Monte Carlo methods for Bayesian Neural Networks

Kamran Javid, Will Handley, Mike Hobson & Anthony Lasenby

April 20, 2020

1 Introduction

For an input x and output y, a neural network is a (approximate) function f which maps from x to y. The goal of neural network training is to learn the function f which correctly maps from training input data $x_{\rm tr}$ to training output data $y_{\rm tr}$ but also generalises well to any x and y which were not used in the training of the network.

Traditionally, neural networks are trained via a forward-backward propagation paradigm. In this framework, the parameters of the model θ, are learned by computing the forward propagation of the network with the training data $f(x_{\rm tr}, \theta)$, and minimising some loss function $J(f(x_{\rm tr}, \theta), y_{\rm tr})$ with respect to θ using a backward propagation procedure, which involves calculating the derivative of the loss function with respect to the various model parameters. This of course requires all the (trainable) parameters of the model to be differentiable.

In probabilistic terms, one can view the forward propagation of the neural network and calculation of the corresponding loss function as evaluating a (log) likelihood function $L(y_{\rm tr}|x_{\rm tr}, \theta, f)$. Thus the minimisation of the loss function J is equivalent to the maximisation of (log) L.

Often in neural network training, a regularisation term $R(\theta)$ is added to the objective function:

$$O(x_{\rm tr}, y_{\rm tr}, \theta) = J(f(x_{\rm tr}, \theta), y_{\rm tr}) + R(\theta). \tag{1}$$

R is generally a monotonic function and seeks to 'penalise' large values of θ by increasing the objective function accordingly (which is now the thing to be minimised).

Again this can be viewed probabilistically. The regularisation term is equivalent to assigning a prior $\pi(\theta)$ on the model parameters. In which case the minimisation of O is equivalent to the maximisation of the posterior probability distribution $P(\theta|x_{\rm tr}, y_{\rm tr}, f)$.

The relative weighting of J and R in O is usually dictated by hyperparameters h_1 and h_2, i.e.

$$O(x_{\rm tr}, y_{\rm tr}, \theta) = h_1 J(f(x_{\rm tr}, \theta), y_{\rm tr}) + h_2 R(\theta). \tag{2}$$

Their values however, cannot be 'learned' by simply minimising O (without further regularisation on their values. In the probabilistic framework this is equivalent to treating the variance of the likelihood and priors as being stochastic, see below), as this would trivially lead to $h_1 = h_2 = 0$ assuming they take positive values or 0. Instead cross validation is used to learn these hyperparameters, which in its simplest case works as follows: θ is learned using x_{tr} and y_{tr} as usual, with given values for h_1 and h_2. A separate dataset x_{cv} and y_{cv} is then used to evaluate the performance of f with respect to some performance metric. This process is repeated for different values of the hyperparameters, and the values used (along with the learned values of θ from that particular run) are the ones which score best using the metric evaluated on dataset x_{cv} and y_{cv}. Similarly, network size (both in terms of depth and width) is a hyperparameter which needs optimising, but doing so on the training data (in the traditional way) would almost surely lead to overfitting, as larger networks will often be preferred. Thus the network size is often also optimised during the cross validation procedure. Many other hyperparameters exist, such as number of epochs, batch size, early stopping amount, and activation type. All of these follow a similar optimisation over the cross validation set, to avoid overfitting to the training data.

The training of the regularisation hyperparameters is equivalent to having stochastic variance parameter(s) on the likelihood function and on the prior(s) of θ. In a Bayesian context, this is usually handled by using hierarchical Bayes in which case a prior on the hyperparameters is assigned and thus the training data $x_{\text{tr}}, y_{\text{tr}}$ decides their values (they are usually marginalised over rather than assigned a 'particular' value). Note that this does force one to 'deterministically' assign the parameters of the priors on the hyperparameters (sort of equivalent to the hyperparameters of the hyperparameters h_1 and h_2, 'hyperparameters squared' if you will). It is interesting to note that the treatment of hyperparameters in this way (i.e. assigning themselves a prior) is uncommon in the traditional works (where cross validation is used instead) but rife in the Bayesian neural network literature (utilised by [MacKay 1991], [Neal 1996], and [Gal 2017] to name a few). Even though there is nothing intuitively wrong with using the approach in the former.

2 Training Bayesian Neural Networks

Speaking probabilistically, as already mentioned the aim of traditional methods is to maximise the value of the likelihood or posterior giving rise to a single estimate of $f(x, \theta)$ where the values of θ are obtained from the maximisation using training/cross validation data (the mode of the posterior/likelihood). Bayesian neural networks however, aim to give a probability distribution over f:

$$p(f(x, \theta_i)|x_{\text{tr}}, y_{\text{tr}}), \tag{3}$$

from which one can look at the mode, mean, or median estimate of $f(x)$, as well as the shape of its distribution, and the uncertainty in the estimate of its value.

By looking at the mean estimate of the function output, one is not just looking at the network which the data is fit to (including the prior) the best, but it is averaging over the ensemble of functions considered in the sampling space, weighted by how well each function fits the data (and the prior probability of that network). One might naturally expect this to overfit the training data less.

Restricting ourselves to training using the posterior distribution (i.e. ignoring cases where regularisation is not used), the key difference between traditional and Bayesian neural network training is that, instead of maximising the posterior in the former, the latter samples the posterior and uses these samples,

$$P(\theta_i|x_{\rm tr}, y_{\rm tr}, f), \tag{4}$$

to obtain the distribution over the function output:

$$p(f(x,\theta_i)|x_{\rm tr}, y_{\rm tr}, f) = f(x,\theta_i)P(\theta_i|x_{\rm tr}, y_{\rm tr}, f). \tag{5}$$

Thus the solutions obtained to a Bayesian neural network is a superset of those obtained from traditional methods (which is simply the mode of $p(f(x,\theta_i)|x_{\rm tr}, y_{\rm tr}, f)$).

Another key advantage of Bayesian neural networks is the opportunity to calculate the normalisation factor of the posterior distribution, better known as the Bayesian evidence or the marginal likelihood. The evidence gives a measure of the model's fit to the data, versus how complex the model is, and thus expresses Occam's razor. This is useful because it gives an indication of how well the network fits the training data, versus (roughly) how likely it is to be overfitting the training data. This gives a measure of how well the model fits the training data vs how well it generalises from the training data itself, without having to resort to cross validation or test data. This can be particularly useful in the low data limit, where cross validation isn't really feasible.

In the past it has been suggested that the evidence correlates directly to how well a network performs on data which it wasn't trained on (evaluated by some performance metric, see [MacKay 1991]). Furthermore, in the case where a correlation doesn't exist, it may suggest that the model needs to be reformulated (so that it does indeed show a correlation).

The key disadvantage of Bayesian neural networks is the computational cost. With the exception of Gal's method, training Bayesian neural networks is equivalent to solving an integral over the model parameter space, while the traditional method reduces to an optimisation problem over the same space. Gal's method uses optimisation to train the network, and the only extra computational cost is directly proportional to the number of samples one wishes to draw from the posterior.

3 Monte Carlo Methods, Nested Sampling, and the PolyChord Algorithm

Monte Carlo sampling methods are a broad class of computational algorithms that rely on repeated random sampling of some distribution to obtain a numer-

ical approximation of the true results. In the context of Bayesian inference, this amounts to representing a posterior distribution via a set of weighted samples

$$\mathcal{S} = \{P(\theta_1|x_{\text{tr}}, y_{\text{tr}}, f),\ ..., P(\theta_n|x_{\text{tr}}, y_{\text{tr}}, f)\}, \tag{6}$$

where $P(\theta_i|x_{\text{tr}}, y_{\text{tr}}, f)$ is the weight of each sample and $\sum_{i=1}^{n} P(\theta_i|x_{\text{tr}}, y_{\text{tr}}, f) = 1$. These samples can be used to plot approximations of the true posterior distribution.

[Skilling 2004] introduced a novel sampling method referred to as nested sampling. This algorithm focuses on calculating the evidence, but also generates samples from the posterior probability distribution. Nested sampling exploits the relation between the likelihood and 'prior volume' to transform the N-dimensional (where N is the number of trainable model parameters) integral giving the Bayesian evidence Z:

$$Z = \int L(y_{\text{tr}}|x_{\text{tr}}, \theta, f)\pi(\theta)\mathrm{d}\theta, \tag{7}$$

into a one-dimensional integral. The prior volume X is defined by $\mathrm{d}X = \pi(\theta)\mathrm{d}\theta$, thus X is defined on $[0, 1]$ and we can set

$$X(\lambda) = \int_{L>\lambda} \pi(\theta)\mathrm{d}\theta. \tag{8}$$

The integral extends over the region(s) of the parameter space contained within the iso-likelihood contour $L(y_{\text{tr}}|x_{\text{tr}}, \theta, f) = \lambda$. Under certain conditions on L and π, the evidence integral can be written as (see [Javid 2018] for proof)

$$Z = \int_0^1 L(y_{\text{tr}}|x_{\text{tr}}, X, f)\mathrm{d}X. \tag{9}$$

PolyChord [Handley et al. 2015] is a novel nested sampling algorithm tailored for high-dimensional parameter spaces. PolyChord utilises slice sampling at each iteration to sample within the hard likelihood constraint of nested sampling. The stepping size of the slices are trivially determined due to the algorithm periodically performing contour whitening, which converts a topologically complex parameter space into a relatively easy to navigate transformed coordinate system.

4 Using PolyChord to train Bayesian Neural Networks

One of the key advantages of using PolyChord is that it focuses on calculating the evidence value accurately, even over complex, high-dimensional parameter spaces. Furthermore the algorithm doesn't require derivative information, meaning the model parameters do not have to be differentiable. This is crucial if one wants to treat hyperparameters such as the number of neurons, the

number of layers, type of activation functions as part of the probabilistic inference problem (without having to resort to more qualitative methods such as cross validation). One can marginalise over these hyperparameters when making predictions to give an ensemble average over the universe of possible models as mentioned above with the neural network parameter marginalisation. Furthermore it means that when choosing the activation functions/size of the model, one can concentrate on their effectiveness in modelling the data, rather than their tendency to handle derivatives well (c.f. vanishing and exploding gradients).

Even without considering the evidence or stochastic hyperparameters, because PolyChord doesn't use derivative information, it is an invaluable tool in evaluating the activation functions/different numbers of layers/neurons ability to model the data by just looking at the test set error for different runs, because one does not have to be concerned with numerical issues associated with a particular architecture, only whether the parameter space is being fully explored.

5 Bayesian Neural Network Inference pipeline

Using Python, TensorFlow, Keras and C++, we have implemented a full (parallelised) Bayesian inference pipeline designed for training Bayesian neural networks. The software comprises of two main parts: the sampling algorithm, PolyChord, and the Bayesian neural network suite. The latter can be used with more or less any sampling algorithm which samples in the unit hypercube (and can easily be adapted for other sampling methods). PolyChord is responsible for efficiently sampling the high-dimensional, complex parameter space associated with the universe of neural networks that form the domain of the problem. The Bayesian neural network suite comprises of six main parts:

- Template neural network architectures.
- Jupyter notebooks for the training of neural networks using traditional methods, and evaluating their performance.
- Code for generating samples from the priors and hyperpriors (of the neural network model parameter(s), i.e. h_2), given hypercube values from the sampler.
- Code for evaluating the forward propagation of neural networks given a set of model parameters sampled from the prior, and calculating the likelihood of the data with that model. Also can be used to model the likelihood noise, i.e. treat the likelihood variance(s) (h_1) as stochastic parameters, as with the prior hyperparameters.
- Jupyter notebooks for making predictions using the trained Bayesian neural networks (including the handling of the chains .txt files).
- Post-inference analytics suite- for performing analysis of the performance of the Bayesian neural networks and making comparisons.

5.1 Template Neural Networks

These simply take the input matrix (x) and neural network parameter vector (which holds the weight matrices and the bias vectors) as their argument, and calculate the forward pass of the neural network. The output layer will have a softmax activation if the problem involves classification. Depending on the architecture's complexity, these templates separate implementations in any of: Keras, TensorFlow, NumPy, and C++.

5.1.1 Residual Neural Networks

Residual neural networks (ResNets) are a form of network where previous layers are fed into the activations of later layers, so that information isn't 'forgotten' during the forward and backward propagation. Of course in the context of Bayesian neural networks trained using PolyChord, the latter of these is unimportant. However, promising work from [Lin & Jegelka 2018] suggests that an infinitely long ResNet with blocks consisting of a one neuron hidden layer followed by an input size hidden layer, satisfies the universal approximation theorem. This architecture is a relatively low dimensional system (2 * input size + 1 per block following the implementation of [Lin & Jegelka 2018]), so is promising for BNNs. This MLP ResNet architecture (with an arbitrary number of blocks) and similar ones have been created in Keras, TensorFlow, NumPy, and C++.

5.2 Traditional training methods for Neural Networks

These are simply Jupyter notebooks which use Keras neural network architectures to train the networks using backward propagation. Obviously support built-in Keras metrics and losses, but also custom ones made for comparison with Bayesian neural networks (see below). Can choose optimisation method, number of epochs, batch size, regularisation factors etc. Training certainly isn't optimised as it uses Keras. Probably best way to optimise it is using a good GPU, as I'm not sure Keras supports distributed computing and even if it does, it's probably a lot of work.

5.3 Prior Sampling

5.3.1 Deterministic Hyperparameter Priors

Basic premise is that the class takes the unit hypercube from PolyChord and returns the corresponding samples of the model parameters.

Deterministic hyperparameter priors are ones where the hyperparameters specifying $\pi(\theta)$ are fixed. This is equivalent to fixing the regularisation constant (h_2) in traditional training which uses 'vanilla' regularisation (i.e. not dropout regularisation or some other exotic method). The prior hyperparameters and prior types are specified upon initialisation of the prior class. Furthermore, different neural network parameters can have different priors assigned to them to a certain extent:

- The least granular setup assigns all parameters to the same prior (with the same hyperparameters). Note though that the values sampled for each parameter is of course different.
- The parameters corresponding to the weights multiplying the same input from the previous layer are drawn from a dependent (with one another) prior. This allows a sorted dependent prior to be used, to prevent a degeneracy in the posterior over these weights (see below).
- Further granularity can be specified at the user's wish (e.g. if one wants each parameter to have a different prior/set of hyperparameters). But the current tools don't generate these settings automatically, and so they must be specified manually.

For a hidden layer in a network, if one set of weights multiplying the same input (from the same neuron in the previous layer) do not have a sorted dependent prior assigned to them, the posteriors will be degenerate by a factor $H!$ where H is the number of neurons in the layer in question. Note also that assigning a sorted prior to the biases is also sufficient to lift this degeneracy.

The choice of priors is vast, and all are available in the sorted dependent prior equivalent. A non-exhaustive list of available priors is:

- Uniform
- Log uniform
- Gaussian
- Laplace
- Cauchy
- Gamma
- Delta

Note that it is usual to use Gaussian priors (with zero mean), which corresponds to L2-norm regularisation in the traditional case. Normalising the inputs and outputs of the data to some range around zero can help justify using zero mean priors.

5.3.2 Stochastic Hyperparameter Priors

With stochastic hyperparameters priors the scale parameters of $\pi(\theta)$ are treated as stochastic, meaning that the data decides the influence of the prior on the posterior (by dictating its 'peakedness'). In traditional methods this is equivalent to having the data decide the regularisation constants. Of course, the stochastic hyperparameters require prior distributions (which have deterministic hyperparameters), which are traditionally Gamma distributions.

Neal justifies the choice of Gamma priors for the hyperparameters (with Gaussian priors on the neural network parameters) by showing that when the hyperparameters are marginalised over, the resultant prior over the neural network parameters is a t-distribution which converges to a stable non-Gaussian distribution on the network parameters which in the limit of an infinitely large hidden layer neural network, each neuron has a non-negligible contribution to the modelling of the data. In contrast when just Gaussian priors are used on the network parameters, in the limit of an infinitely large hidden layer the prior over functions is a Gaussian process where the expected contribution from each neuron is zero.

The hierarchical prior is equivalent to having a regularisation term penalising the regularisation constant in the objective function in traditional methods.

During each sampling phase, the stochastic hyperparameters are sampled from their respective distributions (sometimes referred to as hyperpriors in the code documentation). These values are then used as the hyperparameters for the priors from which the neural network parameters are sampled from. The range of priors available for the hyperparameters is the same as the ones discussed in the previous section. The level of granularity possible for the hyperparameter sampling is as follows:

- A single value for the hyperparameter value is sampled for the whole of the neural network (number of stochastic hyperparameters = 1).
- A hyperparameter value is sampled for each layer of the neural network (excluding the input layer. Number of stochastic hyperparameters = number of layers).
- For a given layer, a hyperparameter is sampled for each input to that layer and shared amongst the priors of all network weight parameters which multiply that input. The biases in that layer also share a common sampled hyperparameter (number of stochastic hyperparameters per layer = number of inputs to that layer + 1 for the biases).
- A more exotic combination of stochastic hyperparameters can be used, but must be designed manually by the user as with the prior case above.

5.4 Neural Network as a Likelihood

This part of the code takes the sampled parameters and arranges them in matrices/vectors to fit the shapes of the neural network weights and biases for each layer of the network. It then uses these sampled values along with the input and output data to evaluate the likelihood associated with the data and the forward propagation of the network. The types of likelihood available are:

- Gaussian (c.f. sum of squared errors loss), for regression tasks.
- Average Gaussian (c.f. mean squared errors loss), for regression tasks.

- Multinomial (c.f. sum of categorical cross entropy loss), for classification tasks.
- Average multinomial (c.f. average categorical cross entropy loss without normalisation factor which is a function of the output of the neural network), for classification tasks.

Note also that the package supports stochastic or mini-batch evaluation, though the effect on the evidence values using this (commonly used technique in traditional methods) is not known.

5.4.1 Stochastic likelihood variance

The variance(s) of the likelihood function (h_1, also commonly referred to as the 'noise') can be treated as stochastic, and learned by the data. Like with stochastic prior hyperparameters, this is common in Bayesian machine learning. In traditional methods its value is learned using cross validation. [Hobson et al. 2002] finds that treating the likelihood variance parameters as stochastic (and marginalising over them) can prevent inherently bi-modal distributions being inferred as uni-modal (with the inferred uni-modal peak appearing in between the true modes), leading to incorrect conclusions. The treatment of stochastic likelihood variance here is treated similarly to the stochastic prior hyperparameters, i.e. the scale factor is assigned a prior (usually Gamma), and the training data is used to learn its value. Like the stochastic prior hyperparameters, the stochastic variance is often treated as a 'nuisance parameter' post-inference, and thus is marginalised over when making predictions.

The current infrastructure supports only one stochastic variance parameter (i.e. one stochastic variance for all outputs of the network), but can easily be adapted to accommodate for a separate stochastic term for each output by implementing likelihood functions which accommodate for non-scalar variances.

5.5 Bayesian neural network prediction tools

These Jupyter notebooks load the neural network weight posterior files so that each sample from the posterior (which is a set of neural network parameters, the associated posterior sample weight, and the likelihood) can be loaded into the neural network architecture to evaluate the network. Mathematically this means that for posterior sample $P(\theta_i|x_{\text{tr}}, y_{\text{tr}}, f)$ the neural network parameters (i.e. all of θ_i apart from any stochastic prior hyperparameters, which are not needed in the prediction phase) are loaded into the neural network, and can be used to evaluate $f(x, \theta_i)$.

Furthermore the samples can be looped over (loaded into the neural network and evaluate $f(x, \theta_i)$ and weighted according to $P(\theta_i|x_{\text{tr}}, y_{\text{tr}}, f)$ to give $p(f(x, \theta_i)|x_{\text{tr}}, y_{\text{tr}})$ which is the posterior distribution over the function output. Of course this also extends to being able to calculate the mode, median and any moments of $p(f(x, \theta_i)|x_{\text{tr}}, y_{\text{tr}})$.

The prediction tools also generate the chains (.txt) files for $f(x, \theta_i)$ (for all n samples) from the posterior over θ so that the posterior over $f(x)$ can be plotted in getdist.

5.6 Performance evaluation tools

The performance evaluation tools can be used on the predictions made by either Bayesian neural networks or networks trained using traditional methods. For test data x_{te} and y_{te} the loss functions mentioned in the likelihood section above can be evaluated to see how well the model generalises to out-of-training data. Note that for Bayesian neural networks the function used can be one associated with any posterior sample (i.e. $f(x_{\text{te}}, \theta_i | x_{\text{tr}}, y_{\text{tr}})$ for all i) or one associated with a statistic of the posterior (i.e. mean, mode, median estimate of f).

Furthermore a number of other evaluation metrics have been implemented using Keras and NumPy. These are:

- Accuracy (for classification problems), tells you how many labels the model correctly predicted.
- Precision (for classification problems) for each output class. High precision means that the classifier is very 'picky', and is reserved about assigning a prediction to the class, thus it misses a lot of the instances. A large proportion of cases which is does attribute to the class, probably are the class.
- Recall (for classification problems) for each output class. High recall means that the classifier is 'generous' in assigning to a class, meaning it captures a lot of the instances. On the other hand, it also assigns a number of examples to the class, which don't actually belong to it.
- Calculates the f1 score (for classification problems) for each output class. There is a trade-off between having high precision and high recall. One generally cannot be 'generous' and 'picky' in assigning to a class at the same time. The f1 score gives a measure of the average of the precision and recall. Maximising it should give an 'optimum' combination of the two.
- Confusion matrix (for classification problems) for the set of output classes. see https://www.youtube.com/watch?v=FAr2GmWNbT0 for a great video explaining the confusion matrix for multiclasses. Rows correspond to true classes, columns correspond to predicted (n.b. wikipedia does it other way round). For the binary case:
  - Top left square is true negatives (TN).
  - Top right square is false negatives (FN).
  - Bottom left square is false positives (FP).
  - Bottom right square is true positives (TP).

Quick way to remember these is:

  - First letter is whether prediction is correct. Second letter is nature of prediction (0 = neg, 1 = pos).

For multiclass case, each class has its own TP, TN, FP, FN, and the number of actual examples belonging to each class is the sum of the elements in its row. The four measures:

  - Values along diagonal are the TPs. The values along a class' (true) row excluding the TP value sum to the FN for that class.
  - The number of FPs for a class is the sum of the elements along its (predicted) column minus the TP value.
  - The number of TNs for a class is the sum of the elements in the matrix excluding the (predicted) column and the (true) row elements corresponding to the class.

Quick way to remember these is:

  - First letter is whether prediction is 'quasi-correct' w.r.t. that class i.e. if true value is that class, predict that class or if true value is not that class, did not predict that class (but didn't necessarily predict correct class).
  - Complimentary cases, i.e. predicted that class but isn't that class or didn't predict that class but true value is that class, are both false.
  - Second letter is nature of prediction (predict other class = neg, predicted class = pos).
  - Multiclass rules apply to binary case, as long as TN is applied before TP rule (and thus diagonal is just one element, so not to double count in TN and TP).

- Explained variance score (probably more suited for regression problems). Not entirely sure what it is, but see the sci-kit learn page on the metric.
- Regression R^2 score (more suited for regression problems). Returns well-known R^2 regression coefficient. Calculates separately for each output, then takes average over these.
- Balanced accuracy (for classification) overcomes the problem of imbalanced data, by normalising true positive and true negative predictions by the number of positive and negative samples, respectively, and divides their sum into two.

5.7 Technical details

- Codebase is roughly:
  - $5k$ lines of C++.

  - $6k$ lines of Jupyter notebooks.
  - $4k$ lines of Python.

- Template nn architectures are implemented in either Keras, TensorFlow, NumPy or C++ (using Eigen library). For complicated architectures, only Keras versions will be implemented due to time constraints. For C++ implementations, matrices associated with nns are stored row wise instead of columnwise (n.b. Eigen's default is columnwise) to be consistent with the other three implementations, which are rowwise by default and difficult to change (I believe).

- The traditional method software is implemented in Jupyter notebooks using the Keras language. Could also be done in TensorFlow, but haven't used nn architectures which can't be done in Keras yet.

- Code for evaluating the neural network likelihoods is implemented in either NumPy and C++. Note in the C++ implementation, the likelihoods also work with rowwise stored matrices. Extreme care was taken in the C++ implementation in particular to make the calculations efficient i.e. not copying over arrays etc.

- The prior/hyperprior sampling classes are also implemented in NumPy and C++. Note in this case no matrices are used (only vectors) so rowwise vs columnwise isn't important. Again, the C++ implementation in particular was designed to be efficient in terms of data handling.

- The notebooks used for making predictions using Bayesian nns are implemented using Keras and NumPy, and aren't particularly fast, but should be good enough since the inference takes up the majority of the computational cost.

- The post-inference analytics suite is a combination of built-in Keras tools, custom made functions using Keras and NumPy, and getdist calls.

- For (very) simple profiling tests done so far, a Keras implementation run of training a BNN took roughly 2.5 minutes, a TF implementation took 1 minute, the NumPy implementation took 30 seconds, and the C++ version took 3 seconds to complete. Note these runs were done on a (very) rubbish laptop.

References

[MacKay 1991] MacKay D. J. C., 1991, Bayesian Methods for Adaptive Models

[Neal 1996] Neal R. M., 1996, Bayesian Learning for Neural Networks

[Gal 2017] Gal Y., 2017, Uncertainty in Deep Learning

[Skilling 2004] Skilling J., 2004, Nested Sampling

[Javid 2018] Javid K., 2018, Physical Modelling of Galaxy Clusters and Bayesian Inference in Astrophysics

[Handley et al. 2015] Handley W. J., Hobson M. P., Lasenby A. N., 2015, PolyChord: next-generation nested sampling

[Lin & Jegelka 2018] Lin H., Jegelka S., 2018, ResNet with one-neuron hidden layers is a Universal Approximator

[Hobson et al. 2002] Hobson M. P., Bridle S. L., Lahav O., 2002, Combining cosmological datasets: hyperparameters and Bayesian evidence