Bayesian learning in multi-layer perceptron neural network using Monte Carlo: `mlp-mc-1`

Carl Edward Rasmussen

September 16th 1996

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

For a full description of the method the reader is referred to (Neal 1996). Here a brief description of the algorithm will be given, along with the heuristics employed. A feed forward multi-layer perceptron neural network with a single hidden layer of hyperbolic tangent units is used; the network is fully connected, including direct connections from the input to the output layer. The output units are linear. All units have biases. A network with a single output, I inputs and H hidden units implements the function

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

where v_h are the hidden-output weights, b_0 the output-bias, w_i the direct input-output weights, a_h the hidden-biases, and u_{ih} the input-hidden weights. All the network parameters are given prior distributions specified in terms of hyperparameters. Predictions are made using Monte Carlo samples of the posterior distribution of weights.

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

$$p(\theta|\mathcal{D}) \propto p(\theta) p(t^{(1)}, \ldots, t^{(n)} | x^{(1)}, \ldots, x^{(n)}, \theta), \tag{2}$$

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

$$p(\tau) \sim \mathrm{Gamma}(\mu, \alpha) \propto \tau^{\alpha/2-1} \exp(-\tau\alpha/2\mu), \tag{3}$$

where μ is the mean and α is known as the shape parameter. We use $\mu = 400$ (corresponding to $\sigma = 0.05$) and $\alpha = 0.5$, which is intended as a vague prior, allowing noise variances down to 10^{-4} and up past unity. The density for the resulting prior is depicted in fig. 1.

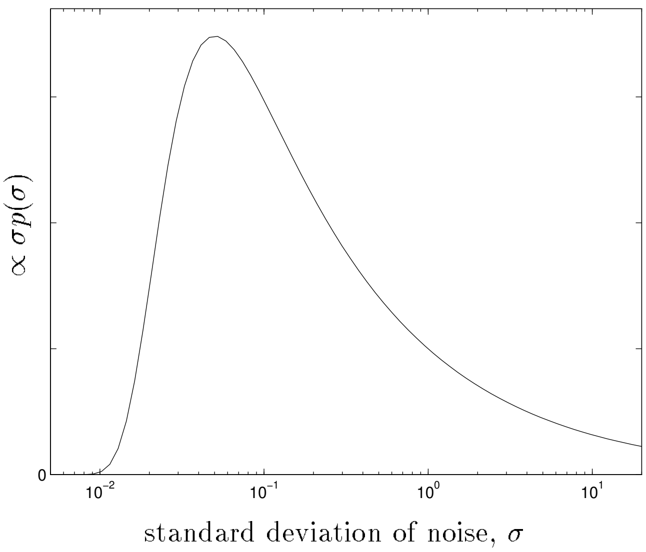

Figure 1: Non-normalised prior on the noise level σ. The function $\sigma p(\sigma)$ has been plotted against σ on a log scale to allow the usual interpretation of probabilities as being proportional to areas under the curve.

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

The output-biases are given zero-mean Gaussian priors $b_o \sim \mathcal{N}(0, \sigma^2)$ with a standard deviation of $\sigma = 1$. Since the targets in the training set will have been normalised by DELVE to have roughly zero mean and unit variance, this prior should accommodate typical values.

The group of hidden-unit biases is given a hierarchical prior consisting of two layers: $a_h \sim \mathcal{N}(0, \sigma^2)$; the distribution of σ is specified in terms of a precision $\tau = \sigma^{-2}$ and given a Gamma form as eq. (3) with parameters: $\mu = 100$ and $\alpha = 0.5$. I have not attempted to plot the resulting prior on the weights (obtained by integrating out τ) since the dependencies between weights through the common value of τ makes it difficult to give a faithful representation of the distribution.

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

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

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

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

$$p\big(t^{(n+1)}\big|x^{(n+1)}, \mathcal{D}\big) = \int p\big(t^{(n+1)}\big|x^{(n+1)}, \theta\big)p(\theta|\mathcal{D})d\theta \simeq \frac{1}{T}\sum_{t=1}^{T} p\big(t^{(n+1)}\big|x^{(n+1)}, \theta^{(t)}\big), \qquad (4)$$

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

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

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

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

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

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

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

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

The Markov chain described above is then invoked and run for as long as desired, eventually producing networks from the posterior distribution. The initial 1/3 of these nets are discarded, since the algorithm may need time to reach regions of high posterior probability. Networks sampled during the remainder of the run are saved for making predictions according to eq. (4). I use 100 samples from the posterior to approximate the integral. Probably the predictions would not get vastly different if more samples were used in the approximation and this has been avoided because the disk requirements for storing these network samples become prohibitive.

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

All runs used the parameter values given above. The only check that is necessary is that the rejection rate stays low, say below 5%; if not, the step size should be lowered. In all runs reported here, $\varepsilon = 0.15$ was adequate. The parameters concerning the Monte Carlo method and the network priors were all selected based on intuition and on experience with toy problems. Thus no parameters need to be set by the user, save the specification of the allowable cpu time. For the results in the DELVE archive, I allowed the time to scale linearly with the number of training cases. I used 30 seconds per training case on our 200 MHz R4400/R4010 processor. This allows the runs to be done in a reasonable amount of time. It is possible that more accurate predictions can be obtained using longer runs.

Implementation

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

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

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

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

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

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

References

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

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

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

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

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

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