2018

# Small-variance asymptotics for Bayesian neural networks

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Thesis

# SMALL-VARIANCE ASYMPTOTICS FOR BAYESIAN NEURAL NETWORKS

by

**SIVARAMAKRISHNAN SANKARAPANDIAN**

B.E., Anna University, 2014

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2018

Approved by

First Reader

Brian Kulis, Ph.D.
Assistant Professor of Electrical and Computer Engineering
Assistant Professor of Systems Engineering
Assistant Professor of Computer Science

Second Reader

Prakash Ishwar, Ph.D.
Professor of Electrical and Computer Engineering
Professor of Systems Engineering

Third Reader

Kate Saenko, Ph.D.
Associate Professor of Computer Science

*My mentality is that of a samurai.*
*I would rather commit seppukku than fail;* Elon Musk

## Acknowledgments

First and foremost, I would like to extend my sincere gratitude to my thesis advisor Prof. Brian Kulis for his continuous support of my thesis and research. I attribute the level of my MS thesis to his encouragement and enthusiasm and without him this thesis would not have been written or completed. One simply could not wish for a better or friendlier advisor than him.

Besides my advisor, I would like to express my thanks to other members of my committee Prof. Prakash Ishwar and Prof. Kate Saenko for their generous insightful comments and valuable guidance.

I would like to thank my parents and my brother who were by my side all the time pushing me forward through hard times. My presence here at Boston University and therefore this thesis would not have been possible without their belief in my decisions till this point. Thank you for being always there for me.

Finally, I would like to thank College of Engineering at Boston University to give me this great opportunity to pursue my thesis.



Sivaramakrishnan Sankarapandian

ECE Department

Boston University

# SMALL-VARIANCE ASYMPTOTICS FOR BAYESIAN NEURAL NETWORKS

SIVARAMAKRISHNAN SANKARAPANDIAN

ABSTRACT

Bayesian neural networks (BNNs) are a rich and flexible class of models that have several advantages over standard feedforward networks, but are typically expensive to train on large-scale data. In this thesis, we explore the use of small-variance asymptotics—an approach to yielding fast algorithms from probabilistic models—on various Bayesian neural network models. We first demonstrate how small-variance asymptotics shows precise connections between standard neural networks and BNNs; for example, particular sampling algorithms for BNNs reduce to standard backpropagation in the small-variance limit. We then explore a more complex BNN where the number of hidden units is additionally treated as a random variable in the model. While standard sampling schemes would be too slow to be practical, our asymptotic approach yields a simple method for extending standard backpropagation to the case where the number of hidden units is not fixed. We show on several data sets that the resulting algorithm has benefits over backpropagation on networks with a fixed architecture.

# Contents

# List of Tables

# List of Figures

# List of Abbreviations

| | | |
|---|---|---|
| BNN | ..... | Bayesian Neural Network |
| BP | ..... | BackPropagation |
| DP | ..... | Dropout |
| EM | ..... | Expectation Maximization |
| GSR | ..... | Group Sparsity Regulaizer |
| HMC | ..... | Hamiltonian Monte-Carlo |
| LMC | ..... | Langevin Monte-Carlo |
| MAP | ..... | Maximum A Posteriori |
| PA | ..... | Proposed Algorithm |
| SVA | ..... | Small-Variance Asymptotics |
| SVM | ..... | Support Vector Machines |

# Chapter 1

# Introduction

## 1.1 Introduction

Bayesian neural networks (BNNs) are a particular kind of neural network in which weights are treated as random variables and therefore have a probability distribution over them. They are particularly useful when uncertainties have to be associated with predictions in the network, to avoid overfitting, and to incorporate prior beliefs about parameters of the model. Further, as the model itself is fully probabilistic, one can tap into the rich toolbox of probabilistic graphical models to extend the model in various ways and in a principled manner. For instance, it is straightforward to extend a BNN to the case where the model size (i.e., number of hidden units) is not fixed, or to other situations such as those arising with sequential data or hierarchical models. Unfortunately, such richness comes at a price: even simple BNN models are difficult to train, often requiring sophisticated sampling schemes that have difficulty scaling to large data sets. Indeed, in practical situations one rarely sees BNNs used as the model of choice.

Our focus in this thesis is to utilize a technique that has recently emerged in the study of large-scale probabilistic models, namely *small-variance asymptotics* (SVA), in order to both study connections between standard neural networks and BNNs as well as to design new algorithms that feature some of the benefits of BNNs but at increased scalability. One simple motivation that is used for SVA is to consider a

standard Gaussian mixture model for clustering. If one fixes the covariance of each cluster as $\sigma I$, and examines the behavior of the model as $\sigma \to 0$, the EM algorithm becomes the standard k-means algorithm [Kulis and Jordan, 2011] and the negative log likelihood of the mixture model becomes the k-means objective function. There has been considerable recent interest in using similar ideas on richer models, including Bayesian nonparametric models, sequential data models, topic models, supervised learning settings, and others. In general, applying SVA yields models that retain many of the properties of the original probabilistic model but are much more scalable.

This thesis is the first to apply SVA in the setting of neural networks. We begin by demonstrating that a particular standard BNN yields, in the SVA limit, the loss function for a standard feedforward network, indicating a simple link between probabilistic and non-probabilistic neural network models. We further demonstrate that an uncorrected Langevin Monte Carlo algorithm on this model yields standard backpropagation in the SVA limit, thus demonstrating connections at both the model and algorithm levels. We then build on this result to explore a richer BNN model where we allow the number of hidden units to be random. Such a model would be difficult to scale to practical settings, but we derive an SVA result indicating that the resulting model asymptotically is equivalent to a neural network loss with an additional penalty on the number of hidden units in each layer. We derive a simple extension of backpropagation to optimize the resulting loss function, yielding a more scalable approach that captures much of the richness of the original BNN.

We then empirically demonstrate that our resulting approach yields a flexible algorithm for training a neural network whose architecture is not fixed upfront. In particular, we show that our algorithms achieve comparable or even better performance than models where we fixed the architecture. Surprisingly, even when we train models whose architectures are the same as those that our algorithm learns (i.e., we retrain

using standard backpropagation on the architectures that our algorithm learns), we find that our error rates are typically lower, indicating that from an optimization perspective there may be benefits to training a network by growing neurons. We believe that our approach will yield further insights in the future to design richer neural network models inspired by probabilistic counterparts.

## 1.2 Related work

Small-variance asymptotics has been applied in a number of settings, including Bayesian nonparametric clustering models, feature learning models, hidden Markov models, Markov jump processes, SVM classification, dimensionality reduction, dynamic clustering models, topic models, and others ( [Roychowdhury et al., 2013], [Jiang et al., 2016] [Wang and Zhu, 2014], [Huggins et al., 2015]). Typically such models yield faster resulting algorithms while maintaining key properties of the original probabilistic models. However, to our knowledge, such analysis has not been applied to neural network models.

Selection of hyperparameters such as the number of hidden units or number of layers is typically chosen manually through grid search [Hsu et al., ] or through random search [Bergstra and Bengio, 2012]. These techniques are computationally expensive, especially if the number of hyperparameters is very large. Bayesian Optimization [Snoek et al., 2012] treats particular settings of hyperparameters as a sample from a Gaussian process, and yields an approach to automatically tuning the hyperparameters. While this method works well for continuous parameters such as the learning rate or regularization coefficients, it cannot be used for discrete parameters such as the number of layers or number of units in a layer.

In terms of automatically choosing the number of units/layers there are two broad categories of techniques available, namely constructive and destructive techniques. On

the destructive side, group sparsity regularizers [Alvarez and Salzmann, 2016] can be used to automatically determine the neurons required from an original network. [Zhou et al., 2016] applied sparsity constraints into the loss function to remove neurons during training. These techniques are generally inefficient in that the training is originally done over a large network and then neurons are pruned. [Hinton et al., 2015, Denil et al., 2013] are two approaches that consider the redundancy in the weights, thereby yielding compact networks. [LeCun et al., 1990, Hassibi and Stork, 1993] deal with the elimination of individual parameters such that there is least effect in the objective function that is being optimized, but both of these techniques require second derivative information.

On the constructive side, existing algorithms are mainly based on reinforcement learning and genetic algorithms. [Stanley and Miikkulainen, 2002, Opitz and Shavlik, 1997, Pujol and Poli, 1998] use genetic algorithms to grow a neural network, but no algorithms have shown proof of performing better than the hand picked values for hyperparameters on real world datasets. [Baker et al., 2016] uses reinforcement learning to build neural network architecture, but this approach requires an additional network that acts as the agent which selects the appropriate models. In contrast, our algorithm learns the number of hidden units as the network is being trained.

# Chapter 2

# Background

We briefly cover relevant material on Hamiltonian Monte Carlo, Langevin Monte Carlo, and small-variance asymptotics.

### 2.0.1 Hamiltonian Monte-Carlo

Hamiltonian Monte-Carlo(HMC) is a sampling method which samples from the target distribution by simulating a fictitious physical system and is dealt in detail in ( [Neal et al., 2011], [Neal, 1993]). The potential energy $U$, kinetic energy $K$, position $\boldsymbol{p}$ and momentum $\boldsymbol{q}$ constitute this system. If we wish to sample from a distribution $p(\boldsymbol{q})$, we define the potential energy to be the negative log probability density of the underlying parameters $\boldsymbol{q}$, kinetic energy to be the negative log probability density of a zero mean multivariate normal distribution and the Hamiltonian to be the sum of potential and kinetic energies:

$$\begin{aligned} U(\boldsymbol{q}) &= -\ln(p(\boldsymbol{q})) + \ln(Z_U) \\ K(\boldsymbol{p}) &= \frac{\boldsymbol{p}^T M^{-1} \boldsymbol{p}}{2} \\ H(\boldsymbol{p}, \boldsymbol{q}) &= U(\boldsymbol{q}) + K(\boldsymbol{p}). \end{aligned} \tag{2.1}$$

The evolution of position and momentum variables with time $t$ are determined by the following differential equations,

$$\begin{aligned} \frac{d\boldsymbol{q}}{dt} &= \tfrac{\partial H}{\partial \boldsymbol{p}} = [M^{-1}\boldsymbol{p}]_i \\ \frac{d\boldsymbol{p}}{dt} &= -\tfrac{\partial H}{\partial \boldsymbol{q}} = -\tfrac{\partial U}{\partial q_i}. \end{aligned}$$

Due to the conservation of Hamiltonian and volume preservation in phase space during the evolution, the joint distribution of position and momentum variables remain invariant. Practical implementation of Hamiltonian equations involve discretization of time using a small step $\epsilon$. The leap frog method is one of the most widely used methods for evaluating the evolution of the Hamiltonian system in discrete time steps $\epsilon$:

$$\begin{aligned} p_i(t+\epsilon/2) &= p_i(t) - (\epsilon/2)\frac{\partial U}{\partial q_i}(q(t)) \\ q_i(t+\epsilon) &= q_i(t) + \epsilon\frac{p_i(t+\epsilon/2)}{m_i} \\ p_i(t+\epsilon) &= p_i(t+\epsilon/2) - (\epsilon/2)\frac{\partial U}{\partial q_i}(q(t+\epsilon)). \end{aligned}$$

Time discretization with non-zero $\epsilon$ introduces numerical errors which do not maintain the Hamiltonian constant; to correct this error, a Metropolis step is added after every proposal is made:

$$\min\left[1, \exp(-(U(\boldsymbol{q}^*) - U(\boldsymbol{q})) - (K(\boldsymbol{p}^*) - K(\boldsymbol{p}))\right].$$

Thus, Hamiltonain Monte Carlo involves sampling random momentum variables from a multivariate Gaussian distribution, then calculating position variables with the gradient information from the log probability density of the target variable (while updating momentum variables) for $L$ steps and then accepting the proposal by calculating the

change in potential and kinetic energies through a Metropolis acceptance step. In an intuitive sense, retrieving samples from HMC is equivalent to determining the position of a frictionless puck that slides over an uneven surface where the height from the surface in its current position determines its potential energy and kinetic energy is propotional to its mass.

### 2.0.2 Langevin Monte-Carlo

If the number of leap frog steps in Hybrid Monte-Carlo is reduced to one, i.e., a new proposal is made after each leap frog step, then HMC is known as Langevin Monte Carlo [Neal, 1993]. Since state transitions here are proportional to the step size $\epsilon$, Langevin Monte Carlo exhibits random-walk behavior. The update equations for position and momentum variables in case of LMC are as follows:

$$\begin{aligned} q_i^* &= q_i - \frac{\epsilon^2}{2}\frac{\partial U}{\partial q_i}(q) + \epsilon p_i \\ p_i^* &= p_i - \frac{\epsilon}{2}\frac{\partial U}{\partial q_i}(q) - \frac{\epsilon}{2}\frac{\partial U}{\partial q_i}(\boldsymbol{q}^*) \end{aligned}$$

Similar to HMC, discretization error is corrected using a Metropolis acceptance step after every proposal. If we choose to eliminate the Metropolis acceptance step, then all the proposals are accepted which becomes known as *uncorrected Langevin Monte Carlo*. In this case, the new momentum values proposed every iteration are replaced immediately by the next iteration, therefore we can remove the momentum values altogether to arrive at a single update step which is as follows,

$$q_i^* = q_i - \frac{\epsilon^2}{2}\frac{\partial U}{\partial q_i}(\boldsymbol{q}) + \epsilon n_i \qquad \text{where} \quad n_i \in \mathcal{N}(0, \sigma_n = 1).$$

### 2.0.3 Small-Variance Asymptotics

In a Bayesian model, there are two complementary approaches to applying SVA. The first approach considers applying asymptotics on the underlying MAP inference problem; this was developed in [Broderick et al., 2013] with a particular emphasis on clustering and feature learning models. At a high level, given data $\mathcal{X}$ and parameters $\theta$, one examines the MAP problem $\text{argmax}_\theta p(\theta|\mathcal{X})$, and examines what happens to this problem when particular variances tend to zero. For instance, suppose we have a clustering model where the parameters are the assignments $\boldsymbol{z}$ of points to clusters and means $\boldsymbol{\mu}$ of all of the clusters. For the Gaussian case, we define the likelihood $p(\boldsymbol{x}|\boldsymbol{z}, \boldsymbol{\mu})$ to be a multivariate Gaussian whose mean and variance are given by the cluster to which $\boldsymbol{x}$ is assigned. The MAP inference problem arises by maximizing the posterior, $p(\boldsymbol{z}, \boldsymbol{\mu}|\boldsymbol{x}) \propto p(\boldsymbol{x}|\boldsymbol{z}, \boldsymbol{\mu})p(\boldsymbol{z})p(\boldsymbol{\mu})$, with respect to the parameters $\boldsymbol{z}$ and $\boldsymbol{\mu}$; it is equivalent to minimize the negative log of the joint probability $p(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\mu})$. If we assume that the variances of each of the clusters is equal to $\sigma I$, a fairly straightforward calculation reveals that as $\sigma \to 0$, the negative log of the joint probability simply becomes the k-means objective function.

The other approach to SVA involves examining the behavior of an algorithm when the variances of particular parameters go to zero. In the case of the clustering model, one can examine the EM algorithm (in the case of a non-Bayesian mixture model) or sampling steps of a Gibbs sampler (in the case of a Bayesian model) and examine what happens to the steps of the algorithm as $\sigma \to 0$. One can obtain the simpler k-means algorithm in both cases in this manner.

We note that SVA models typically yield much faster alternatives as compared to algorithms for the original probabilistic models. For example, EM for Gaussian mixtures scales quadratically in the dimensionality of the data while k-means scales linearly in the dimensionality. These advantages are even more pronounced for richer

probabilistic models; for instance, [Broderick et al., 2013] shows that SVA algorithms can be 1000x faster than sampling algorithms, even on fairly small data sets.

## 2.1 SVA for Bayesian Neural Networks

As a first step, we consider applying SVA to a standard Bayesian neural network. Consider a data set $\mathcal{X} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, where $\boldsymbol{x}_i$ is the $i$-th data point and $y_i$ the corresponding label. We could be considering either regression or classification as the underlying prediction task, but for now let us consider the regression task. Let $\boldsymbol{w}$ comprise the weights of a neural network, and define the likelihood of each data point as

$$p(y_i|\boldsymbol{w}, \boldsymbol{x}_i, \sigma^2) = \mathcal{N}(\mathrm{NN}(\boldsymbol{x}_i; \boldsymbol{w}), \sigma^2),$$

where $\mathrm{NN}(\boldsymbol{x}_i; \boldsymbol{w})$ is the output of a neural network with weights $\boldsymbol{w}$ and input $\boldsymbol{x}_i$. The $y_i$'s are assumed to be i.i.d. We further define a prior over the weights as $p(\boldsymbol{w})$ which could be chosen in various ways, e.g., uniform or Gaussian.

In general, assuming $\boldsymbol{w}$ being independent of $\boldsymbol{x_i}$ the goal is to calculate the posterior distribution of the weights $\boldsymbol{w}$ given the data $\mathcal{X}$. Because we are interested in performing SVA on this model, we consider the MAP inference task, namely

$$\mathrm{argmax}_{\boldsymbol{w}} p(\boldsymbol{w}|\mathcal{X}, \sigma^2) \propto \left[\prod_{i=1}^n p(y_i|\boldsymbol{w}, \boldsymbol{x}_i, \sigma^2)\right] p(\boldsymbol{w}).$$

We can ask what happens to the negative log of the MAP problem as $\sigma \to 0$ inside argmax. This is straightforward to compute, particularly due to the fact that the $y_i$ have Gaussian distributions. One can see that decreasing $\sigma$ causes the impact of the prior to be lessened, and in the limit there is no contribution to the prior, leading to

the standard regression loss with respect to $\boldsymbol{w}$

$$\sum_{i=1}^{n}(\mathrm{NN}(\boldsymbol{x}_i;\boldsymbol{w})-y_i)^2,$$

which is nothing but standard least-squares regression on the output of the neural network. One can easily extend this to the classification scenario by changing the likelihood appropriately and applying a similar asymptotic argument.

A more interesting question is whether existing sampling algorithms become standard neural network backpropagation in the small-variance limit, analogous to how the EM algorithm becomes the k-means algorithm in the small-variance limit. Again considering the regression case, we can explore the uncorrected Langevin Monte Carlo approach to sampling for a Bayesian neural network, and ask what happens to this sampling algorithm in the limit of small variance.

In case of a standard neural network, the weights are typically trained using backpropagation,

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \frac{\partial \mathcal{L}(\mathrm{NN}(\boldsymbol{x}_i;\boldsymbol{w}), y_i)}{\partial \boldsymbol{w}},$$

where $\mathcal{L}$ is the loss function and $\eta$ is the learning rate. The derivative of the loss function with respect to weights of the network is found using backpropogation via the chain rule. For the regression case, we consider the loss function to be the squared error, but the following connection is generalizable to any loss function.

To employ uncorrected Langevin Monte Carlo as our choice of sampling method for carrying out inference in a Bayesian neural network, we need to define a potential energy function for the parameters of the network. Given the likelihood and prior, the potential energy function becomes the logarithm of the posterior distribution of the parameters of the network:

$$U(\boldsymbol{w}) = \sum_{i=1}^{n} \frac{(\mathrm{NN}(\boldsymbol{x}_i;\boldsymbol{w})-y_i)^2}{\sigma} - \ln p(\boldsymbol{w}) + \text{const.}$$

If we plug in the potential energy function to the update rule of uncorrected LMC, we obtain the following update rule:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \frac{\epsilon^2}{2} \cdot \frac{\partial(\sum_{i=1}^{n}(\mathrm{NN}(\boldsymbol{x}_i;\boldsymbol{w}) - y_i)^2/\sigma - \ln p(\boldsymbol{w}))}{\partial \boldsymbol{w}} + \epsilon\mathcal{N}(0, \sigma_n I).$$

This now appears similar to the standard backpropagation update but with two additional terms, one coming from the prior on $\boldsymbol{w}$ and one coming from the Gaussian noise. If we let $\epsilon$ be a function of $\sigma$, namely setting it to $\epsilon^2\sigma$ and thus decreasing $\epsilon$ as we decrease $\sigma$, then in the limit of $\sigma \to 0$, the influence of the $(\mathrm{NN}(\boldsymbol{x}_i;\boldsymbol{w}) - y_i)^2$ term dominates. Indeed, in the limit the update is equivalent to the standard backpropagation update rule. Thus, in the small-variance asymptotic limit, the update rule of uncorrected LMC to this Bayesian neural network becomes the stochastic gradient descent update to a regular neural network.

## 2.2 A Dynamic Neural Network Model

Let us now consider a richer Bayesian neural network model where the number of hidden units is not fixed. From a generative perspective, the standard Bayesian neural network can be viewed as a generative process where the weights are drawn from the prior, and then the outputs are generated from the likelihood distribution given the weights. We can now consider the case where the weights within each layer $\ell$ are further conditioned on a random variable $k^{(\ell)}$ corresponding to the number of hidden units in layer $\ell$. We will place a Poisson distribution on $k^{(\ell)}$; assuming there are $L$

layers total, this makes the model (in the regression setting) therefore

$$\begin{aligned} p(k^{(\ell)}) &= \text{Pois}(\lambda), \quad \ell = 1, ..., L \\ p(\boldsymbol{w}^{(\ell)}|k^{(\ell)}) &= \mathcal{N}(0, I_{k^{(\ell)}}), \quad \ell = 1, ..., L \\ p(y_i|\boldsymbol{w}, \boldsymbol{x}_i\sigma^2) &= \mathcal{N}(\text{NN}(\boldsymbol{x}_i; \boldsymbol{w}), \sigma^2), \end{aligned}$$

where $\boldsymbol{w}^{(\ell)}$ corresponds to the weights in layer $\ell$. The definition of the variables $(k^{(\ell)}, \boldsymbol{w}^{(\ell)})$ imply that the joint probability $p(k^{(\ell)}, \boldsymbol{w}^{(\ell)}) = p(\boldsymbol{w}^{(\ell)}|k^{(\ell)})p(k^{(\ell)})$.

If we make no further assumptions about the rate parameter $\lambda$, then in the small-variance limit, the prior over the number of units will vanish, as the prior over the weights vanishes in the standard BNN case, resulting in an ill-defined model. However, if we assume that $\lambda = \exp(-\gamma/\sigma^2)$, then observe that

$$-\log p(k^{(\ell)}|\lambda) = \frac{\gamma}{\sigma^2} \cdot k + \exp(-\gamma/\sigma_p^2) + \log k^{(\ell)}!$$

As $\sigma \to 0$ (inside argmax) as in the previous section, the first term of this expression dominates; indeed, one can easily see that the MAP inference problem becomes simply

$$\sum_{i=1}^{n} (\text{NN}(\boldsymbol{x}_i; \boldsymbol{w}) - y_i)^2 + \gamma \cdot \sum_{\ell=1}^{L} k^{(\ell)}, \tag{2.2}$$

that is, the standard regression error with an additional term that regularizes the number of hidden units per layer.

### 2.2.1 Algorithm for a dynamic neural network

Now that we have seen that SVA on a BNN with a random number of weights yields an extension of the standard loss function with an additional penalty on the number of weights, we can consider algorithms to optimize (2.2). Note here that standard backpropagation is not appropriate for optimizing this loss function, as it is not clear

**Algorithm 1** Dynamic growing of neural networks

**Input:** $\mathcal{X}$, $\mathcal{Y}$,$H_l$,$[U_{H_1}, ...U_{H_l}]$, $\gamma$,prev_E $= \infty$
**for** $e = 1$ **to** E **do**
  Sample $\boldsymbol{w}$ from $P(\boldsymbol{w}|\mathcal{X}, \mathcal{Y})$
  current_E $= ln(p(\mathcal{Y}|\mathcal{X}, \boldsymbol{w}))$
  **if** prev_E $-$ current_E $> \gamma$ **then**
    $H^* = U[0, H_l]$
    $U_{H^*} = U_{H^*} + dM$
  **end if**
  prev_E = current_E
**end for**

how one would dynamically add or remove nodes from the network.

One option is to develop a sampling algorithm for the underlying BNN and then pass this algorithm through the small-variance limit. Another option is to directly design an algorithm for minimization of the underlying objective function. We take the latter approach here, following some other SVA-based algorithms (e.g., BP-means [Broderick et al., 2013]).

In particular, our algorithm considers starting with a small network of a few nodes per layer. During training, we consider the change in the loss function that occurs if we add some number of nodes $dM$ to the network and train on these additional nodes. If the change in the loss function is greater than $\gamma$, the tradeoff parameter from (2.2), then we keep these new units; otherwise, we maintain the network at its current size. When adding new nodes to the network, one should train long enough that a reliable estimate of the change in the loss function is obtained; we train for a full epoch before deciding whether to accept or reject the new nodes, though it is possible that shorter training times (e.g., a minibatch) may be sufficient for determining the change in the loss. When training multiple layers, we choose uniformly at random a single layer at a time when adding new nodes.

### 2.2.2 Adding neurons is equivalent to adding noise

We finish this section by briefly observing that adding new neurons and applying backpropagation is nearly equivalent to adding Gaussian noise to the gradient, as in uncorrected LMC. Let us consider a simple three layer network (one input, one hidden and one output) with the number of units in each layer being $U_i, U_h, U_o$, respectively. The weights between the input and hidden layer is denoted as $\boldsymbol{w_1}$, between hidden layer and output layer as $\boldsymbol{w_2}$, and the activation function is denoted as $g(\cdot)$. The output of the network for a single input $\boldsymbol{x_i}$ is

$$y_i^* = [g(\boldsymbol{x_i}^T \cdot \boldsymbol{w_1})]^T \cdot \boldsymbol{w_2}.$$

If we add a unit/neuron to the hidden layer, increasing $U_h$ by one, the output of the network changes to

$$y_{i_{mod}}^* = [g(\boldsymbol{x_i}^T \cdot \boldsymbol{w_1})]^T \cdot \boldsymbol{w_2} + [g(\boldsymbol{x_i}^T \cdot \boldsymbol{a_1})]^T \cdot \boldsymbol{a_2},$$

where $\boldsymbol{a_1}$ and $\boldsymbol{a_2}$ denote the added elements to the matrices $\boldsymbol{w_1}$ and $\boldsymbol{w_2}$, respectively, due to the additional neuron in the hidden layer. For simplicity, let us consider the $\mathcal{L}2$ loss. The derivative of the loss $\mathcal{L}$ with respect to weight $\boldsymbol{w_2}$ of the network before the addition of neurons is as follows:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w_1}} = 2([g(\boldsymbol{x_i}^T \cdot \boldsymbol{w_1})]^T \cdot \boldsymbol{w_2} - y_i).g(\boldsymbol{x_i}^T \cdot \boldsymbol{w_1}).$$

After adding the nodes, the update becomes

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \boldsymbol{w_1}} = 2([g(\boldsymbol{x_i}^T \cdot \boldsymbol{w_1})]^T \cdot \boldsymbol{w_2} - y_i).g(\boldsymbol{x_i}^T \cdot \boldsymbol{w_1}) \\ + \underbrace{g(\boldsymbol{x_i}^T \cdot \boldsymbol{a_1})^T \cdot \boldsymbol{a_2} \cdot (\boldsymbol{x_i}^T \cdot \boldsymbol{w_1})}_{\text{noise}}.\end{aligned}$$

Since weights are initialized randomly, this additional term can be viewed as a noise term, similar to noise in uncorrected LMC, with the exception that we apply the non-linear activation function on top of the noise.

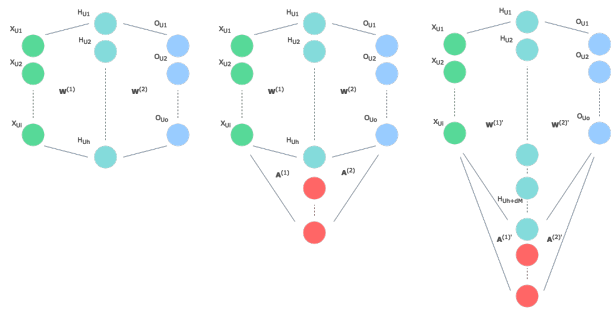

**Figure 2·1:** The red circles indicate the neurons being added to the network and their corresponding addition to matrices $\mathbf{W^{(1)}}$ and $\mathbf{W^{(2)}}$ are indicated as $\mathbf{A^{(1)}}$ and $\mathbf{A^{(2)}}$ repectively

## 2.3 Experiments

We evaluate our proposed algorithm on both single layer and multilayer feed-forward neural networks. To show the effectiveness our of proposed algorithm, we take datasets from [Hernández-Lobato and Adams, 2015] and apply our algorithm to it, along with suitable baselines. We train single layer neural networks for all the datasets in three different ways: 1) using the same architecture as in [Hernández-Lobato and Adams, 2015], 2) using our dynamic algorithm to grow the number of hidden units starting with five hidden units, 3) applying standard backpropagation to the architectures learned by our algorithm. We also train multi-layer networks on MNIST and CIFAR-10 with a fixed number of layers but where the hidden units are grown using our proposed algorithm.

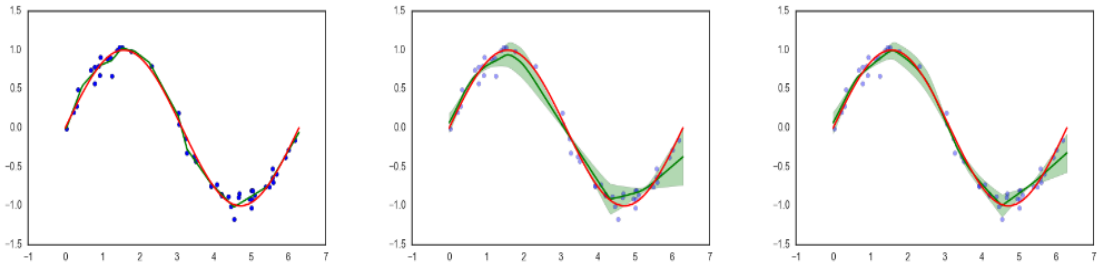

**Figure 2·2:** (From left to right) Regular gradient descent, uncorrected LMC with Gaussian noise and uncorrected LMC with a non-linear function applied to Gaussian noise trained to approximate the sine function. The blue dots indicate the data points and the read line indicates the true function. The dark green line indicates the prediction of the algorithms and the shaded portion is $\pm 1$ standard deviation from the mean of the predictive distribution.

### 2.3.1 Synthetic data

To illustrate empirically that injecting Gaussian noise after passing through a non-linear function still samples from the posterior distribution, we perform two experiments: 1) we implement Bayesian logistic regression using data generated from a bivariate Gaussian distribution and sample weights from the posterior distribution using uncorrected LMC with Gaussian noise and Gaussian noise with a non-linear function applied to it, as described in Section 2.2.2; 2) we attempt to approximate the sine function using standard backpropogation, uncorrected LMC with Gaussian noise, and Gaussian noise with a non-linear function applied.

Similar to [Roychowdhury et al., 2016], we generate 5000 samples from a bivariate Gaussian with means [1,-1], [-1,1] and unit covariance. We apply a linear transformation using the weights [-1,1] before grouping them into two classes. Bayesian logistic regression is performed using uncorrected LMC with Gaussian noise and Gaussian noise after application of the non-linear activation. We use a learning rate of $1e^{-3}$ and

$10^5$ iterations and figure 2·3 shows the traces of both sampling methods. The traces from both the methods looks similar finally converging to the true values of weights [-1,1]. We inject Gaussian noise of the form $\mathcal{N}(0, I)$ to uncorrected LMC steps before applying the non-linear function, since weights of the neural networks are initialized with weights from normal distribution with maximum standard deviation of one.

To further see the effect of the application of a non-linear function to Gaussian noise in uncorrected LMC, we create a synthetic dataset using 50 random data points in the interval [0,2$\pi$] and their corresponding labels using the sine function. A neural network with two hidden layers, each having 20 hidden units, is trained using standard backpropagation, uncorrected LMC, and uncorrected LMC with the incorporation of the non-linear function. Figure 2·2 shows the prediction results for all three methods. For both variants of uncorrected LMC, we include the corresponding predictive distribution, where dark green curve corresponds to the mean and the green shaded region corresponds to $\pm$ one standard deviation. The predictive distribution looks similar for both variants of uncorrected LMC with similar mean and similar $\pm$ one standard deviation. We calculate the standard deviation from the samples after 200 epochs to allow for the "burn in" period. We use a learning rate of $1e^{-3}$ as our learning rate, Adam optimizer($\beta_1 = 0.9, \beta_2 = 0.999$) and $10^4$ epochs.

### 2.3.2 Regression

In [Hernández-Lobato and Adams, 2015], the authors used an architecture with 50 hidden units for all the datasets except for Year Prediction MSD and Protein structure, for which they have used 100 hidden units. We use these architectures as baseline fixed architectures for comparison. We split the dataset into three parts: 60% training, 20% validation and 20% test. Hyperparameter tuning is done on the validation set and, during testing, we combine the validation set with training set. We do not perform

**Table 2.1:** Results of our proposed algorithm on regression datasets.

| Data set | N | D | BP | Hidden Units | Proposed Algorithm | Hidden Units Converged | BP with HU from PA |
|---|---|---|---|---|---|---|---|
| Concrete | 1030 | 8 | 5.977±0.2207 | 50 | 7.660±0.2613 | 25 | 12.346±1.8695 |
| Kin8nm | 8192 | 8 | 0.091±0.0015 | 50 | 0.0870±0.0017 | 43 | 0.091±0.0016 |
| Naval Propulsion | 11934 | 16 | 0.001±0.0001 | 50 | 0.004±0.0000 | 36 | 0.004±0.0040 |
| Powerplant | 9568 | 4 | 4.182±0.0402 | 50 | 4.114±0.0051 | 13 | 4.140±0.0061 |
| Protein | 45730 | 9 | 4.539±0.0288 | 100 | 4.511±0.0009 | 44 | 4.618±0.0559 |
| Wine | 1599 | 11 | 0.645±0.0098 | 50 | 0.609±0.0170 | 14 | 0.671±0.4421 |
| MSD | 515345 | 90 | 8.932±NA | 100 | 8.872±NA | 92 | 8.958±NA |

any preprocessing and run our proposed algorithm with five initial hidden units.

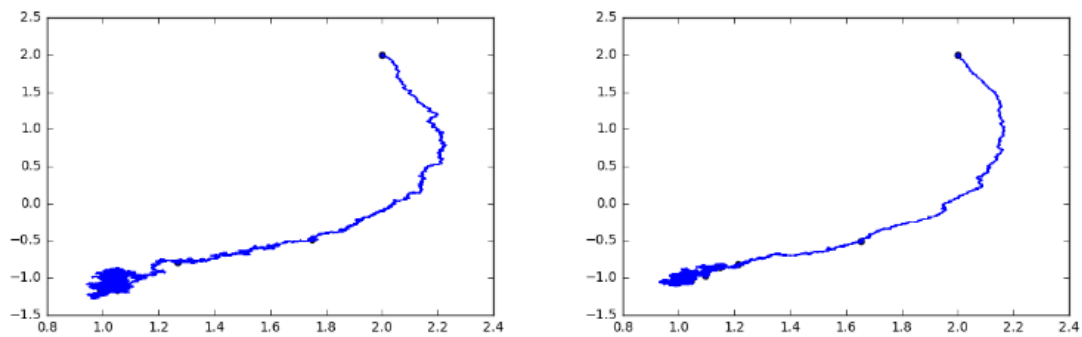


**Figure 2·3:** Trace plots of uncorrected LMC with Gaussian noise and Gaussian noise with the non-linear function applied. Both variants have the same initialization of weights [2,2] which converge to the true value of [-1,1].

We perform train-validation-test split five times after randomly shuffling the data and we include the standard deviation for our results. A single hidden layer is used for all the datasets and we start with five initial hidden units for our proposed algorithm. The best RMSE are underlined for each dataset. BP refers to regular back propogation using stochastic gradient descent. We run our algorithm with rate of increase in hidden units to be one (i.e., $d\mathcal{M} = 1$) and we keep this value constant throughout the training period.

### 2.3.3 Classification(Fully Connected Neural Networks)

**MNIST**

To demonstrate that our algorithm works for multi-layer networks, we consider the task of classifying hand-written digits into 10 classes using the MNIST dataset. We use a three-layer network for the task and we do not use any preprocessing for the classification task. Since starting with a small network with five hidden units in each layer and growing the network with a single unit in each layer is computationally expensive for training and could lead to poor local optima, we modify our algorithm to increase the hidden units by a large proportion during the start of training and gradually reduce this value as the network is trained. We introduce a new discrete parameter $\tau$ to specify the rate of decrease of $dM$. We use 10000 random images from the training set as validation set to tune the hyperparameters and during testing, we merge validation dataset to the training dataset. We use the Adam optimizer with $\beta_1 = 0.9, \beta_2 = 0.999$ to train our network.

Table 2.2 shows accuracies of standard backpropogation, our proposed algorithm and our proposed algorithm with dropout. Our proposed algorithm converges to the given number of hidden units in the first, second and third hidden layer as shown in the third column of the table. Even though the accuracy of our proposed algorithm does not surpass the accuracy by a regular backpropagation, the number of parameters is much smaller than those used by standard backpropagation. We use dropout of 0.5 in all hidden layers.We also run regular backpropogation(incorporating dropout in hidden layers) with number of hidden units our algorithm converged to, and the accuracy is not as good as the proposed algorithm.

To compare the convergence rate of backpropogation and proposed algorithm, we mark the number of epochs when both reach 98.60 %. To be fair, we initialize the weights of the network with the same seed and we maintain the same learning

rate.In [Alvarez and Salzmann, 2016], the authors used group sparsity regularizer over all the weights corresponding to each neuron thereby arriving at a sparse solution where weights corresponding to certain neurons are very small. Those neurons whose weights are negligible can be eliminated to arrive at a compact network. We also compare our proposed algorithm with the neural network with group sparsity regularizer(GSR).While the number of parameters are less in case of group sparsity regularizer, the best accuracy that can be attained is still lower than the proposed algorithm.

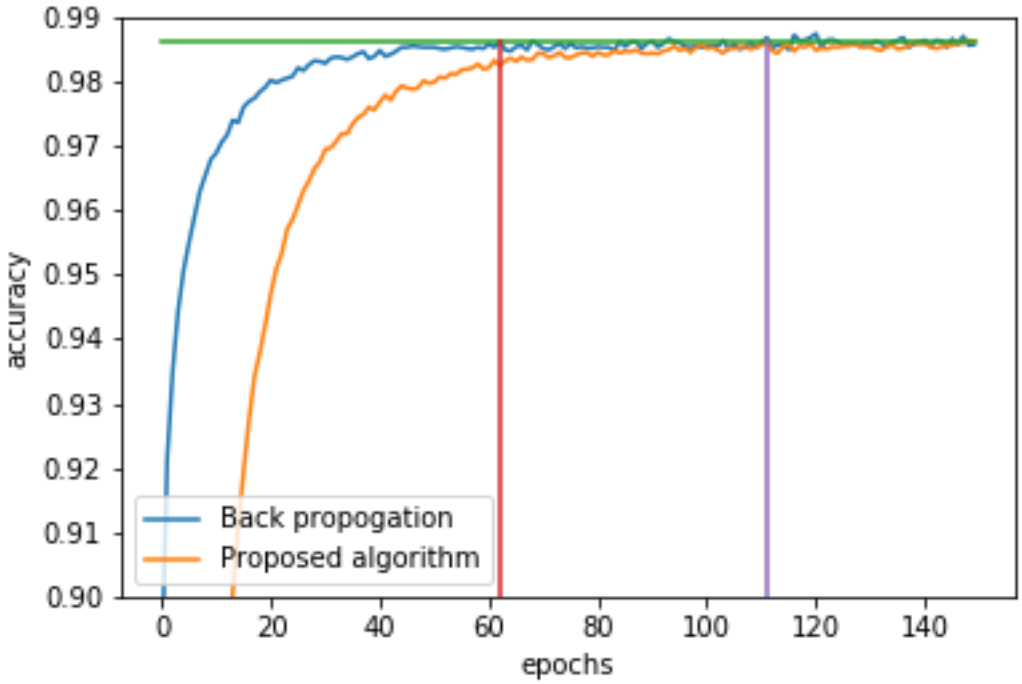


**Figure 2·4:** Convergence rate of backpropogation and proposed algorithm, red line indicates the number of epochs backpropogation requires to reach 98.6%(horizontal green line) accuracy and purple line indicates the same for proposed algorithm

**Algorithm 2** Dynamic growing of neural networks

**Input:** $\mathcal{X}$, $\mathcal{Y}$,$H_l$,$[U_{H_1}, ...U_{H_l}]$, $\gamma$,$\tau$,prev_E $= \infty$
**for** $e = 1$ **to** E **do**
  Sample $\boldsymbol{w}$ from $P(\boldsymbol{w}|\mathcal{X}, \mathcal{Y})$
  current_E $= ln(p(\mathcal{Y}|\mathcal{X}, \boldsymbol{w}))$
  **if** prev_E $-$ current_E $> \gamma$ **then**
    $H^* = U[0, H_l]$
    $U_{H^*} = U_{H^*} + dM$
  **end if**
  prev_E = current_E
  $dM = dM - \tau$
**end for**

**Table 2.2:** Results of our proposed algorithm on MNIST dataset.

| Method | Accuracy | $U_{h_0}, U_{h_1}, U_{h_2}$ | No.of Parameters |
|---|---|---|---|
| BP + DP | 98.75 | 1024-1024-1024 | 2,910,208 |
| PA + DP | 98.72 | 661-917-937 | 1,992,960 |
| BP + DP | 98.42 | 661-917-937 | 1,992,960 |
| GSR | 98.39 | 871-503-972 | 1,614,923 |

**CIFAR-10**

We consider another permutation invariant task of classifying images in the CIFAR-10 dataset into 10 classes. We use a three-layer network similar to MNIST and we do a preprocessing step of global contrast normalization and do not perform any pre-training. The strategy followed for hyperparameter optimization is similar to MNIST, where we take 5000 images at random from the training set for validation. During testing, we merge the validation set to our training set.

Table 2.3 shows the results of back propagation and proposed algorithm with dropout of 0.5 added to all hidden layers. We get similar accuracy with 60% fewer parameters than the standard backpropagation.Similar to MNIST, we run regular backpropogation(incorporating dropout in hidden layers) with number of hidden units our algorithm converged to, and the accuracy is not as good as the proposed algorithm.

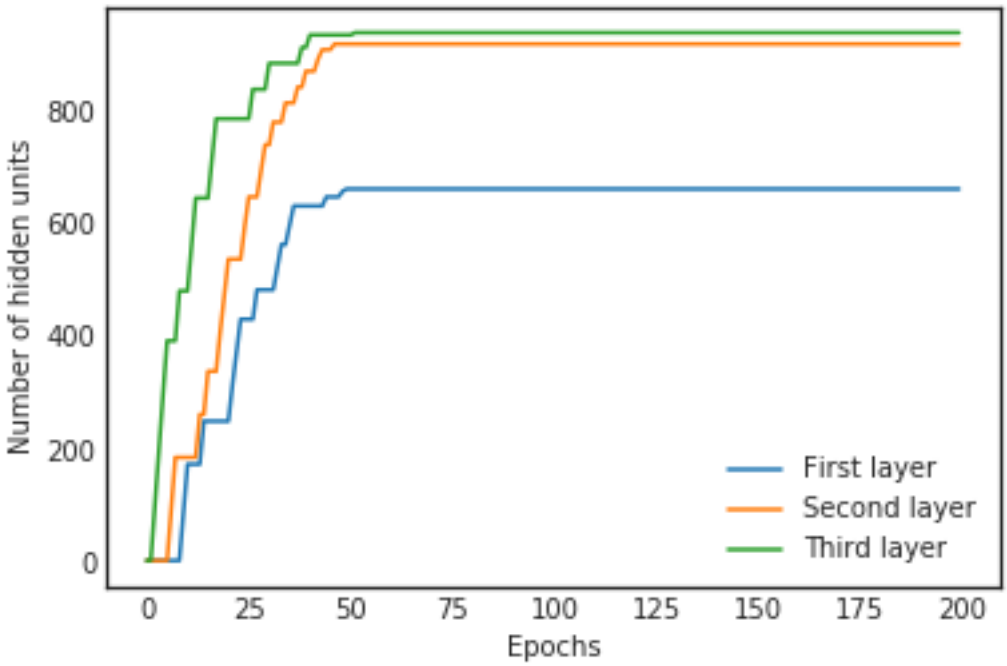


**Figure 2·5:** Evolution of hidden units during training of a 3-layer network on MNIST with dropout.

### 2.3.4 Classification(Convolutional Neural Networks)

To illustrate that the proposed algorithm works well even for Convolutional Neural Networks(CNNs), we create a three layer CNN with a single softmax layer where the number of filters in each layer are not fixed. We compare the results with regular fixed architecture CNN trained using backpropogation and we compare the results in 2.4. We use 3x3 filters and stride of 1 in all the layers and every convolutional layer is followed by maxpool layer with 2x2 filters and stride of 2.

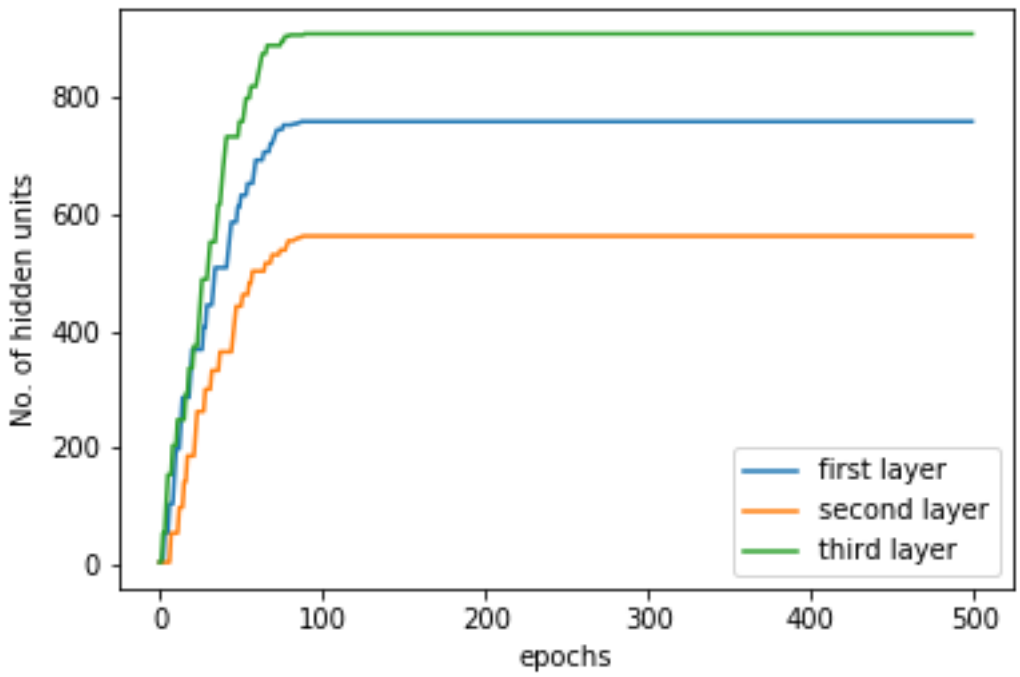


**Figure 2·6:** Evolution of hidden units during training of a 3-layer network on CIFAR-10 with dropout.

**Table 2.3:** Results of our proposed algorithm on CIFAR-10 dataset.

| Method | Accuracy | $U_{h_0}, U_{h_1}, U_{h_2}$ | No.of Parameters |
|---|---|---|---|
| BP + DP | 57.62 | 1024-1024-1024 | 5,253,130 |
| PA + DP | 57.47 | 759-563-909 | 3,279,822 |
| BP + DP | 55.55 | 759-563-909 | 3,279,822 |
| GSR | 53.07 | 1010-1015-716 | 4,861,770 |

**Table 2.4:** Results of our proposed algorithm on MNIST using CNN.

| Method | Accuracy | $U_{h_0}, U_{h_1}, U_{h_2}$ | No.of Parameters |
|---|---|---|---|
| BP + DP | 99.55 | 48-48-24 | 4,920 |
| PA + DP | 99.23 | 16-27-13 | 2,584 |
| BP + DP | 99.11 | 16-27-13 | 2,584 |

# Chapter 3

# Conclusions

## 3.1 Conclusion

In this thesis, we explored applying small-variance asymptotic analysis to Bayesian neural networks. While SVA has been applied to a number of unsupervised learning problems, such as clustering and feature learning, to our knowledge this is the first work that considers utilizing SVA in neural networks. As a first step, we demonstrated connections between standard neural networks and Bayesian neural networks, showing that the loss function for a standard neural network may be obtained in the small-variance limit of a Bayesian neural network. We then considered a richer BNN where the number of hidden units is treated as a random variable in the model. In the small-variance limit, such a model yields a simple loss function where the standard loss is augmented with a penalty on the number of hidden units.

We developed a straightforward algorithm for local convergence of this loss function, and compared this algorithm with standard backpropagation on several data sets. In particular, we found that the proposed algorithm is successful at finding good settings of the parameters, often using far fewer nodes per layer than standard fixed architectures. It is our hope that such analysis may lead further insight into designing richer neural network models.

# References


Alvarez, J. M. and Salzmann, M. (2016). Learning the number of neurons in deep networks. In *Advances in Neural Information Processing Systems*, pages 2270–2278.

Baker, B., Gupta, O., Naik, N., and Raskar, R. (2016). Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*.

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305.

Broderick, T., Kulis, B., and Jordan, M. (2013). Mad-bayes: Map-based asymptotic derivations from bayes. In *International Conference on Machine Learning*, pages 226–234.

Denil, M., Shakibi, B., Dinh, L., De Freitas, N., et al. (2013). Predicting parameters in deep learning. In *Advances in neural information processing systems*, pages 2148–2156.

Hassibi, B. and Stork, D. G. (1993). Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171.

Hernández-Lobato, J. M. and Adams, R. (2015). Probabilistic backpropagation for scalable learning of bayesian neural networks. In *International Conference on Machine Learning*, pages 1861–1869.

Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

Hsu, C.-W., Chang, C.-C., Lin, C.-J., et al. A practical guide to support vector classification. *Unpublished paper, Department of Computer Science, National Taiwan University. Retrieved from https://www.csie.ntu.edu.tw/ ∼ cjlin/papers/guide/guide.pdf*.

Huggins, J. H., Narasimhan, K., Saeedi, A., and Mansinghka, V. K. (2015). Jump-means: Small-variance asymptotics for markov jump processes. *arXiv preprint arXiv:1503.00332*.

Jiang, K., Sra, S., and Kulis, B. (2016). Combinatorial topic models using small-variance asymptotics. *arXiv preprint arXiv:1604.02027*.

Kulis, B. and Jordan, M. I. (2011). Revisiting k-means: New algorithms via bayesian nonparametrics. *arXiv preprint arXiv:1111.0352*.

LeCun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605.

Neal, R. M. (1993). Probabilistic inference using markov chain monte carlo methods. *Technical Report CRG-TR-93-1, Department of Computer Science, University of Toronto. Retrieved from http://bayes.wustl.edu/Manual/RadfordNeal.review.pdf*.

Neal, R. M. et al. (2011). Mcmc using hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2(11).

Opitz, D. W. and Shavlik, J. W. (1997). Connectionist theory refinement: Genetically searching the space of network topologies. *Journal of Artificial Intelligence Research (JAIR)*, 6:177–209.

Pujol, J. C. F. and Poli, R. (1998). Evolving the topology and the weights of neural networks using a dual representation. *Applied Intelligence*, 8(1):73–84.

Roychowdhury, A., Jiang, K., and Kulis, B. (2013). Small-variance asymptotics for hidden markov models. In *Advances in Neural Information Processing Systems*, pages 2103–2111.

Roychowdhury, A., Kulis, B., and Parthasarathy, S. (2016). Robust monte carlo sampling using riemannian nosé-poincaré hamiltonian dynamics. In *International Conference on Machine Learning*, pages 2673–2681.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127.

Wang, Y. and Zhu, J. (2014). Small-variance asymptotics for dirichlet process mixtures of svms. In *In Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence and the Twenty-Sixth Innovative Applications of Artificial Intelligence Conference, Vol 3. Palo Alto, CA: AAAI Press.*, pages 2135–2141.

Zhou, H., Alvarez, J. M., and Porikli, F. (2016). Less is more: Towards compact cnns. In *European Conference on Computer Vision*, pages 662–677. Springer.

# CURRICULUM VITAE

[redacted]