Bayesian Optimisation using Neural Networks

by

Devang Agrawal (Queens')

Fourth-year undergraduate project in
Group F-, 2015/2016

I hereby declare that, except where specifically indicated, the work submitted herein is my own original work.

Signed: _____ Date: _____

Technical Abstract

Bayesian Optimisation has been used for a myriad applications

Contents

1 Introduction

Traditionally machine learning algorithms have focussed on using a set of expertly curated features of inputs to make predictions. However in various tasks, especially in computer vision and natural language processing, the design of these features can be very complex. In many situations it can be difficult to know what features should be extracted. Neural networks excel at such tasks as they can discover not only the mapping from the representation to the output but the representation itself. This approach is called *representation learning.* Learned representation often result in much better performace than can be obtained with hand-designed representations. They also allow machine learning systems to rapidly adapt to new tasks, with minimal human intervention. A representation learning algorithm can discover a good set of features for a complex task in hours to months. Manually designing features for a complex task can require a great deal of human time and effort; it can possibly take decades for an entire community of researchers.

Such representational learning capabilities have allowed neural networks to reach state of the art performance in several applications including computer vision and natural language processing. They have been used get to human-level performance in several video games[28] and have been used in systems to master the game of "Go"[36].

Most neural network applications use maximum likelihood estimates for the parameters of the neural network, such estimates can be found by minimising an error function such as the sum of squared errors between the target and predictions. It is however possible to estimate the posterior distribution on the parameters of the neural networks by using a we

In many applications we are tasked with the zeroth order optimization of an expensive to evaluate function f. Some examples are hyper parameter tuning in expensive machine learning algorithms[37], experiment design[20], optimizing control strategies in complex systems, and scientific simulation based studies. Bayesian optimization finds the optimum of f is found by using as few queries as possible by managing exploration and exploitation[29]

Neural networks have found wide ranging applications in computer vision, natural language processing etc. They are particularly adept at tasks where it is not clear how to extract relevant features

Bayesian Optimisation is a very interesting problem. It has been used in .. . Neural networks can be used for dimensionality reduction and can be used for feature selection. It is not trivial how to build probabilistic models using Neural Networks. Several different options are available and two particular ones were explored in this report. Using a pragmatic approach .... . Alternatively a full Bayesian Neural network was used .... . These functions can be very useful for non-stationary functions.

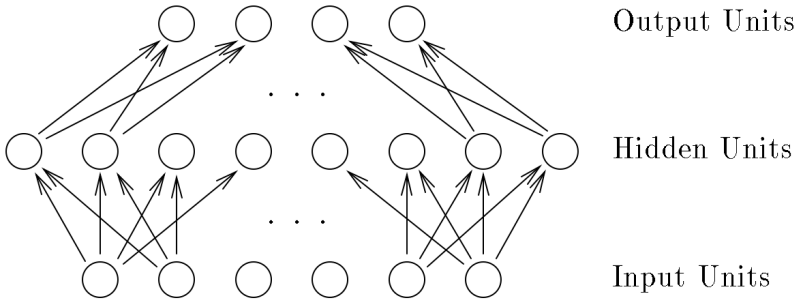


Figure 1: An illustration of a neural network with one hidden layer. The input units at the bottom are fixed to their value depending on the data point. The values of the hidden units are then computed, followed by the values of the output units. The value for a hidden or output unit is a function of the weighted sum of values it receives from the units that are connected to it via the arrows. More hidden layers can be trivially added between the existing hidden units and the output units[32].

2 Neural Networks

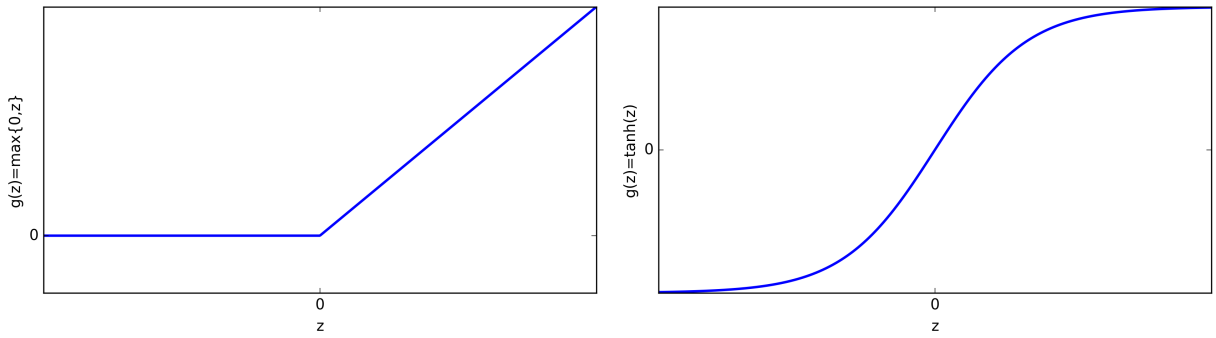
Neural networks use multiple non-linear transforms to map from an input to an output. They are know to have excellent feature selection properties and define the current state of the art in object recognition and natural language processing tasks[5]. For this project we primarily deal with "*feedforward*" networks, these networks take in a set of real inputs, x_i, and then compute one or more output values, $f_k(\boldsymbol{x})$, using some number of layers of *hidden* units. Here, we illustrate the basic concepts of neural networks by considering a simple network with one hidden layer such as the one shown in Figure 1 . The outputs can be calculated as follows:

$$f_k(\boldsymbol{x}) = b_k + \sum_j v_{jk} h_j(x) \tag{2.1}$$

$$h_j(\boldsymbol{x}) = \tanh(a_j + \sum_i u_{ij} x_i) \tag{2.2}$$

Here, u_{ij} is the *weight* on the connection from the input unit i to the hidden unit j; similarly, v_{jk} , is the weight on the connection from the hidden unit j to the output unit k. The a_j and b_k are the *biases* of the hidden units and the output units respectively. These weights and networks collectively constitute the set of parameters of the network.

Each output value, $f_k(\boldsymbol{x})$, is just a weighted sum of the last hidden unit values, plus a bias. Each hidden unit computes a similar weighted sum of input values, and then passes it though a non-linear *activation function.* In this project we choose the hyperbolic tangent (tanh) as the activation function, it is an asymmetric function of sigmoidal shape as shown in Figure 2b. Its value is close to -1 for large negative numbers,$+1$ for large



(a) The Rectified Linear Activation function (b) The Hyperbolic Tangent Activation function

Figure 2: Here we show the rectified linear(ReLU) and the hyperbolic tangent(tanh) activation function. The ReLU function is very popular with deep feedforward networks. Being very close to linear they preserve many properties that make linear models easy to optimize with gradient based methods. However, they are not smooth, the abrupt changes in gradients associated with them can lead to issues when building probabilistic models[38]. However, tanh functions are smooth over their entire domain and are hence suited for building probabilistic models.

positive numbers and zero for zero argument. Use of a non-linear activation function allows the overall function defined by the neural network to be non-linear. It allows the hidden units to potentially represent "hidden features" in the input that can be useful in computing the appropriate outputs.

It has been shown that such a network with one hidden layer can approximate any function defined on a compact domain arbitrarily closely, if enough hidden units are used[19, 9, 15]. Nevertheless, more elaborate network architectures have advantages and are often used. Multiple hidden layers can be easily stacked between the current hidden units and the output units. Such "deep" networks are commonly used for a wide variety of machine learning applications. However it should be noted that in feedforward networks, connections are not allowed to form cycles. This is important to allow the value of the output to be computed in a single forward pass, in time proportional to the number of network parameters.

2.1 Training neural networks

We can define probabilistic models for regression tasks by using network outputs to define the conditional distribution for the targets , $\boldsymbol{y}$, for various values of the input vector $\boldsymbol{x}$. For a regression model with real-valued targets, y_k the conditional distribution of the targets might be defined to be a Gaussian, with y_k with the mean $f_k(\boldsymbol{x})$ and a standard deviation of σ_k. We often assume the different outputs to be independent, given the input. This gives:

$$P(\boldsymbol{y}|\boldsymbol{x}) \;=\; \prod_k \frac{1}{\sqrt{2\pi\sigma_k^2}} \exp(-\frac{(f_k(\boldsymbol{x}) - y_k)^2}{2\sigma^2}) \tag{2.3}$$

The "noise level", σ_k, might be fixed or treated as a hyperparameter. The weights and biases of the neural network can be learnt on a set of *training cases,* $\mathbf{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$, giving independent examples of inputs, $\boldsymbol{x}^{(i)}$, and associated targets $\boldsymbol{y}^{(i)}$. Standard neural network training procedures adjust the weights and biases in the network so as to minimize a measure of "error" on the training cases, most commonly, the sum of the squared differences between the network outputs and targets. Minimization of this error measure is equivalent to maximum likelihood estimation of the Gaussian noise model of equation 2.3, since negative log likelihood with this model is proportional to the sum of squared errors.

A gradient-based approach is commonly used to find the weights and biases that minimize the chosen error function. The derivatives of the error with respect to the weights and biases can be calculated by using the *backpropagation* algorithm[5, 35]. Typically many local minima are present, but good solutions are often found despite this.

Neural networks have a large number of parameters which makes overfitting a significant problem when training neural networks. To reduce overfitting a penalty term proportional to the sum of the squares of the weights and biases is often added to the error function. This modification is known as *weight decay,* since its effect is to bias the training procedure in favour of small weights. Determining the proper magnitude of weight decay is often difficult – too little weight decay, the network may "overfit", but too much weight decay, the network will "underfit" and ignore the data. Adding weight decay can be interpreted as having a Gaussian prior with mean zero and some standard deviation, σ_w, on the weights and biases of the neural network.

Cross validation is often used to find an appropriate weight penalty[40]. In its simplest form, the amount of weight decay is chosen to optimise performance on a validation set separate from the cases used to estimate the network parameters.

In the Bayesian approach to neural network learning, the objective is to find the predictive distribution for the target values in new "test" cases, given the input for that case, and the inputs and targets in the training cases($\boldsymbol{D}$). The predictive distribution is given by:

$$P(y^{(n+1)}|x^{(n+1)}, \boldsymbol{D}) = \int P(y^{(n+1)}|x^{(n+1)}, \boldsymbol{\theta}) P(\boldsymbol{\theta}, |\mathbf{D}) d\theta \tag{2.4}$$

Here, θ represents the network parameters (weights and biases). The posterior density for these parameters is proportional to the product of the prior on them and the likelihood

function given by:

$$L(\boldsymbol{\theta}|\mathbf{D}) = \prod_{c=1}^{n} P(y^{(c)}|x^{(c)}, \boldsymbol{\theta}) \tag{2.5}$$

The distribution of the target values, $\boldsymbol{y}^{(i)}$, given the corresponding inputs, $\boldsymbol{x}^{(i)}$, and the parameters of the network is defined by the type of model with which the network is being used. For the regression model, it is given by equation 2.3.

3 Markov Chain Monte Carlo methods

Markov chain Monte Carlo(MCMC) methods are a class of algorithms for sampling from a desired probability distribution[2, 32]. A Markov chain is constructed with the desired distribution as its equilibrium distribution. These models are widely applied to Bayesian models in statistics. MCMC methods make no assumptions about the about the form of the underlying distribution. In theory they can take account of multiple modes and the possibility that the main contribution to the integral may come from areas not in the vicinity of any mode. In practise though it can under certain circumstances they can take a very long time to converge to the desired distribution. This is the main disadvantage of using MCMC methods.

In Bayesian learning we often encounter situations where we need to evaluate the expectation of a function $f(\theta)$ with respect to the posterior probability density of the model parameters. Writing the posterior as $Q(\theta)$, the expectation can be expressed as

$$E[f] \;=\; \int f(\theta)Q(\theta)d\theta \tag{3.1}$$

Such expectations can be estimated by the *Monte Carlo* method, using a sample of values from Q:

$$E[f] \;\approx\; \frac{1}{N}\sum_{t=1}^{N} f(\theta^{(t)}) \tag{3.2}$$

where $\theta^{(1)}, \ldots, \theta^{(N)}$ are generated by a process that results in each of them having the distribution defined by Q. In simple Monte Carlo methods, the $\theta^{(t)}$ are independent, however when Q is a complicated distribution , generating such independent values is often infeasible. However it is often possible to generate a series of dependent values. The Monte Carlo integration formula of equation (3.2) still gives an unbiased estimate of $E[f]$ even when the $\theta^{(t)}$ are dependent as long as the independence is not too great[32]. The estimate will still converge to the true value as N increases.

Such a series of dependent values can be generated using a *Markov Chain* that has Q

as its stationary distribution. The chain can be defined by giving an *initial distribution* for the first state of the chain, $\theta^{(1)}$, and a set of *transition probabilities* for a new state, $\theta^{(t+1)}$, to follow the current state, $\theta^{(t)}$. The probability densities for these transitions can be written as $T(\theta^{(t+1)}|\theta^{(t)})$. A *stationary (or invariant)* distribution, Q, is one that persists once it is established. This means that if $\theta^{(t)}$ has the distribution Q, then $\theta^{(t')}$ will have the same distribution for all $t' > t$. This invariance condition can be written as follows:

$$Q(\theta') = \int T(\theta'|\theta)Q(\theta)d\theta \tag{3.3}$$

The invariance with respect to Q is implied by the stronger *detailed balance* condition – that for all θ and θ':

$$T(\theta'|\theta)Q(\theta) = T(\theta|\theta')Q(\theta') \tag{3.4}$$

A chain satisfying detailed balance is said to be *reversible.*

An *ergodic* Markov chain has a unique invariant equilibrium distribution, to which it converges from any initial state. If we can find a Markov Chain that has Q as its equilibrium distribution, then we can find find expectations with respect to Q by using equation (3.2) . In this case $\theta^{(1)}, \ldots, \theta^{(N)}$ are the states of the chain, some of the early states can be discarded since they are not representative of the equilibrium distribution.

To use MCMC methods to estimate an expectation with respect to some distribution Q, we need to construct an ergodic Markov chain with Q as the equilibrium distribution. The chain should rapidly converge to this distribution and the states visited once equilibrium is reached should not be highly dependent.

3.1 The Metropolis algorithm

The Metropolis algorithm was introduced by Metropolis et.al in their classic paper in 1953[26] . In the Markov chain defined by the metropolis algorithm, a new state, $\theta^{(t+1)}$, is generated from the previous state, $\theta^{(t)}$, by first generating a *candidate state* using a specified *proposal distribution,* and then deciding whether or not to accept that state based on its probability density relative to the old state, with respect to the desired invariant distribution, Q. If accepted, the candidate state, becomes the next state of the Markov chain; if it is instead rejected, the new state remains the same as the old state.

In detail, the transition from $\theta^{(t)}$ to $\theta^{(t+1)}$ is defined as follows:

1. Generate a candidate state, θ^*, from a proposal distribution. The proposal distribution may depend on the current state, its density is by $S(\theta^* \,|\, \theta^{(t)})$. It is noted that the proposal distribution must be symmetrical, satisfying the condition $S(\theta' \,|\, \theta) = S(\theta \,|\, \theta')$.

2. If $Q(\theta^*) \geq Q(\theta^t)$, accept the candidate state; otherwise accept the candidate state with probability $Q(\theta')/Q(\theta)$. For $\theta' \neq \theta$, the overall transition probability is then given by:

$$T(\theta'|\theta) = S(\theta' \mid \theta) min(1, Q(\theta')/Q(\theta)) \tag{3.5}$$

3. If the candidate state is accepted, let $\theta^{t+1} = \theta^*$. However if it was not accepted, then set $\theta^{(t+1)} = \theta^{(t)}$.

Following the overall transition probability in equation (3.5), the detailed balance condition (3.4) can be verified as follows:

$$\begin{aligned} T(\theta'|\theta)Q(\theta) &= S(\theta' \mid \theta) min(1, Q(\theta')/Q(\theta))Q(\theta) \\ &= S(\theta' \mid \theta) min(Q(\theta), Q(\theta')) \\ &= S(\theta \mid \theta') min(Q(\theta'), Q(\theta)) \\ &= S(\theta \mid \theta') min(1, Q(\theta)/Q(\theta'))Q(\theta') \\ &= T(\theta|\theta')Q(\theta') \end{aligned}$$

Therefore the Metropolis updates leave Q invariant.

Many choices are available for the proposal distribution of the Metropolis algorithm. One simple and popular choice is a Gaussian distribution centred on $\theta^{(t)}$,with the variance chosen so that the probability of the candidate being accepted is reasonably high. Very low acceptance rates can be bad as they lead to successive samples being highly dependent. When sampling from a complex, high-dimensional distribution, the standard deviation of of the proposal distributions typically has to be very small to compared to the extent of Q. This is because large changes will almost certainly lead to regions of low probability. A large number of steps will be required to move to a distant point in the distribution. The problem is made worse by the fact that these movements will take the form of a random walk, rather than a systematic traversal.

Simple forms of the Metropolis algorithm can be very slow when applied to problems such as Bayesian learning for neural networks[32]. As will be seen in section (3.2) , this problem can be alleviated by using the hybrid Monte Carlo algorithm[12], in which candidate states are generated by a dynamical method which can avoid the random walk aspect of the exploration.

3.2 The hybrid Monte Carlo algorithm

The hybrid Monte Carlo algorithm was originally developed by Duane et.al. for application in quantum chromodynamics[12]. Radford Neal in his book on Bayesian Learning for Neural Networks[32]successfully applies this technique to Bayesian learning. The al-

gorithm merges the Metropolis algorithm (reviewed in sec 3.1) with sampling techniques based on dynamical simulation. It generates a sample of points drawn from some specified distribution which can then be used to form Monte Carlo estimates for the expectations of various functions with respect to this distribution. For Bayesian learning, we often wish to sample from the posterior distribution given the training data, and are interested in estimating the expectations needed to make predictions for test cases.

The hybrid Monte Carlo algorithm is expressed in terms of sampling from the *canonical* (or *Boltzmann*) distribution for the state of a physical system, which is defined in terms of an energy function. The algorithm can be used to sample from any distribution for a set of real-valued variables for which the derivatives of the probability density can be computed. For describing the formulation of this algorithm it is convenient to retain the physical terminology even in non-physical contexts. The problem can then be described in terms of an energy function for a fictitious physical system.

Accordingly, suppose we wish to sample from some distribution for a "position" variable, $\mathbf{q}$, which has n real-valued components, q_i. When used for Bayesian neural networks in section (5) , $\mathbf{q}$ will be the set of $n$network parameters. The probability density for this variable under the canonical distribution is defined by :

$$P(\boldsymbol{q}) \propto \exp(-E(\mathbf{q})) \tag{3.6}$$

where $E(\mathbf{q})$ is the "potential energy" function. Any probability density which is not zero at any point can be put in this form, by simply defining $E(\mathbf{q}) = -\log P(\mathbf{q}) - \log Z$, for any convenient Z.

To allow the use of Hamiltonian dynamics, we can introduce a "momentum" variable, $\mathbf{p}$. The canonical distribution over the "phase space" of $\boldsymbol{q}$ and $\boldsymbol{p}$together is then defined to be

$$P(\mathbf{q}, \boldsymbol{p}) \propto \exp(-H(\mathbf{q}, \boldsymbol{p})) \tag{3.7}$$

where $H(\mathbf{q}, \boldsymbol{p}) = E(\mathbf{q}) + K(\mathbf{p})$ is the "Hamiltonian" function, which gives the total energy. $K(\mathbf{p})$ is the "kinetic energy" due to the momentum, for which the usual choice is :

$$K(\mathbf{p}) = \sum_{i=1}^{n} \frac{p_i^2}{2m_i} \tag{3.8}$$

The m_i are the "masses" associated with each component. It is possible to adjust the masses for each component to improve efficiency, however for the rest of the report and project we take all of them to be one.

Since in the distribution of equation (3.7), $\boldsymbol{q}$ and $\boldsymbol{p}$ are independent, the marginal distribution of $\mathbf{q}$ is the same as that of equation (3.6), from which we intend to sample. We can proceed by defining a Markov chain that converges to the canonical distribution

for $\boldsymbol{q}$ and $\boldsymbol{p}$. The values of $\boldsymbol{p}$ can then be simply ignored when estimating expectations of functions of $\boldsymbol{q}$.

The overall dynamics of the system with fixed total energy can be expressed by the following equations:

$$\frac{dq_i}{d\tau} = +\frac{\partial H}{\partial p_i} = \frac{p_i}{q_i} \tag{3.9}$$

$$dq_i = -\frac{\partial H}{\partial q_i} = -\frac{\partial E}{\partial q_i} \tag{3.10}$$

where τ, is the fictitious time in which the state evolves. To be able to run these dynamics, we must be able to compute the partial derivatives of E with respect to the the q_i. Radford Neal in his book[32] shows that the transformation preserves volume and is reversible. Therefore the transformation can be used as the transition operator for a Markov chain and will leave $P(\boldsymbol{q})$ invariant. Evolution under the Hamiltonian dynamics will not sample ergodically from $P(\boldsymbol{q}, \boldsymbol{p})$ because the value of the total energy H is constant. Regions with different values of H are never visited. To avoid this, HMC alternates between performing deterministic dynamical transitions and stochastic Gibbs sampling updates of the momentum. Since $\boldsymbol{q}$ and $\boldsymbol{p}$ are independent , $\boldsymbol{p}$ may be updated without reference to $\boldsymbol{q}$. For the kinetic energy function of equation (3.8), this is easily done, each of the p_i have independent Gaussian distributions which are trivial to sample from. These updates of $\boldsymbol{p}$ change the total energy H and hence allow the entire phase space to be explored.

The length in fictitious time of the trajectories is an adjustable parameter. It is generally better to use trajectories that result in large changes to $\boldsymbol{q}$ to avoid the random walk like effects that would result from randomising the momentum after every short trajectory.

In practice, Hamiltonian dynamics can not be simulated exactly, but can only be approximated by some discretization using finite time steps. This will necessarily introduce numerical errors hence we need a scheme that minimizes the impact of these errors. In the *leapfrog* discretization, a single step finds the approximations to the position and momentum, $(\boldsymbol{q^*}, \boldsymbol{p^*})$ at time $\tau + \epsilon$ from $(\boldsymbol{q}, \boldsymbol{p})$ at time τ as follows:

$$p_i(\tau + \frac{\epsilon}{2}) \;=\; p_i(\tau) \;-\; \frac{\epsilon}{2}\frac{\partial E}{\partial q_i}(\mathbf{q}(\tau)) \tag{3.11}$$

$$q_i(\tau + \epsilon) \;=\; q_i(\tau) \;+\; \epsilon\frac{p_i(\tau + \frac{\epsilon}{2})}{mi} \tag{3.12}$$

$$p_i(\tau + \epsilon) \;=\; p_i(\tau + \frac{\epsilon}{2}) \;-\; \frac{\epsilon}{2}\frac{\partial E}{\partial q_i}(\mathbf{q}(\tau + \epsilon)) \tag{3.13}$$

The leapfrog step consists of a half-step for the p_i , followed by a full step for q_i, and

another half-step for the p_i. To follow the dynamics for some period of time, $\Delta\tau$, a value of ϵ is chosen which is small enough to give an acceptable error, and equations (3.11) - (3.13) are applied for $L = \Delta\tau/\epsilon$ steps in order to reach the target time. The momentum variable is negated at this point to ensure that if we were to perform a dynamical transition starting at the end stage, we will get back to the initial stage.

In the *leapfrog* discretization scheme, the phase space volume remains preserved despite the discretization. The dynamics are also easily reversible. However, the value of H no longer remains exactly constant, this can introduce bias in the simulation.

HMC cancels these effects exactly by adding a Metropolis accept/reject stage after L leapfrog steps. If the original state is $(\boldsymbol{q}, \boldsymbol{p})$, and we get to the new state $(\boldsymbol{q^*}, \boldsymbol{p^*})$ after L leapfrog steps, the new state is then treated as a proposal state and is accepted with probability:

$$min(1, \frac{\exp(-H(\boldsymbol{q^*}, \boldsymbol{p^*}))}{\exp(-H(\boldsymbol{q}, \boldsymbol{p}))} \tag{3.14}$$

In detail , given the number of of leapfrog steps, L, a dynamical transition is performed as follows:

1. Start from the current state, $(\boldsymbol{q}, \boldsymbol{p}) = (\hat{\boldsymbol{q}}(0), \hat{\boldsymbol{p}}(0))$. Perform L leapfrog steps with a step-size of ϵ, resulting in the state $(\hat{\boldsymbol{q}}(\epsilon L), \hat{\boldsymbol{p}}(\epsilon L))$

2. Negate the momentum variables, to produce the proposal state $(\boldsymbol{q^*}, \boldsymbol{p^*}) = (\hat{\boldsymbol{q}}(\epsilon L), -\hat{\boldsymbol{p}}(\epsilon L))$.

3. Accept $(\boldsymbol{q^*}, \boldsymbol{p^*})$ as the new state with probability:

$$min(1, \frac{\exp(-H(\boldsymbol{q^*}, \boldsymbol{p^*}))}{\exp(-H(\boldsymbol{q}, \boldsymbol{p}))}$$

   otherwise let the new state be the same as the old.

It is important to maintain satisfactory acceptance rates by tuning the step-sizes ϵ and the number of leapfrog steps L. A simple adaptive version of HMC is used in this report as implemented in the code accompanying [34]. We track the average acceptance rate of the HMC move proposals, using an exponential moving average. If the average acceptance larger than a target acceptance rate, we can increase the step-size ,ϵ, to improve the mixing rate of the Markov chain. If the acceptance rate is too low, the step-size can be decreased to improve the acceptance rate but yielding a more conservative mixing rate.

4 Bayesian Optimisation

Bayesian optimisation is a natural framework for model-based global optimisation of noisy expensive black-box functions. It relies on querying a distribution over functions defined

by a relatively cheap surrogate model. An accurate model for the distribution over functions is critical to the effectiveness of the approach, and is typically fit using Gaussian processes(GPs). In this work, we will also explore the use of neural-networks as an alternative to GPs to model the distribution over functions.

In this work, we will restrict to using Bayesian optimisation to solve global minimisation problems of the form :

$$\boldsymbol{x}^* = \underset{x \in \chi}{\operatorname{argmin}} f(\boldsymbol{x})$$

where we take χ to be a compact subset of $\mathbb{R}^D$. Maximisation problems can be trivially converted to minimisation problems by negating the objective function.

Bayesian optimisation constructs a probabilistic model for $f(\boldsymbol{x})$, and then exploits this model to make decisions about where in χ to next evaluate the function. The essential philosophy is to use all the information available from previous evaluations of $f(\boldsymbol{x})$ and not simply rely on the local gradient and Hessian approximations. This results in a procedure that can optimize difficult non-convex functions with relatively few evaluations, at the cost of performing more computation to determine the next point to try. When function evaluations are expensive to perform, such as in the case where real experiments need to performed, then it is easy to justify some extra computation to smartly determine what evaluations to make.

Acquisition functions are very important in Bayesian optimisation, they allow us to construct a utility function from the model posterior. This allows us to determine the next point to evaluate.

Figure 3 shows the Bayesian optimisation algorithm on a toy 1D maximisation problem. The algorithm successfully balances exploration and exploitation based on the model to find the maximum in very few function evaluations.

Algorithm 1 Bayesian Optimisation

1: **for** t=1,2,... **do**
2: Find x_t by optimising the acquisition function a : $\boldsymbol{x}_t = \operatorname{argmin}_x a(\boldsymbol{x})$
3: Augment the dataset $\boldsymbol{D}_t = \{\boldsymbol{D}_{t-1}, ((\boldsymbol{x}_t, f(\boldsymbol{x}_t)\}$
4: **end for**

4.1 Acquisition Functions for Bayesian Optimisation

We assume a prior on the function $f(\boldsymbol{x})$. We assume that our observations are of the form $\boldsymbol{D} = \{\boldsymbol{x}_n, y_n\}_{n=1}^N$, where $y_n \sim \mathcal{N}(f(x_n), \nu)$and ν is the variance of the noise introduced into the function observations. The prior and the observations induce a posterior over the functions. The acquisition function denoted by $a : \chi \to \mathbb{R}^+$, determines what point in χ to evaluate next via a proxy optimization $\boldsymbol{x}_{next} = \operatorname{argmin}_x a(\boldsymbol{x})$. Several different

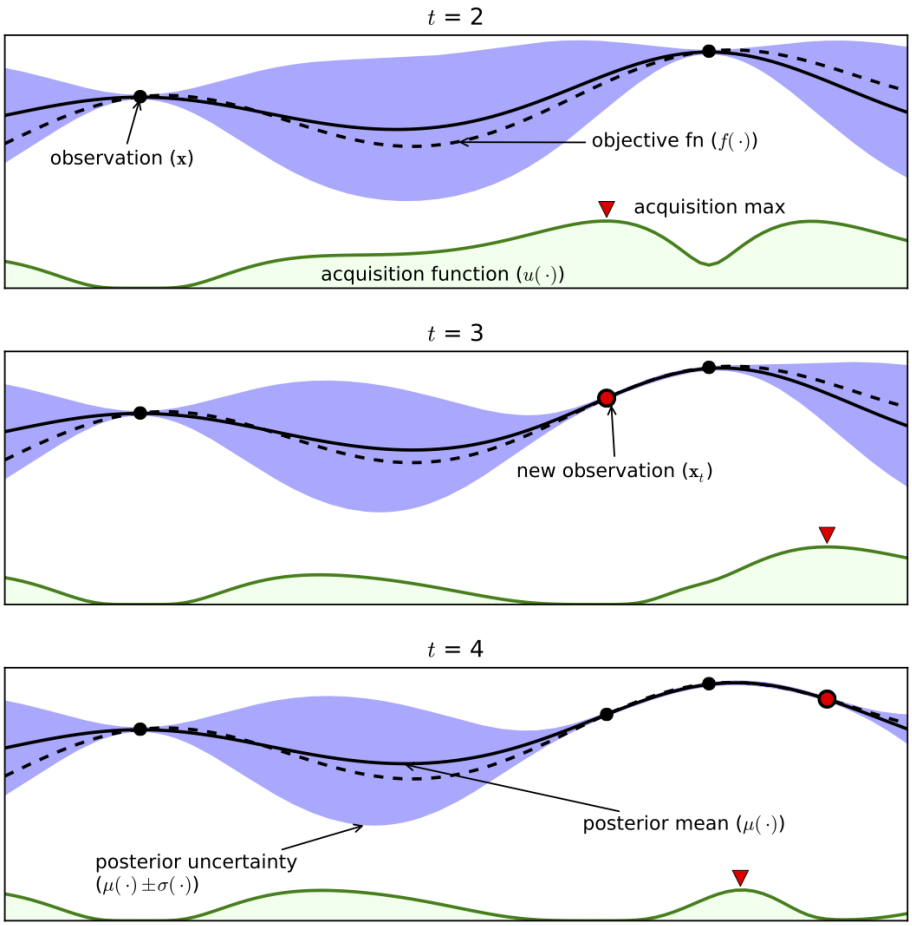


Figure 3: An example of using Bayesian Optimisation on a toy 1D maximization problem[8]. The figures show a Gaussian process(GP) approximation of the objective function over four iterations of sampled values of the objective function. The figure also shows the acquisition function in the lower shaded plots. Since its a maximisation problem, the acquisition function is high where the GP predicts a high objective (exploitation) and and where the prediction uncertainty is high(exploration). We successively evaluate the function at the maximum values of the acquisition function to get close to the true maximum at $t = 4$.

acquisition functions have been proposed[18, 8], for this project we use the GP upper confidence bound.

GP Upper Confidence Bound(GP-UCB) It exploits the idea of exploiting lower confidence bounds (upper, when considering maximisation) to construct acquisition functions that minimize regret over the course of their optimization[39]. These acquisition functions have the form

$$a_{LCB}(x, \boldsymbol{D}, \theta) = \mu(\boldsymbol{x};\, \boldsymbol{D}, \theta) - \kappa\sigma(\boldsymbol{x};\, \boldsymbol{D}, \theta)$$

where θ represents the hyperparameters of the surrogate model being used (GP or Bayesian neural network) , and κ is a tunable parameter to balance exploitation against exploration.

This form of an acquisition function is very well suited for the project since it provides a very convenient way to explicitly balance exploration and exploitation via the tunable parameter, κ . It is also very easy to evaluate for Bayesian neural networks using samples from the posterior of the neural network using an MCMC based approach.

4.2 High Dimensional Bayesian Optimisation

Bayesian optimisation with neural networks can also potentially deal with high dimensional global optimisation problems. Through multiple non-linear transforms neural networks can potentially find a non-linear lower dimensional manifold in a high dimensional spaces which encapsulates most of the variance of the objective function. A neural network based model may be able to take advantage of such structure for Bayesian optimisation. Running Bayesian optimisation on a high-dimensional problem is however very computationally expensive, requiring highly optimized and parallelized code. The efficacy of a Bayesian optimisation framework utilizing neural networks on a high dimensional problem, will be addressed in future work.

Currently there are two known methods for performing Bayesian optimisation on high dimensional problems, we briefly review them here.

4.2.1 High dimensional Bayesian Optimisation via Random Embeddings

Random Embeddings Bayesian Optimisation (REMBO) has been developed by Ziyu Wang et.al.[41] for performing Bayesian optimisation on functions with low effective dimensionality.

The effective dimensionality of a function (d_e) can be defined as :

Definition 1: A function $f : \mathbb{R}^D \to \mathbb{R}$ is said to have ***effective dimensionality*** d_e , with $d_e < D$, if there exists a linear sub-space $\mathcal{T}$ of dimension d_e such that for all $x_T \in \mathcal{T} \subset \mathbb{R}^D$ and $\boldsymbol{x}_\perp \in \mathcal{T}^\perp \subset \mathbb{R}^D$, we have $f(x) = f(x_T + x_\perp) = f(x_T)$, where $\mathcal{T}^\perp$ denotes the orthogonal complement of $\mathcal{T}$. $\mathcal{T}$ is the **effective subspace** of f and $\mathcal{T}^\perp$the **constant subspace.**

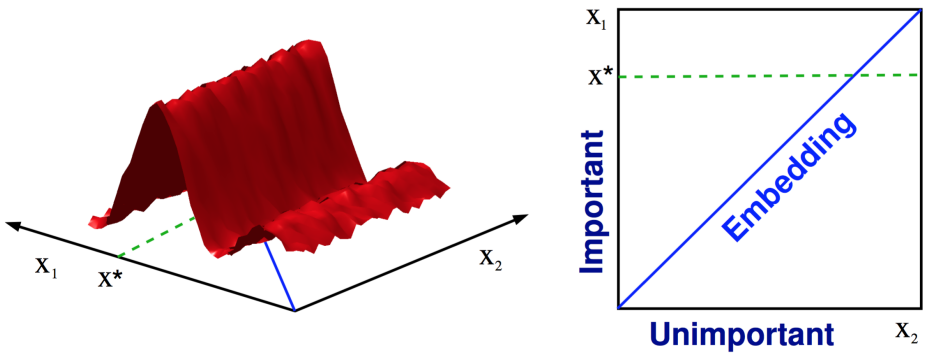


Figure 4: This function in $D = 2$ dimensions only has $d_e = 1$ effective dimension: the vertical axis indicated with the word important on the right hand side figure. Hence, the 1-dimensional embedding includes the 2-dimensional function's optimizer. It is more efficient to search for the optimum along the 1-dimensional random embedding than in the original 2-dimensional space[41].

This definition simply states that the function does not change along the coordinates $x_\perp$, and which is why $\mathcal{T}^\perp$ is referred to as the constant subspace.

For this algorithm, it is assumed that high dimensional functions have a small effective dimensionality and hence the function remains almost constant while moving along most dimensions.

This method performs Bayesian optimisation in a random low dimensional embedding in the original high dimensional space. Figure 4 illustrates this method by using a $D = 2$ dimensional function where only $d_e = 1$ dimension is important. Optimisation is performed in the embedding $x1 = x2$, since it is guaranteed to include the optimum. Importantly, it should be noted, that the method is not restricted to cases with axis aligned intrinsic dimensions.

However this method is restricted to functions for which the low effective dimensionality assumption holds, the performance is limited in functions which do not have low effective dimensionality.

4.2.2 High Dimensional Bayesian Optimisation via Additive Models

This method was developed by Kandasamy et.al.[21], it makes the assumption that the objective function f decomposes into the following additive form :

$$f(x) = f^{(1)}(x^{(1)}) + f^{(2)}(x^{(2)}) + \cdots + f^{(M)}(x^{(M)}) \tag{4.1}$$

Here each $x^{(j)} \in \mathcal{X}^{(j)} = [0, 1]^{d_j}$ are lower dimensional components. Each $\mathcal{X}^{(j)}$ is referred to as a group and the grouping of different dimensions into these groups $\{\mathcal{X}^j\}_{j=1}^M$as the "decomposition". It is important to note that the groups are disjoint.

This method shows excellent performance when the function is indeed additive with

mutually exclusive lower dimensional components but the performance degrades when the function is not well described by these assumptions.

5 Bayesian Optimisation with Bayesian Neural Networks

5.1 Bayesian Neural Networks

Neural networks discussed in Section 2, used maximum likelihood to determine the network parameters. Regularized maximum likelihood can be interpreted as MAP (maximum a posteriori) approach in which the regularizer can be viewed as the logarithm of a prior parameter distribution. However, a full Bayesian treatment of neural networks, requires us to marginalize over the posterior distribution of parameters[7, 32]. In this section we discuss an implementation of a Bayesian neural network in which network parameters are updated using the hybrid Monte Carlo algorithm. The hyperparameters are updated separately by using Gibbs sampling.

Bayesian learning for neural networks is a difficult problem, due to the typically complex nature of the posterior distribution. The hybrid Monte Carlo(HMC) algorithm is particularly suitable for this problem, due to its avoidance of random walk behaviour as discussed in section 3.2.

A neural network can be parametrized by its weights and biases, collectively denoted as $\boldsymbol{\theta}$, that define what network from input to output is denoted by the network. This function can then be written as $f(\mathbf{x}, \boldsymbol{\theta})$, where x is the input to the network. A prior can be defined on the network parameters, this prior can depend on some hyperparameters, γ. The prior density for the parameters can be written as $P(\theta \,|\, \gamma)$ and the prior density for the hyperparameters can be written as $P(\gamma)$. We have a training data-set given by $\mathbf{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$, consisting of independent pairs of input values, $x^{(i)}$, and target values, $y^{(i)}$. We aim to model the conditional distribution of target values given the input values. The required conditional probability density is given by $P(y \,|\, x, \theta, \gamma)$.

The ultimate objective is to predict the target value for new test cases, $y^{(n+1)}$, given the corresponding inputs, $x^{(n+1)}$,using the information in the training set. To make the prediction we require the posterior distribution for $\boldsymbol{\theta}$ and $\boldsymbol{\gamma}$, this is proportional to the product of the prior and the likelihood due to the training cases:

$$P(\boldsymbol{\theta}, \boldsymbol{\gamma} \,|\, \mathbf{D}) \;\propto\; P(\gamma) P(\theta|\gamma) \prod_{c=1}^{n} P(y^{(c)} \,|\, x^{(c)}, \boldsymbol{\theta}, \boldsymbol{\gamma}) \tag{5.1}$$

Predictions on new data-points can then be made by integration with respect to this posterior distribution. The prediction on new data-points is then given by:

$$P(y^{(n+1)}|x^{(n+1)}, \boldsymbol{D}) \;=\; \int P(y^{(n+1)}|x^{(n+1)}, \boldsymbol{\theta}, \boldsymbol{\gamma}) P(\boldsymbol{\theta}, \boldsymbol{\gamma} \,|\, \mathbf{D})\, d\theta\, d\gamma \tag{5.2}$$

For a regression model, the prediction that minimizes the expected squared-error loss is the mean of the predictive distribution. It is given by:

$$\hat{y}^{(n+1)} \;=\; \int f(x^{(n+1)}, \boldsymbol{\theta}) P(\boldsymbol{\theta}, \boldsymbol{\gamma} \,|\, \mathbf{D})\, d\theta\, d\gamma \tag{5.3}$$

Since these integrals take the form of expectation of functions with respect to the posterior distribution, they can be tackled by using a Markov chain Monte Carlo approach. We can approximate the integral with the average value of the function over a sample of values from the posterior.

The hybrid Monte Carlo method discussed in section 3.2 is used to generate samples from the posterior. To apply this method, we first need to formulate the desired distribution in terms of a potential energy function. Since the objective is to sample from the posterior distribution for network parameters, the energy function will be a function of these parameters. The parameters $\boldsymbol{\theta}$, will play the role of the the "position" variables, $\boldsymbol{q}$, of an imaginary physical system. The hyperparameters remain fixed throughout one hybrid Monte Carlo update. Hence we can ignore all energy terms depending only on the hyperparameters. For the generic case described by equation (5.1) , the potential energy can be calculated by taking the negative log of the prior and the likelihood due to the training cases, as follows:

$$E(\boldsymbol{\theta}) = F(\boldsymbol{\gamma}) - \log P(\theta|\gamma) - \sum_{c=1}^{n} P(y^{(c)} \,|\, x^{(c)}, \boldsymbol{\theta}, \boldsymbol{\gamma}) \tag{5.4}$$

where $F(\boldsymbol{\gamma})$ is any convenient function of the hyperparameters. The detailed form of the energy function will obviously depend on the network architecture, the prior, and the data model used. For the purpose of this project we put a Gaussian prior with mean zero and standard deviation σ_p on each of the network parameters. We also assume that the targets are a single real value which can be modelled with a Gaussian noise distribution with standard deviation σ_n. The hyperparameters are then $\boldsymbol{\gamma} = \{\tau_p\,,\, \tau_n\}$, where $\tau_p = \sigma_p^{-2}$ and $\tau_n = \sigma_n^{-2}$. The variances are expressed in terms of the associated precision for convenience. The resulting potential energy function is :

$$E(\boldsymbol{\theta}) \;=\; \tau_p \sum_{i=1}^{k} \frac{\theta_i^2}{2} + \tau \sum_{c=1}^{n} \frac{(y^{(c)} - f(x^{(c)}, \theta))^2}{2}$$

where k is the total number of parameters in the neural net i.e $\boldsymbol{\theta} = \{\theta_1, \ldots, \theta_k\}$, this includes both the weights and the biases associated with the network.

It can be noted that this energy function is similar to the error function which is minimized for training regularized conventional networks. However the objective here is

not to minimise the total energy , but rather to sample from the canonical distribution represented by the energy.

We can proceed by introducing the momentum variable, $\boldsymbol{p}$. The "position" variable, $\boldsymbol{q}$ can be associated with the parameters $\boldsymbol{\theta}$ of the network. We then use the hybrid Monte Carlo algorithm to generate samples from the posterior of the parameters. These samples are then used to approximate the integral in equation (5.3) to make predictions on new data-points.

We plan to use the predictions from the neural network to perform Bayesian Optimisation using the GP-LCB acquisition function[8], which is given by:

$$LCB(x) \quad = \quad \mu(x) - K\sigma(x) \tag{5.5}$$

where $\mu(x)$ is the prediction at point x as given by equation (5.6) , and $\sigma(x)$ is the standard deviation at point x calculated in equation (5.7).

$$\mu(x) \quad = \quad \int f(x, \boldsymbol{\theta}) P(\boldsymbol{\theta}, \boldsymbol{\gamma} \mid \mathbf{D})\, d\theta\, d\gamma \tag{5.6}$$

$$\sigma(x) \quad = \quad \sqrt{\int (f(x, \boldsymbol{\theta}) - \mu(x))^2 P(\boldsymbol{\theta}, \boldsymbol{\gamma} \mid \mathbf{D})\, d\theta\, d\gamma} \tag{5.7}$$

With N , samples from the posterior distribution of the parameters,$\{\boldsymbol{\theta_1}, \ldots, \boldsymbol{\theta_N}\}$, we can approximate the required integrals by using MCMC methods to give:

$$\mu(x) \quad = \quad \frac{1}{N} \sum_{i=1}^{N} f(x, \boldsymbol{\theta_i}) \tag{5.8}$$

$$\sigma(x) \quad = \quad \sqrt{\frac{1}{N} \sum_{i=1}^{N} (f(x, \boldsymbol{\theta_i}) - \mu(x))^2} \tag{5.9}$$

5.1.1 Verifying correctness

To verify the correctness of the above implementation, we show some preliminary results on a dimensional synthetic sinusoidal dataset. The dataset was generated by evaluating the function in equation (5.10) at 20 random points and adding some noise to the observations.

$$f(x) = \sin(7x) + \cos(17x) \qquad \text{for } x \in [-1, 1] \tag{5.10}$$

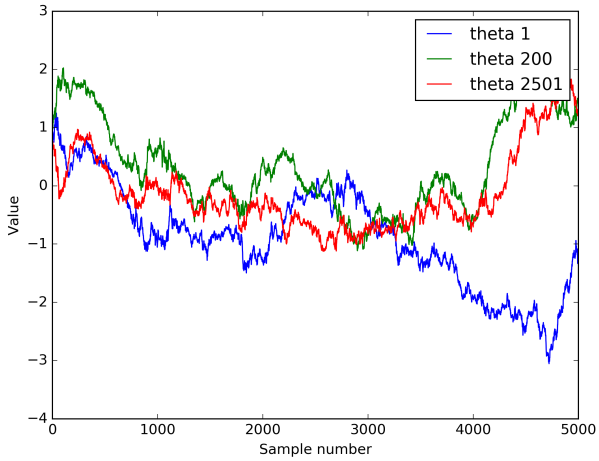


Figure 5: Trace plots of three randomly chosen parameters of the neural network

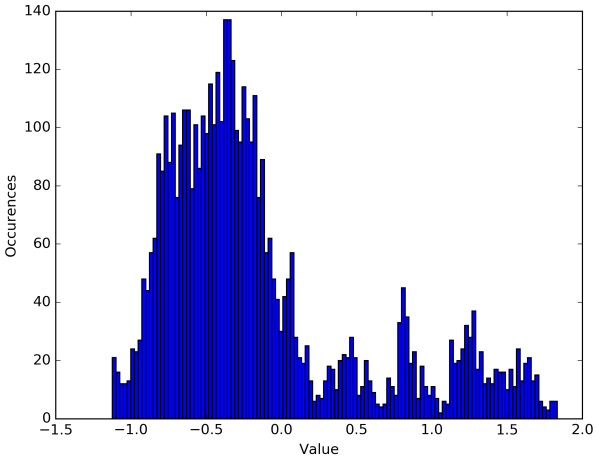


Figure 6: The histogram of the parameter 2501 of the neural network.

A neural network with 3 hidden layers each of width 50 units was used. The *tanh* non-linearity was used in the neural network. The HMC sampler was used to draw 5000 samples from the posterior distribution. Figure 5 shows the trace of three randomly chosen parameters from the neural network. Figure 6 shows the histogram of the of one of the parameters (θ_{2501}) . It can be seen that the Markov chains have not converged to a single mode and instead move between several modes. This can be attributed to the large number of modes that are likely to be present in the posterior. The algorithm is trying to explain 20 training data-points with about 5000 parameters in the neural network. Multiple configurations of the parameters can explain the training data very well. To confirm this we plot the predictions made by the neural networks represented by individual HMC samples. Figure 7 shows the predictions of three well spaced samples from the Markov chain. As expected the predictions fit the training points very well, but show a significant amount of variation far away from them. The predictions of individual samples can then be averaged according to equations (5.8) - (5.9) to find the predictions of the Bayesian neural network. The results are shown in Figure 8, with the credible interval taken to be two standard deviations on either side of the mean. It can be seen that the predictions follow the true function reasonably well. The prediction uncertainty is also modelled very well, it is low near the training data-points and is higher away from them.

5.2 Bayesian Optimisation results

The Bayesian neural network(BNN) model was then used to perform Bayesian optimisation. The LCB acquisition function was used. To illustrate the process of Bayesian Optimisation, we show some intermediate optimisation results on the one dimensional

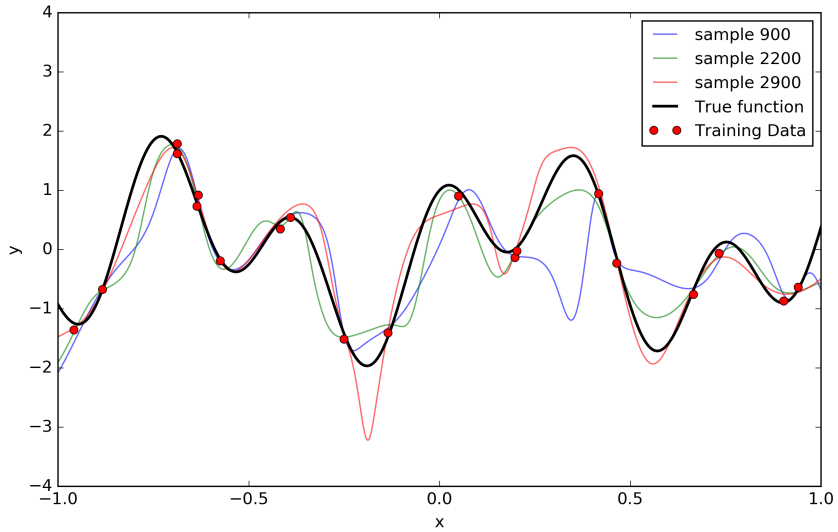


Figure 7: The fit of neural networks from four different samples from the posterior distribution of the network parameters

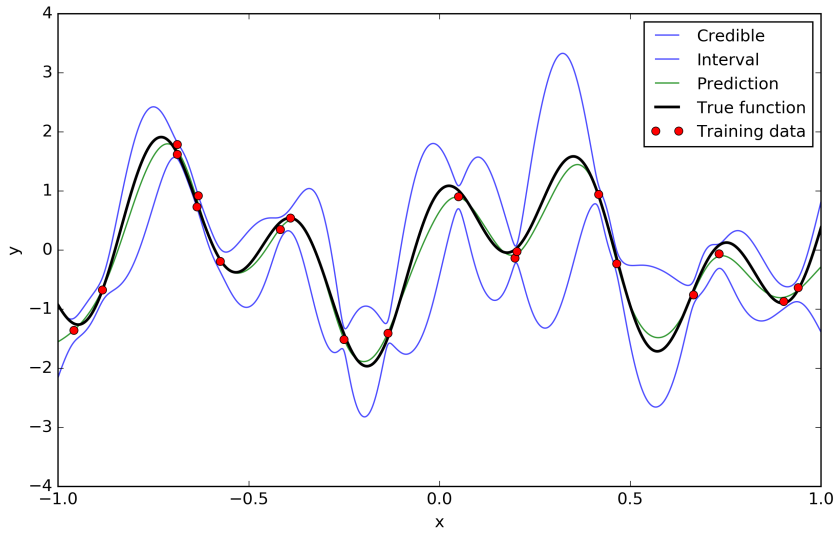
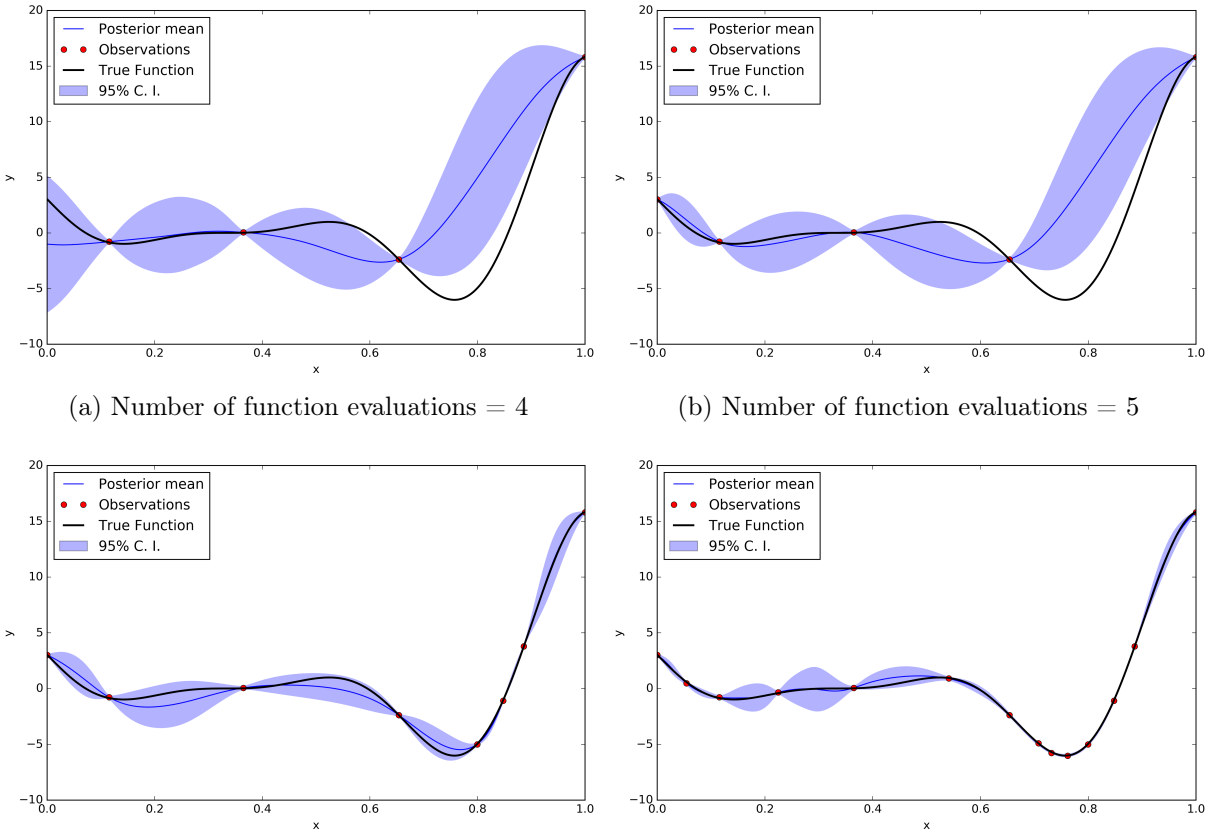


Figure 8: The overall fit of the neural network, averaged over a large number of samples from the neural network



(a) Number of function evaluations = 4

(b) Number of function evaluations = 5

(c) Number of function evaluations = 8

(d) Number of function evaluations = 14

Figure 9: Bayesian Optimisation on Forrester Function. The posterior mean is the averaged prediction from a large number of samples from the posterior of the Bayesian neural network. The standard deviation of the predictions of these samples was used to find the 95% confidence interval(95% C.I).

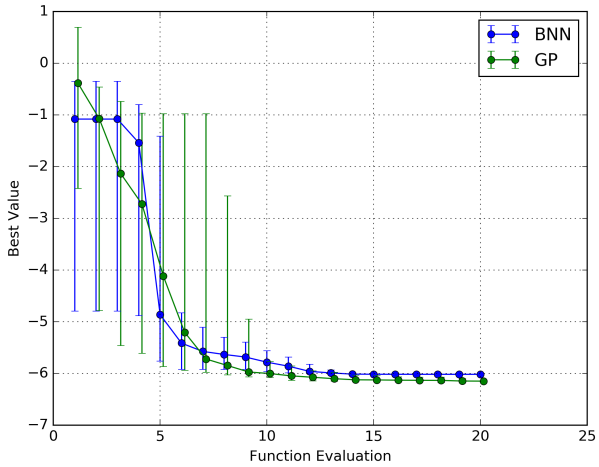
Forrester function[14]. Its global minima is $f(x^*) \approx -6.02074$ at $x^* \approx 0.7572$. The function is given by:

$$f(x) = (6x - 2)^2 \sin(12x - 4) \qquad \text{for } x \in [0, 1] \tag{5.11}$$

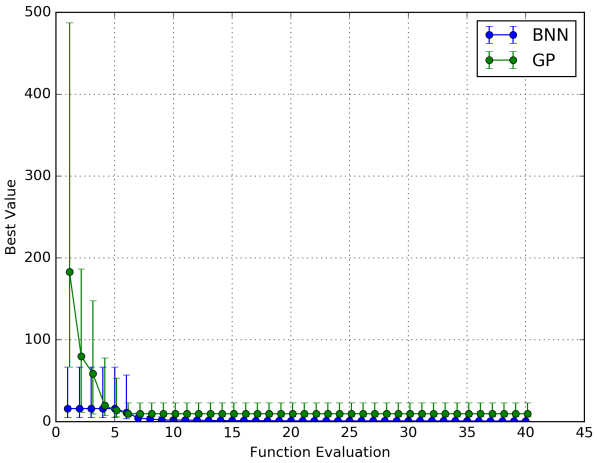
In Figure 9, we show the Bayesian optimisation algorithm in action. We start with four random train points in Figure 9a, then a Bayesian neural network(BNN) model is fit to the data to decide the next proposal point based on the acquisition function. Figure 9b shows the model after querying the Forrester function at the new proposal point ($x^p = 0$). Figures 9c and 9d show the model after a few more function evaluations. The model mirrors the true function reasonably well and finds the true global minima in a reasonable number of function evaluations.

Following this we evaluate the performance of Bayesian Optimisation on a few standard functions, the results are shown in Figure 10.

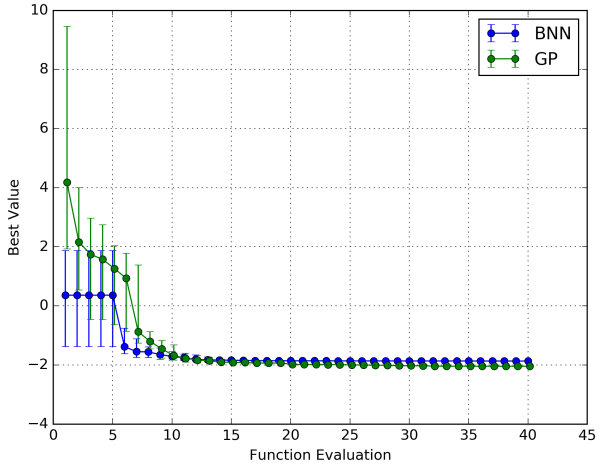
Forrester function : This function has been described by Forrester et.al[14]. Its



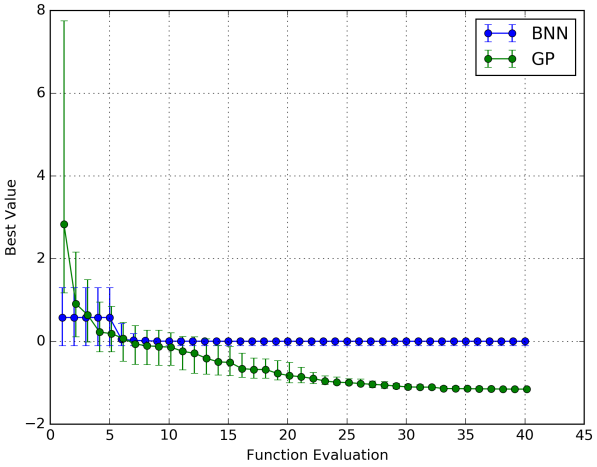
(a) Forrester Function[14]



(b) Rosenbrock(2D) Function[33]



(c) McCormick Function[1]



(d) Six Hump Camel Function[30]

Figure 10: Results of Bayesian Optimisation on some standard test functions. Evaluations using a Bayesian network(blue) and a Gaussian Process(green) are shown. In each subfigure we are minimizing an objective function. The vertical axis represents the running minimum function value. The experiments were repeated 50 times to collect reliable statistics. The circles represent the median of the runs while the top and bottom lines represent the 75^{th} and 25^{th} percentile respectively.

global minima is $f(x^*) = -6.02074$ at $x^* = 0.7572$. It is defined on the interval $[0, 1]$, and has one another local minima in this interval. The Bayesian optimisation process was initialized with two function evaluations at random points in the interval. The BNN model shows competitive performance to a GP based model.

Rosenbrock (2D) function : The Rosenbrock function is a very popular function for evaluating the performance of optimisation algorithms[11] . The function is uni-modal, and the global minimum lies in a narrow, parabolic valley. However, even though this valley is easy to find, convergence to the minimum is difficult[33]. We initialise the Bayesian optimisation process with 5 random function evaluations. Its global minima is $f(x1^*, x2^*) = 0$ at $x1^* = 1$ and $x2^* = 1$, it is defined on the set $\{(x1, x2) : -2.048 \leq x1 \leq 2.048, -2.048 \leq x2 \leq 2.048\}$. The BNN model finds the minima after roughly 7 iterations, the GP based model does not always succeed in finding the the optimum point for this function.

McCormick function : The function is very smooth and uni-modal and is evaluated on the rectangle $\{(x1, x2) : -1.5 \leq x1 \leq 4.0, -3.0 \leq x2 \leq 4.0\}$. Its global minima is $f(x1^*, x2^*) = -1.9133$ at $x1^* = -0.54719$ and $x2^* = -1.54719$. We again start with 5 random function evaluations to initialize the Bayesian optimisation process. Despite better performance initially, the BNN based model converges at a point slightly greater than the global minima, while the GP finds the global minima. However overall performance remains competitive.

Six Hump Camel function : The function is highly multi-modal having 6 local minima, out of which two are also global minima. The function is evaluated on the set $\{(x1, x2) : -3 \leq x1 \leq 3, -2 \leq x2 \leq 2\}$. The global minima are at $x1^* = -0.0898$ & $x2^* = 0.7126$ and at $x1^* = 0.0898$ & $x2^* = -0.7126$, with $f(x1^*, x2^*) = -1.0316$. On this function, the GP model performs much better than the BNN model. The BNN model gets stuck in a local optima and does not manage to find the optima.

5.3 Potential applications

Such as robotics

6 Conclusions

The results are very promising

References

[1] Ernesto P Adorio and U Diliman. Mvf-multivariate test functions library in c for unconstrained global optimization. *Quezon City, Metro Manila, Philippines*, 2005.

[2] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. An introduction to mcmc for machine learning. *Machine learning*, 50(1-2):5–43, 2003.

[3] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[4] Yoshua Bengio, Aaron Courville, and Pierre Vincent. Representation learning: A review and new perspectives. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(8):1798–1828, 2013.

[5] Yoshua Bengio, Ian J. Goodfellow, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2015.

[6] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.

[7] Christopher M Bishop. *Pattern recognition and machine learning*, volume 1. springer, 2006.

[8] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[9] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[10] Misha Denil, Loris Bazzani, Hugo Larochelle, and Nando de Freitas. Learning where to attend with deep architectures for image tracking. *Neural computation*, 24(8):2151–2184, 2012.

[11] LCW Dixon and GP Szegö. The global optimization problem: an introduction. *Towards global optimization*, 2:1–15, 1978.

[12] Simon Duane, A.D. Kennedy, Brian J. Pendleton, and Duncan Roweth. Hybrid monte carlo. *Physics Letters B*, 195(2):216 – 222, 1987.

[13] Robert F Dugan Jr, Rebecca M Dugan, and Corinna Trabucco. Healthy computer use for computer science. *Journal of Computing Sciences in Colleges*, 27(1):9–15, 2011.

[14] Alexander Forrester, Andras Sobester, and Andy Keane. *Engineering design via surrogate modelling: a practical guide*. John Wiley & Sons, 2008.

[15] Ken-Ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192, 1989.

[16] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[17] Geoffrey E Hinton and Ruslan R Salakhutdinov. Using deep belief nets to learn covariance kernels for gaussian processes. In *Advances in neural information processing systems*, pages 1249–1256, 2008.

[18] Matthew D Hoffman, Eric Brochu, and Nando de Freitas. Portfolio allocation for bayesian optimization. In *UAI*, pages 327–336. Citeseer, 2011.

[19] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991.

[20] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

[21] Kirthevasan Kandasamy, Jeff Schneider, and Barnabas Poczos. High dimensional bayesian optimisation and bandits via additive models. 03 2015.

[22] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[23] Daniel J Lizotte, Tao Wang, Michael H Bowling, and Dale Schuurmans. Automatic gait optimization with gaussian process regression. In *IJCAI*, volume 7, pages 944–949, 2007.

[24] David JC MacKay. A practical bayesian framework for backpropagation networks. *Neural computation*, 4(3):448–472, 1992.

[25] David JC MacKay. Probable networks and plausible predictions, a review of practical bayesian methods for supervised neural networks. *Network: Computation in Neural Systems*, 6(3):469–505, 1995.

[26] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[27] Hrushikesh Narhar Mhaskar and Charles A Micchelli. How to choose an activation function. In *Advances in Neural Information Processing Systems*, pages 319–326, 1994.

[28] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[29] Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. The application of bayesian methods for seeking the extremum. *Towards Global Optimization*, 2(117-129):2, 1978.

[30] Marcin Molga and Czesław Smutnicki. Test functions for optimization needs. *Test functions for optimization needs*, 2005.

[31] Radford M Neal. Probabilistic inference using markov chain monte carlo methods. 1993.

[32] Radford M Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 1996.

[33] Victor Picheny, Tobias Wagner, and David Ginsbourger. A benchmark of kriging-based infill criteria for noisy optimization. *Structural and Multidisciplinary Optimization*, 48(3):607–626, 2013.

[34] Marc' Aurelio Ranzato, Alex Krizhevsky, and Geoffrey E. Hinton. Factored 3-way restricted boltzmann machines for modeling natural images, 2010.

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

[36] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

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

[38] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Prabhat, and Ryan P. Adams. Scalable bayesian optimization using deep neural networks. 02 2015.

[39] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

[40] Mervyn Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the royal statistical society. Series B (Methodological)*, pages 111–147, 1974.

[41] Ziyu Wang, Masrour Zoghi, Frank Hutter, David Matheson, and Nando de Freitas. Bayesian optimization in a billion dimensions via random embeddings. *arXiv preprint arXiv:1301.1942*, 2013.

[42] Matthew D Zeiler. Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

A Risk Assessment Retrospective

The risk assessment submitted at the start of the project identified frequent computer use as the main potential hazard. This assessment was accurate and no other hazards were encountered during the execution of the project. To avoid repetitive stress injury from excessive computer use, frequent breaks were taken by the author. Several exercises and best practices[13] were employed to reduce the risk further. If the project is to be carried out again, the risk assessment will remain the same.