PROJECT REPORT

EXPLORING BAYESIAN METHODS TO IMPROVE NEURAL NETWORK LEARNING

February 1, 2017

Santhanakrishnan Ramani, Shane Grigsby, Shirly Montero, Shruthi Sukumar
University of Colorado Boulder

Contents

INTRODUCTION

Neural networks have gained a lot of traction in recent years to solve a range of problems in machine learning, and their application domain is growing. This can mainly be attributed to the ability to build deeper networks thanks to the availability of powerful processors. However, these models often have extremely low interpretability owing to the large number of parameters. For this reason, training these networks and making decisions about its architecture is considered a "black art". As alluded to already, one huge problem associated with building neural networks is tuning its hyperparameters. This has often relied on some expertise with regard to the range of possible values for the parameters and then exhaustively (or randomly) searching over this bounded region. Another problem, closely associated with the *black-box* nature of neural networks, is the training algorithm used to update the weights. The most common method is the Error Backpropagation Algorithm proposed by Rumelhart, Hinton & Williams [9]. Despite its efficiency, neural networks trained using backpropagation are prone to over-fitting and require the use of weight-decay and dropout methods to mitigate this effect. Here, we examine bayesian alternatives to the aforementioned problems.

Our project can be divided into two parts. In *part 1*, we use a Bayesian Optimization based procedure proposed in [10] to tune hyperparameters of a neural network and compare its performance with the conventional methods like Randomized Search Cross Validation. This is done in an attempt to explore an automatic algorithm to tune hyperparameters that uses observed data to intelligently and efficiently explore the search space. In *part 2*, we explore bayesian inference methods like Markov Chain Monte Carlo to infer the weights of the neural network as an alternative to the conventional error backpropagation algorithm. The advantage of this method is that it attempts to sample weights from its true posterior distribution based on the data and addresses the issue of low interpretability. Although the two parts seem fairly disjoint at the moment, the long-term goal would be to provide a bayesian pipeline for a neural network which includes training and model selection.

DATASET

Description

The World Development Indicators dataset is made available by the World Bank, which contains annual indicators of economic development from hundreds of countries around the world. The dataset has also been published on Kaggle for people to perform predictions

with. Our goal here is to predict life expectancy by country in a given year, by using other indicators for all previous years for all the countries present. The Data Science Club at CU had an in-class Kaggle contest set up, where the data was made available in a convenient *.csv* format. The missing values were annotated by -1 in the *.csv* files, since the indicators used all had positive values. Highly correlated indicators with the target labels were removed from the training set like life expectancy among males and females.

Pre-processing

There are missing values in the training data either at random or owing to the unavailability of data. The missing values were imputed with the mean of the non-null values in the field. In addition, we tried imputing with a draw from a Gaussian distribution with the mean and standard deviation of the non-null values in the field. Performance using backpropagation was better with the mean-imputation method and hence was used for the remainder of the project. Other prospective methods to deal with missing values would be to estimate the distribution of the column using Kernel Density Estimation methods and drawing values to perform imputation. However, these methods were not investigated in this project as they proved to be orthogonal to the goal.

In order to avoid the difference in ranges of values in different fields, the values in the training data were normalized by subtracting the corresponding mean of the column and dividing by the standard deviation. In effect, the z-score of the values were returned as training data. For the test data, we subtracted the mean and divided by standard deviation of the training data, to avoid adding any biases the model has not seen during training.

HYPERPARAMETER TUNING

In a neural network (NN), we are training the network by feeding the data (inputs), an activation function, and a distribution of weights forward through the layers of the network to obtain the target values. These, in turn, will be compared to the real labels (observations) and the error will be propagated back to update the weight distributions with the use of the gradient of the cost function. This cycle is repeated until it is decided to stop by reaching a minimum error or a maximum number of iterations. For the first part of this project, the Keras library [1] was used to build and train neural networks.

However, there are parameters other than the weights that strongly influence the performance of an NN that are not related to the data but to the neural network's own architecture. These hyperparameters can be among others, the number of hidden neurons or layers, the weight decay parameter, the density of connections to neurons a target value has (i.e. dropout ratio), and the number of training epochs. The performance of an NN can be thus improved by tuning these hyperparameters. Hyperparameter tuning is an optimization problem, and there are several automatic algorithms available to achieve a solution.

Grid Search Cross Validation

The grid search method exhaustively scans all possible configurations of the selected hyperparameters within the boundaries imposed by the user. During this process, the algorithm evaluates the generalization performance at each configuration by running the training algorithm, using a cross-validation on the training set and a metric like MSE, and returns the configuration of hyperparameters that scores the best. Grid search is very attractive because it can be automated but the computational cost (duration) is high, especially if the training algorithm is slow.

Randomized Search Cross Validation

Randomized search picks points according to some probability distribution to evaluate the model performance using cross validation. Again, the range of parameters, or the parameter grid, is defined by the user. This is more efficient than grid search in terms of computation time. An additional advantage with randomized search is that the efficiency of the algorithm is not reduced with the addition of parameters that do not influence the model performance. Though it performs better than grid search, randomized search still doesn't utilize any information from the points seen so far to decide where to sample next. In our project we used the RandomizedSearchCV [8] method though the Keras Scikit-learn API.

Bayesian Optimization

In this section, the use of bayesian optimization for automatic tuning of hyperparameters of the neural network [10] is explored. The basic idea is to minimize some objective $f(x)$ over some bounded set X. Here, $f(x)$ is essentially the objective defined as function of the hyperparameters. The first step to minimizing this function is to identify what it is, or at least where to sample the function in order to obtain the optimum. Bayesian optimization

performs this by using all the information seen from previous evaluations of $f(x)$ and not just relying on local derivative information of the error surface. Essentially, we now have a method that intelligently selects points at which to evaluate the settings of hyperparameters based on available information.There are two important requirements to be satisfied before implementing bayesian optimization. One is to define a prior over the function space, and the second is to define a rule based on which the algorithm will sample next.

Gaussian Processes

The prior imposed over the function space is chosen to be a Gaussian process, ie., a draw from a Gaussian Process is a function: $f \sim GP(m, K)$. The Gaussian Process is the generalization of the Gaussian distribution and can be considered as an infinite dimensional gaussian. As we know, the gaussian distribution conveniently allows for computation of closed form expressions for conditionals and marginals. Hence, it is easy to find an analytical expression for the posterior over the function space once the data is observed. The gaussian process is completely described by its mean function $m : X \rightarrow \mathbb{R}$ and its kernel or covariance function $K : X\text{x}X \rightarrow \mathbb{R}$. Here, X represents the space of hyperparameters and $f(x)$ is the function to be minimized.

Acquisition Function

This function defines how the algorithm chooses the next point based on the function evaluations seen thus far. The acquisition function is typically expected to balance the exploration-exploitation trade-off while deciding where to sample the next point. The following are some common acquisition functions used:

- Probability of Improvement
- Expected Improvement
- Upper Confidence Bound
- Thompson Sampling

The acquisition function used here in this project is Expected Improvement. This function selects the next point that provides the maximum expected improvement over current best

observed function evaluation. The following is the expression for expected improvement:

$$\begin{aligned} a_{EI}(x) &= \mathrm{E}[u(x)|x, D] \\ &= \int_{-\infty}^{f'} (f - f')\, \mathcal{N}(f; \mu(x), K(x,x))\, df \\ &= (f' - \mu(x))\, \Phi(f'; \mu(x), K(x,x)) + K(x,x) \mathcal{N}(f'; \mu(x), K(x,x)) \end{aligned} \tag{1}$$

where f' is the optimal value of the objective function observed so far. The utility function that encodes improvement is given by

$$u(x) = \max(0, f' - f(x))$$

Equation (1) shows how the exploration-exploitation trade off is encoded in expected improvement. If the mean $\mu(x)$ is reduced, it increases the value of the first term and increases exploitation. If the covariance function $K(x,x)$ is increased, it increases the value of the second term and hence increases exploration.

Spearmint

Spearmint is a python library to perform Bayesian optimization according to the algorithms outlined in the paper Practical Bayesian Optimization of Machine Learning Algorithms [10]. This code is designed to automatically run experiments (thus the code name 'spearmint') in a manner that iteratively adjusts a number of parameters so as to minimize some objective in as few runs as possible.

One of the challenges faced in this project was to interface spearmint with the Keras neural network library in python. To get spearmint to work, the prospective parameters are to be specified in a *config.json* file. In addition, we need to specify a function that builds the architecture of the neural network in question that will take in the values provided by the spearmint code. The performance was decided using as objective the validation loss on a hold-out set which is 30% of the entire dataset.

The covariance function of the GP has its own set of hyperparameters which need to be estimated as well. These hyperparamters are estimated using a fully bayesian approach by integrating out the uncertainty over the GP parameters to compute a monte carlo estimate of the integrated acquisition function.

BAYESIAN NEURAL NETWORK

The second objective of this project attempts to find an alternative to the error back-propagation algorithm to train an NN using Bayesian methods. The idea is to sample from the posterior distribution of the weights upon observing the training data. Since this distribution does not have a closed form and it is not tractable, approximate inference techniques like Markov Chain Monte Carlo (MCMC) are used. MCMC methods are inefficient with respect to time complexity (Cost/duration) and can only be used for very simple NN, with one hidden layer as the number of weights grows with the number of neurons. For this exercise we are only interested in exploring the feasibility and characteristics of the network when being implemented with a Bayesian alternate, in the hope of seeing an improvement in the performance.

The idea for Bayesian Neural Network has been proposed since the early 1990's by David Mackay and Radford Neal in their thesis. Mackay proposed an evidence-based optimisation algorithm for updating the weights of a neural network[4] which aimed to improve the generalisation without the use of the holdout dataset. In his thesis, Radford neal proposed a hybrid Monte Carlo method [7] to sample weights of the NN using Gibbs Sampling, whose idea we are attempting to implement by leveraging recent development in MCMC algorithms.

Approximate Bayesian Inference

As stated before, the main difficulty is to obtain a closed form expression for the posterior distribution over the weights. Hence, we resorted to available Markov Chain Monte Carlo (MCMC) methods. To extend these bayesian methods to deep networks used today like, convolutional neural networks and deep autoencoders, variational inference methods can be used. These have been discussed in detail by Alex Graves [3] and Ghahramani & Gal [2].

Markov Chain Monte Carlo

MCMC methods construct a markov chain whose equilibrium distribution is the required distribution π^* and sample from this distribution by simulating the chain. The first few samples of the MCMC algorithms are typically discarded to allow for *burn-in* cycles. This refers the some iterations where the equilibrium distribution hasn't yet been reached. Some famous MCMC algorithms are Metropolis-Hastings, Gibbs Sampling, Slice Sampling and Elliptical Slice Sampling. In the next few sections we describe some of these methods along with their relevance to this project.

Gibbs Sampling

When we first started out with the project the idea was to use Gibbs Sampling (GS) to infer weights of the neural network. The Gibbs Sampler repeatedly samples each parameter from its distribution conditioned on all the other parameters. This means each weight of the NN will have to sampled sequentially, keeping all other weight values fixed. Even in our simple network, the number of weights is of the order 16,000. This means there will be 16,000 calls to the Gibbs sampler just for one iteration of weight updates. In addition, we are required to provide some cycles for burn in. Hence GS seemed infeasible. The algorithm for the gibbs sampler is given below,

1. set t = 0
2. generate an initial state $x^{(0)} \sim \pi^{(0)}$
3. repeat until t = M
   set t = t+1
   for each dimension i = 1..D
   draw x_i from $p(x_i|x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_D)$

Slice Sampling

The basic idea behind slice sampling [6] is very similar to the Metropolis-Hastings (MH) method. A point to be sampled next is chosen based on the value of the current state and from the prior distribution over the variable. How similar it is to the current state is determined by the step size:

$$f' = \sqrt{1-\epsilon^2} f + \epsilon \nu \qquad \epsilon \in [-1, 1] \tag{2}$$

where, f represents the current state or current value of the sample, ν represents a draw from the prior distribution and ϵ is the step size parameter. The new sample is accepted or rejected based on the ratio of the likelihood of the current state and that of the proposed state. The difference between MH and Slice sampling is that the step size is automatically tuned.

Given a starting point x , a value is chosen uniformly at random from the vertical strip between 0 and $f(x)$, where $f(x)$ is the value of the density at x. This chosen point then defines a horizontal line. Based on which portions of this line lie below the curve $f(x)$, a set of feasible slices are defined. An interval around the current point is defined according to some width w

and grown successively by the same width until ends lie outside the slice. Once, the plausible interval is determined, a new point x_1 is now chosen uniformly in the given interval, provided it lies on the slice. Once a point chosen lies outside the slice, it is used to shrink the interval.

This algorithm still does not overcome the problem of sequentially sampling all the weights individually. However, the next algorithm uses the basic idea of slice sampling while overcoming this hurdle.

Elliptical Slice Sampling

Elliptical slice sampling was proposed by Iain murray, Ryan Adams & David Mackay in 2009 [5]. The main advantage of this algorithm is that a vector of parameters can be updated in one step in contrast to Gibbs and slice sampling.

Elliptical slice sampling is based on the following idea. An ellipse centered at the origin is defined by the current value of the state and a draw from the prior distribution. As in slice sampling, a set of plausible slices is determined based on the likelihood of the observed data. This is referred to as the likelihood threshold. Once the slices are determined, the slice sampler is used to propose and accept or reject the next point by adaptively changing the size of the interval to be sampled from. The sequence of steps is pictorially represented below in figure (1) and takes the following order,

(a) The algorithm receives f = x as input. It draws auxiliary variate ν = +, defining an ellipse centred at the origin (o). In addition, the likelihood threshold defines the green slices, where the next sample can be chosen. A point . is proposed.

(b) The first proposal defines a bracket of angles, however another proposal is chosen from the entire range.

(c) A bracket of points is rejected as both lie outside of the slices such that the retained bracket still contains the point f = x. Proposals are made with shrinking brackets until one lies on the slice.

(d) The proposal in this step falls on the slice and hence is returned as the new point f'.

(e) Shows the reverse configuration where the proposal accepted in the previous step is the current state f = x and f from the previous step is the final accepted proposal.

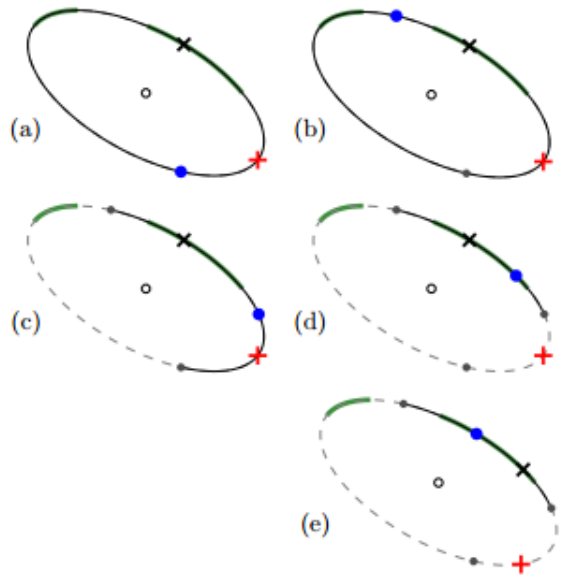

Figure 1: Elliptical Slice Sampling steps [5]

The advantage of slice sampling is that the ellipse we consider remains two-dimensional irrespective of the dimensionality of the points sampled.

Probabilistic Model

First we define an observational model or likelihood model as a draw from log normal distribution. This is because life expectancy is a positive number and we chose a distribution with positive support. This is the likelihood function that will be maximized in our elliptical slice sampler. In addition we assumed a gamma prior on the precision parameter which is one over sigma square, which helps us sample from posterior using the updated form of the gamma distribution based on conjugate priors. The equations (3)& (5) below fully describe the probabilistic model. In both equations f(x) represents the output of the neural network.

$$y|x \sim ln\mathcal{N}(f(x),\sigma^2) \tag{3}$$

$$\tau = 1/\sigma^2$$

$$\tau \sim Gamma(a,b) \tag{4}$$

$$\tau|D,w \sim Gamma(a+\frac{n}{2}, b+\sum_{i=1}^{N}\frac{(y-f(x))^2}{2}) \tag{5}$$

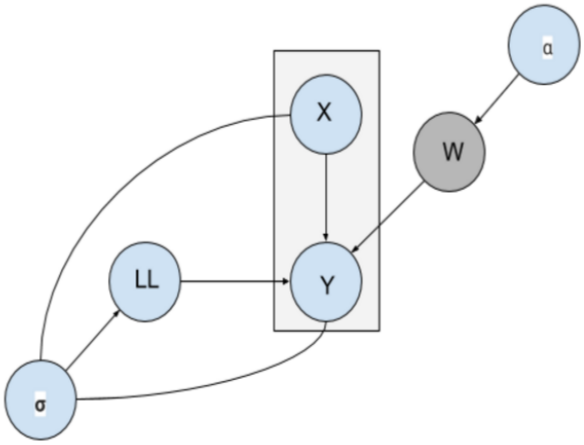

Figure 2: Neural network as Bayes Net

The figure (2) depicts the probabilistic model or Bayes net on which we attempt to perform inference. α defines the variance of the prior distribution of the weights. W is the latent variable which represents the weights. X and Y are the training inputs and target values. LL defines the likelihood of observing the data under the model. σ defines the standard deviation of the observation noise.

Pseudocode

Algorithm 1 MCMC Fit Algorithm

```
procedure MCMC_FIT(data, weights, tau, mcmc_iter, sigma)
    for all iter ∈ mcmc_iter do
        for all layer ∈ (1, no_layers − 1) do
            for all neuron ∈ layer do
                weights[neuron,:] ← eliptical_slice()
            end for
        end for
        tau ← Gamma_Distribution
    end for
    return weights, tau
end procedure
```

RESULTS & CONCLUSION

Hyperparameter Tuning

Parameter	Random Search	Spearmint
Weight Decay	0.0001	0.015206
Error Function	'mse'	'mse'
Number of Epochs	100	100
Activation Functions	['sigmoid', 'relu']	['sigmoid', 'relu']
Neurons	70	100
Weight Initialization	'glorot_normal'	'glorot_uniform'
Dropout	0.35	0.25

The table above shows the results obtained using the two algorithms which we ran for hyperparameter tuning. We were able to see that there is a difference in the number of neurons, weight decay, weight initialization method and dropout rate. The validation loss values of the two algorithms are 1.7967 and 1.669 respectively. Therefore we were able to see the Bayesian Optimization method performing well compared to the conventional Random Search Cross Validation method used for hyperparameter tuning.

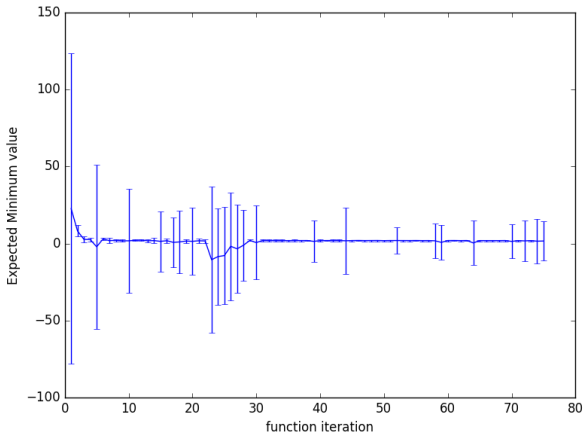

Figure 3: Evaluation of objective function over iterations - Spearmint

From figure (3) we can clearly see the reduction in validation loss which is our objective as well as reduction in the uncertainty. The sudden spikes in the error bars indicate the exploration performed by the acquisition function. The reduction in the size of the error bars indicate that the algorithm is exploring more earlier on and as time goes, exploration reduces.

Bayesian Inference

We are yet to see an improvement in training loss using the MCMC method. One of the problems we are facing is a very low likelihood threshold despite using a small angle range and the acceptance probability was very high. We were not able to obtain an improvement in training loss over the course of training.

Who did What?

- **Shirly Montero:** Baseline Neural Network setup on Keras - Required to run bayesian optimization + Presentation content + Report writing and content
- **Santhanakrishnan Ramani:** Setup of Spearmint and bayesian optimization for neural network + Setup of Randomized search + Presentation making for video + Part of Report
- **Shruthi Sukumar:** Setup of Bayesian Neural network code + Setup of probabilistic model + Script for video + Part of report

Bibliography

[1] François Chollet. Keras. https://github.com/fchollet/keras, 2015.

[2] Yarin Gal and Zoubin Ghahramani. On modern deep learning and variational inference.

[3] Alex Graves. Practical variational inference for neural networks. In *Advances in Neural Information Processing Systems*, pages 2348–2356, 2011.

[4] David JC MacKay. *Bayesian methods for adaptive models*. PhD thesis, California Institute of Technology, 1992.

[5] Iain Murray, Ryan Prescott Adams, and David JC MacKay. Elliptical slice sampling. In *AISTATS*, volume 13, pages 541–548, 2010.

[6] Radford M Neal. Slice sampling. *Annals of statistics*, pages 705–741, 2003.

[7] Radford M Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[9] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[10] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.