 1, 0, 0, 0, 1, 0 and 0, 0, 1. This is done in the lines 10-13.

```
1  % Load the data
2  x=load('demos/cdata');
3  y=repmat(0,size(x,1),3);
4  y(x(:,5)==0,1) = 1;
5  y(x(:,5)==1,2) = 1;
6  y(x(:,5)==2,3) = 1;
7  x(:,end)=[];
8
9  % Divide the data into training and test parts.
10 xt = x(401:end,:);
11 x=x(1:400,:);
```

[5] http://www.cs.toronto.edu/~radford/fbm.software.html

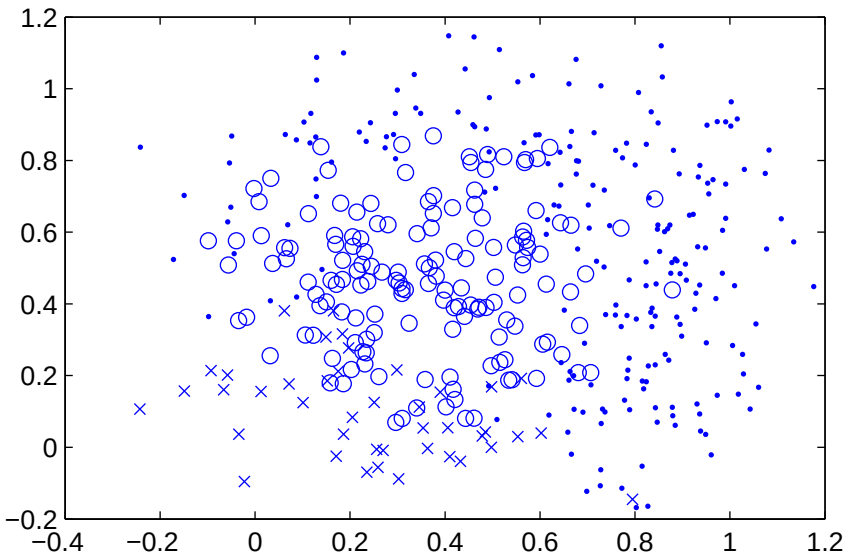


Figure 3: **Classification problem with three classes**. The training data plotted with respect to x_1 and x_2

```
12 yt=y(401:end,:);
13 y=y(1:400,:);
14
15 % create the network
16 nin=size(x,2);
17 nhid=8;
18 nout=size(y,2);
19 % create MLP with logistic output function ('mlp2b')
20 net = mlp2('mlp2c', nin, nhid, nout);
21
22 %Create a Gaussian multivariate hierarchical prior with ARD
23 net=mlp2normp(net, {{repmat(1,1,net.nin) 0.5 0.05 -0.05 1}...
24                     {1 0.5 0.05} ...
25                     {1 -0.5 0.05} ...
26                     {1}})
```

The starting values for sampling are found in the same way as in the demonstration program for two classes. For weights early stop values are found with scaled conjugate gradient optimization algorithm after which the starting values for hyperparameters are defined on the rows 47-68. The variances α_k^2 of the weights are approximated by the variance of early stopped weights $\omega_{k.}$. The starting values for other hyperparameters are defined with one round of MCMC sampling without persistence for weights (see page 62 for persistence).

```
27 % Intialize weights to zero and set the optimization parameters...
```

```
28 w=randn(size(mlp2pak(net)))*0.01;
29
30 fe=str2fun('mlp2c_e');
31 fg=str2fun('mlp2c_g');
32 n=length(y);
33 itr=1:floor(0.5*n);     % training set of data for early stop
34 its=floor(0.5*n)+1:n;   % test set of data for early stop
35 optes=scges_opt;
36 optes.display=1;
37 optes.tolfun=1e-1;
38 optes.tolx=1e-1;
39
40 % do scaled conjugate gradient optimization with early stopping.
41 [w,fs,vs]=scges(fe, w, optes, fg, net, x(itr,:),y(itr,:), ...
42                 net,x(its,:),y(its,:));
43 net=mlp2unpak(net,w);
44
45 % Set the starting values for hyperparameter of the weights
46 % to be the variance of the early stopped weight.
47 shypw1 = std(net.w1,0,2)';
48 shypb1 = std(net.b1);
49 shypw2 = std(net.w2(:));
50
51 net=mlp2normp(net, {{shypw1  0.5 0.05 -0.05 1}...
52                     {shypb1  0.5 0.05} ...
53                     {shypw2 -0.5 0.05} ...
54                     {1}})
55
56 % First initialize random seed for Monte
57 % Carlo sampling and set the sampling options to default.
58 hmc2('state', sum(100*clock));
59 opt=mlp2c_mcopt;
60 opt.sample_inputs=0; % do not use RJMCMC for input variables
61
62 % Do the sampling without persistence.
63 opt.repeat=70;
64 opt.hmc_opt.steps=10;
65 opt.hmc_opt.stepadj=0.2;
66 opt.gibbs=1;
67 opt.nsamples=1;
68 [r,net,rstate]=mlp2c_mc(opt, net, x, y);
```

After the starting values for the sample chain are found we set the sampling options for main sampling on the lines 70-76 and start the main sampling.

```
69 % The starting values are found. Start the main sampling.
70 opt.hmc_opt.stepadj=0.5;
71 opt.hmc_opt.persistence=1;
```

```
72 opt.hmc_opt.steps=40;
73 opt.hmc_opt.decay=0.95;
74 opt.repeat=50;
75 opt.nsamples=250;
76 opt.hmc_opt.window=5;
77
78 [r,net,rstate]=mlp2c_mc(opt, net, x, y, [], [], r, rstate);
```

To find the good sampling parameters we sampled two sample chains from which we tested the convergence and autocorrelation time of the chains. The burn in of the chain was found with PSRF and the autocorrelation time with Geyer's initial monotone sequence estimator. The chain was thinned according to the test results as in line 80. After thinning the convergence of the chains was still tested with Kolmogorov-Smirnov test

```
79 % Thin the sample chain.
80 r2=thin(r2,50,6)
81
82 % test how well the network works for test data.
83 forw = mlp2fwds(r1,xt);
84 for i=1:size(forw,3)
85   tga(:,:,i) = softmax(forw(:,:,i)')';
86 end
87 tga = mean(tga,3);
88 tt = tga==repmat(max(tga,[],2),1,size(tga,2));
89
90 % calculate the percentage of misclassified points
91 missed = (sum(sum(abs(tt-yt)))/2)/size(yt,1)
```

Predictive accurace based on the test data is estimated to be 87($\pm$3)%. The effect of ARD prior can be seen by studying the posterior densities of weights prior α_k. The weights for an input to hidden connection are sampled from $\omega_{kj} \sim N(0, \alpha_k^2)$, where ω_{kj} is a weight from kth input to jth hidden unit. In the figure 4 it can be seen that the distribution of α_3 and α_4 (corresponding the irrelevant inputs x_3 and x_4) is concentrated near zero, whereas the distributions of α_1 and α_2 are more spread. The code for printing the distributions of α_ks is shown on the lines 92-103.

```
92 [pp,xx]=kernelp(r.inputHiddenSigmas(:,1));
93 plot(xx,pp/sum(pp))
94 hold on;
95 [pp,xx]=kernelp(r.inputHiddenSigmas(:,2));
96 plot(xx,pp/sum(pp),'--')
97 [pp,xx]=kernelp(r.inputHiddenSigmas(:,3));
98 plot(xx,pp/sum(pp),':')
99 [pp,xx]=kernelp(r.inputHiddenSigmas(:,4));
100 plot(xx,pp/sum(pp),'-.')
101 legend('a_1', 'a_2', 'a_3', 'a_4')
102 axis([0 12 0 0.03])
103 hold off
```

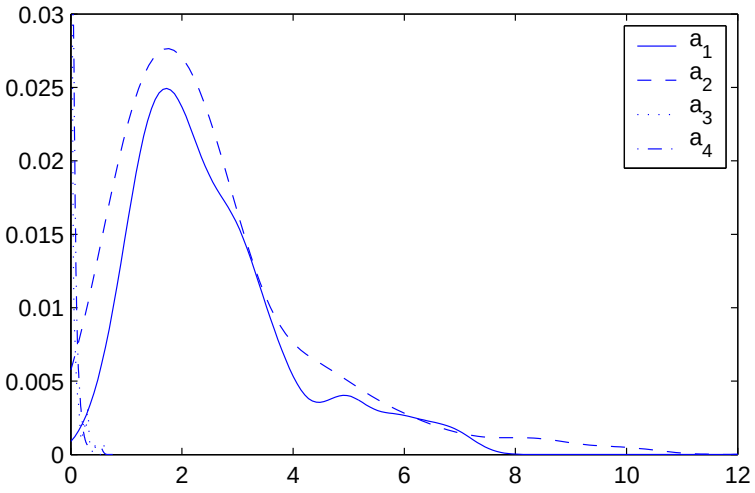


Figure 4: **Results of ARD prior for input to hidden unit weights prior in case of two irrelevant input units** The weights for an input to hidden connection are sampled from $\omega_{kj} \sim N(0, \alpha_k^2)$, where ω_{kj} is a weight from kth input to jth hidden unit. Here are shown the posterior distributions of α_k in case of ARD prior and two irrelevant weights.

3.3.4 Input variable selection with RJMCMC for MLP network

Program `demo_rjmcmc` demonstrates the benefits of RJMCMC sampling in input variable selection. The program is an extension of demonstration program `demo_3class` (section 3.3.3), in which the ARD prior was used in the case of irrelevant inputs. Here, in addition to ARD, we sample also between models with different input variables. RJMCMC is discussed in the sections 3.2.3 and 5.4.

The data used in the demonstration program is the same used by Radford M. Neal in his three-way classification example in Software for Flexible Bayesian Modeling[6]. The data consists of 1000 4-D vectors which are classified into three classes. The data is generated by drawing the components of vector, x_1, x_2, x_3 and x_4, uniformly form (0, 1). The class of each vector is selected according to the first two components of the vector, x_1 and x_2. After this a Gaussian noise with standard deviation of 0.1 has been added to every component of the vector.

The initialization of network is done as in demonstration program `demo_3class` on the lines 1-68. The main sampling and RJMCMC options are set on the lines 69-75. The RJMCMC sampling is done by randomly choosing an input and changing its state (row 81). With probability $1/3$ the state of the input is changed with another input so that the total number of inputs in the model remains (row 82). The main sampling is done on the line 87.

[6] http://www.cs.toronto.edu/~radford/fbm.software.html

```
69 % the starting values are found, start the
70 % main sampling.
71 opt.hmc_opt.stepadj=0.5;
72 opt.hmc_opt.persistence=1;
73 opt.hmc_opt.steps=40;
74 opt.hmc_opt.decay=0.95;
75 opt.repeat=50;
76 opt.nsamples=600;
77 opt.hmc_opt.window=5;
78
79 % use RJMCMC for input variable selection.
80 % The options for RJMCMC are set below.
81 opt.sample_inputs=1;
82 opt.rj_opt.pswitch = 1/3
83 % log of uniform prior for k:s
84 opt.rj_opt.lpk = log(ones(1,net.nin)/net.nin)
85 net.inputii = logical(ones(1,net.nin))
86
87 [r,net,rstate]=mlp2c_mc(opt, net, x, y, [], [], r, rstate);
```

The sampled chain is thinned according to convergence diagnostics and the predictions for test data are done on the lines 92-97. Predictive accuracy based on the test data is estimated to be 86(±3)%. The marginal posterior probability of an input is evaluated on the line 103 and the results are shown in the table below.

Note that the probability of inputs 3 and 4 is not zero even they are irrelevant. The reason to this is that the ARD prior corrects the weights connected to inputs 3 and 4 near to zero and thus their influence to final model is very little even if they are present in the model.

input 1	input 2	input 3	input 4
1.00	1.00	0.15	0.17

Table 2: The marginal posterior probability of inputs in demonstration program `demo_rjmcmc`

```
88 % Thin the sample chain.
89 r=thin(r,50,5);
90
91 % test how well the network works for test data.
92 forw = mlp2fwds(r,xt);
93 for i=1:size(forw,3)
94   tga(:,:,i) = softmax(forw(:,:,i)')';
95 end
96 tga = mean(tga,3);
97 tt = tga==repmat(max(tga,[],2),1,size(tga,2));
98
99 % calculate the percentage of misclassified points
100 missed = (sum(sum(abs(tt-yt)))/2)/size(yt,1)*100
```

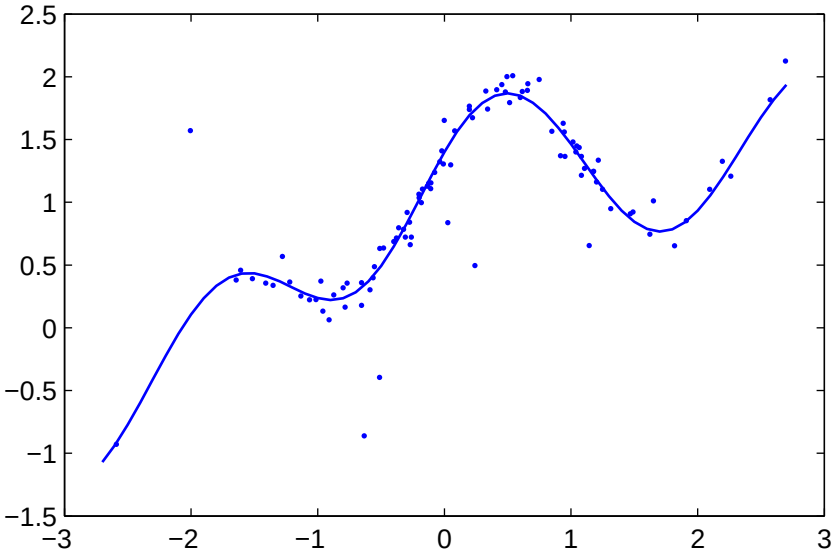


Figure 5: **The training data set and underlying mean of the `demo_tmlp`**

```
101
102 % The posterior probability of input variables
103 probability = sum(r.inputii)/size(r.inputii,1)
```

3.3.5 MLP with Students t residual model in a regression problem

`demo_tmlp` is a regression problem demonstration in which the residual model is Student's t-distribution. The synthetic data used here is the same used by Radford M. Neal in his regression problem with outliers example in Software for Flexible Bayesian Modeling[7]. The problem consist of one dimensional input and target variables. The input data, x, is sampled from standard Gaussian distribution and the corresponding target values come from a distribution with a mean given by

$$y = 0.3 + 0.4x + 0.5\sin(2.7x) + \frac{1.1}{(1+x^2)}.$$

For most of the cases the distribution about this mean is Gaussian with standard deviation of 0.1, but with probability 0.05 a case is an outlier for which the standard deviation is 1.0. There are total 200 cases from which the first 100 are used for training and the last 100 for testing. The training set together with a line of the underlying mean is shown in the picture 5

First the data is loaded and divided into train and test sets (lines 1-7). The train data and underlying mean are plotted on the lines 11-16 and a network with one input, eight hidden

[7] http://www.cs.toronto.edu/~radford/fbm.software.html

units and one output is created on the lines 19-22.

```
1  % load the data. First 100 variables are for training
2  % and last 100 for test
3  x = load('demos/odata');
4  xt = x(101:end,1);
5  yt = x(101:end,2);
6  y = x(1:100,2);
7  x = x(1:100,1);
8
9  % plot the training data in dots and the underlying
10 % mean of it as a line
11 xx = [-2.7:0.1:2.7];
12 yy = 0.3+0.4*xx+0.5*sin(2.7*xx)+1.1./(1+xx.^2);
13 figure
14 plot(x,y,'.')
15 hold on
16 plot(xx,yy)
17
18 % create the network
19 nin=size(x,2);
20 nhid=8;
21 nout=1;
22 net = mlp2('mlp2r', nin, nhid, nout);
```

Network weights are given a Gaussian hierarchical prior on the lines 25-28. For the residual we construct a model with Student's t-distribution as discussed in the section 3.1.3. First residual is given a prior with fixed number of degrees of freedom, ν. after which one round of MCMC sampling is done. This prior is created on the line 33 where it is defined as follows

$$\begin{aligned} e &\sim t_\nu(0, \sigma^2) \\ \nu &= 4 \\ \sigma^2 &\sim \text{Inv-gamma}(0.05, 0.5), \end{aligned} \tag{27}$$

The first number 0.1 in the array gicen to `t_p` defines the initial value of σ^2. In order to get better starting values for the sample chain we sample for one round with fixed number of degrees of freedom. A hierarchical prior for ν is given on the lines 50-52. The number of degrees of freedom is sampled from discretized values given in the vector on the lines 51-52. Now the prior structure for residual is as following

$$
\begin{aligned}
& e \sim t_\nu(0, \sigma^2) \\
& \nu = V j \\
& j \sim U_d(1, J) \\
V1 \quad & J = 2, 2.3, 2.6, 3, 3.5, 4, 4.5, 5, 6, 7, 8, 9, 10, 12, 14, 16, 18, 20, 25, 30, 35, 40, 45, 50 \\
& \sigma^2 \sim \text{Inv-gamma}(0.05, 0.5),
\end{aligned}
$$

where the vector V is approximately uniform on $log(\nu)$.

```
23 % Create a Gaussian multivariate hierarchical
24 %  prior for network...
25 net=mlp2normp(net, {{0.1 0.05 0.5 -0.05 1} ...
26                     {0.1 0.05 0.5} ...
27                     {0.1 -0.05 0.5} ...
28                     {1}})
29
30 % Create students t-distribution prior for residual.
31 % first the prior is created wiht fixed number of degrees
32 % of freedom. After which the sampling is done for one round
33 net.p.r = t_p({0.1 4 0.05 0.5});
34
35 % Set the sampling options
36 opt=mlp2r_mcopt;
37 opt.repeat=50;
38 opt.plot=0;
39 opt.hmc_opt.steps=40;
40 opt.hmc_opt.stepadj=0.1;
41 hmc2('state', sum(100*clock));
42
43 % Sample for one round with fixed nu
44 net = mlp2unpak(net, mlp2pak(net)*0);
45 [r1,net1]=mlp2r_mc(opt, net, x, y);
46
47 % create the prior structure for the final sampling
48 % Here nu (number of freedom) is sampled from discrete
49 % set of values.
50 net1.p.r = t_p({net.p.r.a.s net.p.r.a.nu 0.05 0.5 ...
51             [2 2.3 2.6 3 3.5 4 4.5 5 6 7 8 9 10 12 ...
52             14 16 18 20 25 30 35 40 45 50]});
```

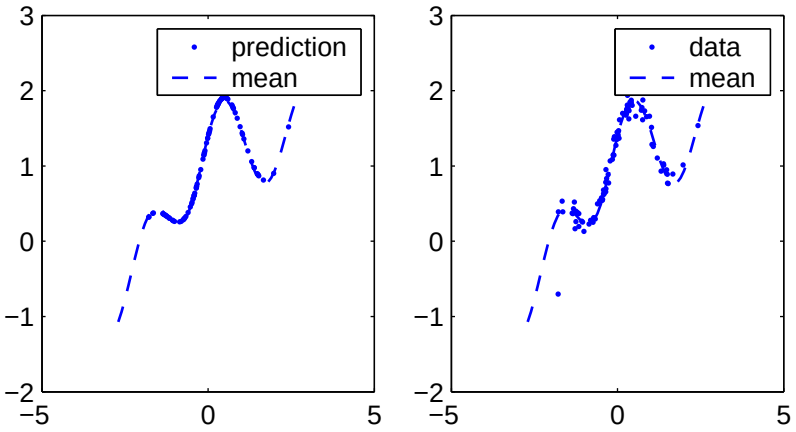
After the residual model is defined we still fine tune the starting point for sampling on the lines 53-66. The main sampling options are set on the lines 69-72 after which the main sampling is started.

```
53 opt.hmc_opt.stepadj=0.2;
54 opt.hmc_opt.steps=60;
```

```
55 opt.repeat=70;
56 opt.gibbs=1;
57 [r2,net2]=mlp2r_mc(opt, net1, x, y, [], [], r1);
58
59 % The sampling parameters are tuned and the sampling with
60 % windowing and persistence is started. After this the starting
61 % values for main sampling are found and main sampling is started.
62 opt.hmc_opt.stepadj=0.3;
63 opt.hmc_opt.steps=100;
64 opt.hmc_opt.window=5;
65 opt.hmc_opt.persistence=1;
66 [r3,net3]=mlp2r_mc(opt, net2, x, y, [], [], r2);
67
68 % Sample for the posterior XX samples .
69 opt.repeat=60;
70 opt.hmc_opt.steps=100;
71 opt.hmc_opt.stepadj=0.5;
72 opt.nsamples=2000;
73
74 [r,net]=mlp2r_mc(opt, net3, x, y, [], [], r3);
```

The test data set is forward propagated through the trained networks and the predictive mean of test data is evaluated on the line 81. A figure of predictive values for test inputs and the underlying mean are plotted on the lines 84-87 and the test data set and mean on the lines 90-93. The plots are shown in the figure 6. The RMSE to mean was 0.029($\pm$0.013) and it is evaluated on the lines 96-97.

```
77 % thin the record
78 rr = thin(r,50,10)
79
80 % make predictions for test set
81 tga = mean(mlp2fwds(r,xt),3);
82
83 % Plot the network outputs as '.', and underlying mean with '--'
84 figure
85 plot(xt,tga,'.')
86 hold on
87 plot(xx,yy,'--')
88
89 % plot the test data set as '.', and underlying mean with '--'
90 figure
91 plot(xt,yt,'.')
92 hold on
93 plot(xx,yy,'--')
94
95 % evaluate the RMSE to the mean
96 yyt = 0.3+0.4*xt+0.5*sin(2.7*xt)+1.1./(1+xt.^2);
97 er = sqrt((yyt-tga)'*(yyt-tga)/length(yyt));
```



(a) The predictions and mean (b) The test data and mean

Figure 6: **The predictions for test data and the test data together with the underlying mean**:

4 Gaussian process in MCMCstuff

4.1 Gaussian process architecture and prior structures

In a Bayesian regression and classification models a prior distribution for model parameters is formulated together with the likelihood of parameters, from which the posterior distribution of parameters is derived (equation (1)). Our primary focus is however in the prediction for a new output, $y^{(n+1)}$ given the training data $((x^{(1)}, y^{(1)}), ...(x^{(n)}, y^{(n)}))$ and new input $x^{(n+1)}$, where $x^{(i)}$ is the input vector and $y^{(i)}$ the corresponding output. The final result is then a predictive distribution for $y^{(n+1)}$, that is obtained integrating over the unknown parameters (equation (3)). Rather than deriving the posterior distribution for the parameters and making the prediction from that, we could specify a prior distribution for the outputs in any set of data. A predictive distribution for an unknown output can then be evaluated by conditioning on the known outputs.

Lets consider a situation where there are n known inputs, $x^{(1)}, ..., x^{(n)}$, and n known outputs $y^{(1)}, ..., y^{(n)}$ each related to corresponding input. Supposing that the distribution of $y^{(i)}$ is Gaussian with mean zero and known covariance function, $\mathrm{Cov}y^{(i)}, y^{(j)}$, a predictive distribution for an unknown output $y^{(n+1)}$ given a new input $x^{(n+1)}$ and outputs $y^{(1)}, ..., y^{(n)}$ can be constructed. The distribution is Gaussian with mean and variance given by (Neal, 1999)

$$\mathrm{E}y^{(n+1)}|y^{(1)}, ..., y^{(n)} = k^T C^{-1} y \tag{28}$$

$$\mathrm{Var}y^{(n+1)}|y^{(1)}, ..., y^{(n)} = V - k^T C^{-1} y \tag{29}$$

where C is the covariance matrix of the known outputs, y is the vector of known values

of these outputs, k is the vector of covariances between new output $y^{(n+1)}$ and the known outputs and V is the variance of $y^{(n+1)}$. The problem is to determine the covariance function and the parameters for it. Neal (1999) has discussed in more detail about covariance functions used in *Gaussian process*. For example a covariance function for a regression model based on a class of smooth functions can be written as follows (Neal, 1999),

$$\mathrm{Cov}(y^{(i)}, y^{(j)}) = \eta^2 \exp\left(-\sum_{k=1}^{K} \rho_k^2 (x_k^{(i)} - x_k^{(j)})^2 \right) + \delta_{ij} J^2 + \delta_{ij} \sigma_\epsilon^2 \qquad (30)$$

where K is the number of components in vector $x^{(i)}$. Here the problem is to find good values for the parameters in the covariance function. The posterior distribution of the hyperparameters can be sampled as in MLP networks with Hybrid Monte Carlo sampling (section 5.2). When the posterior distribution is reached the prediction for new output can be averaged from the predictive distributions (equations (28), (29)) based on the sampled hyperparameters.

4.1.1 Gaussian process in classification problem

The Gaussian process for classification problem is created similar to regression problem. In the case of classification problem we must, though, introduce "latent values", for which the Gaussian process is created. In two class classification problem a logistic model for binary targets, {0, 1}, in terms of latent values z_i is defined by letting the distribution for the target y_i be (Neal, 1999)

$$p(y_i = 1) = \frac{1}{1 + \exp(-z_i)}. \qquad (31)$$

The covariance function for latent values in the software is similar to the one in regression problem (equation (30)). Some amount of jitter, J is needed to make matrix computations easier. The jitter term can be used also to alter the effect of model from logit to probit as discussed by Neal (1999, 1997).

4.1.2 Creating and initializing a Gaussian process

Currently the software provides Gaussian processes only for functions with one output, but multiple inputs can be used. Only one type of covariance function, (30), is provided. The function for creating a Gaussian process is `gp2` and the syntax for it is.

```
gp = gp2(type, nin, varargin);
```

> The input argument `type` is the type of model, which is either `'gp2r'` or `'gp2b'` and `nin` is the number of inputs. In `varargin` is set strings which specifies the elements of the covariance function, these can be `jitter`, or `exp`. The function returns a Gaussian process structure `gp`.

The fields in the Gaussian process structure are following

`type`	string describing the network type

`nin`	number of inputs
`nout`	number of outputs, always 1
`jitterSigmas`	J, jitter term in covariance function
	0.1, if given a string `'jitter'` in `varargin`
`expScale`	η, general scale for exponential part in covariance function
	0.1, if given a string `'exp'` in `varargin`
`expSigmas`	ρ, length scale for each input in covariance function
	0.1, if given a string `'exp'` in `varargin`
`noiseSigmas`	σ_ϵ, scale of residual distribution
	standard deviation (scalar) for normal distribution.
	degrees of freedom (vector) in t-distribution
`noiseVariances`	σ_ϵ^2, variance of standard deviation in covariate dependent
	grouped noise model.
`p`	prior structure containing the above fields for prior also

Fields `noiseSigmas`, `expScale` and `expSigmas` represent parameters σ_ϵ, η and ρ_u in the covariance function of equation (30). Field `jitterSigmas` tells the amount of jitter in a classification problem and corresponds minimum noise level in regression problem. Without jitter the covariance matrix of classification problem would be singular (Neal, 1999). When Gaussian process structure is created only the fields given in input arguments are given values as described above. The other fields are initialized to matrices or structures. The treatment of a linear and a constant term in covariance function (30) can be found from Neal (1999) . At the moment these terms are not in use in the software. The linear term can make the covariance function ill-conditioned and a separate model for linear regression problem is relative easy to construct. The constant term can be excluded if the data is normalized to have zero mean, or the constant term can be included in linear model.

For the classification problem also the latent values are needed. The latent values are set into a field in Gaussian process structure created by `gp2`, for example, as below.

```
gp.latentValues = randn(size(y));
```

As in MLP networks with Bayesian approach there are functions to pack the hyperparameters of Gaussian process into one vector and to unpak them from a vector to the structure fields. The functions are,

```
w = gp2pak(gp)
gp = gp2unpak(gp, w)
```

4.1.3 The prior structure and residual model for a Gaussian process

The prior structure of Gaussian process in the software is treated in detail by Neal (1999) and Lampinen and Vehtari (2001). Here we give only an overview of the prior structure.

The covariance function for regression used in the software is

$$\mathrm{Cov}_{ij} = \eta^2 \exp\left(-\sum_{k=1}^{K} \rho_k^2 (x_k^{(i)} - x_k^{(j)})^2 \right) + \delta_{ij} J^2 + \delta_{ij} \sigma_\epsilon^2, \tag{32}$$

where J is the jitter term. The prior for the scale is

$$\eta^2 \sim \text{Inv-gamma}(\alpha_{\eta^2}^2, \nu_{\eta^2}). \tag{33}$$

For the relevance parameters an ARD prior (see section 3.1.2) is constructed with distributions

$$\rho^2 \sim \text{Inv-gamma}(\alpha_{\rho^2}^2, \nu_{\rho^2}) \tag{34}$$

$$\alpha_{\rho^2} \sim \text{Inv-gamma}(\alpha_{0,\rho^2}^2, \nu_{0,\rho^2}). \tag{35}$$

This prior structure is used, for example, in the demonstration problems `demo_2ingp` (section 4.3.1) and `demo_tgp` (section 4.3.4). The covariance function in classification is similar to (32) with the difference that it does not have the noise term $\delta_{ij}\sigma_\epsilon^2$. The classification is treated in the demonstration progmram `demo_2classgp` (section 4.3.2). For more carefull treatment of classification with GP see Neal (1999).

The residual model. In the GP the noise of the measured data is modeled by the residual term in the covariance function, σ_ϵ. The principal assumption in GP is that the noise has a Gaussian distribution in which case the same noise variance is assumed for each sample. In practice, this is not always a good assumption because the target distribution may contain error sources of non-Gaussian density in which case more robust noise models should be considered. Here a covariate dependent noise model is used to construct an approximation of the Student's t-distribution noise model.

In Gaussian noise model, σ_ϵ in covariance function represents the standard deviation of the noise. For the variance of the Gaussian distribution, a conjugate prior is inverse Gamma, which leads to the following hierarchy

$$e \sim N(0, \sigma_\epsilon^2) \tag{36}$$

$$\sigma_\epsilon^2 \sim \text{Inv-Gamma}(\sigma_0, \nu_\sigma). \tag{37}$$

The Gaussian noise model is demonstrated in the `demo_2ingp` in section 4.3.1.

In covariate dependent noise model each sample $(\mathbf{x}^i, \mathbf{y}^i)$ is given a different noise variance governed by a common prior. This corresponds to prior structure similar to ARD for the relevance parameters (equations (34))

$$\epsilon^i \sim N(0, (\sigma_\epsilon^2)^i) \tag{38}$$

$$(\sigma_\epsilon^2)^i \sim \text{Inv-gamma}(\alpha_{\text{ave}}^2, \nu_\sigma) \tag{39}$$

$$\alpha_{\text{ave}}^2 \sim \text{Inv-gamma}(\alpha_0^2, \nu_{\alpha,\text{ave}}). \tag{40}$$

In this parametrization, as the number of data points increases, the residual is asymptotically same as Student's t-distribution with fixed degrees of freedom, ν_σ (Neal, 1999; Gelman et al., 2004; Lampinen and Vehtari, 2001).

It is also possible to make degrees of freedom a hyperprior with hierarchical prior. In the section 3.1.3 a prior with discretized values for degrees of freedom is discussed in the case of MLP network. For Gaussian process the degrees of freedom is given an Inverse Gamma prior distribution,

$$\nu_\sigma \sim \text{Inv-Gamma}(\alpha_\nu, \nu_0), \tag{41}$$

which corresponds approximately uniform prior in log scale if ν_0 is very small. The disadvantage of Inverse Gamma prior is that the degrees of freedom are not limited above 1. The degrees of freedom should be above one because at $\nu_\sigma = 1$ the residual model represents Cauchy distribution and the expectation value and variance of Cauchy distribution are not defined. In fact, it would be useful to restrict $\nu_\sigma > 2$, but this is not yet implemented in the software. The Student's t-distribution model for the residual is demonstrated in the `demo_tgp` in section 4.3.4.

4.1.4 Creating prior and residual model for Gaussian process

An Inverse Gamma prior can be constructed with the function `invgam_p`, which creates Inverse-gamma distribution as following:

```
q = invgam_p(pr)
```

> `pr` is an array containing the initial values of hyperparameters of prior structure. Function creates function handles for error and gradient evaluation and returns a structure `q` containing the prior information in its fields. The fields in the array `pr` are as following:
>
> pr = $\{\alpha_{h^2} \quad \nu_{h^2} \quad \alpha_{0,h^2} \quad \nu_{0,h^2}\}$
>
> Here h corresponds either scale parameter, η, or relevance parameter, ρ. The prior structure returned is similar to the one of function `norm_p` on the page 12. Note that the hyperparameter given for `invgam_p` is standard deviation α and not the variance α^2.

The residual model. A Gaussian residual model can be constructed by giving σ_ϵ a starting value and inverse Gamma prior as following.

```
gp.noiseSigmas=0.2;
gp.p.noiseSigmas=invgam_p({0.05 0.5});
```

The covariate dependent grouped noise model is asymptotically Student's t-distribution with fixed degrees of freedom. This residual model can be constructed as following.

```
gp.p.noiseSigmas=invgam_p({0.2 4 0.05 1});
gp.noiseSigmas=repmat(0.2,1,n);
gp.noiseVariances=gp.noiseSigmas.^2;
opt.sample_variances=1;
```

Above `n` is the number of data points. The hyperprior for degrees of freedom can be constructed with the following line.

```
gp.p.noiseSigmas.p.nu=invgam_p({6 1});
```

4.2 Markov Chain sampling in a Gaussian process

The parameters of the covariance function are sampled with Hybrid Monte Carlo sampling (section 5.2). Gibbs sampling (section 5.3) is used only for variances. There are two sampling functions for Gaussian processes, `gp2r_mc` for regression problem, and `gp2b_mc` for classification problem with two classes. The difference between these two samplers is that in the classification problems we must sample also for latent values, z, that are used in the place of targets when making predictions.

```
[rec, gp0, rstate] = gp2r_mc(opt, gp, p, t, pp, tt, rec, rstate)
```

> Mandatory input arguments are options structure `opt`, Gaussian process structure `gp`, train data `p`, train target `t` and state of random generator `rstate`. Optionally also test data `pp`, test target `tt` and old record `rec` can be included. The function returns record structure `rec`, Gaussian process `gp0` with parameters of last sampling round and the state of random number generator `rstate`.

The main loop for sampling is carried out as below.

```
for k = 1:opt.nsamples
   for i1=1:opt.repeat
      sample inputs with RJMCMC,
      sample latent values with Metropolis-Hastings
      sample parameters with HMC,
   end
   save sample in record structure
end
```

The record structure returned by `gp2r_mc` is created with an internal function, which is as follows.

```
rec = recappend(rec, ri, gp, p, t, pp, tt, rejs, gl, invC)
```

> Here old record `rec`, record index `ri` and rejection rate `rejs` are taken in. Record structure `rec` is returned, it contains following fields:

type	- type of Gaussian process 'gp'
numInputs	- number of inputs
numOutputs	- number of outputs
jitterSigmas	- jitter term in covariance function
expHyper	- hyperparameter for parameters in the exponential part of covariance function
expHyperNu	- hyperparameter for distribution of hyperparameter above
expSigmas	- length scale for each input
expScale	- general scale for exponential part
inputii	- inputii's for covariate selection
noiseHyper	- standard deviation for hyperparameter of residual model
noiseHyperHyper	- standard deviation for hyperparameter of parameter above
noiseSigmas	- scale of residual distribution standard deviation for normal distribution degrees of freedom for t-distribution (not treated here)
noiseNus	- ν for hyperparameter of residual model
noiseNusHyper	- hyperparameter for Nu above
noiseVariances	- individual variance for each data point, used in covariate dependent grouped noise model
latentValues	- latent values
k	- number of inputs in covariate selection
etr	- RMSE for training data
e	- total energy
edata	- data energy
eprior	- prior energy
etst	- RMSE for test data
rejects	- HMC rejection rate

The sampling options have to be set in a options structure. This structure is created with default options with following function:

```
opt = gp2_mcopt(opt)
```

Empty option fields are set to default, the default ones are:

`nsamples`	= 1
`repeat`	= 1
`display`	= 1
`plot`	= 1
`gibbs`	= 0
`hmc_opt`	= hmc2_opt
`hmc_opt.stepsf`	= 'gp2r_steps'
`persistence_reset`	= 0
`sample_variances`	= 0
`sample_latent`	= 0

The fields and what they represent are identical to the ones after calling `mlp2r_mcopt` (page 3.2.1) with the exception of last field. The fields are:

nsamples The number of samples saved.

repeat How many times the HMC and Gibbs sampling is repeated between two saved samples. The greater the repeat is the further away from the starting sample the next sample is.

display Whether information about sampling process is printed on the screen. The information is printed with value 1.

hmc2_opt The name of function used to initialize the hybrid Monte Carlo sampling options (for HMC options see section (3.2.1)).

opt.stepsf The function to determine the step sizes for HMC.

persistence_reset Whether the persistence is wanted to reset after every `repeat` iteration. The persistence is reset when the value is 1.

sample_variances Optional feature for Student's t residual model and covariate dependent grouped residual model (not treated here). Note please! If it is set to one when using normal distribution model for residual an error occurs in Monte Carlo sampling.

sample_latent Determines whether there are latent values to be sampled in the model. Latent values are needed in classification problems. With values greater than 0 sample latent values (works only with `gp2b_mc`).

The record structure returned from Markov Chain Monte Carlo sampling contains `opt.nsamples` different hyperparameters for covariance function. The prediction for output of new input data is based on the average of Gaussian processes sampled. A thinning of sample chain should be made as described in the section 3.2. The function to forward propagate the data through Gaussian processes is as follows:

```
Y = gp2fwds(gp, TX, TY, X)
```

Here a record structure containing Gaussian processes with parameters sampled from posterior distribution is taken together with training input TX, training output TY and new input X. In the case of classification latent values can be given in place of TY. The function returns a matrix Y containing the outputs of Gaussian processes.

4.2.1 MCMC sampling for classification problem

The sampling in the classification problem differs from the one in regression problem only due to the sampling of latent values. The latent values are updated at each round of sampling by Metropolis-Hastings algorithm. The change to latent value is proposed with the relation (Neal, 1999)

$$z^* = (1 - \epsilon^2)z + \epsilon L\eta, \tag{42}$$

where ϵ is some small constant, L is a Cholesky decomposition of prior covariance of latent values and η is a vector of independent standard Gaussian variates. The value of ϵ

has to be chosen appropriately in order to sample efficiently. The initial value of ϵ is given as a parameter for MCMC sampler in options structure, `opt`. Neal (1999) has suggested that the value of ϵ could be set based on trial runs after which the sampling is done. Here we update the value of ϵ with 100 iterations of adaptive control algorithm.

The parameters of algorithm are set so that the rejection rate of Metropolis-Hastings in latent value sampling is approximately optimal. The optimal rejection rate is 0.56 in one dimension or 0.77 when many parameters are being updated at once (Gelman et al., 2004)[page 306]. With the updated ϵ sampling of the latent values is continued for `opt.sample_latent` times. All The needed additional sampling options are:

opt.sample_latent When set to 0 the latent values are not sampled. With values greater than 0 the latent values are sampled and the value of `opt.sample_latent` defines how many rounds the sampling is done with updated ϵ.

opt.sample_latent_scale Initial value of ϵ. This value is updated during the sampling.

4.2.2 Input variable selection with RJMCMC

The RJMCMC sampling in the case of Gaussian process is done similar to the sampling in case of MLP. The sampling options that have to be set to the options structure `opt` are discussed in the section 3.2.3. The options can be set for example as following

```
opt.sample_inputs=2;
opt.rj_opt.pswitch = 1/3;
opt.rj_opt.pdeath = 0.5;
opt.rj_opt.lpk = log(ones(1,gp.nin)/gp.nin);
gp.inputii = logical(ones(1,gp.nin))
```

4.3 demonstration programs for Gaussian process

4.3.1 Gaussian process in regression problem with 2 inputs

In the demonstration program `demo_2ingp` We are using the same data that was used in the MLP regression model demonstration `demo_2input`. The data is stored in a file from where it is loaded. The two first columns of data matrix contain the inputs and the third column contains the output. The raw data is plotted again to make it easier to compare it with the result from the Gaussian process.

```
1  % Load the data.
2  data=load('dat.1');
3  x = [data(:,1) data(:,2)];
4  y = data(:,3);
5
6  % Draw the data.
7  figure
8  title({'The noisy teaching data'});
9  [xi,yi,zi]=griddata(data(:,1),data(:,2)...
```

```
          ,data(:,3),-1.8:0.01:1.8,[-1.8:0.01:1.8]');
10 mesh(xi,yi,zi)
11
12
13
14 % Greate a Gaussian process for regression model.
15 [n, nin] = size(x);
16 gp=gp2('gp2r',nin,'exp');
17 gp.jitterSigmas=0.01;
18 gp.expScale=0.2;
19 gp.expSigmas=repmat(0.1,1,gp.nin);
20 gp.noiseSigmas=0.2;
21 gp.f='norm';
```

On the lines 17-20 the hyperparameters are given their initial values. Although string `'jitter'` was not in the input argument when calling function `gp2` the field `jitterSigmas` is initialized as a matrix. On the line 17 it is given small value in order to keep the covariance matrix non-singular. The definition `'norm'`, which represents the normal distribution, on the line 21 is needed for the error calculations during the sampling. The prior structures for the hyperparameters are constructed on the lines 22-24.

```
22 gp.p.expSigmas=invgam_p({0.1 0.5 0.05 1});
23 gp.p.expScale=invgam_p({0.05 0.5});
24 gp.p.noiseSigmas=invgam_p({0.05 0.5});
```

Here `expSigmas` (relevance parameters ρ_u in the equation (30)) are given ARD prior with two levels. For the scale η and noise σ_e^2 is given a one level Inv-gamma prior. The sampling options for Gibbs and Hybrid Monte Carlo sampling are set below. On the line 24 the standard deviation of Gaussian residual is given its prior (see section 4.1.3).

```
25 opt=gp2_mcopt;
26 opt.repeat=20;
27 opt.nsamples=1;
28 opt.hmc_opt.steps=20;
29 opt.hmc_opt.stepadj=0.1;
30 opt.hmc_opt.nsamples=1;
31 hmc2('state', sum(100*clock));
32
33 [r1,g1,rstate1]=gp2r_mc(opt, gp, x, y);
34
35 opt.repeat=10;
36 opt.hmc_opt.steps=30;
37 opt.hmc_opt.stepadj=0.3;
38 [r2,g2,rstate2]=gp2r_mc(opt, g1, x, y, [], [], r1, rstate1);
39
40 opt.nsamples=350;
41 opt.hmc_opt.persistence=1;
42 opt.sample_variances=0;
```

```
43 opt.hmc_opt.window=5;
42 r=r2; g=g2; rstate=rstate2;
43 opt.hmc_opt.stepadj=0.75;
44 opt.hmc_opt.steps=10;
45 [r,g,rstate]=gp2r_mc(opt, g, x, y, [], [], r, rstate);
```

Next the new data is created and forward propagated through Gaussian processes. Before the forward propagation it is useful to thin the sample chains of GP parameters.

```
46 % Create new data with the right scale cheched above
47 figure
48 [p1,p2]=meshgrid(-1.8:0.05:1.8,-1.8:0.05:1.8);
49 p=[p1(:) p2(:)];
50 gp=thin(r,50,2);
51 out=gp2fwds(gp, x, y, p);
52 mout = mean(squeeze(out)');
53
54 %Plot the new data
55 gp=zeros(size(p1));
56 gp(:)=mout;
57 mesh(p1,p2,gp);
58 axis on;
```

The effectiveness of Gaussian Process can be seen from the amount of samples needed. Here we needed only 350 samples to reach the approximate convergence when in MLP network with Bayesian learning we needed 2500 samples. The CPU time needed to sample the 350 samples on a 2400MHz Intel Pentium workstation was approximately 30 minutes.

4.3.2 Gaussian process in a 2-class classification problem

The demonstration program `demo_2classgp` is based on synthetic two class data used by Ripley (1996). The data consists of 2-dimensional vectors that are divided into two classes, labeled 0 or 1. Each class has a bimodal distribution generated from equal mixtures of Gaussian distributions with identical covariance matrices. The same date's was used in the demonstration program `demo_2class` in section 3.3.2.

Training data is stored in matrix where each row contains the vector and its label. First we load the data and create the Gaussian process and the prior structure for its parameters.

```
1  x=load('demos/synth.tr');
2  y=x(:,end);
3  x(:,end)=[];
4
5  [n, nin] = size(x);
6  gp=gp2('gp2b',nin,'exp', 'jitter');
7  gp.jitterSigmas=1;
8  gp.expScale=0.2;
```

```
9  gp.expSigmas=repmat(0.1,1,gp.nin);
10 gp.noiseSigmas = 0.2;
11 gp.f='norm';
12
13 gp.p.expSigmas=invgam_p({0.1 0.5 0.05 1});
14 gp.p.expScale=invgam_p({0.05 0.5});
```

The initial values for parameters are found with scaled conjugate gradient algorithm (see section 3.2.4). The initial values for hyperparameters are found by sampling one MCMC round without persistence for momentum in HMC.

```
15 w=randn(size(gp2pak(gp)))*0.01;
16
17 fe=str2fun('gp2r_e');
18 fg=str2fun('gp2r_g');
19 n=length(y);
20 itr=1:floor(0.5*n);     % training set of data for early stop
21 its=floor(0.5*n)+1:n;   % test set of data for early stop
22 optes=scges_opt;
23 optes.display=1;
24 optes.tolfun=1e-1;
25 optes.tolx=1e-1;
26
27 % do scaled conjugate gradient optimization
28 % with early stopping.
29 [w,fs,vs]=scges(fe, w, optes, fg, gp, ...
30                 x(itr,:),y(itr,:), gp,x(its,:),y(its,:));
31 gp=gp2unpak(gp,w);
32
33 opt=gp2_mcopt;
34 opt.repeat=20;
35 opt.nsamples=1;
36 opt.hmc_opt.steps=20;
37 opt.hmc_opt.stepadj=0.1;
38 opt.hmc_opt.nsamples=1;
39
40 opt.sample_latent = 20;
41 opt.sample_latent_scale = 0.5;
42 gp.latentValues = randn(size(y));
43 hmc2('state', sum(100*clock));
44
45 [r1,g1,rstate1]=gp2b_mc(opt, gp, x, y);

46 opt.repeat=10;
47 opt.nsamples = 1000;
48 opt.hmc_opt.persistence=1;
49 opt.sample_variances=0;
50 opt.hmc_opt.window=5;
```

```
51 opt.hmc_opt.stepadj=0.75;
52 opt.hmc_opt.steps=10;
53 [r,g,rstate]=gp2b_mc(opt, g1, x, y, [], [], r1, rstate1);
```

After sampling, training points and decision line are drawn in the same plot. The trained Gaussian process is used to predict the classes of new inputs in test set on the lines 75-82.

```
54 % Thin the sample chain.
55 r=thin(r,50,6);
56
57 % Draw the decision line and training points in the same plot
58 [p1,p2]=meshgrid(-1.3:0.05:1.1,-1.3:0.05:1.1);
59 p=[p1(:) p2(:)];
60
61 tms2=mean((logsig(gp2fwds(r, x, r.latentValues', p))),3);
62
63 %Plot the decision line
64 gp=zeros(size(p1));
65 gp(:)=tms2;
66 contour(p1,p2,gp,[0.5 0.5],'k');
67
68 hold on;
69 % plot the train data o=0, x=1
70 plot(x(y==0,1),x(y==0,2),'o');
71 plot(x(y==1,1),x(y==1,2),'x');
72 hold off;
73
74 % test how well the network works for the test data.
75 tx=load('demos/synth.ts');
76 ty=tx(:,end);
77 tx(:,end)=[];
78
79 tga=mean(logsig(gp2fwds(r, x, r.latentValues', tx)),3);
80
81 % calculate the percentage of misclassified points
82 missed = sum(abs(round(tga)-ty))/size(ty,1)*100;
```

The decision line is shown in the figure 7. Predictive accuracy of the test data is estimated to be 90($\pm$3)%.

4.3.3 Input variable selection with RJMCMC for Gaussian process

The program `demo_rjmcmcgp` demonstrates the input variable sampling in Gaussian Process. The data is the same as in demonstration program `demo_2classgp` with the difference that there are three irrelevant components in the input vector.

First we load the training data and set the three irrelevant input components. After this we create a Gaussian process for classification problem and define prior structure for the parameters on the lines 7-16.

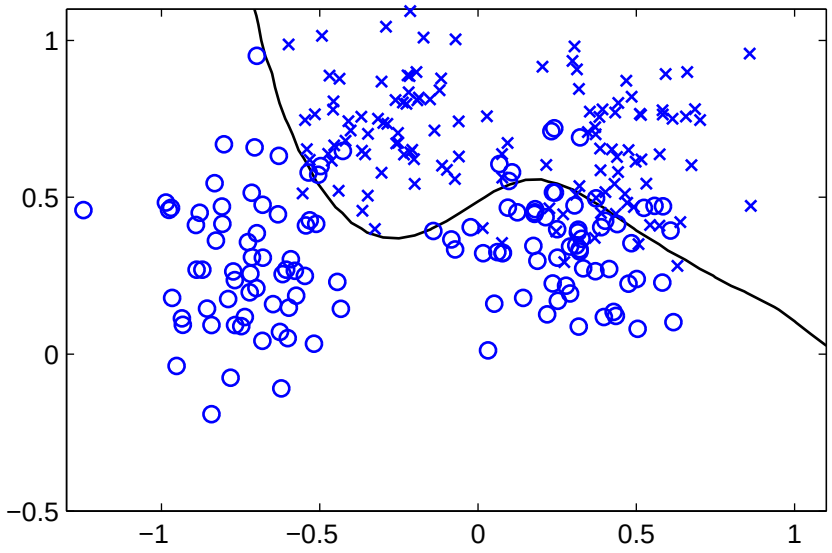


Figure 7: **Classification problem with two classes. The training data and decision line**

```
1  % Load the data
2  x=load('demos/synth2.mat');
3  x = x.x;
4  y=load('demos/synth.tr');
5  y=y(:,end);
6
7  [n, nin] = size(x);
8  gp=gp2('gp2b',nin,'exp', 'jitter');
9  gp.jitterSigmas=10;
10 gp.expScale=0.2;
11 gp.expSigmas=repmat(0.1,1,gp.nin);
12 gp.noiseSigmas = 0.3;
13 gp.f='norm';
14
15 gp.p.expSigmas=invgam_p({0.1 0.5 0.05 1});
16 gp.p.expScale=invgam_p({0.05 0.5});
```

The starting values for the GP parameters are found with early stopped conjugate gradient algorithm after which the starting values for hyperparaters are found by sampling one round without persistence and windowing in HMC (see section 3.2.4).

```
17 % Intialize weights to zero and set the optimization parameters...
18 w=randn(size(gp2pak(gp)))*0.01;
19
20 fe=str2fun('gp2r_e');
21 fg=str2fun('gp2r_g');
```

```
22 n=length(y);
23 itr=1:floor(0.5*n);     % training set of data for early stop
24 its=floor(0.5*n)+1:n;   % test set of data for early stop
25 optes=scges_opt;
26 optes.display=1;
27 optes.tolfun=1e-1;
28 optes.tolx=1e-1;
29
30 % scaled conjugate gradient optimization with early stopping.
31 [w,fs,vs]=scges(fe, w, optes, fg, gp, x(itr,:),...
32                 y(itr,:), gp,x(its,:),y(its,:));
33 gp=gp2unpak(gp,w);
34
35 % set the options for sampling
36 opt=gp2_mcopt;
37 opt.repeat=20;
38 opt.nsamples=1;
39 opt.hmc_opt.steps=20;
40 opt.hmc_opt.stepadj=0.1;
41 opt.hmc_opt.nsamples=1;
42
43 opt.sample_latent = 20;
44 opt.sample_latent_scale = 0.5;
45 gp.latentValues = randn(size(y));
46 hmc2('state', sum(100*clock));
47
48 [r,gp,rstate]=gp2b_mc(opt, gp, x, y);
```

The main sampling parameters and RJMCMC options are set on the lines 50-63 and the main sampling is started on the line 65. The input sampling is done by randomly choosing whether to add or remove one input (line 59) or whether to switch the states of two random inputs (line 60). The probability that we remove one input is set to 0.5 on the line 61. The number of inputs in the model is given a uniform prior. On the line 62 a vector of log prior values is constructed. The first model from which we start the sampling has all the inputs on (line 63).

```
49 % Set the main sampling options
50 opt.repeat=10;
51 opt.nsamples = 1000;
52 opt.hmc_opt.persistence=1;
53 opt.sample_variances=0;
54 opt.hmc_opt.window=5;
55 opt.hmc_opt.stepadj=0.75;
56 opt.hmc_opt.steps=10;
57
58 % Set the sampling options for RJMCMC
59 opt.sample_inputs=2;
60 opt.rj_opt.pswitch = 1/3;
```

```
61 opt.rj_opt.pdeath = 0.5;
62 opt.rj_opt.lpk = log(ones(1,gp.nin)/gp.nin);
63 gp.inputii = logical(ones(1,gp.nin))
64
65 [r,g,rstate]=gp2b_mc(opt, gp, x, y, [], [], r, rstate);
```

After sampling the sample chain is thinned on the line 67. The train data and the decision line are plotted on the lines 69-83. The predictions for the test data and the percentage of misclassified points are evaluated on the lines 87-94. On the line 96 we evaluate the posterior probability of inputs. Predictive accuracy based on the test data was estimated to be 90(±3)% of cases and the marginal posterior probability of inputs is shown in the table below.

input 1	input 2	input 3	input 4
1.0	1.0	0.038	0.053

Table 5: The marginal posterior probability of inputs in demonstration program `demo_rjmcmcgp`

```
66 % Thin the sample chain.
67 r=thin(r,50,6);
68
69 % Draw the decision line and training points in the same plot
70 [p1,p2]=meshgrid(-1.3:0.05:1.1,-1.3:0.05:1.1);
71 p=[p1(:) p2(:)];
72
73 tms2=mean((logsig(gp2fwds(r, x, r.latentValues', p))),3);
74
75 %Plot the decision line
76 gp=zeros(size(p1));
77 gp(:)=tms2;
78 contour(p1,p2,gp,[0.5 0.5],'k');
79
80 hold on;
81 % plot the train data o=0, x=1
82 plot(x(y==0,1),x(y==0,2),'o');
83 plot(x(y==1,1),x(y==1,2),'x');
84 hold off;
85
86 % test how well the network works for the test data.
87 tx=load('demos/synth.ts');
88 ty=tx(:,end);
89 tx(:,end)=[];
90
91 tga=mean(logsig(gp2fwds(r, x, r.latentValues', tx)),3);
92
93 % calculate the percentage of misclassified points
94 missed = sum(abs(round(tga)-ty))/size(ty,1)*100;
```

```
95
96 probability = sum(r.inputii)/size(r.inputii,1)
```

4.3.4 Gaussian Process with Student's t residual model in regression problem

`demo_tgp` is a regression problem demonstration in which the residual model is Student's t-distribution. The residual models are discussed ín the section 3.1.3. The synthetic data used here is the same used by Radford M. Neal in his regression problem with outliers example in Software for Flexible Bayesian Modeling[8]. The same data was used also in demonstration program `demo_tmlp`. The problem consist of one dimensional input and target variables. The input data, x, is sampled from standard Gaussian distribution and the corresponding target values come from a distribution with mean given by

$$y = 0.3 + 0.4x + 0.5\sin(2.7x) + \frac{1.1}{(1+x^2)}.$$

For most of the cases the distribution about this mean is Gaussian with standard deviation of 0.1, but with probability 0.05 a case is an outlier for which the standard deviation is 1.0. There are total 200 cases from which the first 100 are used for training and the last 100 for testing. The training set together with a line of the underlying mean is shown in the picture 5

First the data is loaded and divided into train and test sets (lines 1-7). The train data and underlying mean are plotted on the lines 11-16 and a GP is created on the lines 19-26.

```
1  % load the data. First 100 variables are for training
2  % and last 100 for test
3  x = load('demos/odata');
4  xt = x(101:end,1);
5  yt = x(101:end,2);
6  y = x(1:100,2);
7  x = x(1:100,1);
8
9  % plot the training data with dots and the underlying
10 % mean of it as a line
11 xx = [-2.7:0.1:2.7];
12 yy = 0.3+0.4*xx+0.5*sin(2.7*xx)+1.1./(1+xx.^2);
13 figure
14 plot(x,y,'.')
15 hold on
16 plot(xx,yy)
17 title('training data')
18
19 % create the Gaussian process
20 [n, nin] = size(x);
```

[8] http://www.cs.toronto.edu/~radford/fbm.software.html

```
21 gp=gp2('gp2r',nin,'exp');
22 gp.jitterSigmas=0.1;
23 gp.expScale=0.2;
24 gp.expSigmas=repmat(0.1,1,gp.nin);
25 gp.noiseSigmas=0.2;
26 gp.f='norm';
```

The priors for GP parameters are constructed on the lines 28-30. Prior for GP parameters is Gaussian multivariate hierarchical. The residual is given at first normal prior to find good starting value for `gp.noiseSigmas`. Initial sampling parameters are set on the lines 33–41 and on the line 43 we sample for two rounds.

```
27 % Create a Gaussian multivariate hierarchical prior for GP parameters...
28 gp.p.expSigmas=invgam_p({0.1 0.5 0.05 1});
29 gp.p.expScale=invgam_p({0.05 0.5});
30 gp.p.noiseSigmas=invgam_p({0.05 0.5});
31
32 % set the sampling options for the first two rounds of sampling
33 opt=gp2_mcopt;
34 opt.repeat=20;
35 opt.nsamples=2;
36 opt.hmc_opt.steps=20;
37 opt.hmc_opt.stepadj=0.4;
38 opt.hmc_opt.nsamples=1;
39 opt.hmc_opt.persistence=1;
40 opt.sample_variances=0;
41 hmc2('state', sum(100*clock));
42
43 [r,gp,rstate1]=gp2r_mc(opt, gp, x, y);
```

After sampling two samples with normal residual model we construct hierarchical Student's t-distribution for residual. The prior is constructed by giving each data point different noise variance. By integrating over the priors hyperparameters the likelihood approaches Students t-distribution as the number of data points increases. At first The number of degrees of freedom for residual model is fixed (see section 4.1.3).

```
44 % Create Student's t prior for residual and sample with fixed number of
45 % degrees of freedom.
46 gp.p.noiseSigmas=invgam_p({gp.noiseSigmas 4 0.05 1});
47 gp.noiseSigmas=repmat(gp.noiseSigmas,1,n);
48 gp.noiseVariances=gp.noiseSigmas.^2;
49 opt.sample_variances=1;
50
51 [r,gp,rstate]=gp2r_mc(opt, gp, x, y);
```

After sampling for two rounds with fixed number of degrees of freedom we give an inverse Gamma prior for the number of degrees of freedom on the line 53. After this we do the main sampling on the line 57.

```
52 % Add a hyperprior for degrees of freedom in t-distribution
53 gp.p.noiseSigmas.p.nu=invgam_p({6 1});
54 opt.hmc_opt.stepadj=0.4;
55 opt.nsamples= 300;
56
57 [r,gp,rstate]=gp2r_mc(opt, gp, x, y, [], [], r, rstate);
```

The sample chain is thinned and predictions for the test data are done on the lines 58–62. The predictions are plotted to same figure with underlying mean on the lines 65–70. The mean squared error of predictions to mean is evaluated on the lines 80-82 and the mean of RMSE is 0.019($\pm$0.13).

```
58 % thin the record
59 rr = thin(r,10,2)
60
61 % make predictions for test set
62 tga = mean(squeeze(gp2fwds(rr,x,y,xt)),2);
63
64 % Plot the network outputs as '.', and underlying mean with '--'
65 figure
66 plot(xt,tga,'.')
67 hold on
68 plot(xx,yy,'--')
69 legend('prediction', 'mean')
70 title('prediction from MLP')
71
72 % plot the test data set as '.', and underlying mean with '--'
73 figure
74 plot(xt,yt,'.')
75 hold on
76 plot(xx,yy,'--')
77 legend('data', 'mean')
78 title('test data')
79
80 % evaluate the RMSE error with respect to mean
81 yyt = 0.3+0.4*xt+0.5*sin(2.7*xt)+1.1./(1+xt.^2);
82 er = (yyt-tga)'*(yyt-tga)/length(yyt);
```

5 Sampling methods

5.1 Metropolis-Hastings sampling

The Metropolis-Hastings algorithm uses random walk and an acceptance/rejection rule to converge to the specified target distribution. The algorithm is as follows.

1. Draw a starting point θ^0, for which $p(\theta^0|D) > 0)$
2. for $t = 1, 2, ...$
   (a) Sample a proposal θ^* from jumping distribution $J_t(\theta^*|\theta^{t-1}$
   (b) Calculate the acceptance ratio

$$\alpha = \min\left(1, \frac{p(\theta^*|D)J_t(\theta^{t-1}|\theta^*)}{p(\theta^{t-1}|D)J_t(\theta^*|\theta^{t-1})}\right) \tag{43}$$

3. Set

$$\theta^t = \begin{cases} \theta^* & \text{with probability } \alpha \\ \theta^{t-1} & \text{otherwise.} \end{cases}$$

5.2 Hybrid Monte Carlo sampling

In the hybrid Monte Carlo method the sampling is done from the canonical distribution as reviewed by Neal (1996). We wish to sample from some distribution for a *position* variable q, which has n real valued components q_i. Now if an energy function $E(q) = -\mathrm{Log}P(q)$ is written, we can write the probability density in canonical form

$$P(q) \sim \exp(-E(q)). \tag{44}$$

To be able to use dynamical methods, a *momentum* variable p has to be taken in, it has also n real valued components p_i. With the components of canonical phase space a Hamiltonian function $H(q, p) = E(q) + K(p)$ and a canonical distribution of *phase space* of q and p can be defined. The distribution is

$$P(q, p) \sim \exp(-H(q, p)). \tag{45}$$

Here $K(p)$ represents kinetic energy due to the momentum. The idea in HMC is that the sampling is split into two sub-tasks – sampling for values of q and p with fixed total energy, $H(q, p)$, (dynamic sampling) and sampling states with different values of H (stochastic sampling). The sampling of states with different values of H is done by Gibbs method, as discussed in section 5.3. To sample for values q and p we move in the *phase space* of q and p, starting from the position (q, p), for certain length of time, after which the new position $(q*, p*)$ is reached. This new position is either accepted or rejected and the new position parameter, q, is set to the new or old one depending on the acceptance. By altering stochastic sampling and dynamical transitions an ergodic Markov chain can be achieved.
The dynamical transitions are made by *leapfrog* discretisation, where a single step finds approximations to the position and momentum, $q*$ and $p*$, at time $\tau + \epsilon$ from q and p at time τ as follows.

$$p_i(\tau + \frac{\epsilon}{2}) = p_i(\tau) - \frac{\epsilon}{2}\frac{\partial E}{\partial q_i}(q(\tau)) \tag{46}$$

$$q_i(\tau + \epsilon) = q_i(\tau) + \epsilon\frac{p_i(\tau + \frac{\epsilon}{2})}{m_i} \tag{47}$$

$$p_i(\tau + \epsilon) = p_i(\tau + \frac{\epsilon}{2}) - \frac{\epsilon}{2}\frac{\partial E}{\partial q_i}(q(\tau + \epsilon)), \tag{48}$$

where m_i represents the "mass" associated with the component i. Now, in order to follow the dynamics for some period of time, $\Delta\tau$, a value for the step size, ϵ, has to be chosen after which equations (46) — (48) are applied for $L = \Delta\tau/\epsilon$ steps to reach the target time. Knowing the *leapfrog* discretisation the dynamical transition in HMC is performed using three steps,

1 Starting from the current state, (q, p), perform L leapfrog steps with a step size ϵ to reach the state $(q*, p*)$.
2 Negate the momentum variables producing the state $(q', p') = (q*, -p*)$
3 Regard (q', p') as a candidate for the next state, accepting it with probability $\min(1, \exp(-(H(q', p') - H(q, p)))$

A more complete treatment concerning the Hybrid Monte Carlo method can be found in Neal (1996), Neal (1993).

Hybrid Monte Carlo sampling in the software. The actual sampling is done with function `hmc2`. The syntax for the HMC sampling is following.

```
[samples, energies, diagn] = hmc2(f, x, opt, gradf, varargin);
```

The sampling is done from the distribution $P \sim \exp(-f)$, where f is the first argument to `hmc2`, `x` is the point where the Markov Chain starts, `gradf` is gradient function of energy function `f` and `varargin` contains additional arguments to be passed to `f()` and `gradf()`. `hmc2` returns a matrix `samples` containing the sampled position components, q, matrix `energies` containing the energy values from sampling process and structure `diagn` containing information from sampling process. The fields in `diagn` are:

`pos`	positions matrix
`mom`	momentums matrix
`acc`	acceptance threshold matrix
`rej`	rejection rate
`stps`	step size vector

The states of two random number generators, `rand` and `randn`, and the momentum of dynamical process of `hmc2` can be set to a state `s`. Also the state of process can be returned. These are done by calling the function with the following syntaxes

```
hmc2('state', s);
```

Here `s` is either an integer, and it is passed to the random number generators, or structure containing specified states of generators and momentum.

```
s = hmc2('state')
```

This returns a structure containing the current states of Hybrid Monte Carlo process.

5.2.1 Heuristic choice of step sizes

The idea for heuristic choice of stepsizes arises from the fact that the curvature of *phase space* with respect to the *position* variables changes when moving in the volume V of the space with constant total energy, H (Neal, 1996, chapter A.4). Because the dynamics of Hamiltonian *phase space* can be only approximated, the total energy does not stay constant, but changes due the discretisation error. In order to stay in the wanted *phase space* volume, V, the stepsizes, ϵ_i, of each component, q_i, has to be chosen so that the trajectory of Leapfrog steps does not run away from the volume, V, in that direction. If a Gaussian distribution for q is concerned an expression $E(q) = \sum q_i^2/2\sigma^2$ for the potential energy is obtained. It can be shown that $H(q, p)$ diverges if leapfrog step is repeatedly applied with $\epsilon > 2\sigma$, but remains bounded when $\epsilon < 2\sigma$. Divergence can be avoided choosing the step sizes proportional to the curvature of potential energy with respect to position q in given direction q_i,

$$\epsilon \approx \eta \left[\frac{\partial^2 E}{\partial q_i^2} \right]^{-1/2}. \tag{49}$$

Here η is a step size adjustment factor determined manually. If the energy really was of a quadratic form the stepsize obtained would be $\epsilon_i \approx \eta\sigma_i$ and would be close to optimal when $\eta \approx 1$. The second derivatives can not be determined directly based on values $\partial^2 E/\partial q^2$, since it would brake the reversibility of Markov chain (Neal, 1996, chapter A.4). The current values of hyperparameters together with the values of the inputs and targets in the training cases are fixed during the hybrid Monte Carlo update and can be used to approximate the derivatives. The second derivatives of the log posterior for the parameters can be found summing the second derivatives of log prior and of the log likelihood with respect to the parameters, which can be approximated using the hyperparameters together with inputs and targets. Since this procedure is an approximation, the step sizes may be too large and step size adjustment factor, η, can be set to smaller value than 1 in order to produce an acceptable rejection rate. Neal (1996, chapter A.4) derived an expressions for the derivatives as follows:

$$\frac{\partial^2 L}{\partial (\omega_{i,j}^{S,D})^2} = (\upsilon_i^S)^2 \frac{\partial^2 L}{\partial (\upsilon_j^D)^2} \tag{50}$$

$$-\frac{\partial^2}{\partial (\omega_{i,j})^2} \log P(\omega_{i,j} \mid \sigma_{\omega,i}) = \frac{1}{\sigma_{\omega,i}}. \tag{51}$$

Here $\omega_{i,j}^{S,D}$ is a connection from unit i in layer S to unit j in layer D; υ_i and υ_j are the unit values respectively, $\sigma_{\omega,i}$ is a hyperparameter controlling weights of unit i in layer S. The value of υ_i is approximated to be 1 for hidden layer units. Neal (1996) derived an expression for the second derivatives with respect to output and other units as follows:

$$-\frac{\partial^2 L}{\partial (v_j^O)^2} \approx \frac{1}{\sigma_j^2} \tag{52}$$

$$\frac{\partial^2 L}{\partial (v_i^S)^2} \approx \sum_j (\sigma_{\omega,i}^{S,D})^2 \frac{\partial^2 L}{(v_j^D)^2}. \tag{53}$$

The estimated second derivative of minus log posterior is obtained by adding the second derivatives of *likelihood* and *prior*. An expression for step sizes is then,

$$\epsilon_i = \eta \left[-\frac{\partial^2 L}{\partial (\omega_{i,j}^{S,D})^2} - \frac{\partial^2}{\partial (\omega_{i,j})^2} \log P(\omega_{i,j} \mid \sigma_{\omega,i}, \sigma_{a,j}) \right]^{-1/2}. \tag{54}$$

The stepsizes of HMC are determined at the very beginning of `hmc2` sampler. The function called in this occasion is `mlp2r_steps`, which determines the stepsizes as described above. The syntax for it is,

```
ss = mlp2r_steps(net, x, y)
```

> Here `net` is the network for which the stepsizes are determined, `x` is a matrix containing input vectors and `y` is a matrix of target vectors. It returns a vector `ss`, containing stepsizes.

Heuristic step size determination for classification model. For classification problems a logistic output function is used and there is separate function to evaluate the stepsizes for it; the function is `mlp2b_steps`. Neal (1996) derived an estimate for the second derivative of likelihood function with respect to output units as follows

$$-\frac{\partial^2 L}{\partial (v_j^O)^2} = \frac{1}{4}. \tag{55}$$

The second derivative of likelihood function is the only difference between classification and regression models and the rest of the step size determination is similar to regression problem.

Heuristic step size determination for classification model with more than two classes. For classification problems with more than two possible classes a softmax output function is used and there is own function to evaluate the stepsizes for it. The function is `mlp2c_steps`. Neal (1996) showed that for the softmax activation function an estimate for the second derivative of likelihood function with respect to output units is as in equation (55) and the rest of evaluation of stepsizes follows as in regression model.

5.2.2 Hybrid Monte Carlo with acceptance window

In the windowed HMC the transitions take place between *windows* of states at the beginning and end of the trajectory. A trajectory of L leapfrog steps is regarded as sequence of

$L + 1$ states in which the first W states constitute the *reject* window, R and the last W states the accept window, A. The *free energy* of a window W is defined as follows

$$F(W) = -\log\left[\sum_{s \in W} \exp(-H(q_s, p_s))\right]. \tag{56}$$

The operation in the windowed HMC is analogous to that of standard HMC, with the exception that the acceptance is based on the difference in free energies between *accept* and *reject* windows. If the trajectory is accepted the new state is taken from the accept window, with a particular state from that window being selected at random. Similarly if the trajectory is rejected the new state is taken from reject window. In order to this procedure to be valid, an offset, T, for the start state has to be chosen uniformly from $0, ..., W - 1$. After this the trajectory is computed backwards for T leapfrog steps, after which the forward part of the trajectory is computed for $L - T$ leapfrog steps. With long chains this can lower the rejection rate with small or moderate increase in computing time. For more complete treatment of topic see (Neal, 1996).

5.2.3 Hybrid Monte Carlo with persistence

In HMC the random walk behavior can be suppressed by using long trajectories, consisting of many leapfrog steps. Because the momentum variables are replaced in a Gibbs sampling between each trajectory, this advantage is lost when short trajectories are used. In the Hybrid Monte Carlo method with *persistence* for the momentum the momentum is replaced only partially between trajectories. This causes that the motion will tend to *persist* in largely the same direction from step to step. The persistence in largely the same direction between short trajectories in turn suppresses the random walk behavior. The iteration of HMC with persistence operates as following:

1. Perform a partial replacement of the momentum variables, setting them to new values
$$p_{i,new} = \lambda p_i + (1 - \lambda^2)^{1/2} n_i.$$
2. Perform a dynamical transition, as described in the *leapfrog* method on the page 59.
3. Negate the momentum variables regardless of whether the candidate state was accepted in step 2.

For more complete discussion about persistence see (Neal, 1996).

5.3 Gibbs sampling

Gibbs sampling is a method for sampling from a multi-dimensional parameter distribution. The method samples from the full conditional distribution of θ_n^t at round t given all the other components of θ. The method generates θ^{t+1} from θ^t as follows

sample θ_1^{t+1} from the distribution of θ_1 given $\theta_2^t, \theta_3^t, ..., \theta_p^t$

$\vdots$

sample θ_j^{t+1} from the distribution of θ_j given θ_1^{t+1}, ..., θ_{j-1}^{t+1}, θ_{j+1}^{t}, ..., θ_p^t

$\vdots$

sample θ_p^{t+1} from the distribution of θ_p given θ_1^{t+1}, θ_2^{t+1}..., θ_{p-1}^{t+1}.

Gibbs sampling is discussed in more detail in (Neal, 1996) and (Gilks et al., 1996).

In the software there is a function `gibbs` to handle the sampling. As described above the Gibbs sampler samples for each component of parameter, θ_i, from the full conditional distribution given all other parameters and components of θ. This whole procedure of updating all the components of parameter θ is done once when calling `gibbs`. The function takes care of sampling all the parameters in upper levels of ARD's hierarchy as well, by calling itself again whenever there is upper-level hyperparameters. The syntax for Gibbs sampler is following:

```
p1 = gibbs(p1, x)
```

> The sampler returns one sample for parameters contained in structure `p1` at level n and above it. `x` is a parameter structure from level $n-1$ in hierarchy. The sampler returns a structure `p1` containing new samples in structure `p1`.

As described in context with ARD (section 3.1.2), the fields `p1.f` and `p2.f` contain the distribution information for the parameters at that level. Now in order to sample for posterior distribution of parameter on the level `p1` we need to know the prior of that parameter. The prior is thus the distribution of the hyperparameters of given parameter. So we need distribution information from the parameter itself (likelihood) and from it's hyperparameters (prior) to sample the posterior. In the software there's own sampler for every supported posterior distribution. These samplers all start with `cond_` and they take in arguments telling the parameter name (`a`) and parameter structure (`p1.a`) of given level, hyperparameter structure (`p2.a`) and lower level parameter structure (`x`). supported types of posterior distributions are:

`cond_ginvgam_cat`	inverse gamma *likelihood* for a group and categorical *prior*
`cond_gnorm_invgam`	normal *likelihood* for group and inverse gamma *prior*.
`cond_gnorm_norm`	normal likelihood for a group and normal prior.
`cond_gt_cat`	Student's t *likelihood* for a group and categorical *prior*.
`cond_gt_invgam`	T *likelihood* for a group and inverse gamma *prior*.
`cond_invgam_invgam`	inverse gamma *likelihood* and *prior*
`cond_invgam_cat`	inverse gamma *likelihood* and categorical *prior*.
`cond_laplace_invgam`	Laplace *likelihood* and inverse gamma *prior*.
`cond_mnorm_invwish`	normal *likelihood* for multi-group and inverse Wishard *prior*.
`cond_norm_invgam`	normal *likelihood* and inverse gamma *prior*
`cond_norm_ginvgam`	normal *likelihood* and inverse gamma *prior* for a group
`cond_t_cat`	Student's t *likelihood* and categorical *prior*.
`cond_t_invgam`	Student's t *likelihood* and inverse gamma *prior*.

5.4 Reversible jump Markov chain Monte Carlo sampling

Reversible jump Markov Chain Monte Carlo (RJMCMC) method (Green, 1995) is an extension of Metropolis-Hastings algorithm that allows jumps between models with different dimensional parameter spaces.

Let M_l denote a model l and θ_l the parameter vector of that model with dimension d_l. In order to jump between models we have to introduce an auxiliary random variable that enables the matching of parameter space dimensions across models. If we consider a jump from model M_l to model M_{l^*} we generate a random variable, t_l, with proposal distribution $q(t_l|\theta_l, M_l, M_{l^*})$. Now an invertible function is constructed to define the mapping between parameter spaces, $(\theta_{l^*}, t_{l^*}) = h_{l,l^*}(\theta_l, t_l)$, so that the dimension-matching $d_l + \dim(t_l) = d_{l^*} + \dim(t_{l^*})$ is maintained.

The algorithm of RJMCMC is similar to the Metropolis-Hastings. If the current state of Markov chain is (M_l, θ_l) the jump to the state (M_{l^*}, θ_{l^*}) is accepted with probability

$$\alpha = \min\left(1, \frac{p(D|\theta_{l^*}, M_{l^*})p(\theta_{l^*}|M_{l^*})p(M_{l^*})J(M_l|M_{l^*})q(t_{l^*}|\theta_{l^*}, M_{l^*}, M_l)}{p(D|\theta_l, M_l)p(\theta_l|M_l)p(M_l)J(M_{l^*}|M_l)q(t_l|\theta_l, M_l, M_{l^*})} \left|\frac{\partial h_{l,l^*}(\theta_l, t_l)}{\partial(\theta_l, t_l)}\right|\right), \tag{57}$$

where $J(M_{l^*}|M_l)$ is the probability to jump from model M_l to model M_{l^*} and $p(M_l)$ is the prior probability of model M_l.

5.4.1 Jumping probabilities

In the software RJMCMC is used for sampling from models with different number of input variables. The model space contains all the models having $1...K$ inputs, where K is the maximum number of inputs available. In order to change the model we add or remove inputs connected to MLP or GP or add and remove one input (in which case the total number of inputs does not change).

The acceptance ratio, equation (57), in the software is constructed as follows. When adding a new input the function $(\theta_{l^*}, t_{l^*}) = h_{l,l^*}(\theta_l, t_l)$ is set as identity, that is, $\theta_{l^*} = (\theta_l, t_l)$, and the conditional prior of the new parameters is used as the proposal distribution q. In this case the Jacobian determinant in the equation (57) is 1, the prior terms for the parameters common to both models cancel out and the prior and the proposal distribution for the new parameters cancel out. However, as discussed in the section 3.1.4, the hyperparameters can be scaled according to the number of inputs or hidden layer units. Because of scaling the prior of certain hyperparameters changes when adding or removing inputs which has to be taken into account when evaluating the acceptance ratio. After these changes the equation (57) simplifies to

$$\alpha = \min\left(1, \frac{p(D|\theta_{l^*}, M_{l^*})p(\theta_{l^*}|M_{l^*})p(M_{l^*})J(M_l|M_{l^*})}{p(D|\theta_l, M_l)p(\theta_{l^*}|M_{l^*})p(M_l)J(M_{l^*}|M_l)}\right), \tag{58}$$

where the prior terms $p(\theta_{l^*}|M_{l^*})$ corresponds to the effect of scaling. There are couple of possibilities for the jumping probability. Let k denote the number of inputs in the current model and $M_{l,k}$ the model l with k inputs. we can write the jumping probabilities as follows.

1. Randomly pick one input and change its state. Only the state of the chosen input can be changed or with the probability $p(\text{switch})$ an other input with different state is chosen and the states of these two inputs are switched. The probability $p(\text{switch})$ can be adjusted by user as discussed in section 3.2.3. The jumping probabilities are

$$J(M_{l^*,k-1}|M_{l,k}) = \begin{cases} \frac{1}{K} & \text{if } k = K \\ (1 - p(\text{switch}))\frac{1}{K} & \text{if } 1 < k < K \\ 0 & \text{if } k = 1 \end{cases} \tag{59}$$

$$J(M_{l^*,k+1}|M_{l,k}) = \begin{cases} 0 & \text{if } k = K \\ (1 - p(\text{switch}))\frac{1}{K} & \text{if } 1 < k < K \\ (1 - p(\text{switch}))\frac{1}{K-1} & \text{if } k = 1 \end{cases} \tag{60}$$

$$J(M_{l^*,k}|M_{l,k}) = \quad p(\text{switch})\frac{1}{K}\left[\frac{1}{k} + \frac{1}{K-k}\right] \quad \text{if } k \neq K. \tag{61}$$

2. Randomly add or remove one input or change the state of two random inputs. With the probability $p(\text{switch})$ the state of two random inputs are changed. With probability $1 - p(\text{switch})$ It is randomly chosen whether to remove or add one input, with probabilities $p(\text{death})$ and $1 - p(\text{death})$ respectively. The jumping probabilities are

$$J(M_{l^*,k-1}|M_{l,k}) = \begin{cases} \frac{1}{K} & \text{if } k = K \\ (1 - p(\text{switch}))p(\text{death})\frac{1}{k} & \text{if } 1 < k < K \\ 0 & \text{if } k = 1 \end{cases} \tag{62}$$

$$J(M_{l^*,k+1}|M_{l,k}) = \begin{cases} 0 & \text{if } k = K \\ (1 - p(\text{switch}))(1 - p(\text{death}))\frac{1}{K-k} & \text{if } 1 < k < K \\ (1 - p(\text{switch}))\frac{1}{K-1} & \text{if } k = 1 \end{cases} \tag{63}$$

$$J(M_{l^*,k}|M_{l,k}) = \quad p(\text{switch})\frac{1}{K}\left[\frac{1}{k} + \frac{1}{K-k}\right] \quad \text{if } l \neq N. \tag{64}$$

A Monitoring convergence

Before the samples obtained from the *Markov chain Monte Carlo* sampling are from the equilibrium distribution they cannot be used in evaluation of the expectation of output. The number of samples, *burn-in*, needed to reach the equilibrium and the autocorrelation time can be approximated by various methods. The Markov Chain Monte Carlo diagnostics tools used here are available at http://www.lce.hut.fi/research/compinf/mcmcdiag/.

The first method used here is *potential scale reduction factor* (PSRF) (Brooks and Gelman, 1998). PSRF test estimates when two or more sample chains started from different points are from the same distributions by comparing the between variation and within variation. The PSRF test can be used also for one sample chain, when the factor is calculated between the first and last parts of the chain, for example, first and last third of the chain. Use of only one sample chain will produce over-optimistic results.

Neal (1993, pages 102–114) and Geyer (1992) discuss about the methods to approximate the autocorrelation time, τ, in Markov Chain. After the autocorrelation time is determined we can thin the sample chain by taking in only every τ'th sample. The method used to determine τ is *Geyer's initial monotone sequence estimator*.

As an additional test against non-convergence *Kolmogorov-Smirnov test* (Robert and Casella, 2004, p. 466) is used for two independent sample chains. The KS test approximates if two sample chains are from the same distribution. The thinning of sample chain has to be done before using KS-test, because the test assumes independence of samples. KS test can be used also for several chains by making several pairwise comparisons.

The sampling parameters and the length of the sample chain must be selected so that the test values from above tests are acceptable. Autocorrelation time can be used to tune the sampling options. The greater τ is the longer chain has to be sampled in order to get enough effective samples. The autocorrelation time should not be much more than 5% of all samples; otherwise the estimation of the autocorrelation time is unreliable. Number of samples divided by the autocorrelation time tells roughly the effective sample size. In the demos the PSRF test was used first to approximate the convergence and the size of burn-in. The sample chain is likely to be converged when R value from PSRF is under 1.1; the closer R is to one the better it is. The number of burn-in can be approximated with PSRF by omitting, for example 10%, of samples from the beginning of the sample chain and testing the remaining chain with PSRF. Geyer's Initial Monotone sequence estimator was used to determine the autocorrelation time, after which the sample chain was thinned. Only every τ'th sample after the number of burn-in were retained. After thinning the stored sample chain is approximately independent and the KS-test can be used as an additional test to indicate if there is convergence problem.
The sample chain might also seem as converged to the equilibrium, but for example may be stuck in a local mode.

The posterior distribution used in the regression problem example, `demo_2input` was checked using functions `psrf`, `geyer_imse` and `ks` from MCMC diagnostics toolbox.

B Function references

Here are listed all the functions that are treated in the documentation. For complete function list of software package see the Contents.m file in the software.

function	page	Description
batch	18	Batch MCMC sample chain and evaluate mean/median of batches
gibbs	63	Gibbs sampling.
gp2	40	Create a Gaussian Process.
gp2_mcopt	45	Default options for gp2r_mc and gp2b_mc.
gp2fwds	46	Forward propagation through Gaussian Processes.
gp2pak	41	Combine GP hyperparameters into one vector.
gp2r_mc	44	Monte Carlo sampling for model gp2r.
gp2unpak	41	Separate GP hyperparameter vector into components.
hmc2	59	Hybrid Monte Carlo sampling.
hmc2_opt	17	Default options for hybrid Monte Carlo sampling.
invgam_p	43	Create inverse-Gamma prior.
laplace_p	12	Create Laplace (double exponential) prior.
mlp2	8	Create a 2-layer feed-forward network without activation.
mlp2b_mc	14	Monte Carlo sampling for model mlp2b.
mlp2b_steps	61	Calculate heuristic step sizes for 2-layer network.
mlp2c_mc	14	Monte Carlo sampling for model mlp2c.
mlp2c_steps	61	Calculate heuristic step sizes for 2-layer network.
mlp2fwds	16	Forward propagation through 2-layer networks.
mlp2index	13	Create indexes for mlp2.
mlp2normp	13	Create Gaussian prior for mlp.
mlp2pak	9	Combines weights and biases into one weight vector.
mlp2r_mc	14	Monte Carlo sampling for model mlpr.
mlp2r_mcopt	16	Default options for lp2r_mc.
mlp2r_steps	61	Calculate heuristic step sizes for 2-layer network
mlp2unpak	9	Separates weight vector into weight and bias matrices.
norm_p	12	Create Gaussian (multivariate) (hierarchical) prior.
recappend	15	Record append, internal function in mlp2r_mc
	44	Record append, internal function in gp2r_mc
scges	21	Uses a scaled conjugate gradients algorithm to find a local minimum of the function
scges_opt	21	Default options for scges.
t_p	13	Create student t prior.
thin	18	Delete burn-in and thin in MCMC-chains.

References

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.

Brooks, S. P. and Gelman, A. (1998). General Methods for Monitoring Convergence of Iterative Simulations. In *Journal of Computational and Graphical Statistics*, volume 7, pages 434–455. American Statistical Association, Institute of Mathematical Statistics, and Interface Foundation of north America.

Gelman, A., Carlin, J. B., Stern, H. S., and Rubin, D. B. (2004). *Bayesian Data Analysis, Second Edition*. Capman & Hall/CRC.

Geweke, J. (1993). Bayesian Treatment of the Independent Student-*t* Linear Model. *Journal of Applied Econometrics*, 8(Supplement):S19–S40.

Geyer, C. J. (1992). Practical Markov Chain Monte Carlo. *Statistical Science*, 7(4):473–511.

Gilks, W., Richardson, S., and Spiegelhalter, D. (1996). *Markov Chain Monte Carlo in Practice*. Chapman & Hall.

Green, P. J. (1995). Reversible Jump Markov Chain Monte Carlo Computation and Bayesian Model Determination. *Biometrika*, 82(4):711–732.

Green, P. J., Hjort, N. L., and Richardson, S. (2003). *Highly Structured Stochastic Systems*. Oxford University Press.

Lampinen, J. and Vehtari, A. (2001). Bayesian Approach for Neural Networks – Review and Case Studies. *Neural Networks*, 14(3):7–24.

Liu, J. S. (2001). *Monte Carlo Strategies in Scientific Computing*.

Nabney, I. T. (2001). *NETLAB: Algorithms for Pattern Recognition*. Springer.

Neal, R. M. (1993). Probabilistic Inference Using Markov Chain Monte Carlo Methods. Technical Report CRG-TR-93-1, Dept. of Computer Science, University of Toronto.

Neal, R. M. (1996). *Bayesian Learning for Neural Networks*. Springer.

Neal, R. M. (1997). Monte Carlo Implementation of Gaussian Process Models for Bayesian Regression and Classification. Technical Report 9702, Dept. of statistics and Dept. of Computer Science, University of Toronto.

Neal, R. M. (1999). Regression and Classification Using Gaussian Process Priors. In Bernardo, J. M., Berger, J. O., Dawid, A. P., and Smith, A. F. M., editors, *Bayesian Statistics 6 - Proceedings of the Sixth Valencia International Meeting*, pages 475–501. Oxford University Press.

Neal, R. M. (2005). The Short-Cut Metropolis Method. Technical Report 0506, Dept. of statistics and Dept. of Computer Science, University of Toronto.

Paciorek, C. J. and Schervish, M. J. (2004). Nonstationary Covariance Functions for Gaussian Process Regression. In Thrun, S., Saul, L., and Schölkopf, B., editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA.

Ripley, B. D. (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press.

Robert, C. P. and Casella, G. (2004). *Monte Carlo Statistical Methods, Second edition*. Springer.

Vehtari, A. and Lampinen, J. (2001). Bayesian Input Variable Selection Using Cross-Validation Predictive Densities and Reversible Jump MCMC. Technical Report B28, Helsinki University of Technology, Laboratory of Computational Engineering.

Vehtari, A. and Lampinen, J. (2002). Bayesian Model Assessment and Comparison Using Cross-Validation Predictive Densities. *Neural Computation*, 14(10):2439–2468.

Vehtari, A., Särkkä, S., and Lampinen, J. (2000). On MCMC Sampling in Bayesian MLP Neural Networks. In *Proceedings of the IJCNN'2000*, volume 1, pages 317–322. IEEE Computer Society.