Probabilistic Deep Learning

Alex J. Chan

Department of Statistical Science

University College London

Supervisor

Dr Ricardo Silva

A written report for the STAT0035 Project

April, 2019

Word Count: 14994

Acknowledgements

I'd like to thank Dr. Ricardo Silva for supervising me throughout this project, pointing me in interesting directions and providing a great deal of insight and guidance. Thanks as well to Dr. Sam Livingstone for help in particular with my understanding of Markov chain Monte Carlo methods. Lastly thank you to both Dr. Matina Rassias and Professor Richard Chandler for their continued support during my time at UCL.

Abstract

Deep learning models have found widespread use in the machine learning community recently given their state-of-the-art performance on a range of interesting problems. They can however return a wrong answer with high confidence and little way for the practitioner to understand how the model has come to that conclusion. In this project we will explore attempts to fit deep learning models into a Bayesian framework, thereby making clear uncertainty in predictions. We will consider how useful a number of these methods are as well as ultimately improving the flexibility of one of the approximations (known as Monte Carlo Dropout) by incorporating the dropout rate as a parameter of a variational distribution to be optimised.

Contents

Chapter 1

Introduction

"The beginning is the most important part of the work."

Plato

1.1 Motivations - Understanding Uncertainty

Accurate uncertainty estimates are vitally important for any form of data analysis. Given a model that makes some prediction based on an input we should be very interested in how confident the model is that it has returned the correct answer. This is of most importance when the model is very uncertain - we may not want to act on such a prediction and instead explore the case more. This problem is magnified when decisions based on the output of the model have serious consequences. Deep learning models (neural networks) [Goodfellow *et al.*, 2016] have recently found great use in a number of high impact fields including medical diagnostics and autonomous vehicles [Litjens *et al.*, 2017] but if a model returns that a patient has cancer you want to be sure about it before starting chemotherapy. The structure of neural networks makes the task of understanding why an input has produced a particular output especially hard. In simple models like linear regression parameters correspond linearly to some specific covariate and it is easy to see how slightly changing that part of the input results in the the output changing. In neural networks there is no such intuition about individual parameters as they all interconnect in a manner too complicated for people to usually grasp.

Traditionally in statistics this problem is dealt with from the Bayesian point of view by considering any parameters in a model to be random variables. These parameters are given an explicit prior distribution and when new data is seen their posterior distribution given the data can be calculated. With information about the posterior distribution of the parameters of the model comes the predictive distribution of any output from the model, and knowing about the predictive distribution tells us how uncertain the model is. For example a predictive distribution with high variance means the model thinks there are quite a lot of plausible possible values for the prediction and any particular value will have low relative probability density. This has the added benefit of taking into account any uncertainty in the generative process of the data as well.

In deep neural networks this task becomes very complicated as the number of parameters can often run well into the thousands and often for very complicated models into the millions. To perform Bayesian inference this will end up requiring the calculation of integrals in the dimension of those parameters and will soon become completely intractable. There is then a pressing need for scalable and efficient methods to perform approximate Bayesian inference to get uncertainty information as close to the true underlying uncertainty as possible. In this report we will explore a number of the proposed solutions.

1.2 Objectives

This report will document my STAT0035 Project and as such we have three main objectives:

1. Become familiar with the current literature on Bayesian deep learning.
2. Implement and compare the effectiveness of modern methods for evaluating the posterior distribution in neural networks.
3. Take one of the current approximate methods and try to improve it by making it more flexible. In particular we will look at MC Dropout and how we can optimise the dropout rate.

This report then will address these objectives and aim to bring the reader up to date with recent work on Bayesian deep learning, assuming only undergraduate level statistics knowledge. Then it will detail our original work in both comparing the effectiveness of some of these methods and deriving a more flexible method.

1.3 Contributions and Overview of the Report

In order to achieve our objectives, after this introduction, in Chapter 2 we will discuss all the relevant background knowledge that will be needed for the rest of the report, covering mainly the topics of machine learning and modern computational methods for performing Bayesian inference in complex and high-dimensional problems. Having done the ground work we will move on in Chapter 3 to look at the modern literature on Bayesian deep learning. We will look at how these models where introduced as well as contemporary methods used to efficiently fit approximations to the posterior distributions since the size of neural networks makes exact inference analytically intractable. Then we will move to

Chapter 4 where we will discuss the more practical aspects of our work, implementing a number of the algorithms and discussing how they behave comparatively. We will also derive and implement a new method for incorporating and optimising the dropout rate in an approximation to Bayesian inference, thereby making the approximation more flexible and accurate. Having done this we will conclude in Chapter 5 by discussing how our results can be interpreted as well as summarising the work of the project.

All graphics in this report were produced by me. Additionally, all code used in either experiments or for visualisation purposes was written by me as well, only really relying on the automatic differentiation package Autograd. That is to say I coded the models and algorithms myself from the ground up without relying on packages such as Keras to implement neural network layers or PyMC3 to sample Markov chains. Please see Appendix A for further details.

Chapter 2

Background

"A computer program is said to learn from experience E with some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."

Tom M. Mitchell

Summary

In this chapter we will cover and discuss the essential background material that is required to understand the current literature on probabilistic deep learning and not covered on the BSc Statistics degree program. We will first go over the general machine learning paradigm before looking specifically at neural network models. Then we'll go on to methods to train these models, firstly via deterministic optimisation, and then by considering the parameters as random variables and updating them by Bayes rule. Finally we'll look at the modern computational

methods for evaluating the often analytically intractable posterior distributions obtained as well as briefly covering non-parametric models. For each of the topics there are many resources available that repeat the same thing but key results quoted in this chapter can be considered taken from the textbook by Barber [2012] unless otherwise stated.

2.1 Machine Learning

Machine learning is a sub-field of statistics and computer science that specifically deals with algorithms that improve in capability as more information is given to them. As with statistical inference the goal is to use the available data to deduce the properties of some underlying population or process. With recent great increases in computing power and available data, machine learning has proven itself an immensely powerful tool in a number of areas including medical diagnosis, autonomous vehicles and cyber security [Kononenko, 2001]. Machine learning covers a wide range of very different algorithms but can be broadly split into three groups: supervised, unsupervised, and reinforcement learning. We will only really concern ourselves with first of these but I shall very briefly discuss the other two as well.

2.1.1 Supervised Learning

In supervised learning we consider the general paradigm where we have a set of n observation tuples, $\{(x_i, y_i)\}_{i=1}^n$ where x and y are input and output vectors respectively, elements of some potentially both high and different dimensional spaces, X and Y. Typically supervised learning is divided into two further groups: regression, and classification. In regression the output is considered to be a real valued vector while in classification the output is a category. The task is to learn the mapping $g : X \to Y$ so that given new inputs we can accurately determine the output value. The space of all possible functions is impractically large so we tend to constrain them considerably (for example, in linear regression we consider only functions that are a linear combination of the input) and make our guess, $f(x, \theta)$ where θ is some free parameter(s) that we can change in order that our function is as close to the real g as possible.

In order to get close to g, we define a cost function $C(\theta)$ that based on our data tells us something about the "distance" from g. We can then change θ so that $C(\theta)$ is minimised. The choice of cost function can have a big impact on the capability of the model, and it is important that it is chosen wisely. For regression problems the most common choice is the least squares loss while in classification, functions such as the cross entropy loss have been more popular [Zhang, 2000]. Both of these though can be interpreted as some form of likelihood loss in a probabilistic context.

We will discuss in a later section exactly how we minimise the cost function as the same optimisation techniques can be used in more general scenarios outside of supervised learning.

2.1.2 Unsupervised Learning

Unsupervised learning considers the alternate scenario when we have a set of n observations, $\{x_i\}_{i=1}^{n}$ but this time of only input vectors. Instead of trying to learn some output mapping we try to learn some underlying pattern in the data. Typical tasks include dimensionality reduction, clustering, and generative models.

2.1.3 Reinforcement Learning

In reinforcement learning we have an agent (the machine learning program) and the environment, the idea being for the agent to learn about the environment by trial and error. At each stage the agent is shown the state of the environment and is presented with a choice of actions. The agent takes an action and then receives a reward signal that says how good the state of the environment is now the action has been taken, with the goal of the agent to maximise this long term reward. Reinforcement learning has been recently popularised by Google DeepMind, where their Alpha series of algorithms have had great success on a variety of tasks that people did not expect computers to be able to handle, from Go and Atari games to more recent protein folding predictions. [Silver *et al.*, 2017] [Evans *et al.*, 2018]

2.2 Deep Neural Networks

The focus of this project is the neural network (NN), a particular type of model that has been shown to perform extremely well on tasks for which it is often hard to describe rules such as speech and handwriting recognition [Bishop *et al.*, 1995]. We will focus on the most common type, the multilayer perceptron (MLP), for which the Universal Approximation Theorem [Hornik, 1991] says that a suitably large NN can approximate any given function. Since that is the whole goal of supervised learning it would seem they are a very powerful and useful model indeed.

There are however some significant drawbacks. While theoretically any function can be achieved it may be that the architecture of the NN is infeasibly big and the NN fails to learn or generalise correctly. Additionally NNs require large amounts of both data and training time in order to be accurate, which may mean a simpler algorithm could give you "good enough" results at a fraction of the computational cost.

2.2.1 The Perceptron and MLPs

The initial basic building block for all deep learning is a model called the perceptron [Rosenblatt, 1962]. Inspired by neurons in the brain, the perceptron takes in a weighted sum of inputs, adds a bias (in this context a constant value that relates to how easy it is for the neuron to fire) and then outputs some non-linear transformation of this total (we shall call this transformation the activation function) as shown in Figure 2.1. Originally this non-linearity was taken to be a step function outputting a one or a zero (since neurons in the brain either fire or don't, there is no gradation) but since this is discontinuous it causes problems when evaluating the gradient and so we now make "softer" choices such as the sigmoid function. This function, defined as $s(x) = (1 + \exp(-x))^{-1}$, is monotonic and takes any real value and output a number between zero and one, making it act like a smooth step function. It is also differentiable, which we will see later is very important. Technically the model is only a perceptron as presented by Rosenblatt when using the step function, though we shall call any model of this design with various activation functions a perceptron or neuron.

Modern deep neural networks are all essentially MLPs, which consist of a number of perceptrons arranged in layers. In a given layer each individual perceptron takes as input the output of all the perceptrons in the previous layer and its outputs will in turn be taken as input for the following layer. There will be an input layer and an output layer as well as at least one hidden layer in between, as in Figure 2.2. The "deep" part of deep learning refers to these hidden layers and modern NNs often have a very large number of hidden layers.

Now that we understand the idea we must consider how the model will be

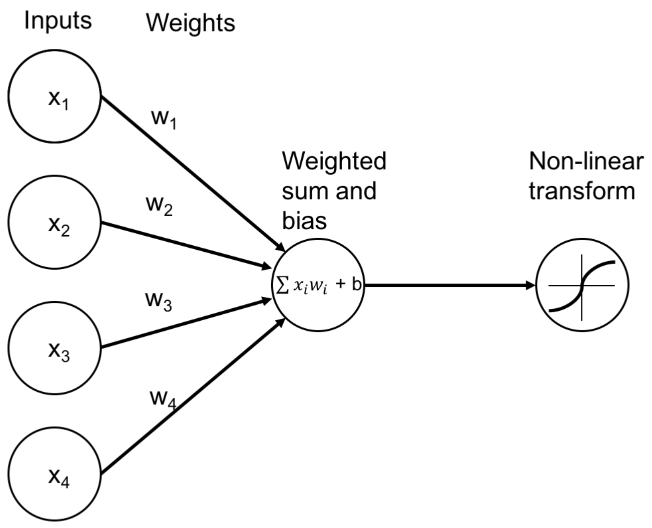


Figure 2.1: A visual representation of a perceptron.

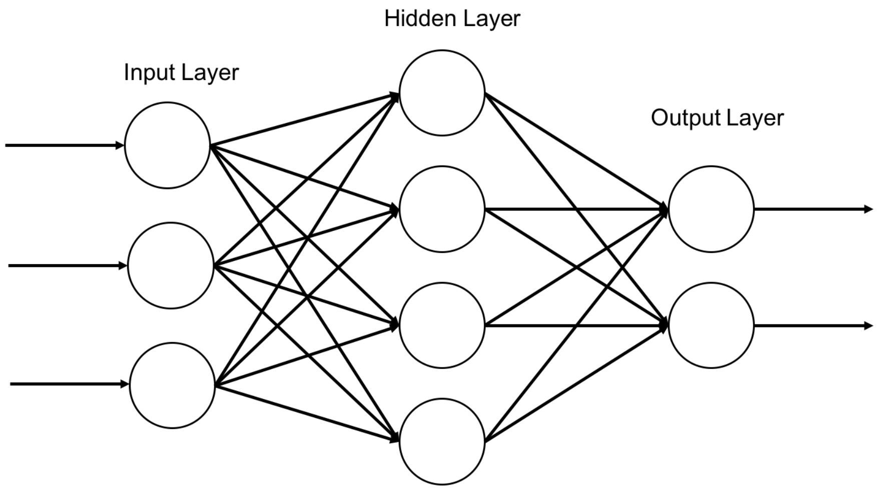


Figure 2.2: A very simple single hidden layer network, in this graph each node represents a single perceptron and each directed edge represents the flow of outputs to inputs.

numerically implemented. We will denote a^l_j as the activation (a.k.a. the output) and b^l_j as the bias of the jth neuron in layer l. Additionally, w^l_{jk} as the weight of the activation of the kth neuron in layer $(l-1)$ as input for the jth neuron of layer l. The subscripts for the weights may seem backwards but they simplify the notation later. Given some activation function $\sigma : \mathbb{R} \to \mathbb{R}$ we can clearly write the activations in one layer as a function of the activations in the previous layer:

$$a^l_j = \sigma\Big(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\Big). \tag{2.1}$$

In order to write this in matrix algebra, for every layer l we define a weight matrix, w^l, which is simply constructed by setting the element in the jth row and kth column of w^l as w^l_{jk}. Additionally we set a^l and b^l to be vectors containing the activations and biases of the lth layer. Now equation 2.1 becomes:

$$a^l = \sigma\big(w^l a^{l-1} + b^l\big), \tag{2.2}$$

where σ is applied elementwise. We now have a very clear way to implement a NN, for each layer multiply the input by a weight matrix, add a bias vector, apply an elementwise function, and repeat for the next layer. We thus have a very flexible class of functions parameterised by a set of weight matrices and bias vectors.

2.2.2 Backpropagation

Now we have defined the model we must consider the gradient of the cost function, ∇C, with respect to the parameters of the model. For a long time after NNs were first proposed this was a significant problem given the vast number of parameters in the model and it wasn't until the introduction of the backpropagation algorithm [Rumelhart *et al.*, 1986] and advances in CPU speeds that NNs became viable for interesting tasks.

Backpropagation is actually a specific example of reverse-mode automatic differentiation, a computational method to get the exact derivative of a function composed of any number of themselves differentiable functions and does not rely on any approximations through the construction of a computational graph and use of the chain rule. Backpropagation works by calculating an error at the end of the network and then *propagating* it backwards though the network to calculate the gradient of each weight with respect to this error. Importantly this only takes time in the order of a forward pass through the network to get the gradient of every weight and as such is much quicker than numerical methods that require that amount of time for every individual weight.

2.2.3 Regularisation

Neural networks have a lot of parameters and as such are very prone to *over-fitting*, that is to say they may approximate the function very well on the training data but fail to generalise well to new data as the NN is learning a very specific map on the training set that does not reflect any overall pattern true to the data and only occurs by chance. To combat this, *regularisation* is applied to the model where some form of constraint is placed on the parameters so that they cannot fit extremely specific functions and are therefore encouraged to find more general patterns. The most common forms in NNs (and indeed supervised learning in general) are L1 and L2 regularisation which add a penalty to the cost function proportional to the L1 or L2 norm of the parameters respectively. This directly relates to examples in traditional statistics when performing lasso or ridge regression respectively. It is not always clear though exactly what sort of regularisation is needed so techniques such as cross-validation (where by you split the data multiple times, for each split fitting a model on one part and testing the fit on the other part) can be used to see how much models over-fit the data and see what the optimal level of regularisation is.

2.3 Optimisation

We will consider the general scenario where we have some function $C : \mathbb{R}^n \to \mathbb{R}$ that takes some n dimensional input vector, θ, and outputs a real number. The task is to select θ such that the value of $C(\theta)$ is minimised, equivalent to maximising $-C(\theta)$.

Basic calculus tells us that we reach a stationary point when the gradient of the function with respect to the input is zero. Thus we may naively try to simply solve the equation: $\nabla C(\theta) = 0$. However we often run into the problem that this equation has no closed form solution, to get around this problem we must often resort to iterative methods.

2.3.1 Gradient Descent and SGD

The idea behind gradient descent is quite simple; given a starting position we take a small step in the direction of steepest descent and repeat until convergence [Bottou, 2010]. We do this by at each step evaluating the gradient of C at the current position and then updating the position by taking away a multiple of the gradient from the current position as in algorithm 1. This will usually take us to a local minimum but we cannot guarantee a global minimum.

There are still some limitations to gradient descent that need to be overcome. Data sets these days can be incredibly large and it can be very computationally expensive to evaluate ∇C over the entire data set. Stochastic Gradient Descent (SGD) is designed to overcome this problem as at each step instead of evaluating the gradient over the entire data set we select a random, much smaller, subset

Algorithm 1: Gradient Descent

```
Result: Optimised parameter $\theta$
Require : $\lambda$ (Step size);
Require : $\theta_0$ (Initial starting point);
$t \leftarrow 0$ (Initialise timestep);
while $\theta_t$ not converged do
    $t \leftarrow t + 1$;
    $\theta_t \leftarrow \theta_{t-1} - \lambda \nabla C(\theta_{t-1})$;
end
return $\theta_t$ (Resulting parameters)
```

of the data (known as a minibatch) and evaluate the gradient over that instead ($\nabla C_{minibatch}$). The key point here is that the cost functions are chosen so that the expectation of $\nabla C_{minibatch}$ is equal to ∇C so that in the long run the average direction of the steps should still take us to a minimum.

2.3.2 Adaptive Methods for Stochastic Optimisation

There are many cases that arise where the objective function to be optimised is stochastic, as in SGD where it changes depending on the exact minibatch chosen. In these cases taking a simple step in the direction of the gradient is not always the best case as the estimators can exhibit very high variance and the parameters may take a long time to converge if they do at all. To get over this a number of extensions to gradient descent have been proposed including with momentum, AdaGrad, and RMSProp [Ruder, 2016] that each take into account past behaviour of the estimate when updating so as to make the trajectory smoother and not be affected so much by extreme evaluations of the objective function. One of the most popular is the algorithm Adam [Kingma & Ba, 2014], which updates the parameter keeping track of running averages of both the gradient and the second moments of the gradient, increasing the speed of convergence and decreasing the likelihood of getting stuck in a local minimum or saddle point. Intuitively Adam is trying to get the benefits of second-order optimisation methods (those that make use of second derivatives) without incurring the computational costs by looking at how the estimates have behaved in the past and how they are currently changing.

Another important aspect is the step size, often it is kept constant but especially in the stochastic case it is important to adapt them. Robbins & Monro [1951] show that when using step sizes ϵ_t that decrease with t and satisfy equations 2.3 that it guarantees convergence to a local minimum.

$$\sum_{t=1}^{\infty} \epsilon_t = \infty \quad \sum_{t=1}^{\infty} \epsilon_t^2 < \infty \tag{2.3}$$

2.4 Bayesian Inference

Admittedly, Bayesian inference is covered in the statistics curriculum but it is crucial for the project and so seems worth going over again here. We consider it as an alternative way to determine the parameters of a model instead of the traditional way of optimisation. The cornerstone of Bayesian inference is of course Bayes theorem, which relates the conditional probabilities of two events, A and B:

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}. \tag{2.4}$$

We can extend this from the conditional probability of an event, A given B, $p(A|B)$ to consider the conditional *distribution* of some parameter, θ, given the data, $\mathcal{D}$. We also note that $p(\mathcal{D})$ is not affected by the value of θ, meaning we can just consider it as a normalising constant such that the resulting distribution integrates to one, leaving us with:

$$p(\theta|\mathcal{D}) \propto p(\mathcal{D}|\theta)p(\theta). \tag{2.5}$$

We can now see that our *posterior* belief about θ having seen $\mathcal{D}$, $p(\theta|\mathcal{D})$ is proportional to the *likelihood* of the data, $p(\mathcal{D}|\theta)$ multiplied by our *prior* belief about θ, $p(\theta)$. Crucially now once we decide on a prior distribution of θ all we have to do is consider the likelihood of the data and multiply the two together before normalising. In some simple models this can be done easily using what are known as *conjugate* priors which ensure the posterior distribution of θ is in the same family as the prior.

In complicated models however we often run into a few problems. For example normalisation can be very tricky as we tend to try to evaluate $p(\mathcal{D}) = \int p(\mathcal{D}|\theta)p(\theta)d\theta$ where the integral can be completely computationally intractable. We thus usually with modern problems have to resort to other techniques in order to evaluate the posterior which we shall cover in the next couple of sections.

2.4.1 Making Predictions

Having evaluated the posterior of the parameter of interest we will often be interested in the distribution of a new observation x^* which by the law of total probability evaluates to:

$$p(x^*|\mathcal{D}) = \int_\Theta p(x^*|\mathcal{D},\theta)p(\theta|\mathcal{D})d\theta, \tag{2.6}$$

which we can numerically approximate as:

$$p(x^*|\mathcal{D}) \simeq \sum_{i=1}^{S} p(x^*|\mathcal{D},\theta^{(i)}), \tag{2.7}$$

where we take S samples of $\theta^{(i)} \sim p(\theta|\mathcal{D})$. Thus it is very useful for us to be able to sample from the posterior distribution even if we can't analytically evaluate it.

2.5 Markov Chain Monte Carlo

We have seen in Bayesian inference that we will often be interested in posterior distributions of some form that are often intractable, as is the case in neural networks. We therefore need some more sophisticated way of sampling other than simple transformation methods. Markov chain Monte Carlo (MCMC) is such a method where we would like to sample from a *target* distribution of the form:

$$p(x) = \frac{1}{Z} p^*(x), \tag{2.8}$$

where $p^*(z)$ is some unnormalised distribution that we can evaluate and the calculation of Z is intractable. The idea in MCMC is to simply sample from a Markov chain that has $p(x)$ as its stationary distribution.

2.5.1 Markov Chains

A Markov chain (MC) is a stochastic process; a sequence of random variables X_1, X_2, X_3... satisfying the Markov property:

$$p(x_{n+1}|x_1, x_2, ..., x_n) = p(x_{n+1}|x_n). \tag{2.9}$$

That is to say that that the conditional distribution of X_{n+1} is independent of X_m $\forall m < n$ given X_n. A *time-homogeneous* Markov chain is therefore completely defined by a transition kernel $q(x'|x)$ which defines a distribution on X_{n+1} given X_n $\forall n$. For any given kernel under certain conditions there exists a *stationary* distribution, $q_\infty(x')$ such that the the following holds:

$$q_\infty(x') = \int_X q(x'|x) q_\infty(x) dx. \tag{2.10}$$

Further if the chain is irreducible the distribution of X_n tends to $q_\infty(x')$ as n tends to infinity. As such in MCMC given a target distribution $p(x)$ we look for a transition kernel which has stationary distribution $q_\infty(x') = p(x)$. There are a number of different algorithms to do this and we will discuss a couple of the important ones next.

2.5.2 The Metropolis-Hastings Algorithm

The first example of MCMC was the Metropolis-Hastings algorithm [Metropolis *et al.*, 1953] where we select a proposal distribution, $\tilde{q}(x'|x)$ that is easy to sample from. We then draw a sample from this distribution before deciding to accept the proposed step with some probability. The acceptance probability is chosen so that the resulting distribution is stationary with respect to $p(x)$ and the derivation of which I will omit. Looking at the acceptance probability:

$$p = \max\{1, \frac{\tilde{q}(x_{i-1}|x')p^*(x')}{\tilde{q}(x'|x_{i-1})p^*(x_{i-1})}\},$$

we see that intuitively we are more likely to accept proposal steps that have higher target density as is seen by the ratio but also for ones that it is more likely to go in the reverse direction under the proposal distribution, helping the chain to not get stuck in places.

This is a very simple and easy to implement algorithm and you can see an example chain in Fig. 2.3. The problem is that when the target is complicated and very high dimensional the sampler is often unable to rapidly explore the state space and will take a very long time to converge to $p(x)$.

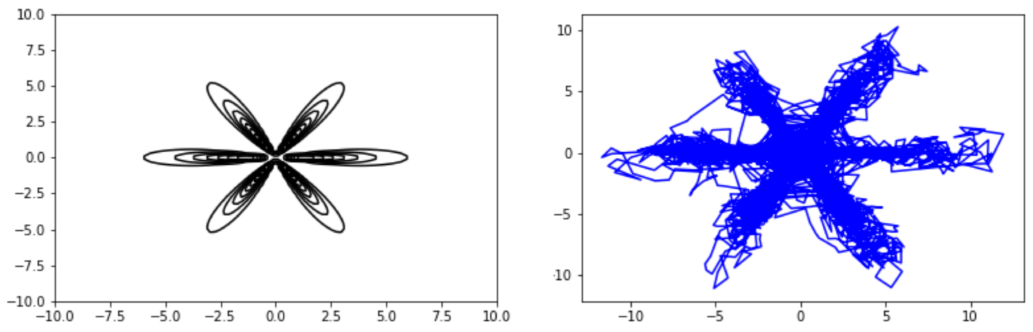


Figure 2.3: Left: A contour plot of a two-dimensional example target distribution. Right: An example of a path of a random walk Metropolis-Hastings sampler.

Algorithm 2: Hybrid Monte Carlo Sampler

```
Result: Samples $x_0$,$x_1$,$x_2$,...,$x_N$
Require : $N$ (No. of samples);
Require : $p(y)$ (Proposal distribution);
Require : $x_0$ (Starting point);
Require : $h$ (No. of Hamiltonian steps);
for $i$ = 1 to $N$ do
    Draw sample $y$ from $p(y)$ ;
    Choose random forward or backward direction;
    From $(x_{i-1}, y)$ follow $h$ Hamiltonian dynamics steps to reach proposal
      $(x', y')$;
    Let $a = \exp(H(x_{i-1}, y) - H(x', y'))$;
    Draw $u$ from a Uniform(0,1) distribution;
    if $u < a$ then
        $x_i = x'$
    else
        $x_i = x_{i-1}$
    end
end
return $x_0$,$x_1$,$x_2$,...,$x_N$
```

2.5.3 Hybrid Monte Carlo

The idea behind Hybrid Monte Carlo (HMC) [Duane *et al.*, 1988] [Brooks *et al.*, 2011], otherwise often known as Hamiltonian Monte Carlo, is to introduce an auxiliary random variable Y and sample from the joint distribution $p(x, y)$. Then we can just consider the Xs as samples but take advantage of properties of Y to allow the sample to make large non-local jumps and explore the state space more rapidly. The main intuition here is to think of the joint variables as the energy of some physical system, with the Xs representing the potential energy and the Ys the kinetic energy or momentum. We consider targets of the form:

$$p(x) = \frac{1}{Z_X} e^{U(x)}, \tag{2.11}$$

and define our distribution of Y similarly as one that is easy to generate samples from:

$$p(y) = \frac{1}{Z_Y} e^{K(y)}, \tag{2.12}$$

giving joint distribution:

$$p(x, y) = p(x)p(y) = \frac{1}{Z} e^{U(x)+K(y)} = \frac{1}{Z} e^{H(x,y)}, \tag{2.13}$$

with some *Hamiltonian* $H(x, y)$. The idea is to propose updates $x' = x + \Delta x$, $y' = y + \Delta y$ such that $H(x, y)$ is conserved, since we are using it to represent energy in a closed system. Following Hamiltonian dynamics it corresponds to setting for some small ϵ:

$$\Delta x = \epsilon \nabla_y H(x, y) \qquad \Delta y = \epsilon \nabla_x H(x, y). \tag{2.14}$$

Thus with HMC sampling at each step we make several Hamiltonian dynamics updates, keeping the Hamiltonian roughly the same, before accepting the new proposal with a Metropolis step. To ensure a symmetric proposal distribution ϵ is chosen to be positive or negative randomly. The details are given in algorithm 2.

HMC proves much more effective than standard Metropolis-Hastings as factoring in the gradient of the Hamiltonian allows the sampler to effectively find its way around the state space. Considering the auxiliary random variable Y as a momentum variable allowing you can see how it allows the chain to pass through areas of low density swiftly and escape local areas of high density.

2.5.4 Advantages and Limitations

The key advantage to MCMC algorithms is that they are asymptotically exact. That is to say in the limit of infinite time we will be getting samples from the exact correct distribution. The problem of course is that we do not have infinite time and so we must simply run the chain for a long finite time and hope that the distribution is “close enough” to the target, something for which there are few theoretical guarantees.

Additionally these methods while very simple to implement are computationally very expensive and so it can take much longer to evaluate than some of the more approximate methods we will talk about later. The trade off is that we can be sure that the distribution will in fact tend to the correct distribution, something that other methods can neither guarantee nor hope to come close to. In terms of accuracy of the distribution, HMC is considered the gold standard among modern computational Bayesian techniques and as we shall see later has been successfully used to sample from the posterior in NN models.

2.6 Variational Inference

An alternative to MCMC is Variational Inference (VI) [Blei *et al.*, 2017] (often also known as Variational Bayes). Here we have an intractable distribution $p(x)$ which we would like to approximate with a simpler distribution $q(x)$ with some free parameters θ. We would like to then select θ such that $q(x)$ is as close to $p(x)$ as possible. A common measure used to assess this is the Kullback-Leibler (KL) divergence, defined as:

$$KL\big(q(x)||p(x)\big) = \mathbb{E}_q\big(\log q(x) - \log p(x)\big). \tag{2.15}$$

This has the useful properties of always being non-negative and equal to zero if and only if p and q are the same distribution. In practice we are often interested in the posterior distribution of some latent variables Z given the data $\mathcal{D}$, written $p(z|\mathcal{D})$, which we aim to approximate with $q(z)$. The KL divergence can still be hard to evaluate so we often rewrite the expression as follows:

$$\begin{aligned} KL\big(q(z)||p(z|\mathcal{D})\big) &= \int_{\mathcal{Z}} q(z) \log \frac{q(z)}{p(z|\mathcal{D})} dz \\ &= \int_{\mathcal{Z}} q(z) \log \frac{q(z)}{p(z,\mathcal{D})} dz + \int_{\mathcal{Z}} q(z) \log p(\mathcal{D}) dz \\ &= \int_{\mathcal{Z}} q(z) \log \frac{q(z)}{p(z,\mathcal{D})} dz + \log p(\mathcal{D}) \end{aligned}$$

$$\begin{aligned} \log p(\mathcal{D}) &= KL\big(q(z)||p(z|\mathcal{D})\big) - \int_{\mathcal{Z}} q(z) \log \frac{q(z)}{p(z,\mathcal{D})} dz \\ \log p(\mathcal{D}) &= KL\big(q(z)||p(z|\mathcal{D})\big) + \mathcal{L}(q), \end{aligned}$$

where $\mathcal{L}(q)$ is known as the Evidence Lower BOund (ELBO), since the *evidence* $p(\mathcal{D})$ is fixed with respect to the parameters of q, maximising $\mathcal{L}(q)$ is equivalent to minimising $KL\big(q(z)||p(z|\mathcal{D})\big)$. In the case where this is computationally tractable we then have the familiar task of selecting the parameters θ of $q(z)$ such that $\mathcal{L}(q)$ is maximised. We can then use any of the optimisation techniques already discussed and this essentially transforms the Bayesian inference problem into one of optimisation, an already extensively researched area and one for which there are many tools available. VI is often significantly less computationally expensive than MCMC but suffers from the fact it will be inherently non-exact as well as the other host of problems that come with non-convex optimisation. It can also require a lot of work on the part of the practitioner to derive and evaluate the ELBO of the suitable approximate distributions used as the integrals involved are very often analytically intractable.

Due to the relative speed of VI methods we shall see that these ideas are the basis of most of the efficient and scalable methods for trying to fit Bayesian neural networks.

2.7 Gaussian Processes

We have so far only considered *parametric* models of the data, Gaussian Processes (GPs) [Rasmussen, 2003] on the other hand are a type of model know as *Bayesian non-parametrics*. These are so-called since they can be thought of as having no parameters or even an infinite number such that it doesn't really make sense to consider the meaning of them. We should also note that NNs are often thought to not exactly be parametric models since the number of parameters is usually so high. The reason we discuss GPs here is that they present an alternate way to consider a distribution over some estimate given by a model, and in fact there has been shown to be many interesting connections to deep learning that we shall explore later.

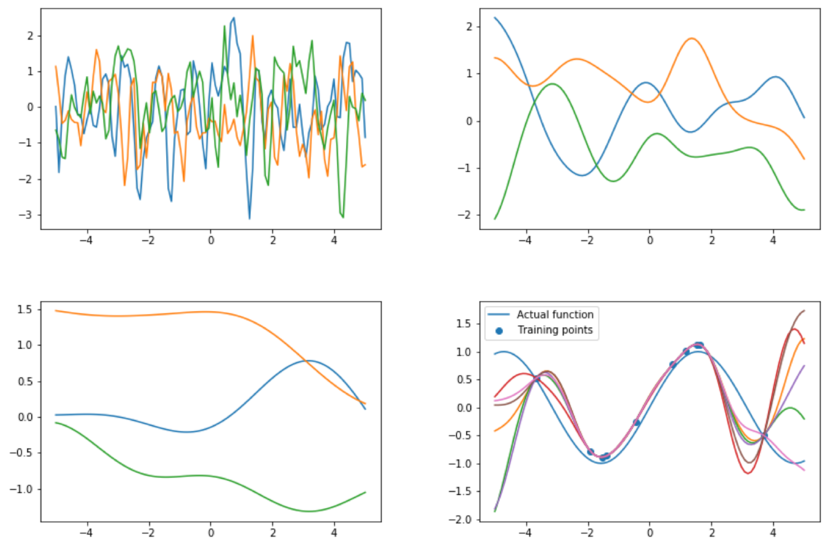


Figure 2.4: Top left to bottom right: Three draws from a squared exponential kernel with parameter a) 0.1 b) 1 c) 3 along with d) six draws from the predictive distribution given a squared exponential prior with parameter 1.

2.7.1 As a Prior over Functions

We consider the general supervised learning paradigm of learning a function $f : X \to Y$. The idea is that any function can be described by a vector of infinite length that at each entry contains the value of the function evaluated at a different point. A GP assumes this vector to be sampled from an infinite dimensional Gaussian distribution, that is to say any finite number collection of the variables is distributed according to a multivariate Gaussian. The distribution is generally considered to be zero mean, though this can be adjusted by using another function to evaluate the mean, with the covariance between any two points x and x' governed by a covariance function, or kernel, $k(x, x')$. A correctly defined kernel therefore completely defines an appropriate multivariate Gaussian which we can draw samples from easily, and with each sample representing a function we can as such consider it a prior distribution over functions.

2.7.2 Covariance Functions

As mentioned above we are looking to model a function, given as a distribution over the functions output values $p(Y|X)$, as a $N(0, K)$. Importantly K must be a positive definite matrix and so the method of constructing it is quite constrained. Still there are a number of different possibilities for the kernel, including the linear and squared exponential kernels. It can also be shown that any sums, products, and vertical rescalings of kernels are also themselves kernels. Fig. 2.4 shows a few different choices of covariance kernels and how they give rise to different priors over functions.

2.7.3 Making Predictions

When making predictions with GPs we simply use Bayes rule as shown in the Bayesian inference section to take our prior and update our distribution. I will omit the derivation for brevity but the key points are that this method is firstly exact in that there are no local minima to worry about (though there may be if we wish to also learn the hyperparameters of the model in an empirical Bayes paradigm). Secondly it is principled in that it is very clear how uncertainty is dealt with and where it comes from. Lastly though it is computationally expensive as for a sample of size n it requires calculating the inverse of an $n \times n$ matrix which is of complexity $\mathrm{O}(n^3)$, making them far too expensive for modern large datasets.

Chapter 3

Bayesian Deep Learning - A Review

"To know what you know and what you do not know, that is true knowledge."

Confucius

Summary

We will now give a brief history of Bayesian deep learning (BDL) as well as review the current state of the art, summarising and discussing important papers while identifying promising future areas of interest. We will go through the literature chronologically, examining how Bayesian inference was first considered in deep neural networks, before looking at the modern approximations that have driven the recent increase in popularity of these methods. Then we will cover important papers in more general variational inference which while not specifically designed for NNs can be applied to them.

While we look at multiple papers which each write things in their own way, I will now establish the notation I shall use throughout this chapter and in the future for clarity and so as to avoid confusion. We will always consider a data set $\mathcal{D}$ made up of n observation tuples $\{(x_i, y_i)\}_{i=1}^n$, where $(x_i, y_i) \in \mathbb{R}^k \times \mathbb{R}^m$ with x_i a k dimensional input and y_i an m dimensional output. We consider NNs / MLPs as functions $f : \mathbb{R}^k \to \mathbb{R}^m$ parameterised by a set of weights $\mathbf{w}$ and biases $\mathbf{b}$, made up of individual weights w_i and biases b_i. On the data set we shall consider the output of a NN as an estimator for the corresponding output, writing $\hat{y}_i = f(x_i|\mathbf{w}, \mathbf{b})$ with a Gaussian likelihood for the data, that is $Y_i|X_i = x_i \sim N(\hat{y}_i, \sigma^2)$.

3.1 The Origins of Bayesian Neural Networks

Since the introduction of deep neural networks there has been interest in a Bayesian fitting of the model, however for a long time even a deterministic fitting of the model was difficult when it came to large modern architectures. Calculating a distribution over a weight or bias will certainly require more work than a single point estimate since the distribution will itself be parameterised by usually at least a location and scale parameter.

3.1.1 General Framework and a Laplace Approximation

The first significant attempt to incorporate NNs into a Bayesian framework was done by MacKay [1992] which also introduced a Laplace approximation for some distributions of interest. MacKay asks us to consider the values:

$$E_D(\mathcal{D}) = \sum_{i=1}^{n} \frac{1}{2}(\hat{y}_i - y_i)^2 \qquad E_W(\mathbf{w}) = \sum_i \frac{1}{2} w_i^2,$$

where E_D is an error term for the accuracy of the model and E_W is a regularisation term to penalise large weights. We might then be interested in minimising $M = \alpha E_W(\mathbf{w}) + \beta E_D(\mathcal{D})$ where α and β are hyperparameters to be chosen. MacKay shows then that assuming Gaussian error in the data the likelihood of an observation in the model can be written:

$$p(y_i|x_i, \mathbf{w}, \mathbf{b}, \beta) = \frac{\exp[-\beta E_D((x_i, y_i))]}{Z_m(\beta)}, \tag{3.1}$$

with $Z_m(\beta) = \int \exp[-\beta E_D((x_i, y_i))]dy$ and a prior:

$$p(\mathbf{w}|\alpha) = \frac{\exp[-\alpha E_W(\mathbf{w})]}{Z_W(\alpha)}, \tag{3.2}$$

with $Z_W(\alpha) = \int \exp[-\alpha E_W(\mathbf{w})]d^k\mathbf{w}$ (With k weights). This yields a posterior:

$$p(\mathbf{w}|\mathcal{D}, \alpha, \beta) = \frac{\exp[-\alpha E_W(\mathbf{w}) - \beta E_D(\mathcal{D})]}{Z_M(\alpha, \beta)}, \tag{3.3}$$

with $Z_M(\alpha, \beta) = \int \exp[-\alpha E_W(\mathbf{w}) - \beta E_D(\mathcal{D})]d^k\mathbf{w}$. In this interpretation minimising M can be though of as finding the most probable singular values for the weights, $\mathbf{w}_{ML}$. A key point MacKay makes next is that if you assume the distribution to have a single mode at $\mathbf{w}_{ML}$ and that it is quadratic around the mode then Z_M can be approximated as:

$$Z_M \simeq e^{-M(\mathbf{w}_{ML})}(2\pi)^{\frac{k}{2}} det^{-\frac{1}{2}}\mathbf{A}, \tag{3.4}$$

where $\mathbf{A} = \nabla\nabla M$ is the Hessian of M evaluated at $\mathbf{w}_{ML}$. Thus MacKay introduces the method of evaluating point estimates of the weights using a typical method such as backpropagation before then building a Gaussian distribution around them using an evaluation of the Hessian for the variances. While we don't have any reason to believe these assumptions hold, or this is a good approximation to the posterior distribution, empirically it seems to produce good results when all the weights are considered together.

3.1.2 Minimising Descriptive Length

While this traditional Bayesian framework was being formulated, Hinton & van Camp [1993] came at the problem from a slightly different, information theoretic, point of view. Despite this they essentially became the first to use a form of variational inference, where the posteriors are approximated with a Gaussian or mixture of Gaussians.

The *Minimum Descriptive Length Principle* (MDL) [Rissanen, 1986] says that the best model for some data is the one that uses the least information to describe both the model and the discrepancy between the model output and the seen data. The idea is to minimise a loss function made up of a complexity loss (that says how much information is required to transmit the model) and an error loss (that relates to describing the deviations of the data from the model). Thus Hinton & van Camp encoded the weights of a NN as a Gaussian distribution and sought to optimise the accuracy of the model while keeping the KL divergence between this posterior and a Gaussian prior as small as possible. They did this by updating the means and variances of the weights in a similar way to the traditional backpropagation algorithm, taking the error at the end and calculating the partial derivatives with respect to each weight mean and variance layer by layer. This then models the posterior over all of the weights as a multivariate Gaussian distribution with a diagonal covariance matrix. This was improved on later by Barber & Bishop [1998] who showed you could get a similar deterministic algorithm that would permit a more general covariance matrix, allowing interactions between weights.

Following this method then results in distributions over weights, as we would expect from Bayesian inference. While this is done by “minimising descriptive length” and not multiplying a prior by a likelihood there are clear parallels and are often considered very similar if not the same idea just by different names. Indeed the complexity loss in MDL is comparable to the terms for the prior in Bayesian inference as well as to any regularisation terms in a traditional setting while the error loss relates to the likelihood of the data as well as standard cost functions.

3.1.3 Practical Hybrid Monte Carlo

In his PhD thesis, Neal [1994] addressed two important aspects of BDL. First was the issue of the meanings of priors over the weights in NNs. Priors are usually chosen because of previous knowledge about the distribution of the parameter but in NNs we really don’t have any intuition about what the shape of the distribution should look like and so in some sense it doesn’t make much sense for us to arbitrarily select a prior. To solve this, Neal considered how individual priors over weights interact and give rise to a prior over functions when together in the NN. In order to then assess them he looked at the limit of these functions as the number of hidden units goes to infinity. Neal showed that when considering Gaussian priors over the weights, if using a smooth activation function such as the hyperbolic tangent function (tanh), the priors would converge on a Gaussian process. Additionally if using a step-function they would instead converge on a Brownian process. This gives us at least a principled reason to select priors over individual weighs even if we have no reason to think the weight actually has such a distribution.

Secondly Neal introduces the idea of using Hybrid Monte Carlo (algorithm 2) to sample from the posterior distribution. Before this, only approximations had been tried but Neal thought that it was important to have a method which did not rely in a specific parameterisation of the posterior. As such Neal used HMC to accurately sample from the posterior of the weights as well as using Gibbs sampling in order to optimise the hyperparameters.

While this method undoubtedly gives more accurate solutions, there is a cost. Neal noted that on the hardware of the time finding a solution to the “robot arm problem” took about 20 hours of computation time, while MacKay [1992] found his method took only 6 minutes, leading to a hugely significant difference.

3.2 Modern Approximations

After the initial flourish of excitement around BDL there was a bit of a period where very little progress was made. These initial ideas seemed to have little scope as the approximations would break down with scale or simply take too long do be of any practical value. This also coincided with a time where NNs in general fell a little out of favour with the research community as little progress was being made and models like the *Support Vector Machine* [Scholkopf & Smola, 2001] seemed to be the way forward.

That being said there has recently been another burst of interest in the area, with a number of effective and scalable algorithms proposed, as well as ways to interpret common practices as a form of Bayesian inference, which we shall now survey.

3.2.1 Practical Stochastic Variational Inference

Building on the work of Hinton & van Camp [1993], Graves [2011] introduced a simple idea to make training more computationally practical. Essentially Graves decided to ignore any analytic solution and focus on variational distributions that have expectations that are well numerically approximated. A good example that can be applied to any differentaible log-loss model is a diagonal Gaussian. For a NN considering the negative log-likelihood as the network loss, $L^N(\mathbf{w}, \mathcal{D}) = -\log p(\mathcal{D}|\mathbf{w})$, this is clearly differentiable as done in backpropagation. Thus also considering a prior $p(\mathbf{w}|\alpha)$ parameterised by α, and a variational distribution $q(\mathbf{w}|\beta)$ paramterised by β we can write the ELBO:

$$\mathcal{L}(q) = \underbrace{\mathbb{E}_q(L^N(\mathbf{w}, \mathcal{D}))}_{L^E(\beta,\mathcal{D}),\text{ error loss}} + \underbrace{KL(q(\mathbf{w}|\beta)||p(\mathbf{w}|\alpha))}_{L^C(\alpha,\beta),\text{ complexity loss}}, \tag{3.5}$$

arriving at a clear formulation of an MDL loss function:

$$L(\alpha, \beta, \mathcal{D}) = \mathcal{L}(q) = L^E(\beta, \mathcal{D}) + L^C(\alpha, \beta),$$

showing that optimising $L(\alpha, \beta, \mathcal{D})$ is the same as variational inference in the model. All that's left then is to derive the partial derivatives of each loss expression with respect to the parameters of the variational distribution, which in a diagonal Gaussian consists of two vectors containing a mean μ_i and variance σ_i^2 for every weight and with a Gaussian prior with $\alpha = \{\mu, \sigma^2\}$. In this case the complexity loss can be evaluated analytically:

$$L^C(\alpha, \beta) = \sum_{i=1}^{W} \log \frac{\sigma}{\sigma_i} + \frac{1}{2\sigma^2}\big[(\mu_i - \mu)^2 + \sigma_i^2 - \sigma^2\big],$$

$$\begin{aligned}\frac{\partial L^C(\alpha, \beta)}{\partial \mu_i} &= \frac{\mu_i - \mu}{\sigma^2},\\ \frac{\partial L^C(\alpha, \beta)}{\partial \sigma_i} &= \frac{1}{2}\big[\frac{1}{\sigma^2} - \frac{1}{\sigma_i^2}\big].\end{aligned}$$

The error loss on the other hand requires a Monte Carlo approximation, as an expectation we take the mean of S samples drawn from q and can then take derivatives:

$$L^E(\beta, \mathcal{D}) = \mathbb{E}_q(L^N(\mathbf{w}, \mathcal{D})) \simeq \frac{1}{S}\sum_{k=1}^{S} L^N(\mathbf{w}^k, \mathcal{D}),$$

$$\begin{aligned}\frac{\partial L^E(\beta, \mathcal{D})}{\partial \mu_i} &\simeq \frac{1}{S}\sum_{k=1}^{S} \frac{\partial L^N(\mathbf{w}^k, \mathcal{D})}{\partial w_i},\\ \frac{\partial L^E(\beta, \mathcal{D})}{\partial \sigma_i} &\simeq \frac{1}{2S}\sum_{k=1}^{S} \Big[\frac{\partial L^N(\mathbf{w}^k, \mathcal{D})}{\partial w_i}\Big].\end{aligned}$$

where the partial derivatives of the network loss are obtained in the standard, backpropagation manner. Graves also derives the derivatives for a number of other variational distributions as well as for optimising α so that we can learn the parameters of the prior under an empirical Bayesian framework.

3.2.2 Bayes by Backprop

The estimators from Graves [2011] suffer from being slightly biased, building on the work though Blundell *et al.* [2015] introduced a "backpropagation-compatible algorithm for learning a probability distribution on the weights of a neural network" which calculates unbiased gradients and updates the weights in a backpropagation style algorithm. The algorithm, called *Bayes by Backprop*, focuses again on a diagonal Gaussain posterior but relaxes limitations on the prior that the complexity cost be analytically calculated.

Again, a diagonal Gaussian is parameterised by a vector of means μ and variances σ^2 though here they parameterise $\sigma = \log(1 + \exp(\rho))$ ensuring σ is always positive and so the parameters of the variational distribution q are $\theta = \{\mu, \rho\}$. They also note that the weights can be written $\mathbf{w} = \mu + \log(1 + \exp(\rho))\epsilon$ where $\epsilon \sim N(0, 1)$. While calculating the derivatives of the ELBO they consider the intermediate function:

$$f(\mathbf{w}, \theta) = \log q(\mathbf{w}|\theta) - \log p(\mathbf{w})p(\mathcal{D}|\mathbf{w}), \tag{3.6}$$

which is a standard NN cost function with regularisation, the derivatives of which can again be calculated with backpropagation. Sampling ϵ the gradients of the ELBO can then be calculated and updated as follows:

$$\begin{aligned}
\frac{\partial \mathcal{L}(q)}{\partial \mu} &= \frac{\partial}{\partial \mathbf{w}} f(\mathbf{w}, \theta) + \frac{\partial}{\partial \mu} f(\mathbf{w}, \theta), \\
\frac{\partial \mathcal{L}(q)}{\partial \rho} &= \frac{\partial}{\partial \mathbf{w}} f(\mathbf{w}, \theta) \frac{\epsilon}{1 + \exp(-\rho)} + \frac{\partial}{\partial \rho} f(\mathbf{w}, \theta).
\end{aligned}$$

Thus to calculate the gradients with respect to the mean and variance you simply need to calculate the standard gradients with respect to the weights before scaling and shifting them appropriately.

With Blundell *et al.* coming out of DeepMind it's only natural that they highlight the applications of their method in reinforcement learning, particularly *contextual bandits* where the associated uncertainty when using BDL to evaluate agents *Q functions* can help in the common exploration/exploitation problems.

3.2.3 Probabilistic Backprop

While methods so far have mostly come from an MDL perspective, Hernández-Lobato & Adams [2015] introduced a new approach and algorithm they call *probabilistic backpropagation* (PBP). Similar to the traditional backpropagation algorithm, PBP has two phases. In the first, the data is input and then propagated through the network. This gives rise to intractable distributions over the weights which PBP approximates with a Gaussian by *matching moments* (choosing the mean and variance to be equal to the ones of the intractable distribution). This is a resultant property of the KL divergence when considering Gaussians. After the first phase the output is given as the negative log-likelihood of the data and so in the second phase the gradients of this are taken with respect to the means and variances of the Gaussians and propagated backward through the network using reverse-mode automatic differentiation so that they can be used to update the parameters.

3.2.4 Dropout as a Bayesian Approximation

Srivastava *et al.* [2014] introduced *dropout* as a technique for regularising a neural network. The idea was fairly simple, with each pass through the network during training time each neuron would be randomly "dropped" with some probability $1 - p$ (i.e. the activation would be set to zero). Mathematically, and keeping notation from section 2.2, every layer this amounts to sampling a vector d^l from a binomial distribution with the same length as the number of neurons in the layer and with probability p and multiplying the activations elementwise by this vector (note p can be a global parameter but we will usually consider a separate p^l per layer):

$$a^l = \sigma\big(w^l a^{l-1} + b^l\big) \odot d^l. \tag{3.7}$$

Only weights connecting non-zero neurons will be updated after the pass and so for every training example this is essentially sampling from an exponential number of smaller neural networks to update. The main point here is that specific neurons don't become co-dependant on each other and the network is still able to perform accurately without all of the neurons being present. At test time, instead of sampling and setting activations to zero all outputs are kept but are scaled by p so that the overall magnitude of the outputs stays the same during

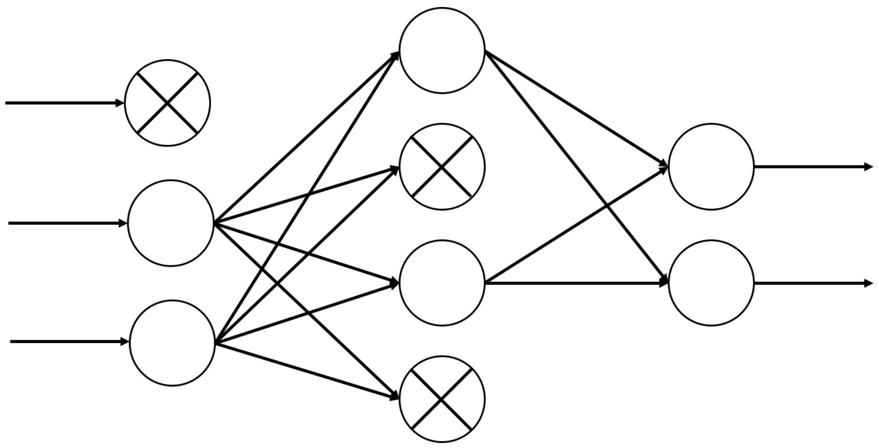


Figure 3.1: An example of the effect of dropout; each node with a cross is "dropped" and so no output from it is passed to the next layer.

both training and test time. Srivastava *et al.* showed this technique significantly reduced over-fitting and produced more accurate results at test time.

Given the ease of implementation as well as clear benefits dropout saw wide use in the community, but was just seen as a particularly useful regularisation technique and nothing more. That changed when Gal & Ghahramani [2016] showed that actually training a network with dropout was mathematically equivalent to approximate Bayesian inference in a deep Gaussian process.

Gal & Ghahramani showed that in both the case of training a NN with dropout and L2 regularisation, and performing variational inference you arrive at the same objective function to be optimised up to a constant:

$$\mathcal{L} \propto \frac{1}{N}\Big[\sum_{i=1}^{N}(\hat{y}_i - y_i)^2\Big] + \tau \sum_{j=1}^{l} p^{l-1}||w^j||^2, \tag{3.8}$$

where τ is made equal in each case through careful selection of hyper-parameters. In order to then use dropout to assess uncertainty, Gal & Ghahramani suggests *Monte Carlo (MC) dropout* whereby instead of scaling the activations at test time you simply make a large number of passes through the network, sampling every time, and then taking the empirical distribution of the output to get information about how certain the output is. Thus they provided a very fast and principled way to perform BDL with which many people were already familiar and did not require much further modification.

3.3 Important Work in Bayesian Computation

We will now take a look at some important ideas in computational Bayesian methods that while not specifically designed for use in NNs have strong connections that are either directly related or can be applied with small modifications.

3.3.1 Stochastic Gradient Langevin Dynamics

Welling & Teh [2011] introduce a novel way of sampling from a posterior distribution by combining standard stochastic optimisation with Langevin dynamics to efficiently generate samples in a style quite similar to MCMC. First Welling & Teh note that when considering a parameter θ with prior $p(\theta)$ along with some data $\mathcal{D} = \{x_i\}_{i=1}^{N}$ and consequent likelihood $p(\mathcal{D}|\theta)$ a standard optimisation procedure at step t updates θ according to:

$$\Delta\theta_t = \frac{\epsilon_t}{2}\Big(\nabla \log p(\theta_t) + \frac{N}{n}\sum_{n}^{N} \log p(x_{ti}|\theta_t)\Big), \tag{3.9}$$

with ϵ_t a Robbins & Monro [1951] sequence. The idea in the paper is quite simple, take the standard update rule and add a noise term:

$$\Delta\theta_t = \frac{\epsilon_t}{2}\Big(\nabla \log p(\theta_t) + \frac{N}{n}\sum_{n}^{N} \log p(x_{ti}|\theta_t)\Big) + \xi_t, \tag{3.10}$$

where $\xi_t \sim N(0, \epsilon_t)$ and the variance is chosen in accordance with the stochastic estimator of the likelihood to give the appropriate resulting variance for the posterior distribution. The noise term prevents θ from converging on any ML/MAP estimate of the parameter and as Welling & Teh show it actually allows convergence on the correct posterior distribution as in a similar way to Langevin Monte Carlo (An MCMC variant that proposes steps based on a discretisation of the Langevin stochastic differential equation) but with a decaying step size that removes the need for any Metropolis-Hastings acceptance step. This is the part that saves a lot of computation time since in MCMC this step requires an evaluation over all the data. Thus it we can start optimising θ and after a suitable amount of time to allow the Langevin dynamics to take effect we can take the values achieved as samples form the posterior (like in MCMC) for use in Monte Carlo estimates.

3.3.2 Black Box Variational Inference

Black box variaional inference (BBVI) [Ranganath *et al.*, 2014] was introduced as a simple and effective way to perform VI in a wide variety of models. Ranganath *et al.* noted that in many cases it took practitioners a long time to derive an appropriate form of the lower bound for every different model they used and a general case algorithm was missing. BBVI therefore aims to fill this gap, the main idea is considering the ELBO as an expectation with respect to the variational distribution:

$$\mathcal{L}(q) \triangleq \mathbb{E}_{q_\theta(z)}[\log p(x,z) - \log q(z|\theta)]. \tag{3.11}$$

They then show that you can write the gradient of this ELBO as an expectation with respect to the variational parameters as well:

$$\nabla_\theta \mathcal{L}(q) = \mathbb{E}_{q_\theta(z)}[(\nabla_\theta \log q(z|\theta))(\log p(x,z) - \log q(z|\theta))]. \tag{3.12}$$

Thus both of these expressions can be approximated by the sample mean of S Monte Carlo samples from the variational distribution, in particular:

$$\nabla_\theta \mathcal{L}(q) \simeq \frac{1}{S}\sum_{s=1}^{S}[(\nabla_\theta \log q(z_s|\theta))(\log p(x, z_s) - \log q(z_s|\theta))], \tag{3.13}$$

where $z_s \sim q(z|\theta)$. In practice we may make use of samples of size 1, making the algorithm computationally simpler although of course resulting in a high variance estimator of the ELBO.

This means we can then perform standard stochastic optimisation on the ELBO with very few restrictions. Crucially all that is required is that you can evaluate the log of the joint, $\log p(x, z)$, which one is usually able to do. Of course we are also assuming we can sample from, and evaluate the score function of, the variational distribution but since those are simple distributions chosen for convenience we should be able to do that. This means then that we can apply BBVI in NN models as well, although it does tend to be the case that the stochastic estimators of the gradient have quite large variance. This requires some tricks to decrease the variance, a couple of which Ranganath *et al.* details in the paper.

3.3.3 The Reperameterisation Trick

In another attempt to scale up VI, Kingma & Welling [2013] introduced a new way to parameterise the ELBO so as to yield an easy stochastic estimator of the gradient. As part of this, and probably the most significant contribution of the paper, is the introduction of the *reperameterisation trick* that provides a way of introducing differentiability with respect to the parameters of distributions. Suppose there is a latent variable z of interest with variational posterior distribution such that $z \sim q_\phi(z|x)$. The key observation by Kingma & Welling is that it is usually possible to write $z = g_\phi(x, \epsilon)$ where g is some deterministic function and ϵ a noise variable. For example if $z \sim N(\mu, \sigma^2)$ then we may write $z = \mu + \sigma\epsilon$ where $\epsilon \sim N(0, 1)$. This retains the important property that:

$$\mathbb{E}_z\big[f(z)\big] = \mathbb{E}_\epsilon\big[f(g_\phi(x, \epsilon))\big], \tag{3.14}$$

so that we can still sample the noise variable to get Monte Carlo approximations. The important thing is that any direct realisation of z is not differentiable with respect to the distributional parameters. However we can now instead sample from the noise distribution and pass it through g_ϕ to get an equivalent sample but one which we can differentiate.

There are a few standard ways to pick a suitable g function. First, like with the Gaussian example, any "location-scale" family can be picked in the same manner of location + ($\epsilon \times$ scale). Second is in the case of a tractable inverse CDF, where $\epsilon \sim U(0, 1)$ and g is the inverse CDF. Lastly some random variables (for example the log-normal or Gamma) can be written as the composition of more basic auxiliary variables.

In the same paper, Kingma & Welling present some more techniques for optimising the ELBO in unsupervised learning with an algorithm for learning an encoding and decoding distribution for the latent variables including when using a neural network to arrive at the variational auto-encoder, a now popular and powerful generative model.

3.3.4 Normalising Flows

In an attempt to create a very flexible class of variational distribution Rezende & Mohamed [2015] introduced the use of *normalising flows* to take a standard base distribution such as a Gaussian and then transform it multiple times by passing it through a series of invertible mappings until you result in a potentially very complex and flexible distribution.

Given an invertible function f and latent variable z with distribution $q(z)$ the resulting random variable $z' = f(z)$ had distribution:

$$q(z') = q(z)\left| \det \frac{\partial f^{-1}}{\partial z'} \right| = q(z)\left| \det \frac{\partial f}{\partial z} \right|^{-1}. \tag{3.15}$$

Repeating this a finite N number of times with different invertible functions f_i so as to obtain the new random variable $z_N = f_N \circ ... \circ f_1(z)$ results in a distribution of:

$$\log q_N(z_N) = \log q(z) - \sum_{i=1}^{N} \log \left| \det \frac{\partial f_i}{\partial z_{i-1}} \right|. \tag{3.16}$$

Thus we can sample for $q(z)$, pass it through the function and arrive at a new sample z_N from q_N which we can then use in a numerical approximation of the ELBO. Then, given suitable functions, we can take the derivative of the ELBO with respect to the parameters of these functions and update them accordingly.

Rezende & Mohamed also consider the case when the flow is infinitely long. In this case, instead of being described as a sequence of transformations, we consider a partial differential equation that describes how the density changes over time. As with Welling & Teh [2011] they consider the Langevin stochastic differential equation to show that it can be used to sample from the exact posterior in the limit of infinite time.

Chapter 4

Experiments and Contributions

"If we knew what it was we were doing, it wouldn't be called research, would it?"

Albert Einstein (Apparently)

Summary

Now that we have discussed Bayesian deep learning extensively and considered a variety of different approximate methods for evaluating the posterior distributions, we will now detail the practical aspects of our work on this project. First we will implement a number of these algorithms to examine how they work in practice as well as consider the trade-offs in using different methods. Then we will get to the main part of this project, improving the flexibility of the dropout approximation by incorporating the dropout rate into the variational distribution, and seeing how this works in practice.

4.1 Comparison of Methods

Having looked in depth at Bayesian neural networks and examined a few different methods for how to approximate the posterior, it might be useful to see how they work in practice. We will look specifically at four implementations: Hybrid Monte Carlo, Bayes by Backprop, MC Dropout, and Graves' variational inference method. These are interesting as they cover the main approaches from Markov chain Monte Carlo to fitting a mean-field approximation in backpropagation style as well as an interpretation of dropout as a Bayesian fit and a minimum descriptive length framework.

4.1.1 Predictive Distribution Analysis

In order to get an idea for how accurate these approximate methods are we implement a few of them on a simple one-dimensional regression problem. In the set up we consider the data set $\{x_i, y_i\}_{i=1}^{100}$ where the x_i are sampled from a $N(0, 4)$ distribution truncated at -4 and 4 with $y_i = 100\sin(x_i) + 10\epsilon$, $\epsilon \sim N(0, 1)$. This set up is beneficial since we can visualise the posterior predictive distributions well, in higher dimensions this becomes impossible.

For all of the methods we consider a standard set up for the neural network. We use a single hidden layer with 128 neurons connected by a hyperbolic tangent activation function. All prior distributions are chosen as standard Gaussian. Each method though had a number of parameters unique to that method and we tried to find the optimal parameters in each case though it can't be guaranteed an optimum was reached. Once trained, in order to visualise the uncertainty, we took 100 samples from the posterior and evaluated the neural network on a test set of 100 equally spaced points on the interval $[-4, 4]$. For HMC we use a slightly different method of producing 100,000 samples from the chain, using half

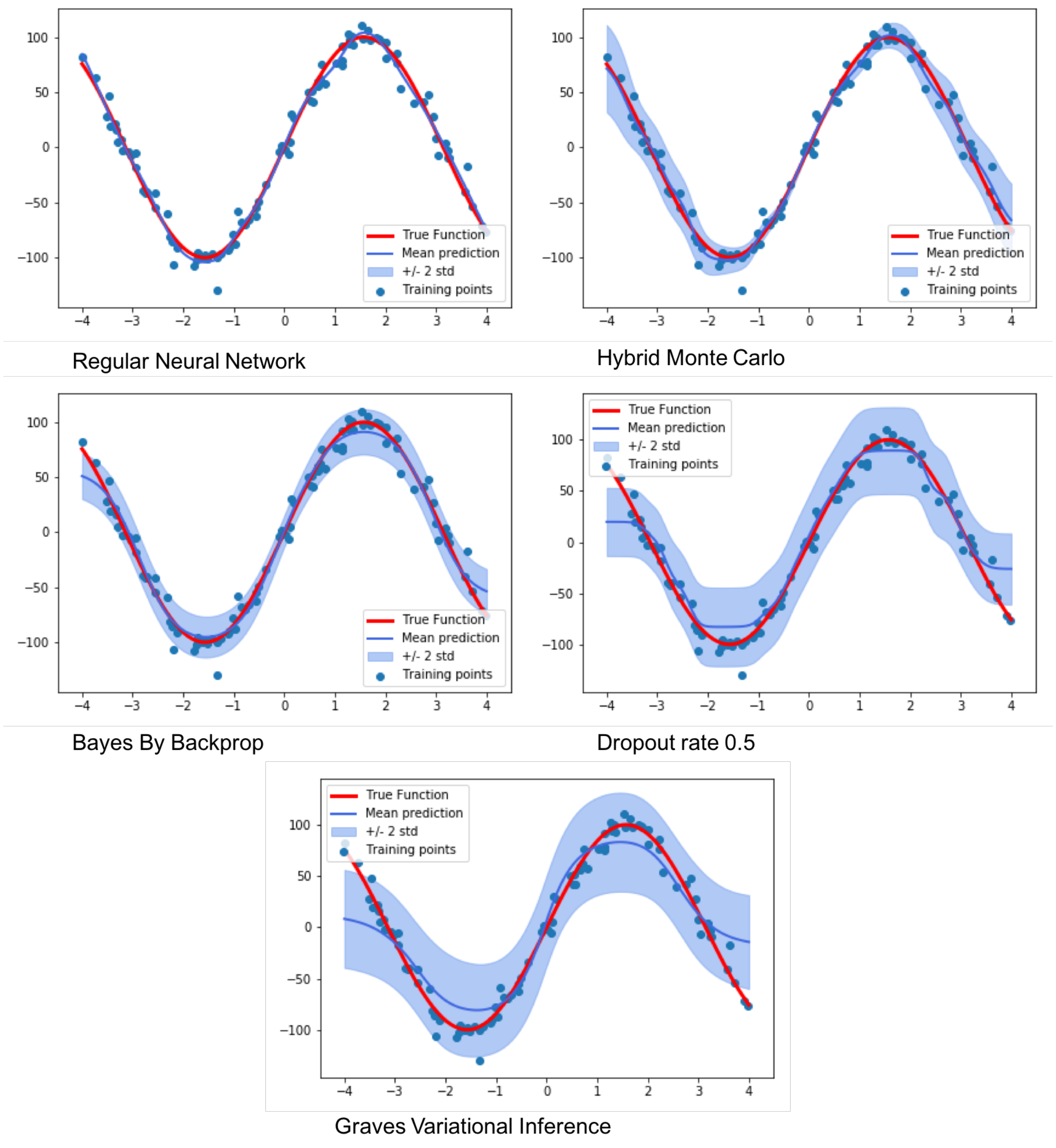


Figure 4.1: Results of a toy regression problem using a few different Bayesian deep learning methods. The red line represents the true underlying function with the blue dots the training set. The dark blue line represents the mean prediction while the shaded blue area represents plus minus two standard deviations from the mean. [See Appendix A]

of those as burn-in and using the other half for prediction (Acceptance probability was tuned to 65%, as suggested by Beskos *et al.* [2013]). Given these predictive realisations we plot the empirical mean prediction as well as +/- 2 standard deviations, as shown in Fig. 4.1.

Before looking at the Bayesian approach we first note the traditional neural network first the data very well though there does look to be slight over-fitting happening as the prediction is not as smooth everywhere as the true function. Of course in simple cases like this we would expect the traditional method to do well and it is not a case where we would really be able to make good use of a Bayesian approach's benefits.

Unlike the other methods we do know that HMC is asymptotically exact. We shall thus use it as the standard by which we measure the other methods in terms of accuracy, as we assume it to be close enough to the "truth" given the number of samples taken and burn-in period used to make fairly accurate comparisons. Given this, and looking at Fig. 4.1 we see that all the methods tend to have a higher predictive variance than HMC but with a fairly accurate mean. However Bayes by Backprop produces the closest predictive distribution, though it appears still to have slightly too much variance in the turning points of the function. Second comes MC Dropout which seems to have quite a high variance at all points but otherwise a fairly accurate mean prediction. Graves' method performs the worst, which is not particularly surprising as it's an old method with a lot of approximations.

With that being said it's important to remember this is only a very simple example and we can't really be sure how the posterior distributions will behave in a higher dimensional or more complicated task. For example dropout in its traditional use (i.e. not as a Bayesian approximation) tends to work better for bigger problems and in networks with more layers and neurons so we might see an improvement in a different setting.

4.1.2 A Note on Speed

For any method the key things that matter are how accurate they are and how fast we can implement them. Having looked at the accuracy of the methods we may now want to compare the speeds. From our experience implementing them it was clear the regular neural network and dropout where comparable and by far the fastest. This was followed by Bayes by Backprop and Graves with HMC taking by far the longest to run. We haven't included any specific timings as they wouldn't be very useful, since when it comes to these large scale algorithms a lot of work can be done to numerically optimise how fast they can run. This includes tricks with the mathematics as well as multi-threading and GPU implementation and we simply aren't sophisticated enough to get all of the algorithms running at optimal speed. However as we have done we can comment generally on at least comparatively how fast they run when implemented in a fairly basic fashion which should hopefully be useful.

4.2 Dropout Rate Optimisation

We shall now move on to the main result of this project, answering the question as to how we can and should optimise the rate of dropping neurons in dropout.

4.2.1 The Problems with a Fixed Rate

Since the original paper where dropout was formulated it has always been recommended to use 0.5 as the dropout rate (p) at non-input layers. This decision has always been based on the empirical evidence that 0.5 appears to work well, as Srivastava *et al.* say, "[p] can simply be set at 0.5, which seems to be close to optimal for a wide range of networks and tasks". They note also though that it can be chosen based on a validation set, as Gal & Ghahramani also suggest, via grid search.

In the first case of just using one half this appears problematic in that it seems we are unnecessarily reducing the flexibility of our approximation by restricting ourselves to only one value of p. The second presents a different problem of seeming rather unprincipled when in the Bayesian context but also computationally expensive, having to evaluate the model on a wide range of values of p.

We will now present two different ways to optimise the dropout rate based on the data and see how they can be used in the wider context of both standard and Bayesian neural networks.

4.2.2 Coordinate Ascent

We first ask the question why is p not optimised, like all the other parameters, by taking the gradient of the cost function and updating accordingly? Simply, it doesn't show up in the expression for the model when we consider the original formulation of dropout (i.e. not Bayesian). Without loss of generality, if we consider the forward pass at training time for a single hidden layer neural network with dropout we get:

$$f_1(x) = (\sigma((x \odot d_1)w_1 + b_1) \odot d_2)w_2 + b_2, \tag{4.1}$$

where w_1, w_2 weight matrices, b_1, b_2 bias vectors, σ an elementwise activation function, $\odot$ symbolising elementwise product, and d_1, d_2 vectors of Bernoulli random variables sampled with probability p_i (the dropout rate in layer i). Importantly there is no occurrence of p in this expression and so no gradient can be taken, p is instead used to sample a random variable which takes values not dependant

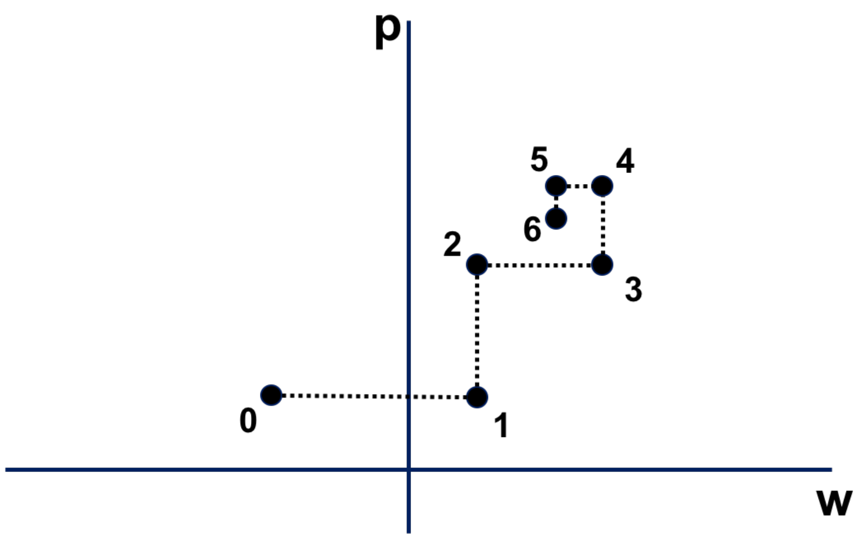


Figure 4.2: A simplified look at how coordinate ascent works, we take a step to update w followed by a step to update p. Crucially only one of the parameters is changed at each update.

on p. Having said that, when it comes to test time a different function is used:

$$f_2(x) = (\sigma((xp_1)w_1 + b_1)p_2)w_2 + b_2. \tag{4.2}$$

Suddenly p becomes involved and so any cost function C involving f_2 will be differentiable in p. It might be tempting then to optimise solely based on f_2, but this of course is just a regular neural network with scaled weights and so loses the dropout benefits. Additionally any value of the weight has infinite combinations of w and p that come to the same value, giving no preference to any particular combination.

Given this, we propose a method for optimising all the parameters via coordinate ascent (Fig. 4.2). This amounts to having two steps to the optimisation procedure, one gradient step for the weights and biases based on f_1, and one gradient step for p based on f_2, as demonstrated in algorithm 3.

```
Algorithm 3: Coordinate Ascent Dropout
  Result: Optimised parameters w,p
  Require : $\lambda$ (Step size);
  Require : $w_0, p_0$ (Starting points);
  $t \leftarrow 0$ (Initialise timestep);
  while $w_t, p_t$ not converged do
      $t \leftarrow t + 1$;
      $w_t \leftarrow w_{t-1} - \lambda \nabla_w C(f_1(x; w_{t-1}, p_{t-1}), y)$;
      $p_t \leftarrow p_{t-1} - \lambda \nabla_p C(f_2(x; w_t, p_{t-1}), y)$;
  end
  return $w_t, p_t$ (Resulting parameters)
```

4.2.3 Practical Behaviour

Applying this algorithm "works" in that our parameters converge and our training error gradually decreases (see Fig. 4.3). This is by no means a guarantee as we are optimising two different objective functions and so it would be perfectly possible to not reach a joint global optimum. The behaviour of the weights behaves exactly how we might expect given the training process is the same as before and is consequently not of much interest to us. The behaviour of p however is interesting and of most importance to our question, and indeed depends on the problem setting.

In the case of a regression problem we see that the rate goes to one (or very, very close) in every case. This is a problem really because when p equals one then we're just left with a regular neural network with no dropout and so completely defeats the purpose in using it. The reason for this is unclear.

On the other hand when we consider a classification problem we find the opposite, that p goes very close to zero (it never actually goes to zero since that would give an output of zero which is clearly not optimal in any context). This again is a problem as with a low rate of retaining neurons it introduces a lot of randomness into the weights and so the variance of the outputs increases

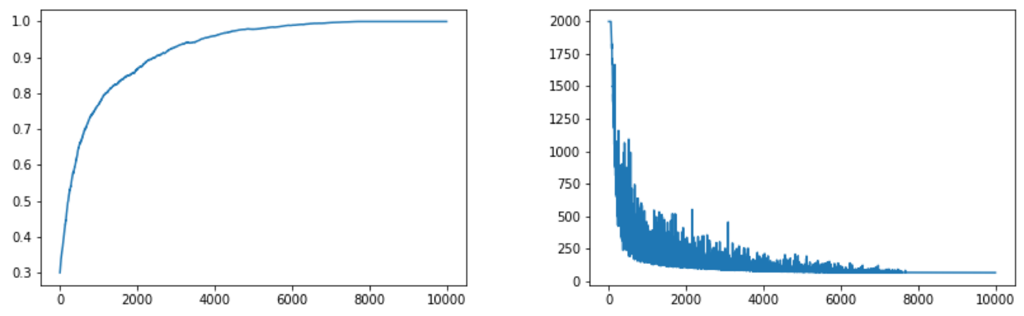


Figure 4.3: Results of a toy regression problem. Left: The change in p during training. Right: Euclidean loss change during training

dramatically. The reason this might happen in classification problems is probably to do with output layer and the use of the softmax function to scale all the outputs so they add to one. Thus if all of the inputs to a softmax function are scaled by the same amount (i.e. by p) then it won't have an impact on the output of the function. Thus the network is perfectly fine with p becoming very small which there is at least some justification for given the appearance of p in the L2 regularisation term.

In summary then we see that the algorithm experiences practical issues with the dropout rate going to either zero or one depending on the situation. Additionally there are reasons to be worried from a theoretic point of view. For example, we are essentially using two different objective functions to train the network so we might be concerned we will not reach a joint optimum. The objective for optimising p is also not well theoretically justified, in the original paper introduced only so that the expected output of the neurons remained the same during training and test time.

4.2.4 Incorporating the Bayesian Framework

Given the not particularly nice properties of the coordinate ascent algorithm we look for an alternative and more theoretically justified way to optimise the dropout rate. To do so we return the Bayesian interpretation of dropout. Gal & Ghahramani [2016] consider a variational distribution with a fixed value of p, but we define ours, with p a flexible parameter, to be of the form:

$$q_W(w; m, p) = p^{\frac{w}{m}}(1-p)^{\frac{m-w}{m}}, \tag{4.3}$$

for $w \in \{0, m\}$, 0 otherwise. That is w takes the value m with probability p and 0 with probability $(1-p)$. This leads us to an objective function to maximise:

$$\begin{aligned}\mathcal{L}(q) &= \mathbb{E}_q[\log p(X, W) - \log q(W; m, p)] \\ &= \mathbb{E}_q[\log p(X|W) + \log p(W) - \log q(W; m, p)] \\ &= \mathbb{E}_q[\log p(X|W)] + \mathbb{E}_q[\log p(W)] - \mathbb{E}_q[\log q(W; m, p)].\end{aligned} \tag{4.4}$$

Fortunately the three expectation terms of the ELBO are possible to evaluate either by MC approximation or analytically. For MC approximations we will consider only samples of size one where $\hat{w}$ will signify that realisation. The first, the expected log likelihood evaluates to:

$$\mathbb{E}_q[\log p(X|W)] = \frac{1}{2}\sum_{i=1}^{n}(f(x_i; \hat{w}) - y_i)^2. \tag{4.5}$$

The second, the expectation of the log prior (given a Gaussian prior) evaluates to:

$$\mathbb{E}_q[\log p(W)] \propto p||m||^2. \tag{4.6}$$

We say proportional too since we can tune the variance of the prior as a precision parameter. Lastly the entropy of the variational distribution evaluates analytically:

$$\mathbb{E}_q[\log q(W; m, p)] = k(p \log p + (1 - p) \log(1 - p)), \tag{4.7}$$

where k is the number of weights. Thus since $q_W(w; m, p)$ is easily sampled from we can get a good approximation to the ELBO at any point. Rather less fortunately though we run into problems when trying to evaluate the gradient of the ELBO. We notice that samples from q are either m or zero and so p never shows up in the right hand side of expression 4.5, meaning the gradient with respect to p is just zero and we can't optimise them with respect to p, which is the whole point. We also run into problems trying to take the gradient of the ELBO directly using a trick such as black box VI which takes the gradient of q with respect to both m and p. In this the problem is with m as the q is not continuous in m, it's not differentiable.

Note as above we can optimise with respect to m but not with respect to p. In order to overcome these issues, and get a gradient in p, we consider a slight reperameterisation of the weights, writing $W = M * B$ where we have $B \sim Bernoulli(p)$ and M is a constant or considered a random variable with all its density placed on the value m. Given this we use Jensen's inequality to derive an expression for the ELBO, starting with an expression given the distribution over W before getting to one we can more easily handle in B. We use a slight abuse of notation with the integrals but the meaning should be clear:

$$\begin{aligned}\log p(X) &= \log \int_W p(X, W) dW \\ &= \log \int_B \int_M p(X, B, M) dM dB \\ &= \log \int_B \int_M p(X, B, M) \frac{q(B)}{q(B)} dM dB \\ &= \log \int_B p(X, B|M = m) \frac{q(B)}{q(B)} dB \\ &= \log(\mathbb{E}_q\Big[\frac{p(X, B|M = m)}{q(B)}\Big]) \\ &\geq \mathbb{E}_q\Big[\log \frac{p(X, B|M = m)}{q(B)}\Big] \\ &= \mathbb{E}_q\big[\log p(X, B|M = m) - \log q(B)\big] = \mathcal{L}_B(q),\end{aligned}$$

with $\mathcal{L}_B(q)$ the variational lower bound with respect to the variational distribution over B. In order to get the gradient of this with respect to p we employ the black box VI trick to get the gradient as an expectation with respect to the variational distribution:

$$\nabla_p \mathcal{L}_B(q) = \mathbb{E}_q\big[(\nabla_p \log q(B))(\log p(X, B|M = m) - \log q(B))\big], \tag{4.8}$$

which is justified by the dominated convergence theorem [Çınlar, 2011] where all of the expressions in the expectation are calculable given a realisation b of B, in particular $\log q(b) = \log p$ if $b = 1$ and $\log(1 - p)$ if $b = 0$ which are also clearly differentiable in p. We can thus take an MC approximation of this gradient and update p accordingly as in algorithm 4.

Algorithm 4: Bayesian Dropout Optimisation

```
**Result:** Optimised parameters $m, p$
**Require** : $\lambda$ (Step size);
**Require** : $m_0, p_0$ (Starting points);
$t \leftarrow 0$ (Initialise timestep);
**while** $m_t, p_t$ *not converged* **do**
    $t \leftarrow t + 1$;
    Sample $\hat{b} \sim Bernoulli(p_{t-1})$;
    $m_t \leftarrow m_{t-1} - \lambda \nabla_m (\frac{1}{2} \sum_{i=1}^{n} (f(x_i; m_{t-1}\hat{b}) - y_i)^2 + \alpha p_{t-1} ||m_{t-1}||^2)$;
    $p_t \leftarrow p_{t-1} - \lambda (\nabla_p \log q(\hat{b}))(\log p(X, \hat{b}|M = m_{t-1}) - \log q(\hat{b}))$;
**end**
**return** $m_t, p_t$ (Resulting parameters)
```

4.2.5 A Practical Example

Now that we have derived a theoretically justified algorithm to optimise the dropout rate we would like to see how it behaves in practice. Implementing this algorithm on a toy regression problem as in section 4.1 we see some interesting properties for how p converges. The good thing is that it at least does always seem to converge, just not always to the same thing depending on it's starting point. This may not be too surprising - it is well known that local gradient based optimisation methods will find a local and not necessarily global optimum. Indeed, as shown in Fig. 4.4, there appears to be a large catchment area where the dropout rate will converge on a specific value, in this case 0.37. Outside of this though the rate will go to either 1 or 0 depending on which is closer.

The rate shows some interesting behaviour in that whenever it converges to 0.37 it doesn't go there directly, instead it initially moves in the opposite direction before coming back to the centre.

We note that whenever the rate hits either 0 or 1 it's immediately trapped there by the practical implications of what the dropout rate actually does in the network. There may be reason to believe then that if p wasn't constrained to $[0, 1]$

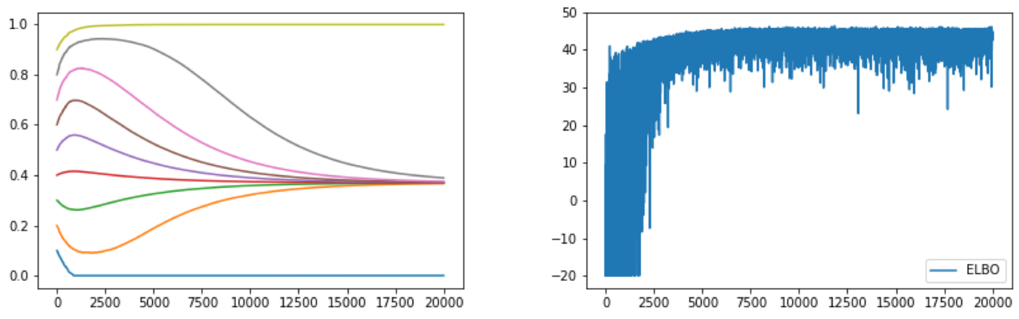


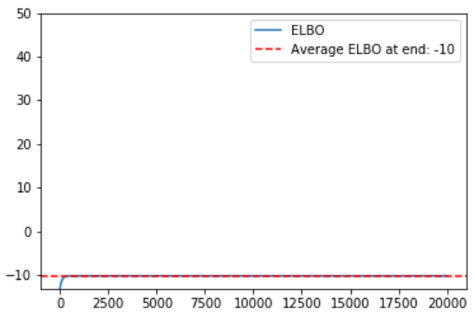
Figure 4.4: Results of a toy regression problem. Left: The change in p during training, each line corresponds to different starting values of p. Right: Example change in the ELBO during training

it might converge back to 0.37 since it looks initially like it's following the general pattern of moving in the opposite direction before coming back to the centre, the problem is it then hits the barrier of either 0 or 1, not letting it change further.

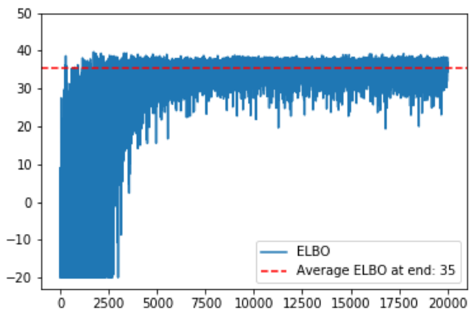
Regardless of whether or not the algebra might make the rate converge back to 0.37 though we now see it's clear that the rate will converge practically to one of 0, 0.37, or 1.

4.2.6 Validation by Grid Search

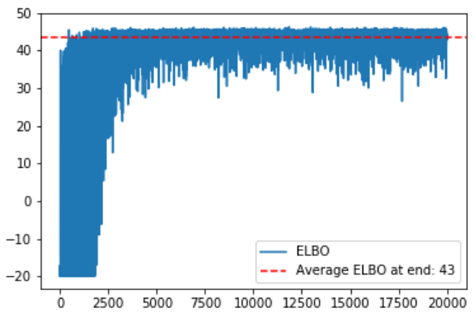
To validate whether these convergent values of the dropout rate are reasonable we also use grid search to evaluate the ELBO at different values of p. We evaluate every 0.1, as well as also looking at 0.37 since that is the value we have found is converged on. Fig. 4.5 shows the resulting plots of the ELBO during training while keeping p fixed at a starting point as well as a plot of the average ELBO achieved for different values of p. Our evaluations of the ELBO are necessarily noisy due to the MC approximations so it is not always clear the exact value of the ELBO being converged on (except fortunately in the case of zero and one where there is no randomness). Thus for $p \in (0, 1)$ we approximate the value of the



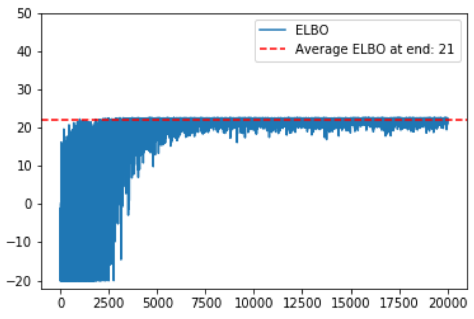
p = 0.00



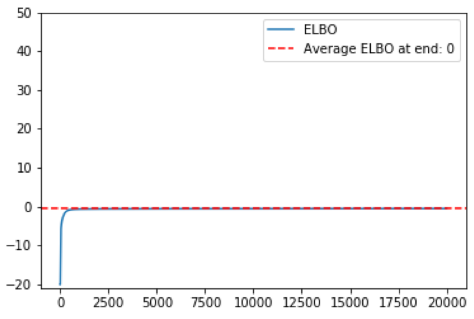
p = 0.20



p = 0.37



p = 0.80



p = 1.00

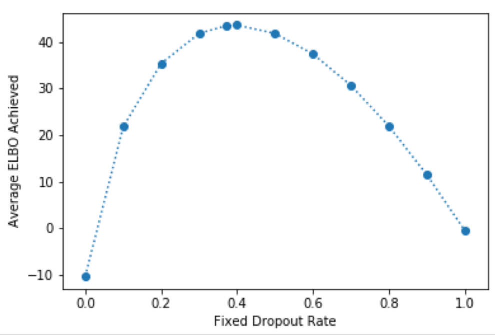


Figure 4.5: Plots of the ELBO during training while the dropout rate is fixed and not optimised. The red dotted line shows the value of the ELBO achieved at convergence by averaging the last 20% of the values. Last plot summarises the average ELBO reached against fixed dropout rate.

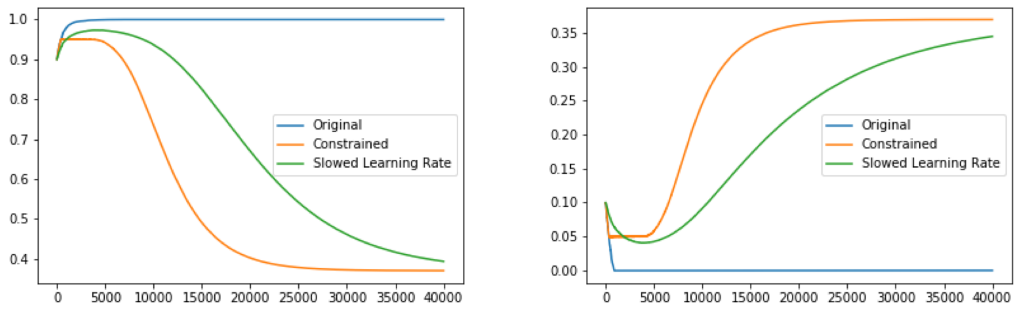


Figure 4.6: Plots of how the dropout rate changes originally, when p is constrained to $[0.05, 0.95]$, and when the learning rate is halved. Left: p starting at 0.9. Right: p starting at 0.1.

ELBO by averaging the last 20% of the evaluations once it appears convergence has been reached.

We see when p is set to 0.37 the ELBO reaches an average of 43.409, this is clearly much higher than when p is 1 with an ELBO of -0.484, and when p is 0 with an ELBO of -10.243. This gives us very clear evidence that values of 0 and 1 are not optimal, the values of the ELBO they achieve are clearly not a maximum. Additionally it verifies that a rate of 0.37 credibly results in the highest value for the ELBO compared to other values in (0,1). We note though that technically in our experiments the rate of 0.4 achieved an average of 43.552, very slightly higher than the 43.409 of 0.37 but it seems perfectly possible this is just due to the inherent noise in the approximation.

4.2.7 Preventing Convergence to Zero or One

It is not very satisfying that the rate sometimes converges to 0 or 1 when we now know them to not be optimal. We suggest that the reason for this is that once the rate hits 0 or 1 the value for $\log q(B)$ will always be zero given a realisation of B and so our estimator for the gradient will also be zero. Thus we might expect that if p never gets to 0 or 1 it will eventually converge on the optimal rate. To test this we try two different things; first we artificially constrain p to be in the closed interval $[0.05, 0.95]$ so that it can't go to 0 or 1, and second we try a slowed learning rate with the hope that the direction will change before we get to 0 or 1. Fig. 4.6 shows what happens when we try this starting at a rate of 0.9 or 0.1. As we've already seen from Fig. 4.4 if we start at 0.9 we normally hit 1 and starting at 0.1 we normally hit 0, but when we use these two techniques we actually do see that the rate does actually begin to converge to the optimum.

Chapter 5

Conclusions

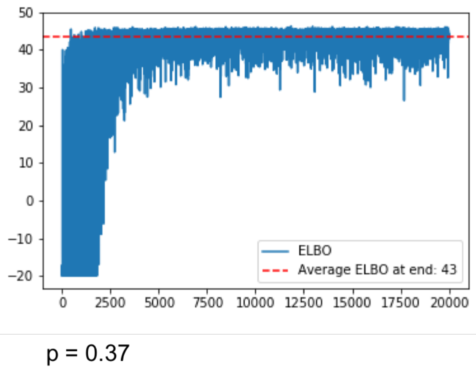
"Enough research will tend to support your conclusions."

Arthur Bloch

5.1 How Should We Use Dropout Now?

We have seen then that when considering dropout in the context of approximate variational inference in a Bayesian fitting of a neural network the dropout rate will converge to a value $\hat{p} \in (0, 1)$ or 0 or 1. Moreover it seems that when it converges to 0 or 1 it is on its way to $\hat{p}$ but is stopped by the boundary conditions on p. We have also shown that 0 and 1 are not optimal from the point of view of the ELBO and so if p is found to converge to one of these it may be worth restarting training with a different value of p so that it goes to $\hat{p}$. Indeed it would seem that best practice would be too always start p at a value of 0.5, as far from 0 and 1 as possible, making it unlikely to converge to them. Alternatively employing a technique from Section 4.2.7 may also work.

These findings challenge the common practice of simply setting p to 0.5, though as can be seen from Fig. 5.1 the difference between the ELBOs reached



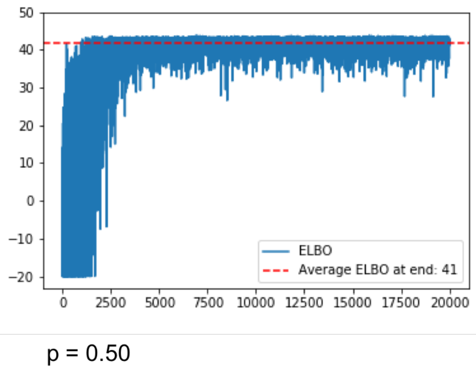


Figure 5.1: Plots of the ELBO during training. Left: p starting at 0.37. Right: p starting at 0.5.

between p at its optimal value and at 0.5 is not very much, especially considering the inherent variance due to the Monte Carlo evaluation. Thus it seems 0.5 would give a reasonable approximation, despite not being optimal, and so it is unsurprising that 0.5 is found to work quite well in practice. While our method is not very computationally expensive, there would still be a saving in not optimising p which could reasonably be traded for the slight decrease in ELBO by setting p to 0.5.

Our work is not entirely conclusive though, for a start this has only been implemented on a couple of different toy regression problems as well as classification on the MNIST handwritten numbers data set (a common neural network benchmark) so we cannot conclusively say that the rate will converge to something other than 0 or 1 in all situations. However that does not detract from our method which would still stand and is applicable in any neural network architecture where dropout can be applied and the likelihood of the data can be calculated. It is also trivially extended to architectures of more than one hidden layer (or indeed applied on the input layer) as the mathematics are independent of the numbering of the layer, also allowing for different dropout rates between layers. It could also be generalised to a different rate per individual neuron but would certainly incur more of a computational cost.

5.2 Further Research

We have made the dropout approximation more flexible by using the rate as another parameter to be optimised. This then naturally leads to other potential ways to improve the flexibility of the distribution such as by not fixing the dropped value to be zero. In other words taking a value a with probability p and a value b with probability $(1-p)$, where in normal dropout $b=0$. Further extending this to maybe a multinomial distribution could work, of course remembering though that the more flexible the distribution the more computationally expensive it becomes and one of the key advantages to dropout is its speed.

Further extensions to dropout have been proposed such as instead of multiplying the weights by $B \sim Bernoulli(p)$ we multiply them by some noise say $Z \sim N(1, \sigma^2)$, understanding this in the Bayesian context could be interesting and seems like it might be closer than traditional dropout.

Of course this is all in the context of trying to fit Bayesian neural networks both quickly and accurately. All the fast algorithms out there at the moment rely on some pretty strict assumptions on the posterior distribution and so more work on increasing their flexibility will be useful.

There are also a number or applications of Bayesian neural networks (and thus our dropout method too) that it would be interesting to explore. For example deep neural networks are used extensively in reinforcement learning, where accurate evaluation of the uncertainty is very useful in order for the agent to decide whether it should explore as opposed to exploit. If the agent has high uncertainty over an action it may be useful to take that action so in the future it has a more clear idea of how good that action is. Thus seeing how dropout can be used in this context may be very useful.

5.3 Summary of Work

Now that we've reached the end of the project all that remains is to sum up all the work that has been done and really detail what I've learned from the experience.

Over the course of the year I've learned and taught myself about machine learning, and specifically deep neural network models. I've familiarised myself with stochastic optimisation methods as well as modern methods in Bayesian computation including Hamiltonian Monte Carlo and variational inference. I've read through the literature on Bayesian deep learning, getting a good view on the modern capabilities and limitations of the field, before looking at a specific problem - that of optimising the dropout rate. Here I took the original interpretation of dropout as a Bayesian approximation and adapted it, making the variational distribution more flexible by incorporating the dropout rate as a parameter of that distribution to be optimised in training. Having derived the mathematics I then implemented this in code.

Throughout the whole project I implemented most of the theoretical techniques I encountered in code (see Appendix A). This not only improved by programming skills but also gave me more of an understanding of the techniques and allowed me to see how they really behave in practice, not just in theory.

This project has been a very useful and interesting exploration of a field I had little knowledge of previously. It's taught me a lot of technical knowledge as well as developed my ability to do independent study and research, and for that I'm very grateful.

Appendix A

Programming Details

A significant part of this project involved programming, implementing the algorithms that were looked at and producing graphics for the report. All code was written in Python, open source package use was limited to only the more general mathematical and scientific programming tool-kits. NumPy for general mathematics, Matplotlib for plotting and Autograd for automatic differentiation. This was for my benefit, I wanted to really know what the code was doing and implementing these algorithms was a good way to test my understanding of them.

If you are interested I have uploaded the Python scripts directly used in the production of this report into a Google drive folder (I was going to use GitHub but that wouldn't have been anonymous) which can be accessed by following this link:

`https://drive.google.com/open?id=1ve9Qh5DzDRC7TL3H7eSR39kjs6cpxItD`

The folder contains the following files, and we detail where they are used:

GaussianProcesses.py : Fig. 2.4

MetHastings.py : Fig. 2.3

CoordinateAscent.py : Fig. 4.3

DropoutSingleLayer.py : Fig. 4.1

DropoutRateOptim.py : Fig. 4.4, Fig. 4.5

BayesByBackprop.py : Fig. 4.1

GravesVI.py : Fig. 4.1

HMCNeuralNet.py : Fig. 4.1

Running the scripts will not always produce exactly the plots used in the report since in some cases scripts were run multiple times with different values to produce multiple plots. They have however been written so they can be run by themselves in isolation to produce example plots. It's worth emphasising these are non-trivial algorithms and it took a significant amount of work to implement and tune this code.

References

Barber, D. (2012). *Bayesian Reasoning and Machine Learning*. Cambridge University Press. 6

Barber, D. & Bishop, C.M. (1998). Ensemble learning in bayesian neural networks. *NATO ASI Series F Computer and Systems Sciences*, **168**, 215–238. 29

Beskos, A., Pillai, N., Roberts, G., Sanz-Serna, J.M., Stuart, A. *et al.* (2013). Optimal tuning of the hybrid monte carlo algorithm. *Bernoulli*, **19**, 1501–1534. 45

Bishop, C.M. *et al.* (1995). *Neural networks for pattern recognition*. Oxford university press. 8

Blei, D.M., Kucukelbir, A. & McAuliffe, J.D. (2017). Variational inference: A review for statisticians. *Journal of the American Statistical Association*, **112**, 859–877. 22

Blundell, C., Cornebise, J., Kavukcuoglu, K. & Wierstra, D. (2015). Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424*. 33, 34

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, 177–186, Springer. 13

Brooks, S., Gelman, A., Jones, G. & Meng, X.L. (2011). *Handbook of Markov Chain Monte Carlo*. CRC press. 20

Çinlar, E. (2011). *Probability and stochastics*, vol. 261. Springer Science & Business Media. 53

Duane, S., Kennedy, A., J. Pendleton, B. & Roweth, D. (1988). Hybrid monte carlo. *Phys. Lett. B*, **195**, 216. 20

Evans, R. *et al.* (2018). De novo structure prediction with deep-learning based scoring. *Thirteenth Critical Assessment of Techniques for Protein Structure Prediction (Abstracts).* 8

Gal, Y. (2016). *Uncertainty in Deep Learning*. Ph.D. thesis, University of Cambridge.

Gal, Y. & Ghahramani, Z. (2016). Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *International Conference on Machine Learning*, 1050–1059. 36, 47, 51

Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep Learning*. MIT Press, `http://www.deeplearningbook.org`. 1

Graves, A. (2011). Practical variational inference for neural networks. *Advances in Neural Information Processing Systems*, 2348–2356. 31, 33

Hernández-Lobato, J.M. & Adams, R. (2015). Probabilistic backpropagation for scalable learning of bayesian neural networks. *International Conference on Machine Learning*, 1861–1869. 34

Hinton, G.E. & van Camp, D. (1993). Keeping neural networks simple. *ICANN93*, 11–18. 29, 31

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, **4**, 251–257. 8

Kingma, D.P. & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. 14

Kingma, D.P. & Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*. 39, 40

Kononenko, I. (2001). Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in medicine*, **23**, 89–109. 6

Litjens, G., Kooi, T., Bejnordi, B.E., Setio, A.A.A., Ciompi, F., Ghafoorian, M., Van Der Laak, J.A., Van Ginneken, B. & Sánchez, C.I. (2017). A survey on deep learning in medical image analysis. *Medical image analysis*, **42**, 60–88. 1

MacKay, D.J. (1992). A practical bayesian framework for backpropagation networks. *Neural computation*, **4**, 448–472. 27, 28, 31

Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H. & Teller, E. (1953). Equation of state calculations by fast computing machines. *The journal of chemical physics*, **21**, 1087–1092. 18

Neal, R. (1994). *Bayesian Learning for Neural Networks*. Ph.D. thesis, University of Toronto. 30, 31

Ranganath, R., Gerrish, S. & Blei, D. (2014). Black box variational inference. In *Artificial Intelligence and Statistics*, 814–822. 38, 39

Rasmussen, C.E. (2003). Gaussian processes in machine learning. In *Summer School on Machine Learning*, 63–71, Springer. 23

Rezende, D.J. & Mohamed, S. (2015). Variational inference with normalizing flows. *arXiv preprint arXiv:1505.05770*. 40, 41

Rissanen, J. (1986). Stochastic complexity and modeling. *The annals of statistics*, 1080–1100. 29

Robbins, H. & Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*, **22**, 400–407. 15, 37

Rosenblatt, F. (1962). *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Report (Cornell Aeronautical Laboratory), Spartan Books. 9

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*. 14

Rumelhart, D.E., Hinton, G.E. & Williams, R.J. (1986). Learning representations by back-propagating errors. *nature*, **323**, 533. 11

Salimans, T., Kingma, D.P., Welling, M. *et al.* (2015). Markov chain monte carlo and variational inference: Bridging the gap. In *ICML*, vol. 37, 1218–1226.

Scholkopf, B. & Smola, A.J. (2001). *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press. 31

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T. *et al.* (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*. 8

SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A., SUTSKEVER, I. & SALAKHUTDINOV, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, **15**, 1929–1958. 35, 36, 46

WELLING, M. & TEH, Y.W. (2011). Bayesian learning via stochastic gradient langevin dynamics. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 681–688. 37, 38, 41

ZHANG, G.P. (2000). Neural networks for classification: a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, **30**, 451–462. 7