MCMC Training of Bayesian Neural Networks

Radford M. Neal

University of Toronto

radford@stat.utoronto.ca
https://www.cs.utoronto.ca/~radford
https://radfordneal.wordpress.com

Virtual seminar, Fields Institute, 16 May 2022

I. Bayesian neural network models

II. Implementaton with Markov Chain Monte Carlo

III. Some experiments

IV. Future work

Multilayer Perceptron Neural Networks

I'll be talking about regression and classification models that utilize multilayer perceptron neural networks.

The network computes outputs, $o_1, \ldots, o_q$, from inputs, $x_1, \ldots, x_p$, using some number of layers of hidden units.

For example, with two hidden layers of N and M units:

$$o_k(x) \;=\; c_k + \sum_{j=1}^{M} w_{jk} g_j(x), \quad g_k(x) \;=\; G(b_k + \sum_{j=1}^{N} v_{jk} h_j(x)), \quad h_k(x) \;=\; F(a_k + \sum_{j=1}^{p} u_{jk} x_j)$$

F and G are the activation functions for the layers. I will be using either tanh or $\text{softplus}(s) = \log(1 + \exp(s))$ as activation functions.

The parameters of this network are the weights, u, v, and w, and biases, a, b, and c. I'll use θ to denote all of them.

More possibilities:

- Direct connections from inputs to hidden layers beyond the first (or to outputs).
- Skip connections from a layer to a layer beyond the next one (or to outputs).
- Connections between layers with sparsity or weight-sharing (eg, convolutional layers).

Models From Neural Networks

I'll use multilayer perceptron networks to build a non-linear regression or classification model, which specifies the distribution of a target variable given the input variables.

For regression, we use a network with one output, $o(x)$, and let

$$y \,|\, x, \theta, \sigma \quad \sim \quad \text{Gaussian}(o(x), \sigma^2)$$

Note that $o(x)$ depends on the network parameters, θ, and that σ is an additional model parameter.

For binary classification, we can use one output in a logistic model:

$$P(y = 1 \,|\, x, \theta) \quad = \quad \frac{1}{1 + \exp(-o(x))}$$

For classification with q classes, we use as many outputs as classes in a "softmax" (multinomial logit) model:

$$P(y = k \,|\, x, \theta) \quad = \quad \frac{\exp(o_k(x))}{\sum_{j=1}^{q} \exp(o_j(x))}$$

Non-Bayesian Training — Old and New

Neural network models can be trained by batch gradient descent in some error measure. When the error is minus log probability of the targets, this is maximum likelihood estimation. Or one can do on-line gradient descent (look at one case at a time).

Overfitting the training data is quite likely. To avoid this, typical practice circa 1990 might involve

- Limiting complexity of the network (eg, number of layers, number of hidden units in each layer).
- Using regularization (eg, weight decay), with batch or on-line gradient descent.
- Stopping training early — either deliberately, using a validation set, or by accident, because computers were too slow to train to the point of overfitting.

More modern practice:

- Much more complex networks, much more data.
- Gradient descent with mini-batches.
- Stochastic modifications such as dropout.
- Early stopping, ensembles.

But why does it work?

The Bayesian Approach to Neural Network Models

A network with only a few hidden units is not a reasonable model — seldom is the real function we need a sum of three tanh functions, for example.

But a network with many hidden units is a reasonable "non-parametric" model, since such a network can approximate any function arbitrarily closely, with enough hidden units.

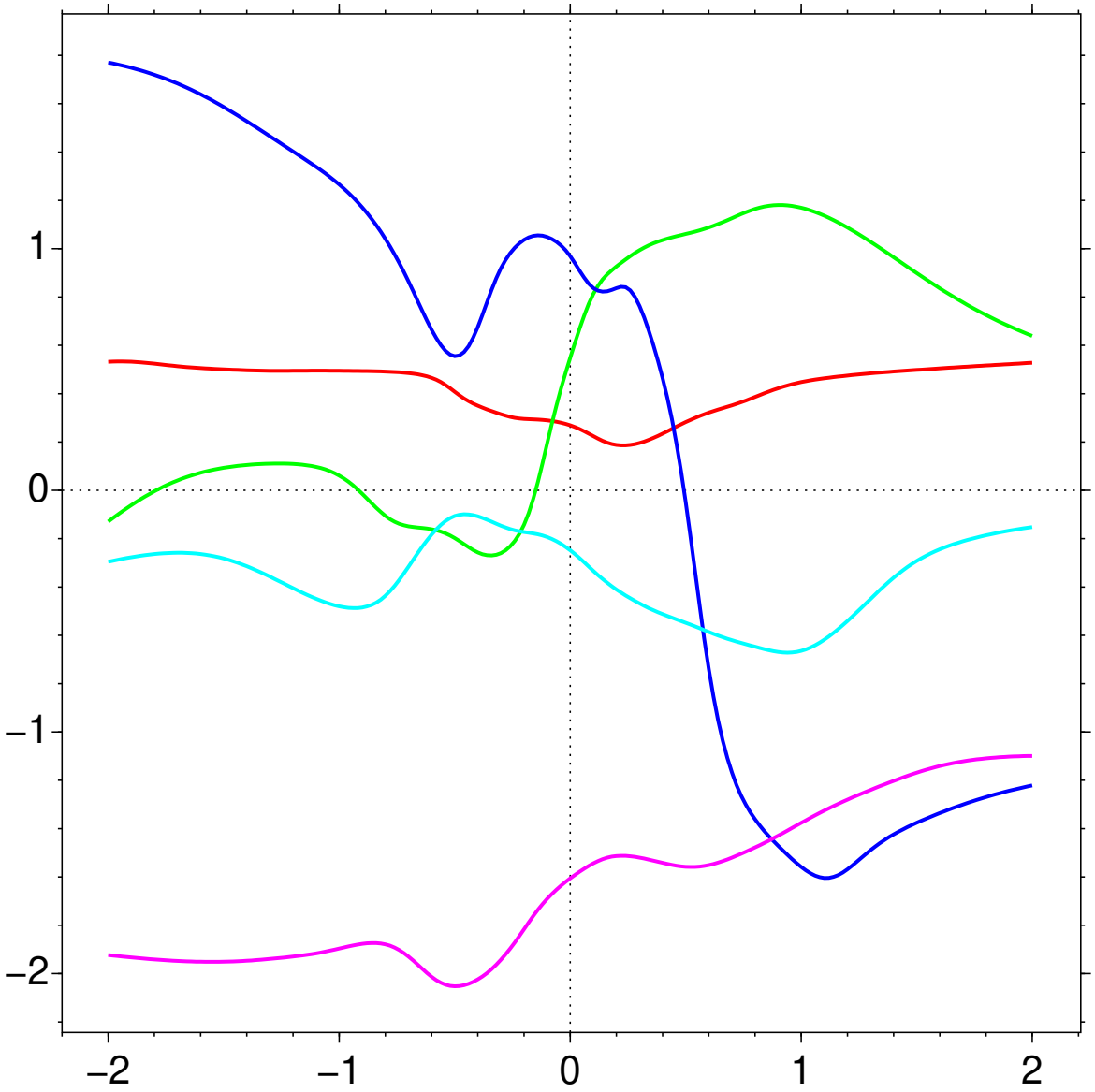
So we should use a network with many hidden units. But then maximum likelihood estimation will lead to drastic overfitting, and poor generalization.

The Bayesian approach [MacKay 1992, Neal 1995] uses probability throughout:

- Define a *prior distribution* over network parameters, representing what we think the actual function might be before seeing any data.
- Find the *posterior distribution* for parameters, which incorporates the information from both the prior and the training data.
- Find *predictive distributions* for test cases, by averaging the predictions the models makes when using parameters drawn from this posterior distribution.

Ideally, the choice of network and prior are based on our knowledge of the task being solved — though in practice, we're limited by our understanding of what these network models mean, and by our computational resources.

Four Functions Drawn According to a Prior on Network Parameters



Network with 1 input, 1 output, 2 hidden layers, 100 tanh units in each hidden layer

Prior distributions for parameters:

$$
\begin{aligned}
u_{ij} &\sim \text{Gaussian}(0, 4^2) \\
a_k &\sim \text{Gaussian}(0, 3^2) \\
v_{jk} &\sim \text{Gaussian}(0, 0.2^2) \\
b_k &\sim \text{Gaussian}(0, 2^2) \\
w_{jk} &\sim \text{Cauchy}(0, 0.01) \\
c_k &\sim \text{Gaussian}(0, 0.5^2)
\end{aligned}
$$

Gaussian Limit of the Prior Over Functions

[Neal 1995]

Consider the prior over functions defined by a network with one hidden layer, in which the parameters have independent Gaussian prior distributions. We'll look at the limit as the number of hidden units, N, goes to infinity.

For one component, k, of the function, evaluated at a single point, $x^{(1)}$:

$$o_k(x^{(1)}) \quad = \quad b_k \; + \; \sum_{j=1}^{N} v_{jk} h_j(x^{(1)})$$

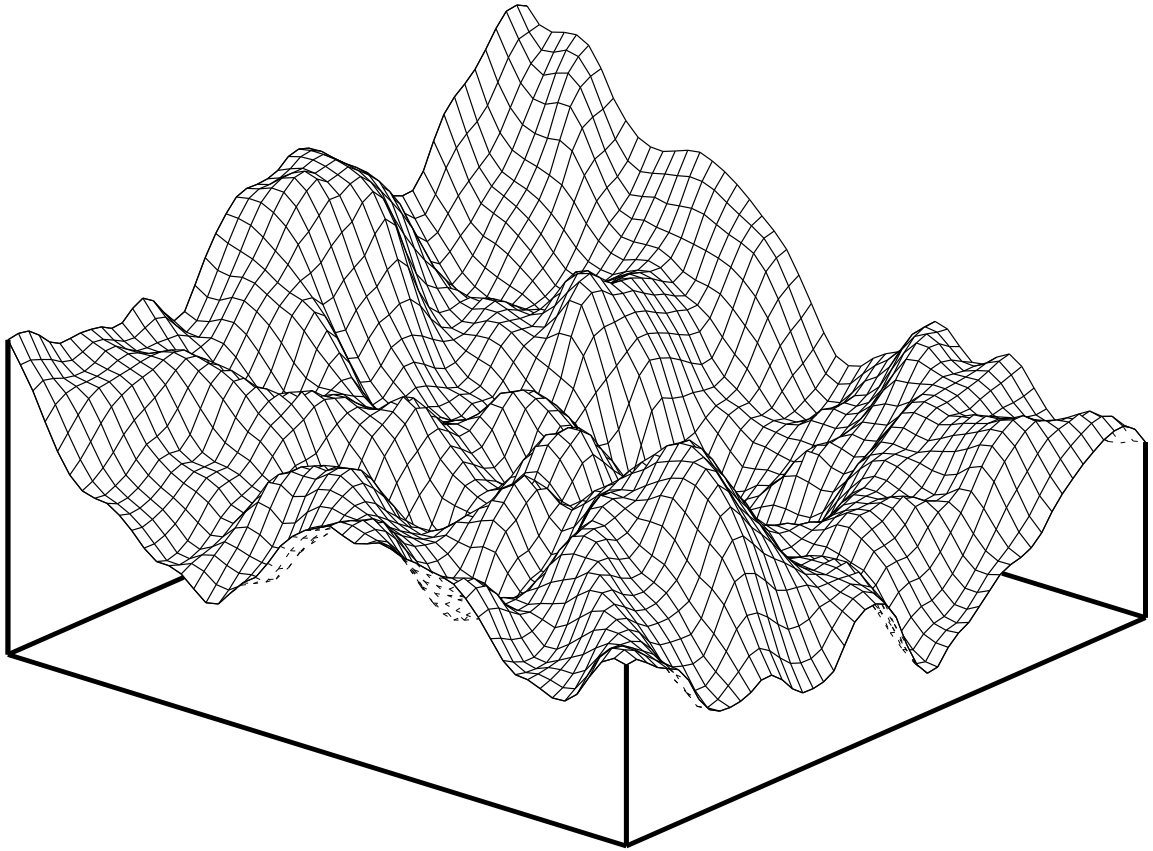
The first term above is Gaussian.

By the Central Limit Theorem, the second term becomes Gaussian as $N \to \infty$, provided each term has finite variance. The variance is finite when u_{jk} is Gaussian and the hidden unit activation function is bounded (such as tanh), and for other common activation functions when using Gaussian priors for weights and biases on input-hidden connections.

Hence $o_k(x^{(1)})$ becomes Gaussian for large N. Its distribution will reach a limit if we make σ_v scale as $N^{-1/2}$.

Similarly, the joint distribution of the function at any number of input points converges to a multivariate Gaussian — i.e. we have a *Gaussian process* prior over functions.

A Function Drawn From a Gaussian Network Prior



The network had two inputs, one output, and 10000 tanh hidden units.

Some Properties of the Gaussian Network Prior

- The hidden-to-output weights go to zero as the number of hidden units goes to infinity. So hidden units are individually of no importance.
- With a smooth hidden unit activation function, the functions are smooth, with

$$\text{Corr}\left[o(x^{(1)}),\, o(x^{(2)})\right] \;\approx\; 1 - |x^{(1)} - x^{(2)}|^2$$

- If the hidden units use a step function, the functions are locally Brownian, with

$$\text{Corr}\left[o(x^{(1)}),\, o(x^{(2)})\right] \;\approx\; 1 - |x^{(1)} - x^{(2)}|$$

- The functions computed by different output units are independent.
- Even with more hidden layers, we still get a Gaussian process prior (with a different covariance function) — the CLT argument can be applied to each hidden layer in turn [Lee, et al, 2017].

Note that if you can compute the covariance function for the Gaussian process, you can do inference directly using matrix operations — no actual network needed!

Priors Based on Non-Gaussian Stable Distributions

[Neal 1995, Der and Lee 2005]

If $X_1, \ldots, X_N$ are i.i.d. from a symmetric stable distribution of index $\alpha \in (0, 2]$,

$$(X_1 + \cdots + X_n)\, N^{-1/\alpha}$$

has the same stable distribution. The same is true as $N \to \infty$ if the X_i are in the "normal domain of attraction" of the stable distribution — a generalization of the Central Limit Theorem.

If we give the weights on connections between layers a prior such that the distribution of the contribution of a unit to a unit in the next layer is in the normal domain of attraction of the stable distribution of index α, and scale the width of the prior as $N^{-1/\alpha}$, we get a well-defined $N \to \infty$ limit.

Example: For tanh units (bounded value), use a Cauchy prior with the width parameter scaling as N^{-1}. More generally, we can use a t distribution with $\alpha \in (0, 2)$ degrees of freedom and scale by $N^{-1/\alpha}$.

Properties of Non-Gaussian Stable Priors

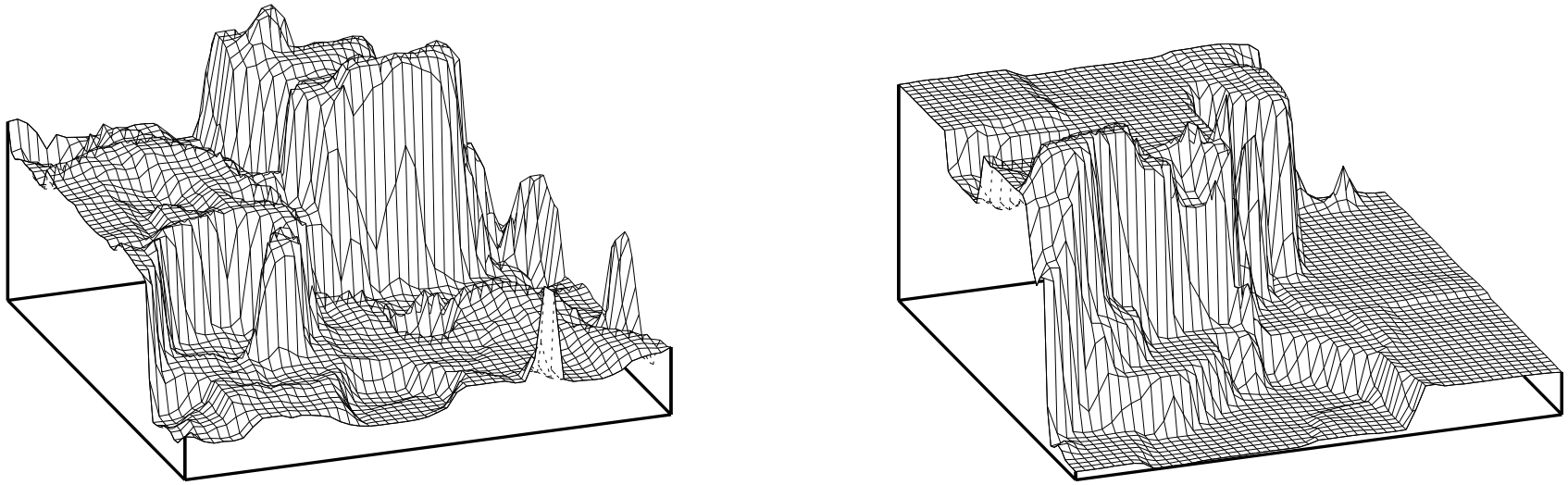
- The hidden-to-output weights do not go to zero as $N \to \infty$, but asymptotically come from a Poisson process (which is an alternative way of defining the prior). So some individual hidden units have significant effects — there can be "hidden features" found from the data.
- The functions computed by different output units are independent. But they can be made dependent without being correlated, by making the weights from the same hidden unit be dependent. "Hidden features" can be shared between outputs.
- Networks with more than one hidden layer (perhaps mixing Gaussian and non-Gaussian priors) can produce interesting new effects.

Analysing non-Gaussian stable processes is much more difficult than analysing Gaussian processes.

There's no feasible way to do inference directly in terms of the limiting process.

So actual networks might be useful after all!

Functions From a Network With Two Hidden Layers

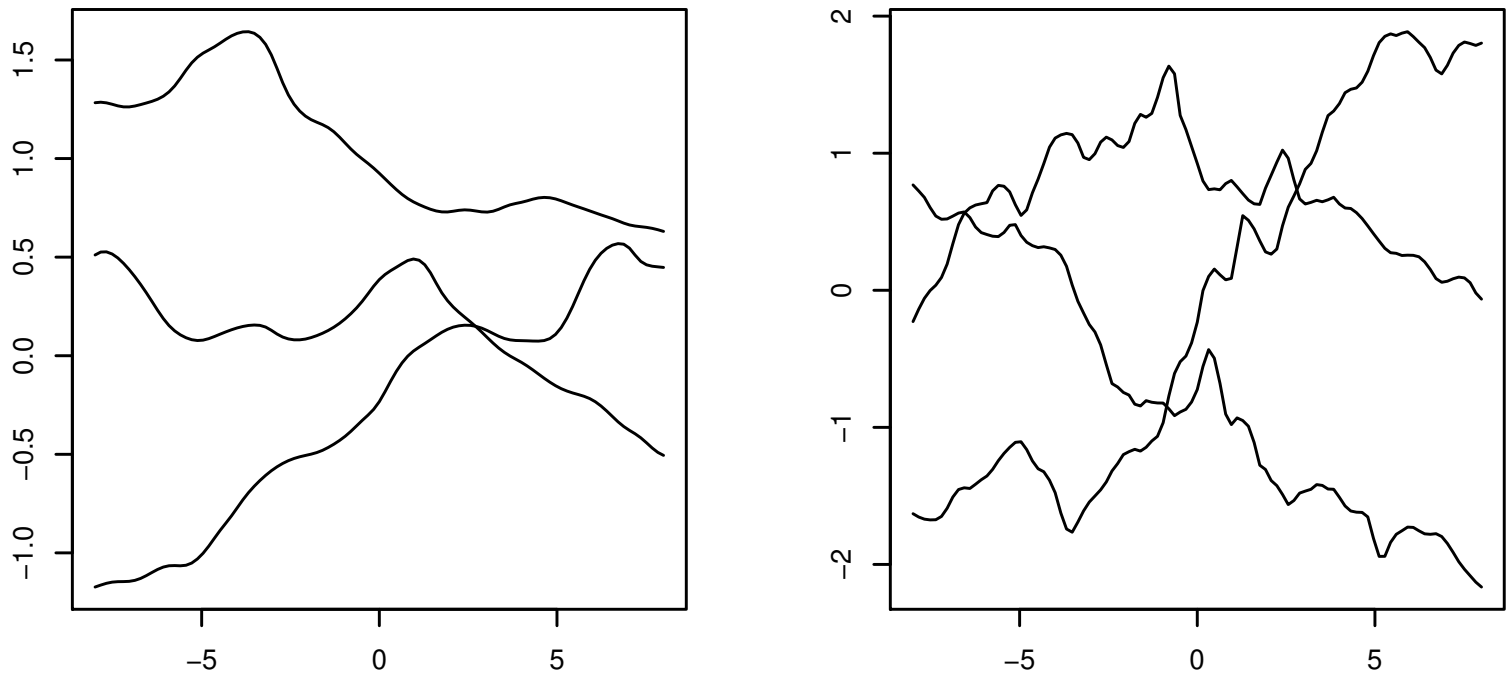


The networks had two inputs and one output. Gaussian priors were used for weights into the first hidden layer (with 500 tanh units) and second hidden layer (300 tanh units), but the prior on the weights from the second hidden layer to the output had a t-distribution with $\alpha = 0.6$. The two functions differ only in the random number seed used when generating weights from their priors.

Hyperparameters for Neural Network Models

The priors we give to weights, such as $u_{ij} \sim \text{Gaussian}(0, \sigma_u^2)$ for input-to-hidden weights, affect the nature of functions drawn from the network prior. Here are samples of three functions (of one input) drawn using Gaussian priors for networks with one hidden layer of 1000 units, using two different σ_u^2:

$\sigma_u = 1$ $\qquad$ $\sigma_u = 5$



A larger σ_u produces "wigglier" functions. Usually, we won't know exactly how wiggly the function should be. So we make σ_u a variable *hyperparameter*, and give it a prior distribution that spans a few orders of magnitude.

Hierarchical Hyperpriors: Automatic Relevance Determination

More elaborate priors for hyperparameters can be used to allow for various possible high-level characteristics of the data. One very useful example is the *Automatic Relevance Determination* prior.

When we have many inputs, we are often uncertain how relevant each is to predicting the target variable. We can express this uncertainty by using a prior for input-to-hidden weights such as the following:

$$
\begin{aligned}
u_{ij} \mid \sigma_{u,i} &\sim \text{Gaussian}(0, \sigma_{u,i}^2), \quad \text{for } i = 1, \ldots, p \text{ and } j = 1, \ldots, N \\
1/\sigma_{u,i}^2 \mid \sigma_{u,*} &\sim \text{gamma}(\text{mean} = 1/\sigma_{u,*}^2, \ \text{shape} = \ldots), \quad \text{for } i = 1, \ldots, p \\
1/\sigma_{u,*}^2 &\sim \text{gamma}(\ldots)
\end{aligned}
$$

Here, $\sigma_{u,i}$ controls the magnitude of weights from input i to the hidden units. If it is small, this input will largely be ignored. The top-level hyperparameter $\sigma_{u,*}$ controls the distribution of the lower-level $\sigma_{u,i}$.

Prediction with a Bayesian Neural Network Model

To make a prediction on a new test case, we average the the model's predictions over the posterior distribution of the parameters. For a classification model, a prediction for the class, y^*, given inputs, x^*, based on n independent training cases, has the form

$$P(y^* = k \,|\, x^*,\, x^{(1)}, y^{(1)}, \ldots, x^{(n)}, y^{(n)}) \;\; = \;\; \int P(y^* = k \,|\, x^*,\, \theta)\, P(\theta \,|\, x^{(1)}, y^{(1)}, \ldots, x^{(n)}, y^{(n)})$$

where the posterior distribution for θ is given by

$$P(\theta \,|\, x^{(1)}, y^{(1)}, \ldots, x^{(n)}, y^{(n)}) \;\; \propto \;\; P(\theta) \prod_{i=1}^{n} P(y^{(i)} \,|\, \theta, x^{(i)})$$

The integral is not going to be analytically tractible, so we must either use some approximation, or use a Monte Carlo method, replacing the integral by an average over values for θ drawn from the posterior distribution.

Benefits of the Bayesian Approach to Neural Network Models

- The meaning of a Bayesian model and prior can be understood by methods such as looking at samples from the prior — it's not an uninterpretable "black box". (Though interpreting complex models may be difficult in practice.)
- The predictions from Bayesian inference incorporate not just uncertainty due to the process being modelled, but also the uncertainty due to our incomplete knowledge of the process (reflected in the posterior distribution of network parameters).
- The model can learn about high level aspects of the problem, such as which inputs are relevant, using hyperparameters that are part of the probabilistic model. In theory, one doesn't need to reserve a validation set for this. (In practice, holding out some data for validation is prudent.)
- Overfitting should not be a problem, if the model and prior are set up correctly. With a good model, that should be so regardless of how much or little data we have. There's no "bias variance tradeoff".

Of course, these benefits can be obtained in practice only if we can feasibly perform the computations needed for Bayesian inference.

I. Bayesian neural network models

II. Implementaton with Markov Chain Monte Carlo

III. Some experiments

IV. Future work

Implementing Bayesian Neural Networks

Integrating over the posterior distribution of network parameters to produce predictions is a very challenging problem. Some approaches:

- Make a Gaussian approximation to the posterior around some mode, based on second derivatives of the log likelihood [MacKay 1992]. But this doesn't capture the true, highly non-Gaussian posterior very well.
- Use a "variational" approximation based on a Gaussian or some other simple class of approximating distributions. Again, not likely to capture the true posterior well.
- Use Markov chain Monte Carlo (MCMC), which is asympotically exact, in the limit of long runs.

  Methods using Hamiltonian Monte Carlo [Duane, et al 1987, Neal 2010] work very well for problems of moderate scale [Neal 1995, Neal 2006, Neal and Zhang 2006].

I am interested in how far (asymptotically) exact MCMC methods can be pushed to work on bigger problems, using bigger networks.

Markov Chain Monte Carlo (MCMC)

The idea:

- Initialize the network parameters and hyperparameters in some way (possibly random).
- Simulate an ergodic Markov chain from that starting state, designed so it converges to the posterior distribution of parameters and hyperparameters, given the training data.
- Use iterations after the point of (approximate) convergence to estimate the predictive distribution for a test case, by averaging the predictions using networks with parameters from each iteration.

Optionally, one might initialize and simulate several chains, and combine states from all chains when making predictions.

But how can we define a Markov chain that converges to the posterior distribution?

One way is to use a random-walk Metropolis algorithm [Metropolis, et al, 1953]:

> For each transition, propose a small change in the state, then accept or reject the change based on the change in posterior probability.

This is a very general technique, applicable to a wide class of models.

Hamiltonian Monte Carlo for Bayesian Neural Networks

[Neal 1995]

Unfortunately, for any but "toy" Bayesian neural network problems, convergence to the posterior distribution using simple random-walk Metropolis updates is hopelessly slow.

But Hamiltonian Monte Carlo (HMC) works much better — using the gradient information obtained by the usual "backprop" method, it can change network parameters by a substantial amount, without the penalty of random walk behaviour.

Each HMC update consists of the following steps:

- Randomly sample "momentum" variables that establish a direction to move in parameters space. The momentum is part of the state, with a Gaussian distribution.
- Simulate (approximately) a Hamiltonian dynamics trajectory for a period of simulated time likely to produce a substantial change in parameters. This will require doing many "leapfrog" steps, which are similar to steps for the standard "gradient descent with momentum" method (using "batch" gradients).
- Accept or reject the end-point of the trajectory by the usual Metropolis criterion: $\Pr(\text{accept}) = \min(1, P(\text{end})/P(\text{start}))$.

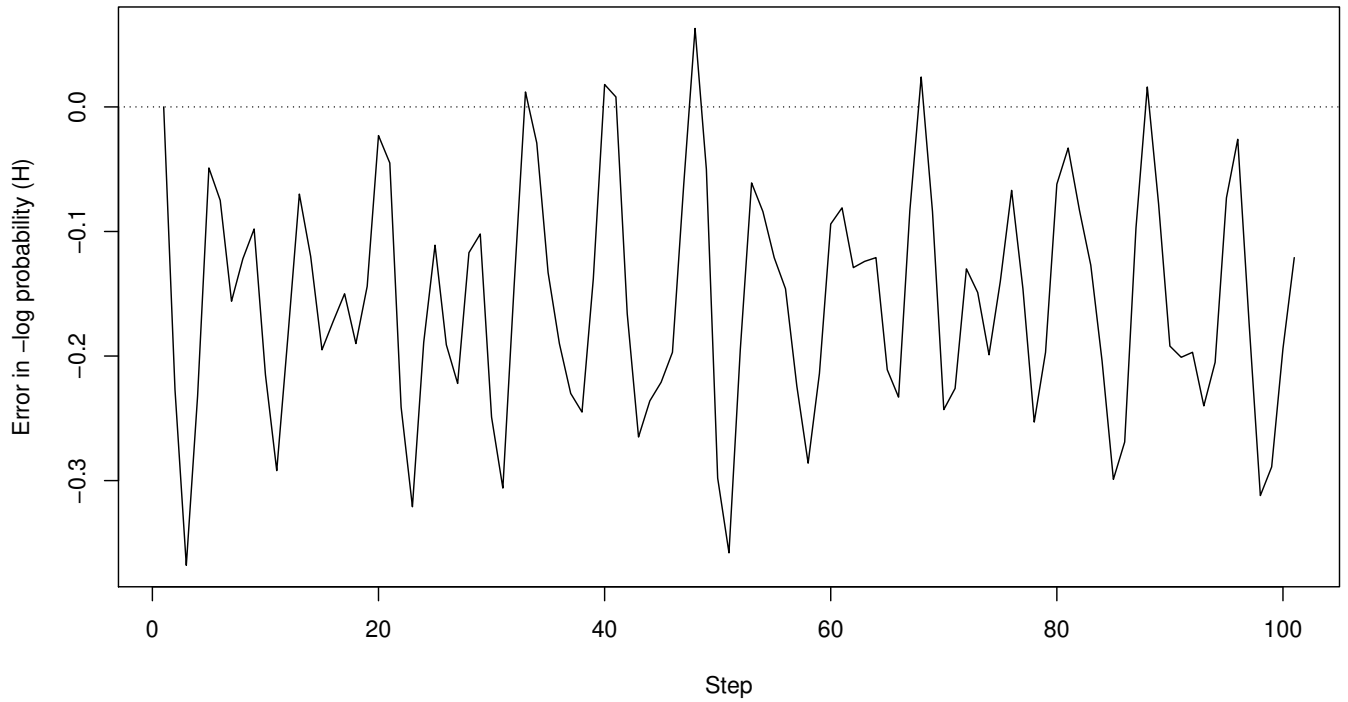
Why Does HMC Work Well?

By simulating Hamiltonian dynamics, we can propose a point in parameter space that is *distant* from the current point, and we have a good chance of *accepting* this point.

If the dynamics were simulated exactly, we would always accept. We actually simulate it inexactly in small steps, which introduces error.

But remarkably, the error often stays bounded — just oscillating along the trajectory.

Example: Error in log probability (of parameters & momentum) along an HMC trajectory, for a convolutional model for CIFAR-10 (with 239,370 parameters):



Setting the Stepsizes for Parameters

Using good stepsizes for leapfrog steps is crucial to getting HMC to work well. I set stepsizes using a heuristic procedure [Neal, 1995, Neal, 2022], based on estimating second derivatives of the log posterior probability with respect to each parameter.

Note that we should set different stepsizes for different parameters — for example, the stepsize on a weight out of an input that is typically large should be less than that for a weight out of an input that is typically small.

For hidden units, we need to estimate how big they typically are. However, it is not valid to set stepsizes based on the *actual* values of hidden units in training cases — that would undermine the reversibility condition needed to prove that HMC samples correctly.

Instead, typical values of hidden units are found assuming that weights in the network have values that are typical given the current hyperparameter values. For this to be valid, the hyperparameters must *not* be updated by HMC. Instead, they are updated in separate Gibbs sampling steps.

(Updating hyperparameters with HMC causes other problems as well.)

Ways of Speeding up HMC

- We can accept/reject based on the *average* probability in "windows" of states at the beginning and end of the trajectory [Neal 1994].

  Advantage: Lower rejection rate. Averaging reduces the change in probability (and possible rejection) when the error in the trajectory has high-frequency variation.

- Instead of avoiding random walks by using long trajectories, we can only partially update the momentum before each short trajectory [Horowitz 1991].

  Advantage: More frequent opportunity to update hyperparameters.

  Disadvantage: For correctness, when a trajectory is rejected, the momentum must be negated, so we head back where we came from. To keep the rejection rate low, we need a small stepsize, hence more leapfrog steps. This can be alleviated by clustering the rejections [Neal 2020].

- Each leapfrog step can be split into K leapfrog steps, each of which uses the gradient on a fraction $1/K$ of the training cases. Analogous to the use of "mini-batchs" in stochastic gradient descent learning.

  Advantage: For large, redundant data sets, this allows larger steps (per full scan).

- Use inexact methods with no accept/reject correction early in a run, when the equilibrium distribution hasn't been reached anyway.

I. Bayesian neural network models

II. Implementaton with Markov Chain Monte Carlo

III. Some experiments

IV. Future work

Software Used for Experiments

Since 1995, I've distributed a package for "Flexible Bayesian Modeling" using neural networks and other models. A new version released 2022-04-21 aims at making the neural network models more useful for larger-scale problems.

Architectural features: Connections can be configured in arbitary fashion, allowing for convolution and weight sharing. Skip connections between all layers are allowed.

Performance improvements: Optimized for using SIMD instructions on CPUs, allows for computation on GPUs with CUDA, has many advanced MCMC methods.

You can get it at `http://www.cs.utoronto.ca/~radford/fbm.software.html`

Also at `gitlab.com/radfordneal/fbm`

Two goals:

- Extend the scale of problems for which Bayesian neural network models are a preferred practical solution.
- Learn about characteristics of Bayesian methods in larger problems (even where the computation needed may not be practical).

The "Bioresponse" task

This is a Kaggle dataset. The task is to predict if a molecule produces some biological response, based on features of the molecule. The details are not documented.

There are 1776 input features (some numerical, some binary), one binary target.

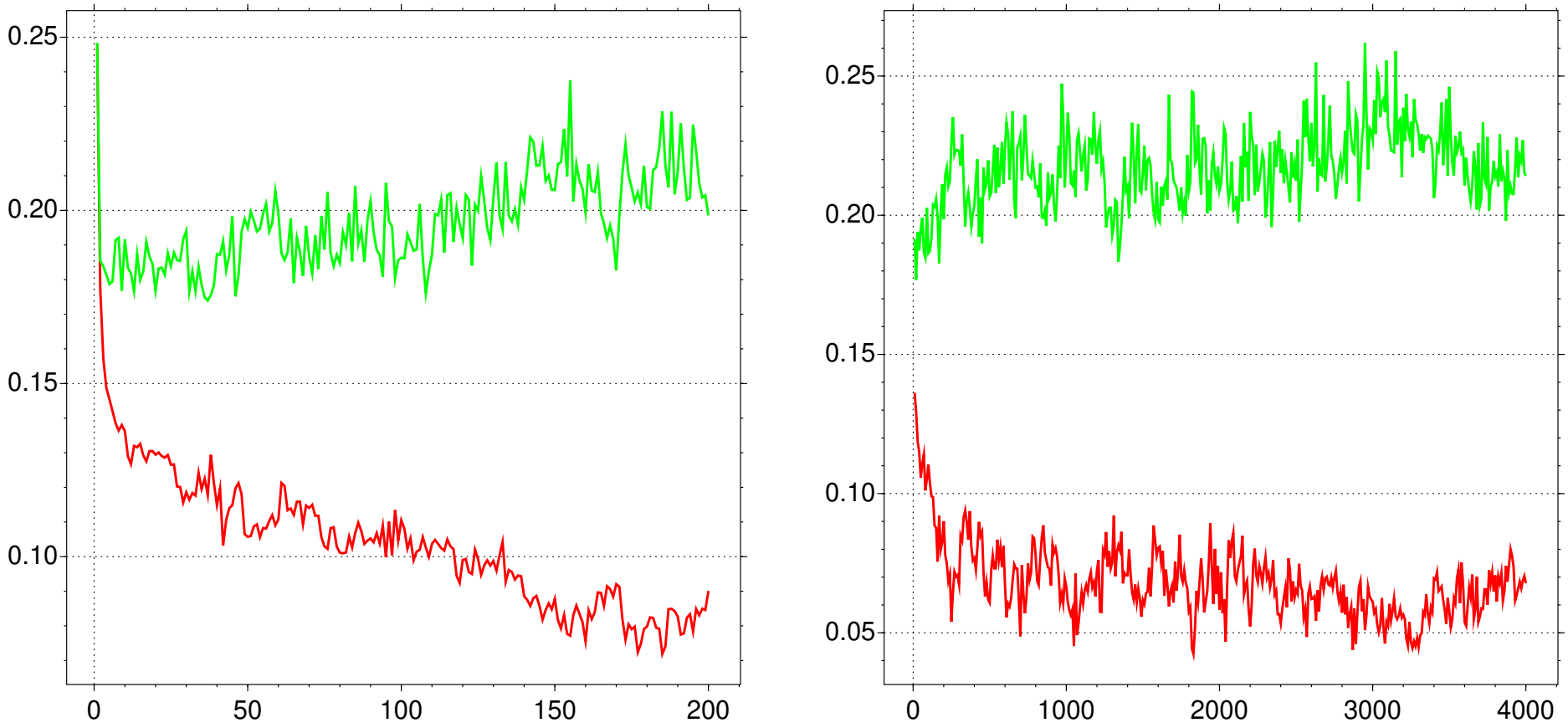
Examination of the data shows that features vary by index, with features in six groups having distinctly different properties. These are presumably different categories of features, which might have different relevance.

3751 training cases are available. I used 3000 for training and 751 for a validation set.

The model used has 17,457 parameters, with:

- Six initial hidden layers (8 softplus units each), each connecting only to one of the six groups of input features. Within each group hyperparameters are used to model differing relevance of inputs. The possibly different relevances of the groups are also modeled with hyperparameters.
- A layer of 48 tanh units connecting to all six initial layers.
- Another layer of 16 tanh units connecting to the previous layer.
- An output unit used for a logistic model, connecting to the previous two hidden layers, with hyperparameters controlling the contribution from each layer.

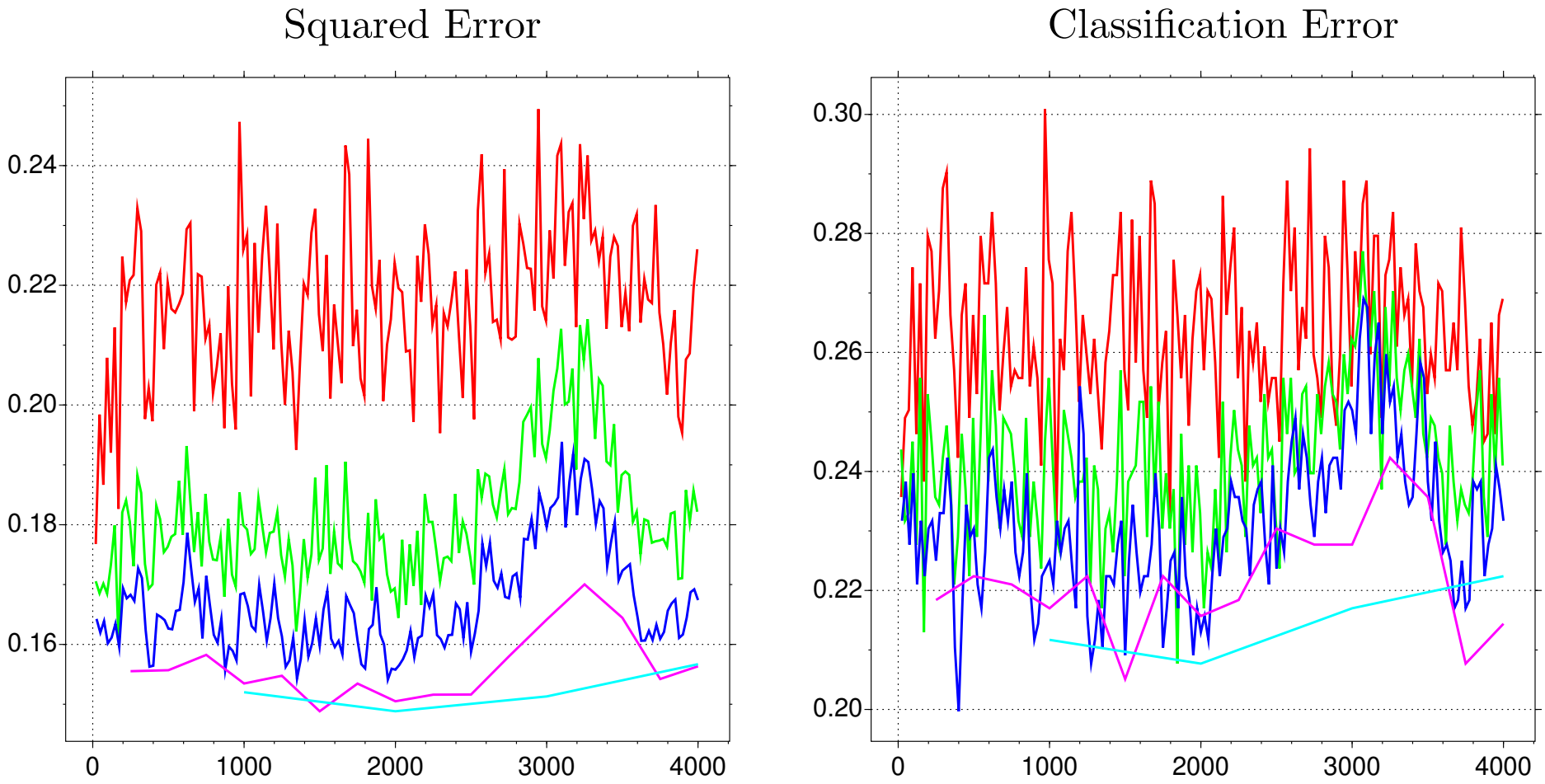
Training and Validation Error During MCMC Run



Squared error on training sed (red) and validation set (green), during the early part of run (left) and full run (right).

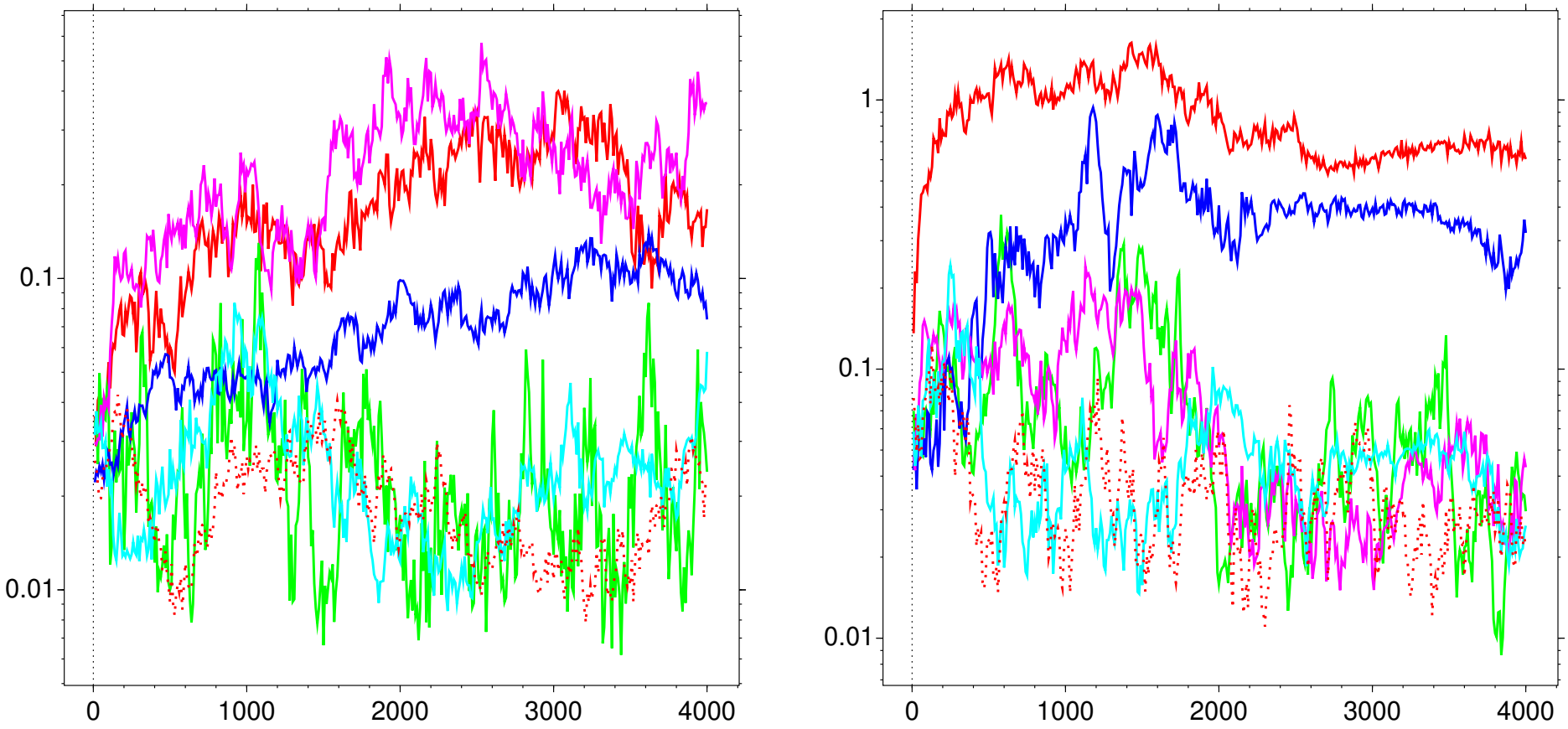
Each iteration does 2500 leapfrog steps (the main computational cost). The full run of 4000 iterations takes 10 hours, using an RTX A4000 GPU (15 GB, 19 TFLOPS).

Predictive Performance Averaging Over Iterations



Peformance using predictions from single iterations (red), and using averages of groups of 5 (green), 25 (blue), 250 (magenta), and 1000 (cyan) iterations.

Hyperparameter Values



Values over the run of the hyperparameters controlling the magnitude of weights from each of the six groups of features to the six initial hidden layers (left), and the hyperparameters controlling the magnitude of weights from each of these hidden layers to the next hidden layer (right).

CIFAR-10

I have also tried Bayesian training of convolutional networks on the CIFAR-10 image classification task (32x32 colour images, 10 classes).

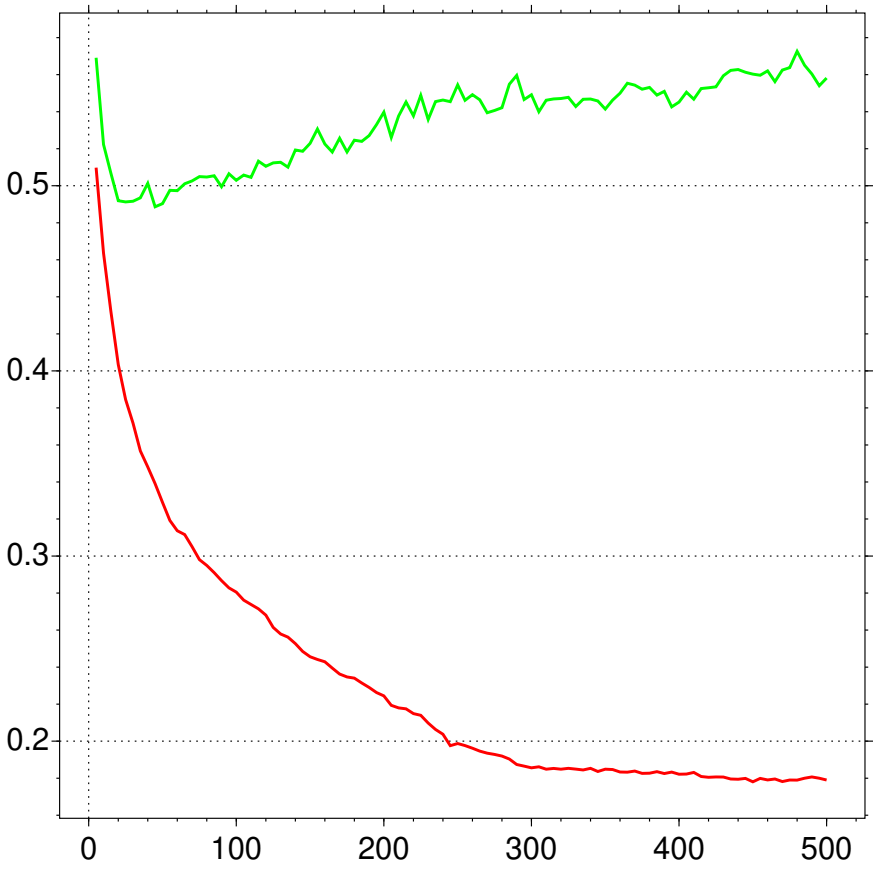
I randomly picked 5000 of the 50,000 training cases for a validation set, and trained on the remaining 45,000 cases. (I haven't been looking yet at the 10,000 case test set.)

The network had 239,370 parameters, with three convolutional and four fully connected hidden layers, as follows:

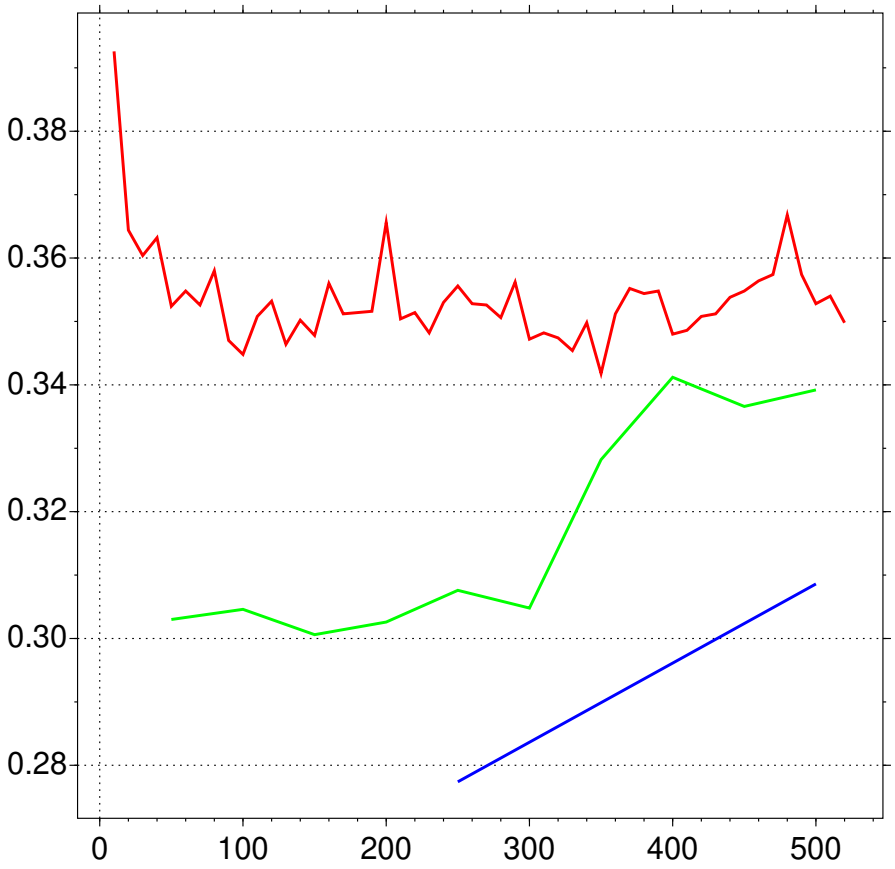
- A convolutional layer (softplus) looking only at intensity information, using 16 filters on 4x4 patches of input.
- A convolutional layer (solftplus) using 48 filters, seeing colour information from 12x12 patches, with stride of 4, and the previous layer, in 9x9 patches with stride 4.
- A convolutional layer (softplus) of 16 filters, looking at 3x3 patches of previous layer.
- A fully-connected layer of 48 tanh units, looking at the two previous layers.
- A fully-connected layer of 128 softplus units, looking at the two previous layers.
- A fully-connected layer of 64 softplus units, looking at the two previous layers.
- A fully-connected layer of 32 tanh units, looking at the three previous layers.
- A softmax output layer, looking at the last four hidden layers.

Training and Validation Errors on CIFAR-10

Squared Error (Training & Validation)



Valid. Classification Error Using Averages



Averages are over 1 (red), 50 (green), and 250 (blue) iterations.

The 500 iterations (each scanning through the training cases 200 times) took 11 days, using an RTX A4000 GPU. The method switched from inexact to exact after iteration 300.

I. Bayesian neural network models

II. Implementaton with Markov Chain Monte Carlo

III. Some experiments

IV. Future work

Insights from Bayesian Learning by Brute Force

Izmailov, Vikram, Hoffman, and Wilson [2021] implemented Bayesian neural network learning with HMC using large computational resources — 512 TPUv3 devices — and assessed how it compared with alternative methods.

- They found very good performance (at high computational cost), on various problems, including CIFAR-10 (with the ResNet-20-FRN architecture).
- Two approximate Bayesian methods (stochastic gradient Langevin dynamics and mean field variational inference) did not mimic the result found with HMC very well (though their results were sometimes fairly good).
- They found that the value of the prior variance for parameters (fixed in their runs) had little effect.
- They found that Bayesian predictions were poor when there was "distribution shift" (test cases from a modified distribution).

The last two results seem puzzling to me. I'd like to investigate what the results are using a better HMC implementation (e.g., a good heuristic for stepsize), with hyperparameters that aren't fixed.

Other Future Work

- Work on how best to update hyperparameters — separately, or within HMC? If within HMC, how best to do it?
- Investigate which combinations of variations on HMC are actually best for large-scale problems.
- Compare with non-exact MCMC methods (eg, [Welling and Teh, 2011]).
- Gain insight into non-Bayesian deep learning. For example, are the very deep networks used necessary to represent the relationship modelled, or is network depth an implicit method of regularization?
- Look at applications of moderate scale, where Bayesian methods may give superior predictive performance at acceptable computational cost.

[Bibliography follows]

Bibliography

Der, R. and Lee, D. (2005) "Beyond Gaussian Processes: On the Distributions of Infinite Networks", *Advances in Neural Information Processing Systems 18*.

Duane, S., Kennedy, A. D., Pendleton, B. J., and Roweth, D. (1987) "Hybrid Monte Carlo", *Physics Letters B*, vol. 195, pp. 216-222.

Horowitz, A. M. (1991) "A generalized guided Monte Carlo algorithm", *Physics Letters B*, vol. 268, pp. 247-252.

Izmailov, P., Vikram, S, Hoffman, M. D., and Wilson, A. G. (2021) "What Are Bayesian Neural Network Posteriors Really Like?", `https://arxiv.org/abs/2104.14421`

Lee, J., Bahri, Y., Novak, R., Schoenholz, S. S., Pennington, J., and Sohl-Dickstein, J. (2017) "Deep Neural Networks as Gaussian Processes", `https://arxiv.org/abs/1711.00165`

MacKay, D. J. C. (1992) "A practical Bayesian framework for backpropagation networks", *Neural Computation*, vol. 4, pp. 448-472.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953) "Equation of state calculations by fast computing machines", *Journal of Chemical Physics*, vol. 21, pp. 1087-1092.

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

Neal, R. .M. (1995) *Bayesian Learning for Neural Networks*, PhD thesis, University of Toronto.

Neal, R. M. (2006) "Classification with Bayesian neural networks", in *Machine Learning Challenges. Evaluating Predictive Uncertainty, Visual Object Classification, and Recognising Textual Entailment*, Springer.

Neal, R. M. (2010) "MCMC using Hamiltonian dynamics", in the *Handbook of Markov Chain Monte Carlo*, S. Brooks, A. Gelman, G. L. Jones, and X.-L. Meng (editors), Chapman & Hall / CRC Press, pp. 113-162. Also available at `arxiv.org/abs/1206.1901`

Neal, R. M. (2020) "Non-reversibly updating a uniform [0,1] value for Metropolis accept/reject decisions", `arxiv.org/abs/2001.11950`

Neal, R. M. (2022) "Neural network models implemented in the Flexible Bayesian Modelling software", included in the documentation for the FBM distribution, and also available at `http://www.cs.utoronto.ca/~radford/fbm.2022-04-21.doc/net-models.PDF`

Neal, R. M. and Zhang, J. (2006) "High dimensional classification with Bayesian neural networks and Dirichlet diffusion trees", in *Feature Extraction: Foundations and Applications*, Springer.

Welling, M. and Teh, Y. W. (2011) "Bayesian Learning via Stochastic Gradient Langevin Dynamics", `http://www.icml-2011.org/papers/398_icmlpaper.pdf`