

Uncertainty Quantification in Bayesian Neural Networks vs Gaussian Processes

Written by Jasmine Thompson with the support and guidance of Prof. Peter Challenor

1. Contents

1. Contents.....	1
2. Motivation.....	2
What is Uncertainty?.....	2
What is our Main Goal?.....	3
Note on Structure.....	3
3. Methodology.....	4
What Methods are being Evaluated?.....	4
How are we Evaluating Accuracy?.....	5
4. What is Bayesian Probability or Bayesian Statistics?.....	6
Comparison to Frequentist Statistics.....	6
Bayes' theorem.....	7
How do we choose our Prior Distribution?.....	8
5. What is a Gaussian Process?.....	9
What is a Gaussian Process?.....	9
Training a Gaussian Process.....	11
6. What is a Neural Network?.....	12
What is a Neural Network?.....	12
Backpropagation.....	14
7. What is a Bayesian Neural Network?.....	16
8. What is MCMC?.....	18
What is MCMC?.....	18
Metropolis-Hastings Algorithm.....	19
Brief History of MCMC.....	21
9. What is Variational Inference?.....	21
What is Variational Inference?.....	21
Bayes by Backpropagation.....	24
10. What is Dropout?.....	25
11. Torchbnn Implementation.....	26
Model Structure and Prior Selection.....	26
Model Training.....	27

Model Inference.....	27
Further Details.....	28
12. Pyro Implementation.....	28
Model Structure and Prior Selection.....	29
Model Training.....	30
Model Inference.....	30
Further Details.....	31
13. Results.....	31
Standard Neural Network in Pytorch.....	32
Torchbnn Implementation of a Bayesian Neural Network Trained with Variational Inference.....	33
Pyro implementation of a Bayesian Neural Network Trained using Variational Inference.....	35
Torchbnn and Pyro Bayesian Neural Network Trained using Variational Inference...	36
Pyro Implementation of a Bayesian Neural Network Trained using MCMC.....	37
GPytorch Implementation of a Gaussian Process.....	38
Comparison of Best Methods.....	38
14. Conclusion.....	39
Limitations.....	39
Torchbnn vs Pyro Implementation.....	40
Pyro Variational Inference vs MCMC.....	40
Dropout Effect.....	40
Pyro MCMC vs Gaussian Process.....	41
Further Adjustments.....	41
15. References.....	42
Historical.....	42
Tutorials Used.....	42
Key Explanations.....	43
Key Packages Used.....	43
Supplementary.....	44
16. Code.....	44

2. Motivation

What is Uncertainty?

Uncertainty can be described as giving the range of values in which the true value of the measurement lies. This variability in the data may arise due to inherent measurement error or imperfect information making it difficult to predict the actual outcome. Uncertainty captures our lack of ability to perfectly predict the true outcome with a probability distribution.

Capturing uncertainty is useful for safety reasons as we may only want to allow a particular margin of error over which we must go back and collect more data. Another time it is useful

is in explaining how seriously to take predictions about unpredictable phenomena such as rainfall or earthquakes.

Currently, many machine learning models work as essentially a black box where the models generated are too complex to understand in totality. Uncertainty quantification can help this lack of understanding by augmenting them with indications of where the model is likely to fall short.

The majority of the time, speed and accuracy are considered to be the most important factors when doing machine learning. However, in certain cases where safety is paramount such as with self-driving cars then uncertainty quantification becomes essential. For example, in a self-driving car if the model indicates that it is uncertain then safety measures can be taken such as handing off to a human driver. Thus it is unsurprising that the Pyro Python library for probabilistic programming used in this paper to create Bayesian neural networks was developed by Uber who work on creating autonomous cars.

Another focus of uncertainty quantification is in capturing uncertainty when simulating computer models. This is something that Gaussian processes are commonly used for.

What is our Main Goal?

To understand how uncertainty is quantified in a Bayesian neural network compared to a Gaussian process. Gaussian processes are more straightforward and thus their uncertainty-generating process is more easily understood. Both models serve to predict the underlying mean function of the data and evaluate their own uncertainty in this prediction with an uncertainty distribution from which we can set uncertainty boundaries.

Uncertainty boundary

- The real value for the data is expected to fall inside this boundary with a particular percentage of the time
- For example, with a 95% uncertainty boundary, the real value is expected to fall inside it 95% of the time

Essentially, we want to check how well both kinds of function approximators perform and also whether they are able to capture when they have too little information to give an accurate prediction.

Note on Structure

This report will aim to describe in a high-level way the underpinning mathematics behind the methods used and how they are implemented in practice. This should give insight into any particular quirks of the different methods. In the end, the methods used will be evaluated with visuals and metrics.

Many sections describe in detail key concepts required for understanding what a Bayesian neural network actually is, how it is trained, how it works and why we use them at all. The reason for this was to understand the final results as clearly as possible. The biggest difficulty during this project was not in obtaining relevant results but in gaining the relevant background of how all of these processes worked mathematically to compare with the Python code implementations used. Understanding the mathematics behind the key processes in a sufficiently detailed way led to having enough understanding to explain some of the results found in the conclusion.

For this work, I relied heavily on the paper “Hands-on Bayesian Neural Networks – A Tutorial for Deep Learning Users” by Jospin L. V. et al. [J1] which gave the most comprehensive introduction to the many varied types and implementations of Bayesian neural networks out of any material reviewed. Much of the initial code was taken from various tutorials before refining and integrating it. These tutorials have been included in the references clearly in their own section.

3. Methodology

What Methods are being Evaluated?

The methods we are interested in evaluating are Gaussian processes and Bayesian neural networks which are both methods capable of function approximation which provide an uncertainty boundary as already described.

Gaussian processes are more well established and have a more simplified implementation but they can be computationally intensive to implement. Similarly to Bayesian neural networks they use Bayesian concepts for training.

Standard non-Bayesian neural networks are very efficient at getting accurate mean predictions but are in many ways some of how they work is too detailed to understand in an exact way making them a black box and they don't give an uncertainty boundary.

Bayesian neural networks are an extension of standard neural networks. Similarly to Gaussian neural networks they are computationally expensive but they are also able to capture the model's uncertainty.

It is important to note that in many ways Bayesian neural networks is a catch-all term which encapsulates a multitude of neural network structures and probabilistic structures with the common features that a neural network is used and Bayesian methods are used to train that neural network and get an output result when evaluating new data. [J1]

We used 3 different types of Bayesian neural network:

1. A torchbnn implementation trained using variational inference
2. A Pyro implementation trained using variational inference
3. Pyro implementation trained using MCMC (Markov Chain Monte Carlo).

We created our Gaussian process using GPyTorch. The details of these implementations will be further explained once key concepts have been covered.

How are we Evaluating Accuracy?

The function underlying our data with the noise included	The distribution of the noise in our dataset
$f(x) = x \times \sin(x + 5) + \epsilon$	$\epsilon \sim \mathcal{N}(0, 1)$

Figure 3.1

To create our predictable noisy data with an underlying one-dimensional function we used the function shown in figure 3.1 with 10000 data points generated for the training set and another 10000 for the test set. The function was calculate between the boundaries of x=-5 to x=5.

This means that we have control over the actual origin of the data so are able to assess model performance objectively. As the model is unable to account for the random noise and get perfectly accurate predictions, we can use this idea to assess the model on its ability to capture the random noise with an uncertainty boundary.

$$MSE(\mathbf{y}) = \frac{1}{n} \sum_{i=1}^n (y_i - \tilde{y}_i)^2$$

n	The number of draws to assess the model on
y_i	The true value as given by the mean function
\tilde{y}_i	The predicted mean value as given by the model

Figure 3.2

Assessing how well each method is able to capture the mean function underlying some data is relatively simple as we can simply take a large number of draws from the real underlying function and compare these to the corresponding mean prediction from the model using Mean Squared Error (MSE) as shown in figure 3.2. Mean squared error can be seen as the difference between the true value and the predicted value then squared for all predictions and averaged across them.

Quantifying whether the model is able to capture the variation in the data using an uncertainty boundary is more challenging but a simple test is to evaluate the 95% confidence

interval. Essentially if 95% of the noisy data falls within some boundary then in an ideal scenario this portion of the data should fall within the model's 95% uncertainty boundary also. If more of the data falls within that boundary then the model is underconfident in its prediction and if less of the data falls within that boundary then the model is overconfident.

4. What is Bayesian Probability or Bayesian Statistics?

Comparison to Frequentist Statistics

Bayesian can be thought of as a new way of conceptualising what we mean when we talk about probabilities. It gives us new ways of quantifying probability and of adjusting our probability given new data.

Supporters of Bayesian methodology give a new name to the more common form of doing statistics which people are familiar with from school. It is called 'frequentist statistics' to contrast with Bayesian statistics.

The theory behind Bayesian statistics is to point out that we cannot know the true probability of an event. In deterministic reality the event has a probability of 1 for what actually occurs and a probability of 0 for what doesn't occur. In other words if we say an event will occur with 50% certainty we are not capturing the reality which is whether the event does or doesn't occur which is fixed but unknowable.

An analogy for understanding is that if we roll a dice and hide it under a cup then that dice has a fixed value which is simply unknown to us. The idea given by Bayesian statistics then is that in frequentist statistics we are not capturing the true result on the dice but instead the long run frequency of results from tossing the dice many times. We actually cannot know the true result on the dice but only make subjective beliefs about it.

Long run frequency

- The frequency of different outcomes of some data generating process (e.g. tossing a dice) when performed many times
- When divided by the number of times the data generating process has been performed this gives us our frequentist probability

Bayesian statistics doesn't throw out this idea of using a long run frequency of observed results to predict future behaviour but instead modifies it. It is weighted with our beliefs about the data.

Weighted

- When the idea of a weight is used it is simply a variable which that other quantity is multiplied by used to moderate or amplify it

- If a quantity is weighted by some variable it means that one variable adjusts the value of the other through multiplication
- Usually used when we want to adjust something

This idea of including our beliefs in the calculating of final probabilities allows us to adjust our probability to be more sceptical of the rare cases when the data predicts that something is the case by simple chance due to a lot of variables being compared or analysed. For example in frequentist statistics there can exist spurious correlations between two sets of data despite there being no real mechanism for them to interact.

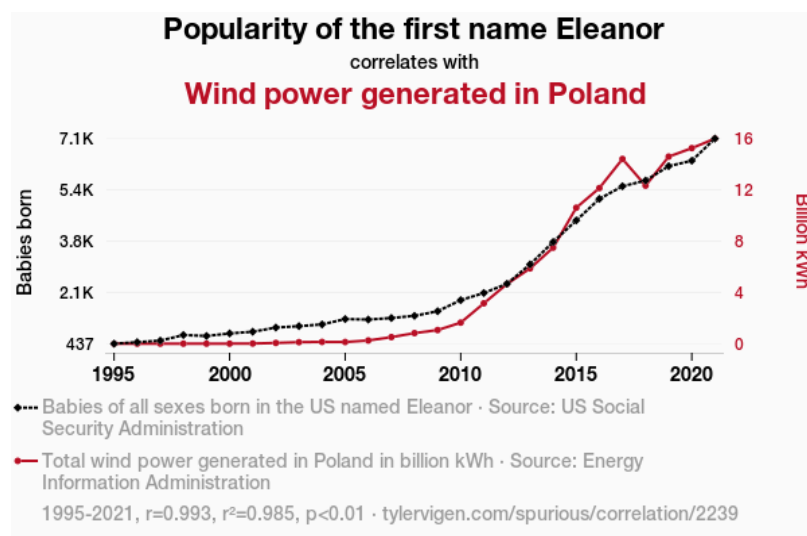


Figure 4.1 [V1]

In figure 4.1 we can see that there is a high degree of correlation between the popularity of the first name Eleanor and the amount of wind power generated in Poland despite these things being completely unrelated. It demonstrates the failings of using a hypothesis test with a p-value cut off to determine correlation as the p-value here indicates a less than 1% chance that these two variables are uncorrelated yet by common sense we can tell that there can be no relation between them. The combined inclusion of our own beliefs can mean we are more sceptical when events like this occur.

Bayes' theorem

Bayesian incorporates our previous knowledge or belief in the form of a probability distribution indicating our expectation of the outcome of some data generating process or experiment. This can be thought of as representing our belief in what we expect the result of an experiment to be which we call the prior distribution.

Following this the process of generating the final probability distribution given the data is that of refining our initial beliefs with the addition of new data. We call this final outcome probability distribution the posterior distribution. It combines our beliefs and observations of the data to give a probability distribution capturing both. This can be described using Bayes' formula which is the key formula used in Bayesian inference.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

P(A B)	Posterior	Our updated belief about A given the evidence B
P(B A)	Likelihood	Probability of observing evidence B given that our prior belief is the case
P(A)	Prior	Our initial belief about the underlying process
P(B)	Evidence/ Data / Marginal likelihood	Probability of observing the data regardless of the prior

Figure 4.2

Bayes' theorem as shown above indicates that using the probability that evidence/data will occur given our current beliefs, our current beliefs and the probability that the evidence/data will occur in general we are able to generate an updated belief using the new evidence/data.

My understanding of Bayes' theorem was informed by the video "Bayes theorem, the geometry of changing beliefs" [S3]

How do we choose our Prior Distribution?

There are a variety of methods and approaches for choosing this prior distribution and there is no set way to do it but there are concepts that inform our choices.

Our prior distribution indicates our belief of what the outcome will be as a distribution. Usually the prior distribution will include all possible values which the outcome might be with some probability but values which are considered extremely unlikely will be given a very low probability of occurring. This is because it is a good idea not to give 100% certainty in beliefs as realistically it is impossible to know all of the information leading up to an outcome. [D2]

It can be noted that when we use a prior distribution the idea is that with the right amount of data even a very skeptical prior belief in an event occurring gets 'overwhelmed by the data' such that the final posterior distribution comes to reflect the data.

Sometimes for ease of calculation we choose our prior distribution to be a conjugate prior meaning it is chosen such that when combined with our data we have a posterior distribution of the same type as the prior. [D2] This is often used for the purposes of having an easily calculable closed-form solution to Bayes' theorem.

Alternatively, we may choose an objective prior if we don't want our prior choice to reflect our subjective beliefs. This may occur in the case we are more interested in the training method

of a Bayesian process and how it allows us to incorporate new data and calculate uncertainties. We want to be careful about claims of subjectively influencing our final outcome. In this case we may choose our prior with the idea that it would not favour some predictions or outcomes over others and for this purpose a uniform distribution is often used. However it is rare to get a truly objective prior. [D2]

5. What is a Gaussian Process?

What is a Gaussian Process?

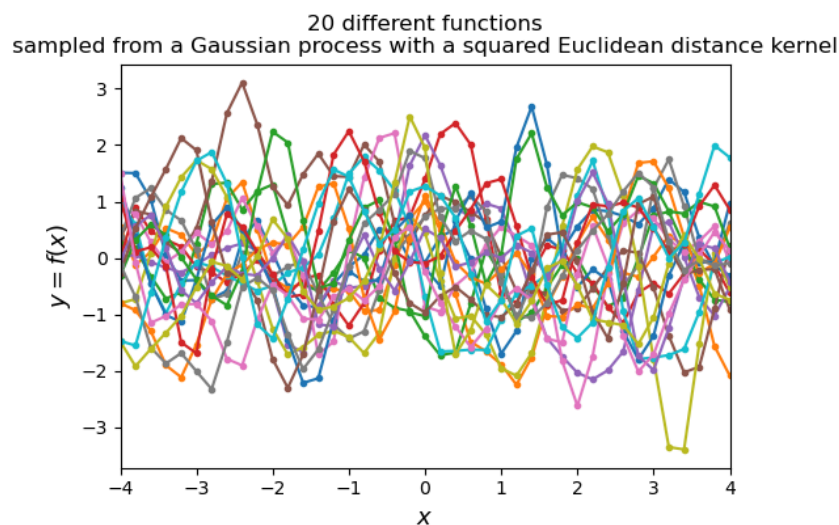


Figure 5.1

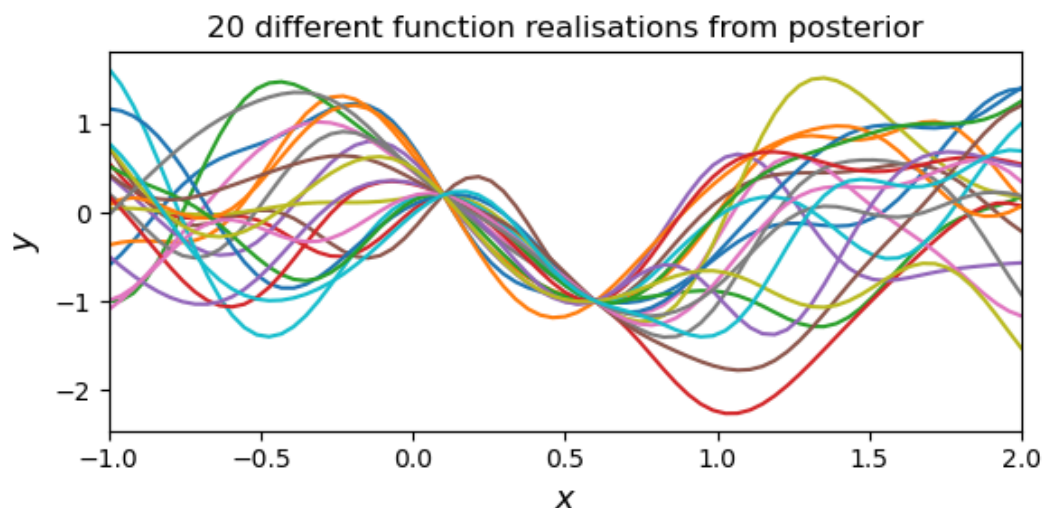


Figure 5.2

A Gaussian process can be thought of in two ways. We can consider it as a joint multivariate Gaussian distribution of an infinite number of random variables or as a set of smooth continuous functions which pass through the given data as shown in figure 5.2.

A joint multivariate Gaussian/normal distribution

- The probability distribution of many normally distributed random variables which are correlated
- Allows us to model how these variables are related so they don't have to be independent and identically distributed

We can think of a Gaussian process as being a multivariate Gaussian distribution over the data which tells us the probability of any other location.

If we then extend if we have continuous input values then the Gaussian process can be conceptualised as the average of all possible smooth continuous functions for the x values we provide. We can see representations of these smooth continuous functions before conditioning on some data points in figure 5.1 and after conditioning on some data points in figure 5.2.

Function being modelled by a Gaussian process	Formula for the mean function	Formula for the most common kernel function the mean covariance function
$f(x) \sim \mathcal{GP}(m(x), k(x, x'))$	$m(x) = \mathbb{E}[f(x)]$	$k(x, x') = \sigma^2 \exp\left(-\frac{(x - x')^2}{2l^2}\right) + \tau^2$

$f(x)$	The function we are modelling
$m(x)$	The mean function which is the expected value of the function we are modelling for any input
$k(x, x')$	The kernel function meant to represent the covariance between two data points
x	The data point we are interested in
x'	The other data point
l	The character length scale parameter which can control the smoothness of the functions in the Gaussian process
τ^2	The nugget which the square of the expected measurement noise added to make sure the Gaussian process doesn't fit the training data too exactly

Figure 5.3

As shown in figure 5.3 our Gaussian process is fully specified with a mean function and a kernel function.

The kernel function represents a variety of different ways of capturing the covariance between any two data points. Usually this is calculated as some function of the distance between them.

Providing a nugget as shown in figure 5.3 supplies this kernel function with some additional unavoidable variability inherent in the data. This allows the model to account for the fact that the data have some unavoidable measurement error or variability included in it so that it cannot be fit with perfect accuracy. This means that our predictions need not predict our actual data with perfect certainty but instead our model is conditioned on the data we use.

A function drawn from the Gaussian process modelled as a joint multivariate normal distribution (our prior distribution)	The covariance matrix as applied from applying the kernel function
$f \sim \mathcal{N}(m(\mathbf{x}), K(\mathbf{x}, \mathbf{x}))$	$K_{ij} = k(x_i, x_j)$

f	A function drawn from the Gaussian process
x	A vector of our training data points
m(x)	The mean vector of the set of training data points
K(x, x)	The covariance matrix between our training data points
k(x_i, x_j)	The kernel function between the two points

Figure 5.4

As shown in figure 5.4 a function drawn from the Gaussian process can be represented by a joint probability normal distribution created using a mean function/vector of the training data points as the mean value and a kernel function representing covariance between our data points as the joint covariance matrix.

In this case our joint multivariate normal distribution is characterised by a mean vector $m(\mathbf{x})$ and a kernel matrix $K(\mathbf{x}, \mathbf{x})$ capturing the kernel values between any two variables as depicted in figure 5.4.

This distribution is used because the points of training data we have must be assumed to be related in order to give structure to our final prediction distribution.

Training a Gaussian Process

The posterior distribution modelled as a normal distribution
--

$$f(x^*|\mathbf{x}, \mathbf{y}) \sim \mathcal{N}(\mu(x^*|\mathbf{x}, \mathbf{y}), \sigma^2(x^*|\mathbf{x}, \mathbf{y}))$$

x^*	The new location we are interested in
\mathbf{x}	A vector of the inputs of the training data
\mathbf{y}	A vector of the outputs of the training data
$f(x^* \mathbf{x}, \mathbf{y})$	The distribution of predictions for a new location x^* given the training data
$\mu(x^* \mathbf{x}, \mathbf{y})$	Our posterior mean function trained on the training data
$\sigma^2(x^* \mathbf{x}, \mathbf{y})$	Our posterior variance trained on the training data

Figure 5.5

As we know a Gaussian process can be understood as a multivariate Gaussian distribution. However in order to make it fit our data we must get a posterior distribution for it using Bayesian. This posterior distribution represents all the possible smooth continuous functions which pass through some given data points as an ensemble providing a probability distribution. The form for this posterior distribution is given in figure 5.5.

Ensemble

- The set of all possible outcomes of a process
- Can be thought of as a probability distribution from which we can draw infinitely many individual functions as shown in figure 5.1.

In order to train the Gaussian process using Bayesian inference where we have a predefined prior belief which is then adjusted using Bayesian inference to condition our distribution on the data so that the new posterior distribution is influenced by the data points.

If there is no nugget indicating unpredictable (cannot predict) variability in the data that cannot be captured by the model, then the functions must pass exactly through the points provided.

In order to perform Bayesian inference we often use MCMC (Markov Chains Monte Carlo) which will be discussed in a later section.

6. What is a Neural Network?

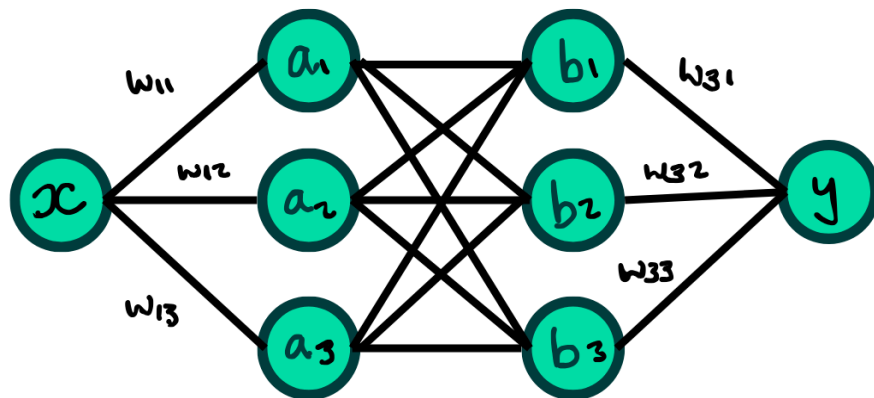
What is a Neural Network?

A neural network is a way of fitting functions that involves creating a particular network structure of preliminary values connected by weights and adjusting the weights in this

structure until it is able to produce the right output for a given input. In our case we are only interested in acyclic feedforward neural networks.

Acyclic feedforward neural networks

- We have an output that is generated by feeding data progressively through our neural network from our input to our output without doubling back and there are no loops in the structure



w_{2ij} given for weight connecting a_i and b_j

The first layer of hidden nodes	The second layer of hidden nodes	The output node
$a_i = w_{1i}x + \beta_{1i}$	$b_i = \sum_{n=j}^3 w_{2ji}a_j + \beta_{2i}$	$y = \sum_{n=i}^3 w_{3i}b_i + \beta_3$

Figure 6.1 [S1]

Figure 6.1 demonstrates the process of producing an output for any given input of a neural network after it has been trained. This is also referred to as the forward propagation process contrasting with the method of training which is known as backpropagation.

As shown in figure 6.1, when we want to predict an output for a given input using our neural network structure we first calculate the first row of hidden nodes (nodes which aren't the input or the output) as a linear combination of any input nodes with the weights connecting them to the next layer which is then fed through an activation function not shown in figure 6.1 but added in figure 6.2.

Then every subsequent layer of hidden nodes is also a linear combination of the previous layer of hidden nodes and the layer of weights in between also fed through an activation function. The output is calculated in the same way from the last layer of hidden nodes. Thus it is easy to conceptualise how we can have a varied number of inputs and outputs and how we can vary the width and number of hidden layers of nodes by simply adding more nodes to the diagram.

Although in this diagram the weights and biases are separate for all other purposes throughout this report the biases are considered to be an additional weight which is trained and so aren't mentioned much directly but are implicitly included when weights are mentioned.

Nodes

- Shown by green circles in figure 6.1 the nodes are the preliminary values we calculate in order to get our final prediction

Weights

- The connections between the nodes as shown by lines in figure 6.1
- These are adjusted in order to train our model

Hidden layers

- The layers of nodes which are not included in the input or output

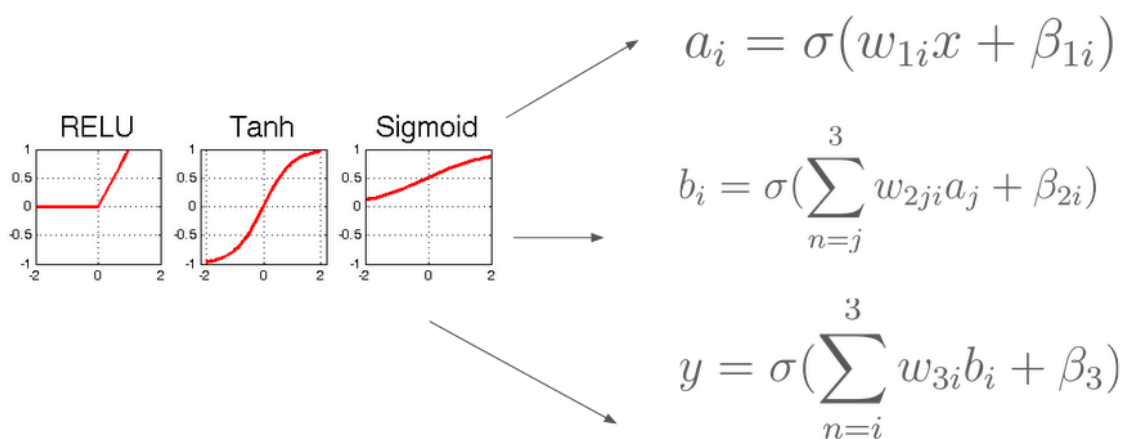


Figure 6.2

Activation functions are simply functions applied to the linear combinations shown before to get the final value for the node. The purpose of an activation function is to introduce non-linearity into the network so it can learn more complex patterns in the data. Figure 6.2 shows some different types of activation functions being applied to the formulas from figure 6.1 in the form of the function σ .

Backpropagation

$$MSE : L = (y_r - y)^2$$

L	The Loss
y_r	The real value the output is being compared to

y	The output of the neural network as shown in figure 6.1

Figure 6.3

In order to train the neural network we start by initialising the weights randomly. We then need to use a measure of loss which represents how badly the neural network performed at making a prediction. A common type of loss called mean squared error is shown in figure 6.3. By minimising this loss we can train the model to perform better.

$y = \sigma(z)$	$z = \sum_{n=i}^3 w_{3i}b_i + \beta_3$
The output of the neural network after being fed through the activation function	The value before the activation function as expressed as the sum of previous hidden nodes b_i , the weights w_{3i} and the bias β_3

Figure 6.4

$\frac{\delta L}{\delta w_{31}} = b_1 \times \sigma(z)' \times 2(\sigma(z) - y_r)$
The formula for the gradient of the loss given the weight w_{31} calculated from the weights and values of the previous hidden nodes, the activation function and the true value of y

Figure 6.5

We then use a process known as backpropagation where we calculate the gradients of the last layer of weights with respect to the loss function representing the error in the current prediction of the neural network. These gradients can then inform how we adjust these weights to minimise our loss function. Figure 6.4 and 6.5 depict the key formulas for calculating the value of the gradient for a specific weight given the mean squared error loss function shown in figure 6.3.

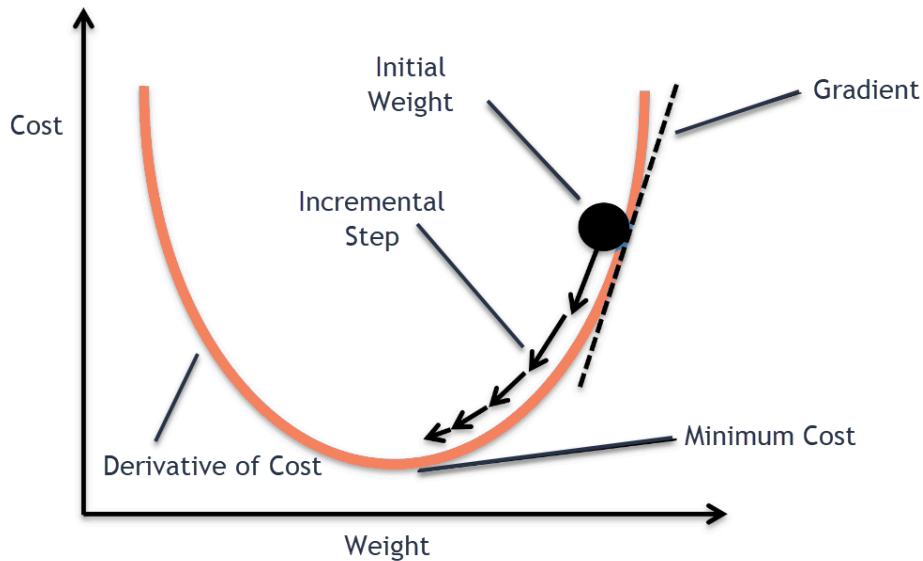


Figure 6.6

For optimisation we usually use a method known as stochastic gradient descent roughly depicted in figure 6.6 whereby we explore the parameter space in order to reach a minimum loss by computing the gradient of the loss function with respect to randomly drawn samples from the training data. These gradients can be used to adjust the parameters to minimise the loss before calculating the gradients again. The speed in which descent occurs is referred to as the training rate.

Stochastic

- There is an element of randomness to the process i.e. in what direction and how large of step we take is not perfectly determined by the size and direction of the gradient

Using stochastic gradient descent introduces noise into the parameter updates which can help avoid getting stuck in a local minima. We usually use something called the Adam optimiser which just means that when we carry out this formula for gradient descent we usually define a learning rate which defines how quickly we descend but with an Adam optimiser this is adjusted as a part of the optimisation process which leads to quicker convergence. [K1]

7. What is a Bayesian Neural Network?

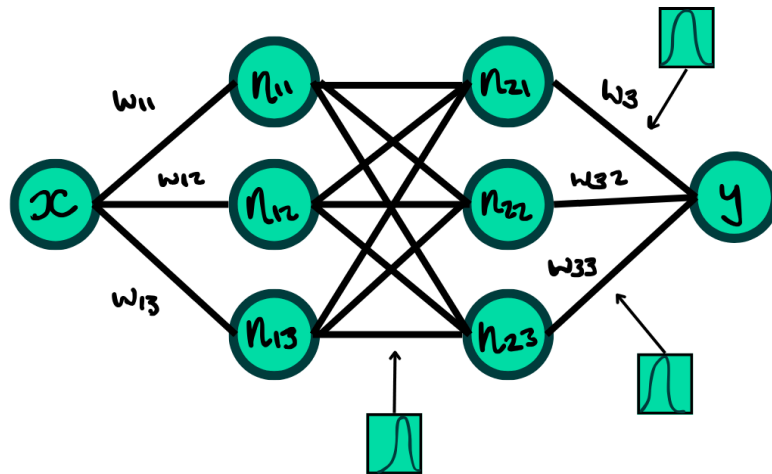


Figure 7.1

Bayesian neural networks usually train the weights as a distribution rather than adjusting to find the single optimal value as depicted loosely in figure 7.1. [J2] [R2] [M1] This allows us to introduce uncertainty into our predictions and means that we can train our neural network using Bayes' theorem. As we are using Bayes' theorem we have to give each of the weights a prior distribution which then gets adjusted based on the data.

There also exists an alternative structure where instead of having the weights be stochastic variables we have the activations as stochastic variables and this method allows us to combine the variables involved for variational inference. [J1] However, having the weights be stochastic variables is more commonly used in practice. [J1] In fact both implementations of a Bayesian neural network used in this paper of torchbnn and Pyro use stochastic weights as the structure rather than stochastic activations.

Stochastic variables

- A variable defined with a probability distribution rather than a fixed value

Variational inference

- A method for approximating the posterior distribution by confining it to a particular type of distribution
- Will be discussed in more detail in a later section

Activations

- A name for final value received for a node after applying the activation function

Bayesian neural networks have a lot of different formats on top of the regular neural network structure depending on which parameters we are trying to train and which are fixed. It is possible to only have weights be stochastic variables in the final layers of a neural network in order to decrease training time while capturing uncertainty. [J1]

It is also possible to have Bayesian neural networks using a method called dropout which introduces additional uncertainty based on the structure of the network. [J1] This will be discussed in a later section.

Finally a multitude of different prior distributions for the stochastic variables can be used although in practice normal distributions are used most often. One reason for this is as normal distributions are mathematically tractable and have convenient and simple properties for inference. In fact normal distributions are often required when performing variational inference which will be discussed later. Normal distributions are also commonly used and intuitively understood which makes them easier to use when we want to capture our beliefs.

There are two ways to make training more efficient when training a Bayesian neural network and these are MCMC and Variational Inference. These will be discussed in detail in the following sections.

8. What is MCMC?

We usually use Markov Chains Monte Carlo (MCMC) for Bayesian inference. This means we sample from the posterior directly rather than needing to approximate the posterior.

What is MCMC?

Markov chains are a stochastic process similar to a random walk where the probability of moving from one state to another is determined only by the current state, not on the entire history of the chain. Markov chains are constructed to explore possible values of a probability distribution. The idea is that if high probability areas of the probability distribution are sufficiently sampled then we can construct a good representation of it.

Monte Carlo is a method which can perform integration by using random sampling to simulate the process.

When using the Metropolis-Hastings algorithm we construct a Markov chain which explores the space of values in such a way that we eventually converge and are able to capture the desired distribution. This is used for approximating a posterior distribution given a Bayesian neural network.

In other words we generate a sequence of values using the Markov chain in such a way that with an infinite number of values the distribution of the values in our sequence is approximately the same as the distribution we are trying to estimate.

$$P(\theta|x) = \frac{P(x|\theta)P(\theta)}{P(x)}$$

P(Θ x)	Posterior	The new distribution of the parameters given the data
P(x Θ)	Likelihood	Probability of observing the data x given the parameters
P(Θ)	Prior	The distribution representing our beliefs about the

		parameters before observing the data
P(x)	Evidence/ Data / Marginal likelihood	The probability of observing the data x regardless of the parameter values

Figure 8.1

Figure 8.1 shows another version of Bayes' theorem specifically tailored to be closer to how it is used to train a Bayesian neural network where any stochastic variables/parameters (usually just the weights) are adjusted using the formula.

Calculating the evidence using the likelihood and the prior by integration.

$$P(x) = \int P(x|\theta)P(\theta) d\theta$$

Figure 8.2

$$P(x) \approx \frac{1}{N} \sum_{i=1}^N P(x|\theta^{(i)})$$

Figure 8.3

One difficulty of using Bayes' theorem is that to compute the marginal likelihood $P(x)$ we have to integrate over the product of the likelihood and prior over all possible parameter values as shown in figure 8.2. MCMC methods can be used to approximate this integral using random sampling so that we can calculate the desired posterior distribution as shown in figure 8.3. Here the integral can be estimated by taking N samples $\theta^{(i)}$ from the posterior distribution and averaging the likelihood of the data given this set of parameters.

Metropolis-Hastings Algorithm

In general the Metropolis-Hastings algorithm for performing MCMC involves moving from state to state on a Markov chain until the samples converge to the posterior distribution. These samples are then used for prediction.

The idea here is that we start with some values for the stochastic parameters $\Theta^{(t)}$ and we want to generate another set of stochastic parameters from the same distribution $\Theta^{(t+1)}$ with the intention of creating a posterior distribution of our stochastic parameters trained on the data.

Then we have a candidate Θ' for a new set of parameters from the same distribution which we generate from a proposal distribution which is given by $q(\Theta' | x)$. This distribution depends on the current state of the Markov chain $\Theta^{(t)}$. Typically the proposal distribution is a normal distribution centred on the current state $\Theta^{(t)}$. [N2]

$$A(\theta', \theta^{(t)}) = \min \left(1, \frac{P(\theta'|x) \cdot q(\theta^{(t)}|\theta')}{P(\theta^{(t)}|x) \cdot q(\theta'|\theta^{(t)})} \right)$$

$\Theta^{(t)}$	Current state
Θ'	Candidate new state
x	The data we have currently
q	The proposal distribution
$P(\Theta' x)$	The probability of the new state given the data
$q(\Theta' \Theta^{(t)})$	The probability of a candidate new state given the old state
$q(\Theta^{(t)} \Theta')$	The probability of proposing our original state given the candidate new state
$P(\Theta' x)$	The probability of the new candidate state based on our current data
$P(\Theta^{(t)} x)$	The probability of our current state based on the data

Figure 8.4

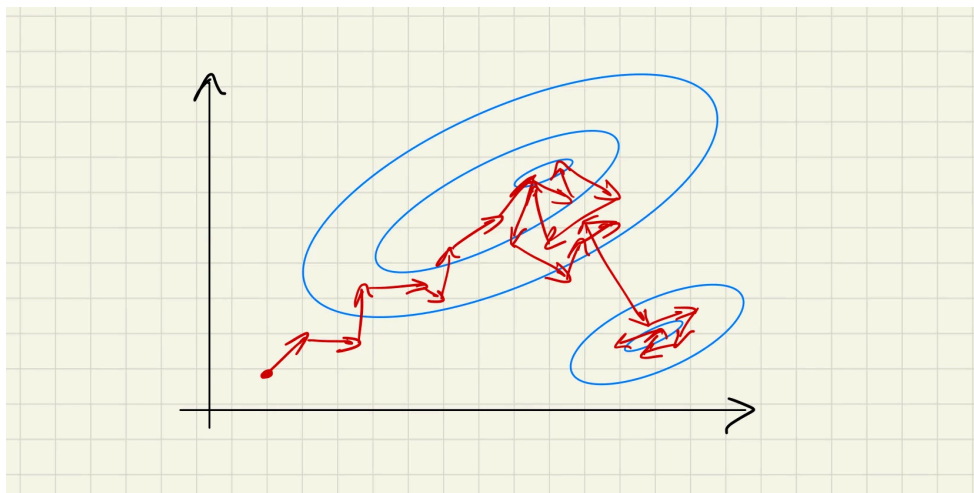


Figure 8.5

Then we decide whether we want to accept or reject this next step by calculating the acceptance probability which determines whether we accept our candidate Θ' as our next step $\Theta^{(t+1)}$.

This acceptance probability captures our likelihood of moving to the next step and is depicted in figure 8.4. Essentially we choose the probability of moving to the next step based

on the probability of the new position. We want to explore areas of the posterior with a high probability more extensively in order to capture key areas of the distribution, but we also want to move around the space. This movement around a 2-dimensional probability of distribution intended to capture areas of high probability is depicted in figure 8.5.

Given this, intuitively it is important to have a period we refer to as burn-in for which the parameter space is explored before we start collecting values from the posterior so that areas of high probability representative of the posterior distribution can be found. Any steps taken during burn-in are then discarded and not included in our set of posterior distribution draws.

These explanations benefited from the straightforward explanation of a Metropolis-Hastings Algorithm by Navarro D. [N2]

Brief History of MCMC

MCMC originated in the paper “Equation of State Calculations by Fast Computing Machines” by Metropolis et al. which described a general method of calculating the properties of a system made from many interacting particles. [M1] [N1] Following this another approach with deterministic molecule movement encapsulated by Hamiltonian dynamics was created which also aimed to calculate the behaviour of many interacting particles. [I1] [N1] Finally these two approaches were combined in the paper “Hybrid Monte Carlo” by Duane et al. where they formed what we usually refer to as “Hamiltonian Monte Carlo” (HMC). [D1] Hamilton Monte Carlo’s performance is however sensitive to the setting of the step size and desired number of steps. [H2] NUTS or “The No-U-Turn Sampler” is a form of the MCMC algorithm which is an addition to HMC where we eliminate the need to set a number of steps and we adapt the step size as we go. This is done by exploring the space of the target distribution and stopping when the algorithm begins to double back and retrace its steps. [H2]

9. What is Variational Inference?

What is Variational Inference?

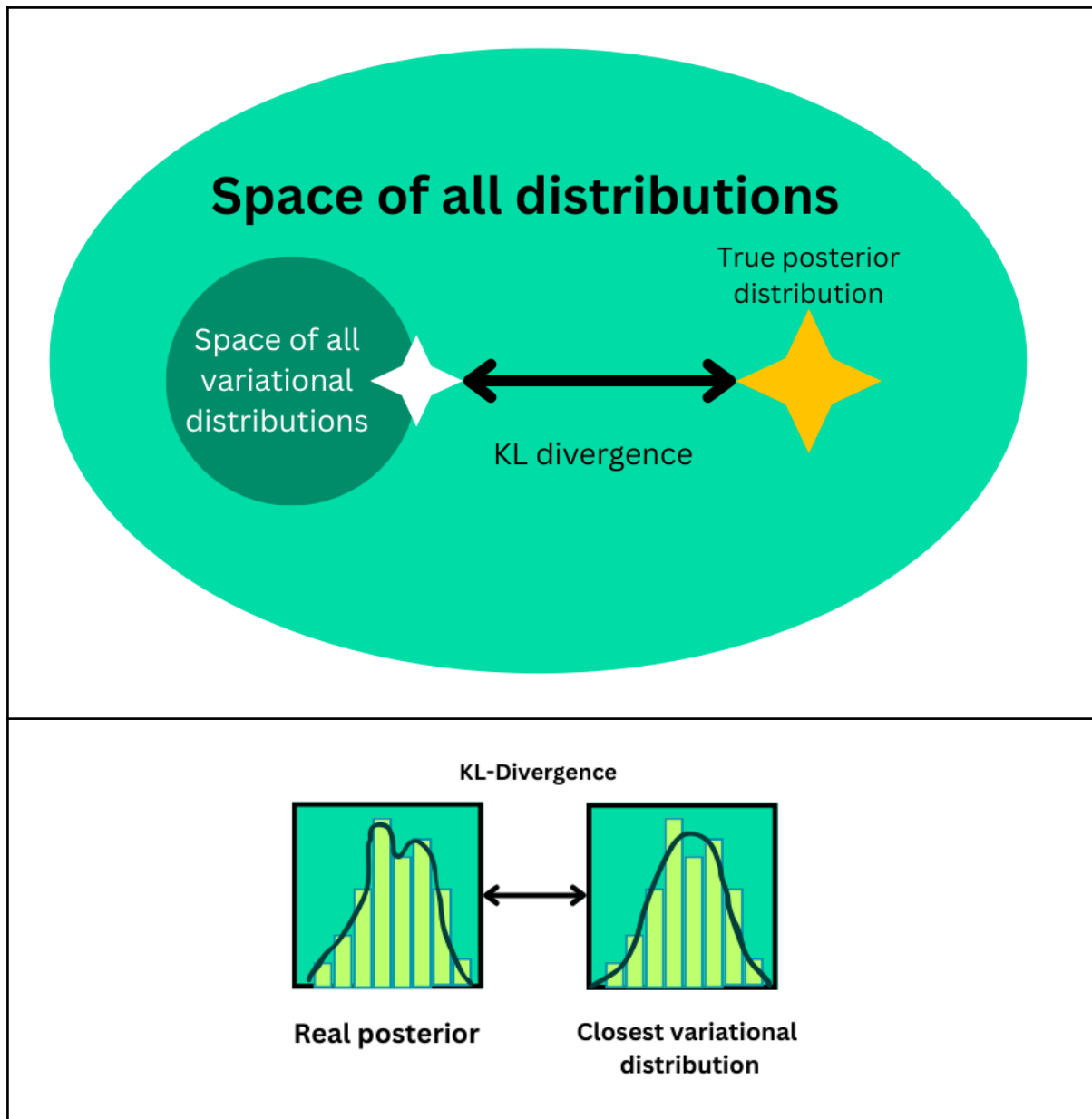


Figure 9.1

In variational Bayes we don't work out the actual posterior but instead we use a variational distribution structure that is easier to sample from in place of the real posterior.

Variational distribution

- We use a simple distribution that we are familiar with that can approximate the posterior
- This can be specified by only a few parameters
- Usually we use the normal distribution

In practice we use the normal distribution as our variational distribution so this means all of our prior distributions for our parameters are normal and all our posterior distributions are also normal.

We have a term called KL-divergence (Kullback–Leibler divergence) which measures the difference between the true posterior and the variational distribution we are using as shown in figure 9.1. By minimising this we obtain optimal values for the variational distribution and approximate the true posterior.

The KL-divergence (Kullback-Liebler-divergence)

$$D_{\text{KL}}(q_{\phi}||P) = \int_{\theta} q_{\phi}(\theta) \log \left(\frac{q_{\phi}(\theta)}{P(\theta|x)} \right) d\theta$$

$q_{\phi}(\Theta)$	The variational distribution	The simpler distribution which we are trying to approximate the posterior with
$P(\Theta x)$	The true posterior distribution	The actual posterior distribution of the parameters given the data
Θ	The stochastic parameters in the model	The parameters which we are trying to fit using the model

Figure 9.2

As we can see in the formula for KL-divergence as in figure 9.2 we have to calculate the true posterior distribution in order to calculate and minimise the KL-divergence. This is an issue as the true posterior is what we want to obtain. We get around this by maximising a new value known as the ELBO.

The ELBO (Evidence Lower Bound)

$$\int_{\theta} q_{\phi}(\theta) \log \left(\frac{P(\theta, x)}{q_{\phi}(\theta)} \right) d\theta = \log(P(x)) - D_{\text{KL}}(q_{\phi}||P)$$

$P(x)$	Evidence/ Data / Marginal likelihood	The probability of observing the data x regardless of the parameter values
$P(\Theta, x)$	Joint distribution of the data and the parameters	Note: Can be easily calculated from the likelihood and the prior of the parameters

Figure 9.3

Maximising the ELBO shown in figure 9.3 is equivalent to minimising the KL-divergence as $\log(P(x))$ only depends on the prior. The ELBO provides the tractable difference between our KL-divergence and the log-likelihood of the data.

Then if we want to train on more data and we are using variational inference with a normal distribution to train our model then we can initialise the new prior distributions in our Bayesian neural network as normal distributions with the values we already trained.

This has accuracy issues when the true posterior for these values isn't normally distributed or has a radically different distribution because in these cases the constraints of the type of distribution used can create inaccuracies.

Bayes by Backpropagation

Variational inference is often implemented in a process called Bayes by backpropagation when using deep learning. This is most likely how variational inference is carried out in both torchbnn and Pyro implementations due to the fact that it is a good way to efficiently perform variational inference. It offers the most easy to comprehend method of doing variational inference as it is a close replication of how traditional backpropagation works as described in the section on neural networks.

Algorithm 5 Bayes-by-backprop algorithm.

```

 $\phi = \phi_0;$ 
for  $i = 0$  to  $N$  do
  Draw  $\varepsilon \sim q(\varepsilon);$ 
   $\theta = t(\varepsilon, \phi);$ 
   $f(\theta, \phi) = \log(q_\phi(\theta)) - \log(p(D_y | D_x, \theta)p(\theta));$ 
   $\Delta_\phi f = \text{backprop}_\phi(f);$ 
   $\phi = \phi - \alpha \Delta_\phi f;$ 
end for

```

Φ	Variational parameters (usually the mean and standard deviation of the normal distribution we are fitting)
$\mathcal{E} \sim q(\mathcal{E})$	Our random variable drawn from a source of noise
$\Theta = t(\mathcal{E}, \Phi)$	Our value sampled from the current prior parameter distribution defined by our variational parameters Φ and obtained deterministically from \mathcal{E}
$f(\Theta, \Phi)$	The estimate of the ELBO from a single sample
$\Delta_\phi f$	The gradient of how the ELBO changes based on changing our variational parameters
α	The learning rate of how much the variational parameters are adjusted with each iteration

Figure 9.4 [J1]

Variational parameters

- The parameters required to fully specify the variational distribution we are using
- Usually can just be understood as the mean and variance (or mean and standard deviation) of the normal distribution

In order for traditional backpropagation to work we need the process to be deterministic in order to calculate the gradient of our parameters with respect to the loss we are using.

In order to make Bayes by backpropagation deterministic, instead of sampling from our prior parameter distributions directly we instead use a random variable \mathcal{E} as a source of noise independent of the process. Thus even though \mathcal{E} changes, we can have other parameters which do not change which we can use to fit the parameter distribution. For example if our distribution values are given by $\theta = \mu + \sigma * \mathcal{E}$ then we can still calculate the gradient of the ELBO with respect to μ and σ and fit them disregarding the noise and use this to fit θ .

The algorithm can be briefly described as simply performing something akin to backpropagation where we adjust the variational parameters Φ (can be understood as the mean and standard deviation of the normal distribution so μ and σ) based on calculating the ELBO on each particular iteration of draws from our current prior distribution $P(\Theta)$ and using the gradient of the ELBO to adjust the variational parameters as just described with μ and σ . The amount to which we change the variational parameters at each iteration is based on the size of the learning rate α .

The algorithm used here and the explanations were lifted from a tutorial on Bayesian neural networks and supplemented with my own understanding for the sake of clarity. [J1]

It is important to note that although I know that the inner workings of both torchbnn and Pyro implementations of a Bayesian neural network use an Adam optimiser when training and perform some form of variational inference, it is not altogether clear that they are using Bayes by backpropagation over a different kind of implementation. However it is one of the most intuitive and applicable ways to implement variational inference on a neural network.

10. What is Dropout?

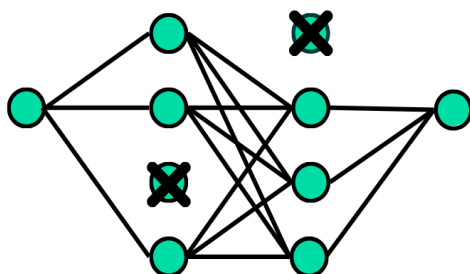


Figure 10.1

Dropout is a form of regularisation for a neural network which can be most straightforwardly understood as simply removing nodes from the neural network at each training or inference

iteration with each node being removed with a certain probability. This can be carried out during both training and testing steps with different consequences.

A more probabilistic explanation is that we can understand dropout as applying a certain kind of Bernoulli noise to the nodes in each layer causing them to be removed with a particular probability. Bernoulli noise indicates that the key nodes will be removed from the network in a binary way (either removed or not removed) with a particular probability.

We can also perform this with Gaussian noise and get Gaussian dropout. This means that instead of completely disappearing, the strength of the node's contribution is varied with a certain probability. [J1]

Dropout works as a form of regularisation as it simplifies the network but it is different from regularisation by introducing a term to the loss to ensure model simplicity. [J1] Dropout regularisation works by using dropout during training in order to make the neural network more robust to being able to provide a good prediction with missing nodes.

Usually we just use dropout during training but if we leave it on when we take samples from the distribution it can provide even a standard non-Bayesian neural network with an uncertainty distribution. [J1] It can be seen as a part of the posterior over all the data.

11. Torchbnn Implementation

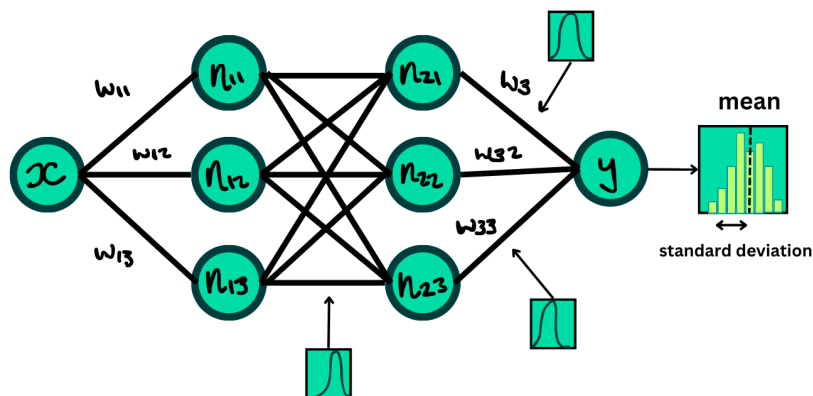


Figure 11.1

Here it is explained how the particular torchbnn implementation of a Bayesian neural network fits together with the concepts we have been discussing.

Model Structure and Prior Selection

As shown in figure 11.1 the type of Bayesian neural network used in my torchbnn implementation is a acyclic feedforward neural network with the weights set initially as

stochastic variables with a normal distribution prior. The prior for every weight is set specifically as the normal distribution $N(0, 0.1^2)$ which has a 0 mean. This is because we don't know anything about the weights initially. Therefore as not to influence whether the weight is positive or negative then 0 is the obvious choice.

A small standard deviation of 0.1 was used for multiple reasons. Firstly, in the torchbnn regression tutorial included with the library this standard deviation is used [G3], it was used in another tutorial using torchbnn [J2] and it was also used as the standard deviation value in the tutorial "From theory to practice with Bayesian neural network, using Python" which I modified for my implementation. [P2]

I also found that increasing this standard deviation to 0.2 or 1 meant that the model struggled to fit to the data and gave a very under confident uncertainty boundary for where the test data would lie. It was difficult to understand why this occurred. I guessed that if there is too large a standard deviation in the prior distribution indicating large uncertainty then optimising the parameters would require far more training steps to overcome the uncertainty. Thus as default I stuck to this selection for standard deviation as it gave me well fitting results.

Given I had 10000 training data points the choice of an ideal prior became less essential as with the large amount of data the prior selection was more important to ensure the model would be able to converge to a distribution representing the data over a great many training steps as the data is expected to overwhelm whatever initial prior we set. As I already knew the underlying mean function and expected uncertainty then whether the model fit the data would be obvious.

Model Training

Although there is no clear indication of exactly what training methods were used in the documentation for torchbnn, it is clear from the structuring of the code that variational inference combined with a form of optimization using the Adam optimizer is used to train the Bayesian neural network. This combination indicates that it is likely that variational inference is implemented using the Bayes by backpropagation (Bayes-by-backprop) algorithm discussed in the section on Variational Inference.

Given the training loop for Bayes-by-backprop is very similar to any traditional gradient descent training loop used for training a non-stochastic neural network, then it is acceptable to use the Adam optimizer as a replacement for other forms of gradient descent when performing Bayes-by-backprop. [J1]

We also implemented dropout with a certain percentage on the nodes of the network during training in some cases.

Model Inference

Inference including a probability distribution is performed by running the model multiple times and then calculating the mean and standard deviation of the responses. The means are then used as a mean function of the output distribution and the standard deviations are used to construct a boundary containing 95% of the data by creating an uncertainty boundary between the lower limit: **mean - 2 standard deviations** and the upper limit: **mean + 2 standard deviations**.

This was continued based on the usage in the torchbnn regression tutorial [G3] and the other tutorial making use of torchbnn to construct a Bayesian neural network. [P2]

When dropout was used during training of the network we would also use dropout during inference so that nodes would be removed according to the probability of removal at each iteration. This would add increased uncertainty when creating the probability distribution.

Further Details

The torchbnn library was used for this implementation. [G4] It cites a paper "GradDiv: Adversarial Robustness of Randomized Neural Networks via Gradient Diversity Regularization" indicating that it was designed in order to protect from adversarial attacks when using neural networks by using a Bayesian neural network. [L1]

Adversarial attacks

- In the context of neural networks this refers to when a neural network (easily understood if we consider an image detection neural network) is given examples which look normal to humans but are designed to fool the neural network into making a mistake.

12. Pyro Implementation

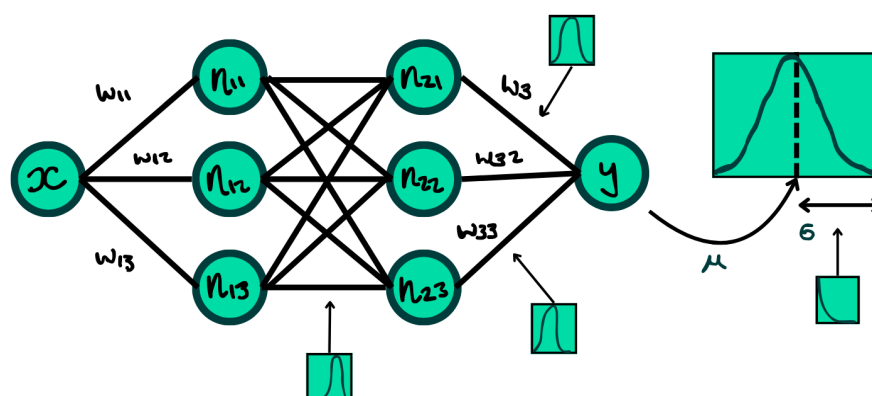


Figure 12.1

Our weights which are normally distributed	The mean of output distribution	The standard deviation of our output normal distribution modelled from a Gamma distribution	Our output distribution
$w_{ij}/w_{ijk} \sim \mathcal{N}(0, 0.1^2)$	$\mu = y$	$\sigma \sim \text{Gamma}(0.5, 1)$	$\mathcal{N}(\mu, \sigma^2)$

Figure 12.2

This section explains how the Pyro implementation [I2] aligns with the concepts of a Bayesian neural network, variational inference and MCMC as discussed in previous sections.

Model Structure and Prior Selection

As shown in figure 12.1 our Pyro implementation of a Bayesian neural network also has the weights as stochastic variables with a normal distribution prior. In fact they have the same prior as in the torchbnn implementation of the normal distribution $\mathcal{N}(0, 0.1^2)$ for similar reasons. Essentially the prior choice was sufficient for the data to fit well and provided some similarity to aid comparison with the torchbnn version.

However our forward propagation process is different in this case. Instead of simply averaging over these distributions to get probabilistic output, we instead feed the output value (y in the diagram) into an output normal distribution as its mean value. Essentially the structure of the Bayesian neural network used in the torchbnn implementation is the same but the Pyro implementation is extended with an additional normal distribution at the end.

For the standard deviation of our output normal distribution we draw from a prior gamma distribution $\text{Gamma}(0.5, 1)$ which indicates that the expected value is close to 1. Then we draw our output value from this output normal distribution which has been generated slightly differently with each iteration. This is following the Pyro tutorial on how to make a Bayesian neural network with some adjustments. [A1]

More clearly we can think of this structure as instead of simply averaging over all possible predictions of the neural network given the adjusted distributions of the weight parameters, we instead introduce a new parameter for the standard deviation of our output which can also be adjusted. Intuitively this may lead to being able to capture uncertainty better as getting an accurate standard deviation model is something we train for with Bayesian explicitly. However it is also noticeable that this indicates there will be no interaction between our output normal distribution standard deviation value (σ in the diagram) and our input value (x in the diagram).

Forward propagation process

- The steps that are taken to get an output from the neural network for a given input as defined previously particular in the section on neural networks

Essentially we can think of our final overall prior distribution as being a complex function with a bunch of different parameters fed into it.

Model Training

We can train this distribution using either MCMC or variational inference. However it is important to note that the mechanisms for these two processes are different and this comes with different consequences in how well the model fits the data and ease of model retraining.

When we use variational inference we have to have all of the parameters be a normal distribution so the Gamma distribution which is the prior distribution for σ must be turned into a normal distribution also.

The variational inference model is trained using a `AutoDiagonalNormal` mean field guide using a process which involves stochastic variational inference using the Adam optimiser which has been explained more in the section on variational inference.

As we are simply adjusting a number of normal distributions for each of the stochastic variables including all of the weights and the parameter σ then each of these normal distributions is fully specified when only knowing a value for the mean and for the variance.

When we train using MCMC our sets of parameter samples are from the posterior distribution but otherwise they do not have to follow a normal distribution nor any sort of smooth known distribution. This means that the model can fit the real posterior exactly once the process of MCMC has converged but it means that it may take a long time to train and given there are limited samples the result may be somewhat erratic.

Model Inference

This section was surmised through knowledge of these processes and reading of the source code for the Pyro class Preferences. [S2]

Model inference looks fairly similar for both MCMC and variational inference as they both use the function from Pyro called `Predictive` but in fact the way that predictions are made for our input values is very different.

After running our MCMC training we have a bunch of posterior samples collected from this process. We then go over each of these samples, apply those parameters to the original model and run the model process. This is called conditioning on the posterior samples and can be essentially understood as forward propagation over the model structure with a stochastic component. This conditioning process runs the neural network with the posterior samples to produce the mean of our output normal distribution, uses a posterior sample for the standard deviation of our new distribution σ sampling and samples in a stochastic way from the output normal distribution defined by these two parameters.

Given the conditioning process includes our input value we are able average over our many posterior samples for each of the test data inputs. This gives us our distribution capturing uncertainty based on our training values over our test values.

In variational inference instead of having these sets of posterior samples, we instead have fit a posterior normal distribution for the stochastic weight parameters in the Bayesian neural network and the stochastic parameter for σ . Thus we can simply run the model in a similar way as with MCMC but instead make draws from these normal distributions for each sample. Thus we can get predictions for each of our test data points based on sets of draws.

We can see that both of these processes involve using values for the stochastic parameters in our Bayesian neural network model to get our predictions. However it is notable that with variational inference our limited normal posterior distributions are known directly whereas with MCMC we only have a set of samples from the posterior distributions.

Further Details

Given the differences between MCMC and variational inference in the way that the model is trained in the Pyro implementation, it is clear why it is easier to update the variational Bayes model with new parameters. This is because what we get from training is the mean and standard deviation of each of the posterior distributions of our key stochastic variables. As each of these distributions as normal this is enough to completely define these distributions and so this small amount of data can be used to define prior distributions of these parameters before running the model again.

To contrast this, if we want to retrain an MCMC trained Bayesian neural network we must store all of the posterior samples and use these to help define the prior for the new model which is more time consuming as a process as they do not completely define the prior distributions with a few simple parameters. The saved posterior samples are also more difficult to store given the large number of samples.

All of this is demonstrated in the tutorial on training a Bayesian neural network with Pyro. [A1]

13. Results

Often the standard defaults for the graphs shown here have 2 hidden layers of 100 nodes with 4000 trained epochs and if dropout is used it is done with 10% probability that a node will be removed from the network during both training and inference. This is as I found that this worked best when working on the torchbnn implementation and so where possible this was continued.

The number of training steps or epochs used for each model and the size/shape of the neural network is variable between the methods used.

Here I am attempting to show what makes a good fit and how these different attributes influence the success of the fit for different methods. Thus as I am attempting to find the best trade off between training time, accuracy of mean function fit (measured by how small the MSE is) and accuracy of the coverage (the ideal coverage should be 0.95 as the uncertainty level is set to 95% certainty so it should cover 95% of the data). I mainly only included examples of successful or notable results from using each method and how changing parameters leads to better results.

Standard Neural Network in Pytorch

2 hidden layers of width 20

Epochs: 100

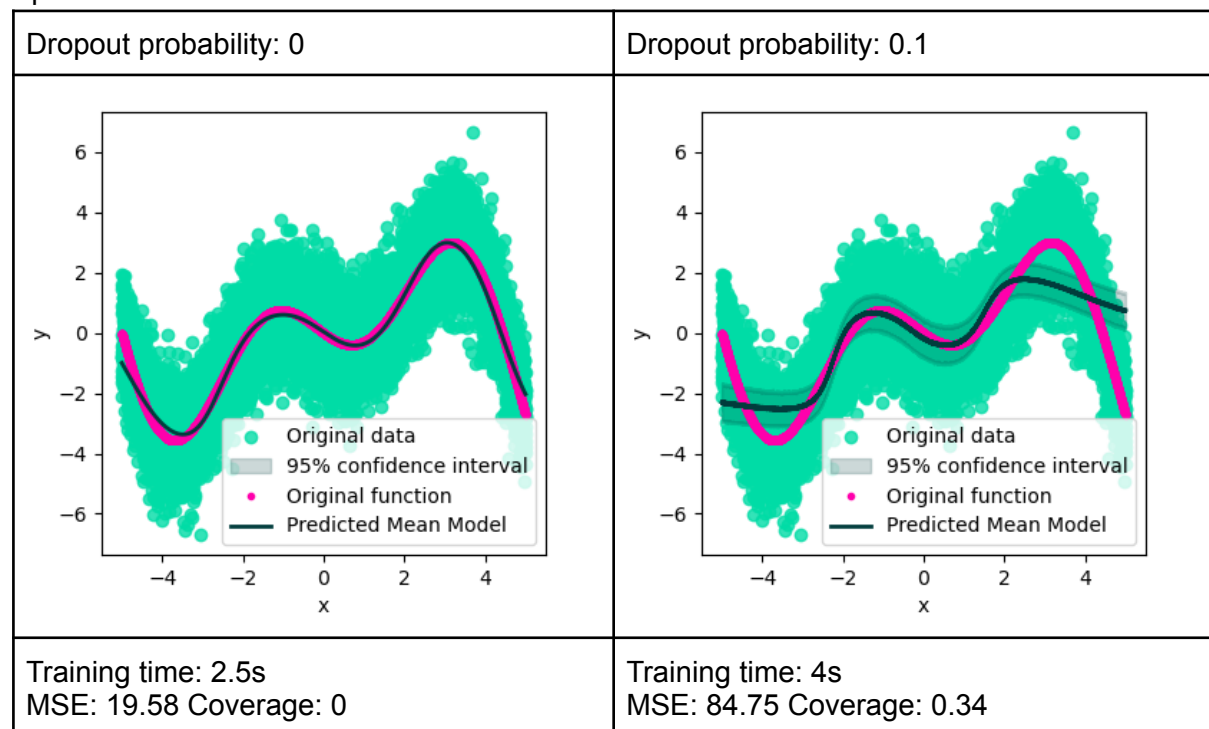


Figure 13.1

In figure 13.1 we can see that introducing 10% dropout in our network of 2 hidden layers with 100 nodes makes it more difficult to fit the data with the number of epochs we are using and increases the mean squared error. It does however introduce an uncertainty boundary from using dropout during prediction and averaging over lots of predictions made. This is able to capture some of the uncertainty in the data in the areas where the mean function fit is accurate.

Torchbnn Implementation of a Bayesian Neural Network Trained with Variational Inference

2 hidden layers of width 100

Epochs: 4000

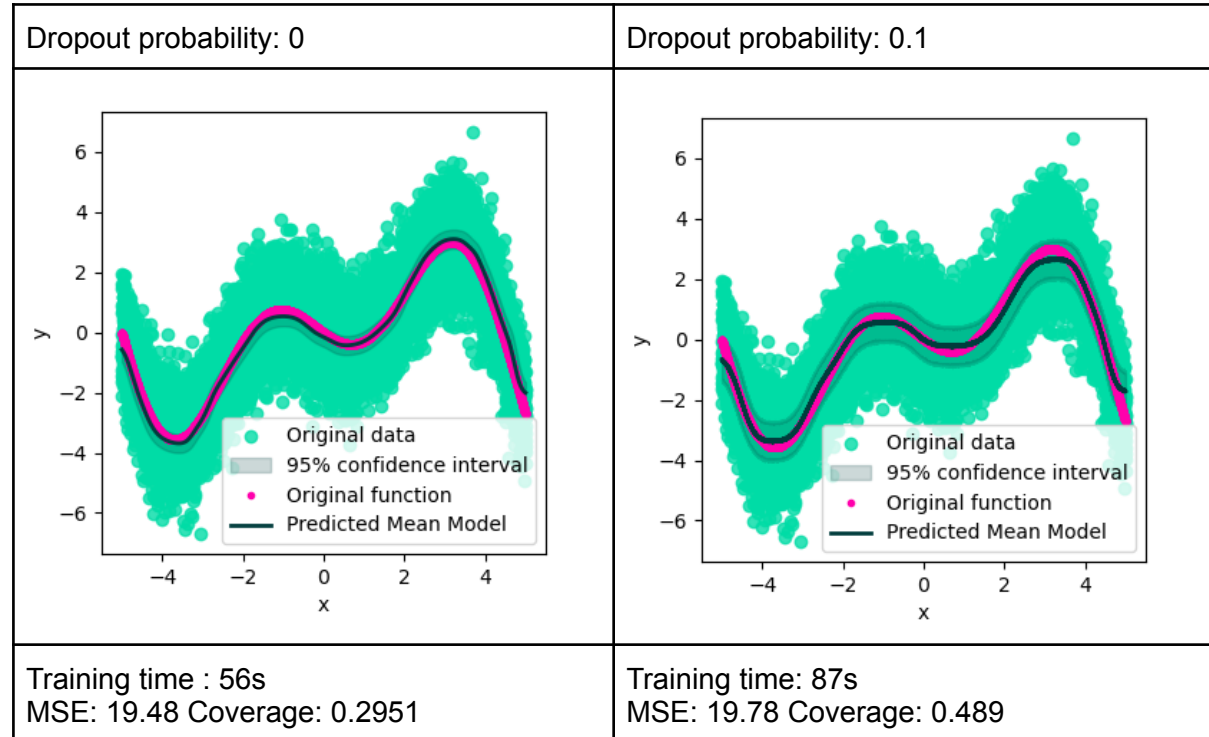


Figure 13.2

In figure 13.2 we can see that if we leave the dropout probability as 0 when using a torchbnn variational inference trained Bayesian neural network we are able to get an accurate fit but the uncertainty has poor coverage of only 0.29. By introducing dropout in this case the accuracy of the fit is mostly preserved but there is introduced an uncertainty boundary which captures some amount of the data but is still very inaccurate based on the desired coverage of 0.95.

Dropout probability: 0.1

Epochs: 4000

2 hidden layers with width 20	2 hidden layers with width 100
-------------------------------	--------------------------------

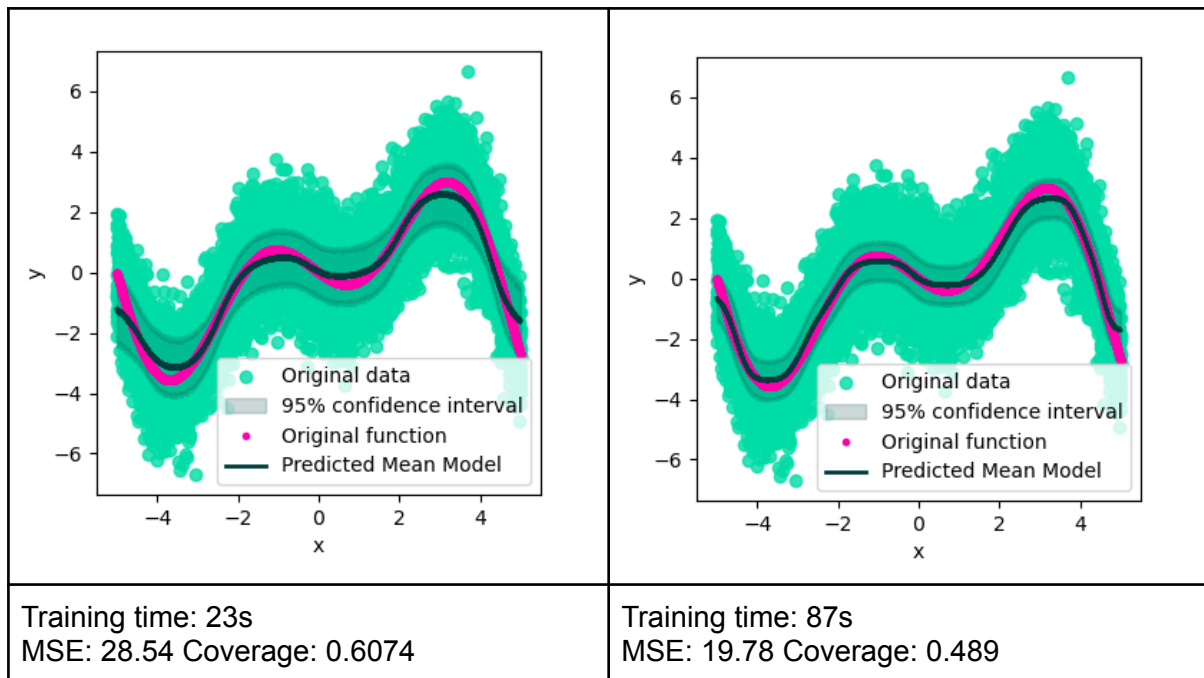


Figure 13.3

In figure 13.3 we can see that using wider hidden layers improves the accuracy of the mean function fit but in fact shrinks the coverage so it is even worse at capturing the uncertainty in the data.

Dropout probability: 0.1

Epochs: 4000

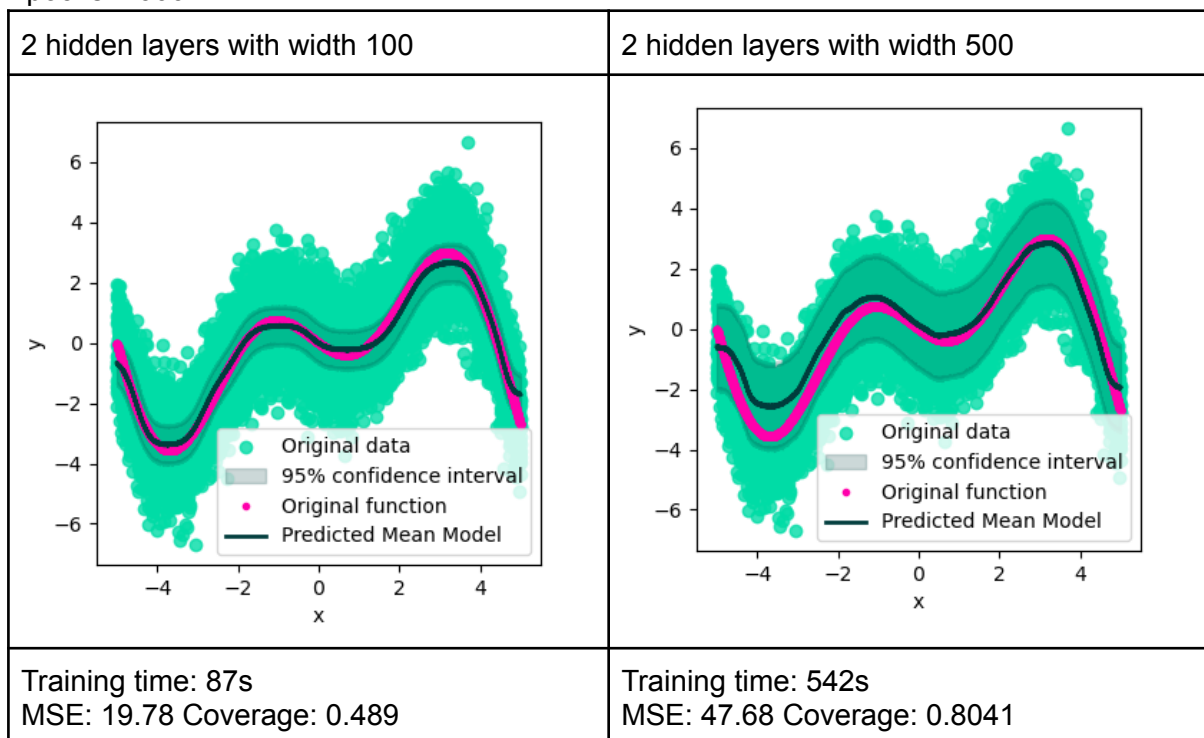


Figure 13.4

In figure 13.4 we see that by adding a huge number of nodes to the hidden layers we are able to get a better coverage but we sacrifice the accuracy of the mean function fit and it takes an extremely long time to train.

Pyro implementation of a Bayesian Neural Network Trained using Variational Inference

2 hidden layers with width 100

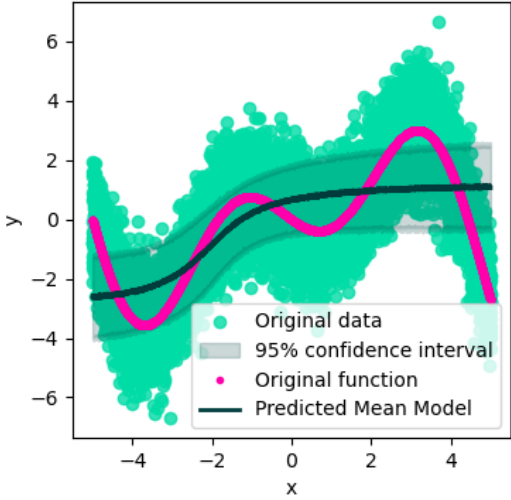
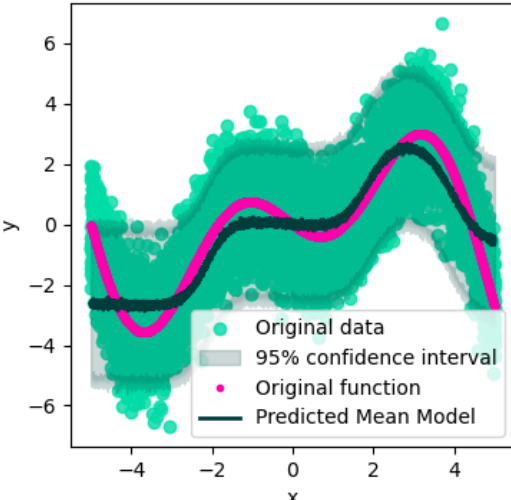
4000 training epochs (training time 66s)	20000 training epochs (training time 342s)
	
MSE: 115.85 Coverage: 0.6016	MSE: 67.97 Coverage: 0.9688

Figure 13.5

In figure 13.5 we see that using the usual 100 layers width of a neural network isn't able to get a successful fit at all with this implementation unless we drastically increase the training epochs and here although the coverage is good the model struggles to fit the underlying mean function particularly at the start and end of the data.

2 hidden layers with width 20 4000 training epochs (training time 37s)	2 hidden layers with width 100 20000 training epochs (training time 342s)
---	--

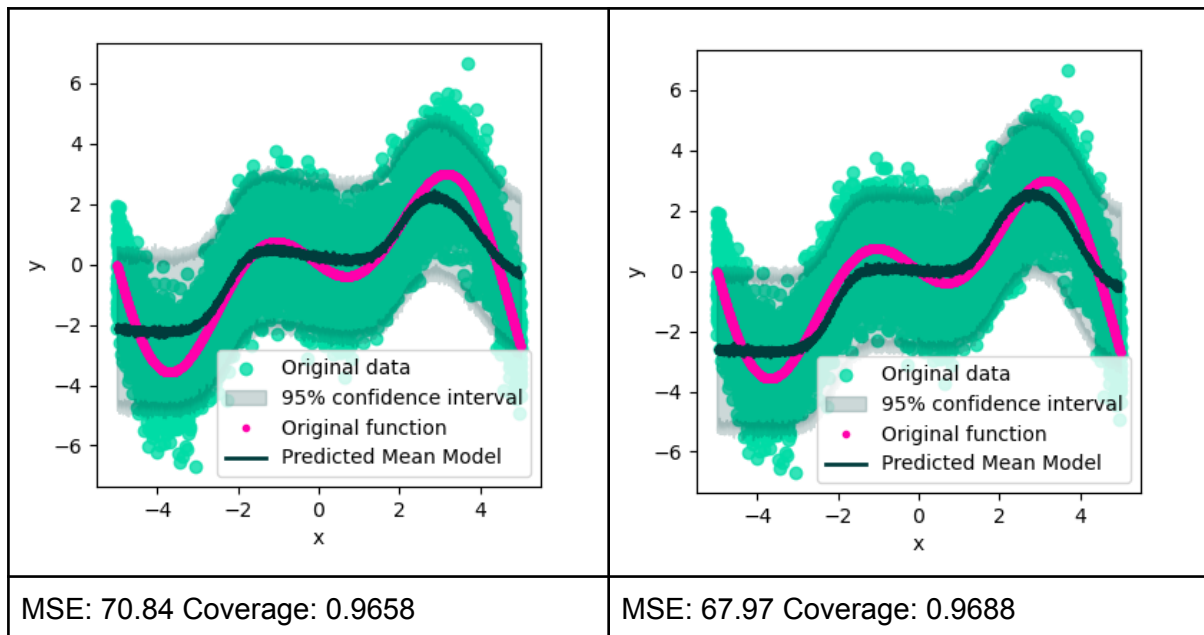
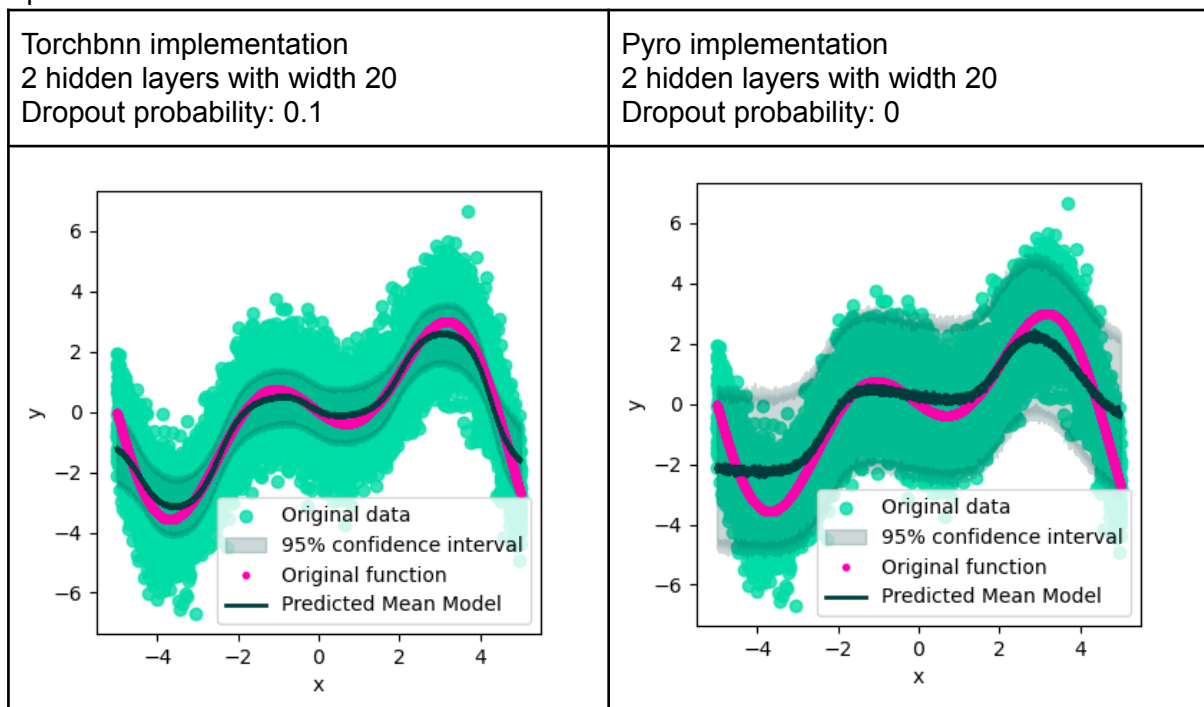


Figure 13.6

We can see in figure 13.6 that by reducing the width of the hidden layers and decreasing the number of training steps accordingly we are able to get a similar fit to using a huge number of training epochs and have roughly the same results both in terms of coverage and accuracy of mean function fit.

Torchbnn and Pyro Bayesian Neural Network Trained using Variational Inference

Epochs: 4000



Training time: 23s MSE: 28.54 Coverage: 0.6074	Training time: 37s MSE: 70.84 Coverage: 0.9658
---	---

Figure 13.7

We can see key differences between the two implementations. Given roughly the same training time and number of epochs we find that the torchbnn implementation fits the mean function significantly more accurately but that the coverage is a poor reflection of the uncertainty in the data. In contrast, the Pyro implementation struggles to fit the mean function accurately throughout but it has a good reflection of uncertainty in the data with the coverage.

Pyro Implementation of a Bayesian Neural Network Trained using MCMC

Burn in steps: 50

Posterior sample steps: 50

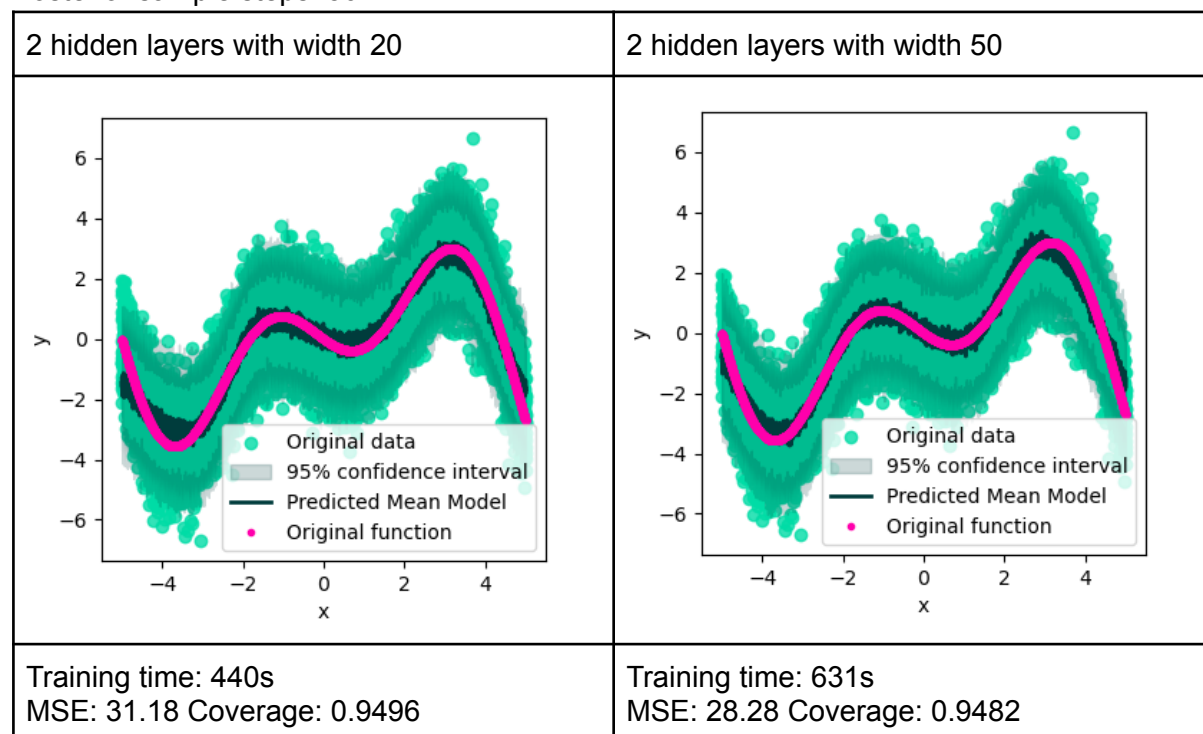


Figure 13.8

Here we can see that the width of the hidden layers of the neural network has a noticeable effect on the accuracy of the mean function fit and has a moderate effect on training time. Due to the noisiness of the fit the mean squared error is similar between the two plots but given we can adjust for and smooth this noise the second option visually is significantly closer to the true function. The coverage of both is accurate to capturing roughly 95% of the data.

The Pyro implementation of a Bayesian neural network is the best out of the options surveyed for having both an accurate uncertainty boundary and a fairly accurate mean function prediction.

GPtorch Implementation of a Gaussian Process

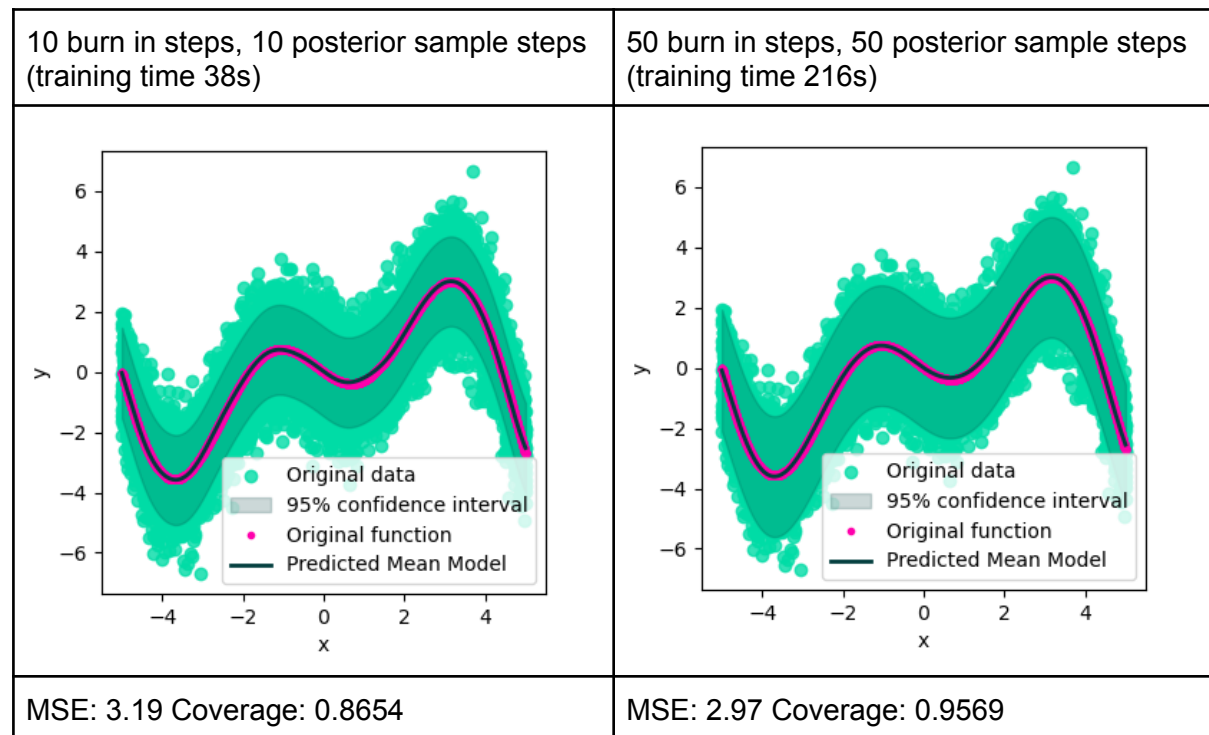


Figure 13.9

We can see that the model is overconfident with insufficient coverage for a lower number of MCMC training epochs but is still able to fit the mean function with a very low mean squared error when requiring only a fraction of the training time. It's significant to note that the coverage is still fairly large and captures 86% of the data given when fewer iterations are used on the MCMC training method.

It is also interesting to note that having insufficient training epochs lead only to underconfidence and not to overconfidence in the uncertainty interval.

Comparison of Best Methods

MCMC Bayesian neural network with 50 burn in steps, 50 posterior sample steps 2 hidden layers of width 20	Gaussian processes with 50 burn in steps, 50 posterior sample steps
--	---

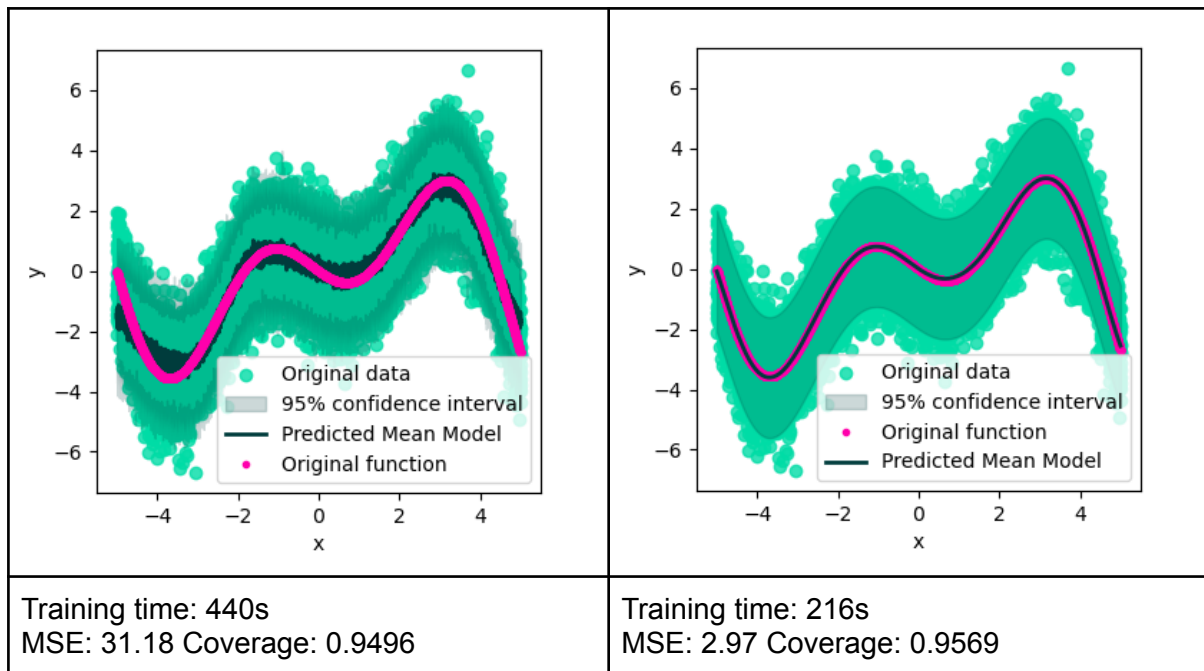


Figure 13.10

As we can see, if we pick the versions of a Bayesian neural network and Gaussian process that get close to accurate coverage of 95% of the data then the training time of the Gaussian process is about half that of the Bayesian neural network. We can also see that the Gaussian process predicts the function and uncertainty boundaries as smooth which may be advantageous when working with a smooth underlying function.

We can see that the Bayesian neural network has a higher mean squared error with respect to the ability to fit a mean function and in particular struggles to fit the function at the boundaries near $x=-5$ and $x=5$ where the data ends.

Overall the Gaussian process performs better in terms of both training time and accuracy and they both have fairly accurate coverage of the uncertainty in the data.

14. Conclusion

Limitations

It is important to note the limitation of any conclusions I am making here. Some of the conclusions are speculative due to the difficulty of perfectly combining my understanding of the underlying Bayesian neural networks with their implementations.

In addition any conclusions are limited based on the fact that I used only one type of function and that that function was one-dimensional so if trained on more complex function shapes or higher dimensions the results found may change.

Torchbnn vs Pyro Implementation

The difference between these implementations is mainly the adding of an additional parameter which is just used for capturing the expected error in the data (the stochastic variable for the standard deviation). This means that the uncertainty in the data is captured better in the Pyro distribution with both the variational inference and MCMC versions because we are specifically interested in fitting the model to capture it.

We can see that adding an additional parameter to fit may cause issues with being able to easily fit the mean function in the data. This is seen by the mean function fit being worse in the Pyro implementation using variational inference compared to the torchbnn implementation using variational inference. However there may also be underlying differences in the way that Bayesian inference is carried out between the two libraries which I wasn't able to discover.

It is interesting that the torchbnn implementation seems to capture uncertainty in terms of the model's ability to fit the mean function accurately. Alternatively the Pyro implementation is good at capturing the error in the model's ability to accurately predict the test data itself.

Pyro Variational Inference vs MCMC

MCMC clearly is better at capturing the posterior distribution. This is by design as it allows us to sample from the actual posterior rather than a variational derivative. Thus when we have a more complex distribution shape than the standard normal distribution this can be captured.

Due to the limited number of posterior samples the Pyro MCMC implementation is more noisy but it also fits better than the variational inference version. This is both in terms of the mean function fit having a lower mean squared error and the coverage being closer to that of the training data. However, it also took much longer to fit the Bayesian neural network in the attempts made. This is about as we would expect given variational inference is designed as a more efficient process.

Dropout Effect

From the examples with the standard neural network and the torchbnn implementation the effect of using dropout during both training and prediction can be observed. While it introduces some uncertainty in the predictions based on altering the structure, it causes the mean function of the data to fit more poorly because we are removing some of the key information during the step when we are trying to make predictions.

If I had time to try more configurations I would be interested in whether the dropout method had a more positive effect on the fit of the data if it was used only during training and not used to contribute to the uncertainty by continuing during the prediction stage.

It appears that while dropout technically is an option for adding some Bayesian aspects to a neural network, doing so appears to lead to a worse mean function fit at least in some cases.

Pyro MCMC vs Gaussian Process

The Pyro MCMC actually took a longer time to fit than a Gaussian Process but was able to produce something which visually structurally appears to be capable of becoming a Gaussian process shape with more iterations. This fits with the concept that with enough nodes and complexity that a Bayesian Neural Network can become a Gaussian process. [J1]

Therefore this shows promising results of being able to approximate the Gaussian process with a Bayesian Neural Network using MCMC.

We can notice that with the Pyro MCMC that both the mean function and uncertainty boundary are incredibly noisy. This can be partially explained as we make a different draw from a different output normal distribution each time we make a prediction so there is a lot of variability. Also when we use MCMC we make limited draws from the posterior this causes our stochastic parameter samples to be a limited number of somewhat randomised draws. This can lead to additional noise as opposed to using a variational distribution where we draw parameters from a smooth normal distribution.

However visually we can see that the ability of Pyro MCMC to fit the mean function of the data in general appears fairly accurate so there is some scope for using function smoothing after the fact to improve the fit.

Considering our results it appears to be plausible that with some adjustment a version of a Bayesian neural network trained using MCMC might be created which replicates a Gaussian process in a way that is more efficient with a reasonable sacrifice for accuracy of smooth continuous function fitting.

However currently the Gaussian process provides a noticeably better fit in terms of both the mean function and the uncertainty coverage than any Bayesian neural network model used. In addition to this Gaussian processes take far less time to train than a high accuracy Bayesian neural network trained using MCMC from the Pyro implementation.

In addition, any improvements in the efficiency of MCMC benefit both Bayesian neural networks and Gaussian processes equally so it is possible that Gaussian processes may become more efficient in time with more research into MCMC. Also, there may be alternatives to MCMC for fitting Gaussian processes more efficiently.

Further Adjustments

It is possible that efficiency could be increased by using a format where only the final nodes are given distributions on the weights. Therefore instead of having a large, complex, difficult to fit structure which gives us the posterior we have a big “prior structure” instead which we train using the data and then employ stochastic parameters towards the end of our network.

If given more time to extend this project I would have investigated the use of TyXe, another Python library built on top of Pyro. This in order to have a better framework for how to create Bayesian neural networks in a more standardised way. This might allow different configurations to be tried without as many low level details.

Alternatively I could also explore different structures of creating a Bayesian neural network in Pyro, especially trying to find the most efficient Bayesian neural network format which would give results comparable to the Pyro implementation I used. I might also look into using a smoothing function to improve the accuracy of the noisy Pyro MCMC output.

I would also be interested in trying out more different kinds of multidimensional functions for generating training data. Due to the long time that it took to train the current Bayesian neural networks and Gaussian processes only one type of one-dimensional function was studied in the end.

15. References

Historical

- [N1] Neal, R.M. (2011) 'MCMC Using Hamiltonian Dynamics', *Handbook of Markov Chain Monte Carlo*. Chapman & Hall/CRC, pp. 113–162. Available at: <http://www.mcmchandbook.net/HandbookChapter5.pdf>.
- [M1] Metropolis, N. et al. (1953) 'Equation of state calculations by fast computing machines.' *Journal of Chemical Physics*, 21:1087–1092.
- [I1] Ider, B. J. and Wainwright, T. E. (1959) 'Studies in molecular dynamics. I. General method.' *Journal of Chemical Physics*, 31:459–466.
- [D1] Duane, S. et al. (1987) 'Hybrid Monte Carlo.' *Physics Letters B*, 195:216–222.
- [H2] Hoffman, M.D. and Gelman, A. (2014) 'The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo', *Journal of Machine Learning Research*, 15. Available at: <https://stat.columbia.edu/~gelman/research/published/nuts.pdf>.
- [K1] Kingma, D. and Ba, J. (2014). 'Adam: A Method for Stochastic Optimization. Computer Science.' [online] doi:<https://doi.org/10.48550/arXiv.1412.6980>.

Tutorials Used

- [A1] Auzina, I.A. et al. (2023) 'Tutorial 1: Bayesian Neural Networks with pyro', *Tutorial 1: Bayesian Neural Networks with Pyro - UvA DL Notebooks v1.2 documentation*. Available at: https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/DL2/Bayesian_Neural_Networks/dl2_bnn_tut1_students_with_answers.html (Accessed: 19 April 2024).
- [P2] Paialunga, P. (2022) 'From theory to practice with Bayesian neural network, using Python', *Towards Data Science*. Available at: <https://towardsdatascience.com/from-theory-to-practice-with-bayesian-neural-network-using-python-9262b611b825> (Accessed: 16 December 2023).

- [G1] 'GPyTorch regression tutorial' (2023) *GPyTorch Regression Tutorial - GPyTorch 1.12.dev26+g58c0335 documentation*. Available at: https://docs.gpytorch.ai/en/latest/examples/01_Exact_GPs/Simple_GP_Regression.html (Accessed: 16 December 2023).
- [G3] Gold, H. (2020) 'Demo - Bayesian Neural Network Regression', *GitHub*. Available at: <https://github.com/Harry24k/bayesian-neural-network-pytorch/blob/master/demos/Bayesian%20Neural%20Network%20Regression.ipynb> (Accessed: 11 May 2024).

Key Explanations

- [S1] Sanderson, G. (2017) 'But what is a neural network? | Chapter 1, Deep Learning', *YouTube*. Available at: <https://www.youtube.com/watch?v=aircAruvnKk> (Accessed: 16 December 2023).
- [S2] Sanderson, G. (2017b) 'Gradient descent, how neural networks learn | Chapter 2, Deep Learning', *YouTube*. Available at: <https://www.youtube.com/watch?v=IHZwWFHWa-w> (Accessed: 16 December 2023).
- [R1] Rahman, A. (2023) 'Neural network is nothing but a linear regression', *Medium*. Available at: <https://medium.com/@asifurrahmanaust/lesson-3-neural-network-is-nothing-but-a-linear-regression-e05a328a0f23> (Accessed: 16 December 2023).
- [R2] Raina, V. (2021) 'Bayesian neural network | deep learning', *YouTube*. Available at: <https://www.youtube.com/watch?v=OVne8jDKGUI> (Accessed: 16 December 2023).
- [M1] Ma, E.J. (2017) 'Eric J. Ma - an attempt at demystifying Bayesian deep learning', *YouTube*. Available at: <https://www.youtube.com/watch?v=s0S6HFdPtIA> (Accessed: 16 December 2023).
- [S3] Sanderson, G. (2019) 'Bayes theorem, the geometry of changing beliefs', *YouTube*. Available at: <https://www.youtube.com/watch?v=HZGCoVF3YvM> (Accessed: 16 December 2023).
- [J1] Jospin, L.V. *et al.* (2022) 'Hands-on Bayesian neural networks—a tutorial for Deep Learning Users', *IEEE Computational Intelligence Magazine*, 17(2), pp. 29–48. doi:10.1109/mci.2022.3155327.
- [D2] Davidson-Pilon, C. (2023) '*Bayesian Methods for Hackers*', *GitHub*. Available at: <https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers> (Accessed: 07 May 2024).
- [J2] Johnson, M. (2022) *Understanding a bayesian neural network: A tutorial*, *nnart*. Available at: <https://nnart.org/understanding-a-bayesian-neural-network-a-tutorial/> (Accessed: 09 May 2024).
- [N2] Navarro, D. (2023) 'The Metropolis-Hastings algorithm' Available at: https://blog.djnavarro.net/posts/2023-04-12_metropolis-hastings/ (Accessed: 15 May 2024).

Key Packages Used

- [G4] Gold, H. (2022) 'Harry24k/Bayesian-neural-network-pytorch: Pytorch implementation of bayesian neural network', *GitHub*. Available at:

<https://github.com/Harry24k/bayesian-neural-network-pytorch> (Accessed: 11 May 2024).

- [L1] Lee, S., Kim, H. and Lee, J. (2023) 'GradDiv: Adversarial robustness of randomized neural networks via gradient diversity regularization', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(2), pp. 2645–2651. doi:10.1109/tpami.2022.3169217.
- [I2] 'Introduction to Pyro - Pyro Tutorials 1.8.6 documentation.' Available at: https://pyro.ai/examples/intro_long.html (Accessed: 16 December 2023).
- [G2] 'Gpytorch.mlls' (2023) 'gpytorch.mlls - GPyTorch documentation.' Available at: https://docs.gpytorch.ai/en/stable/marginal_log_likelihoods.html (Accessed: 17 December 2023).

Supplementary

- [V1] Vigen, T. (2024) 'About spurious correlations', *Tyler Vigen's personal website*. Available at: <https://tylervigen.com/about-spurious-correlations> (Accessed: 08 May 2024).
- [S2] 'Source code for pyro.infer.predictive pyro.infer.predictive - Pyro documentation.' Available at: https://docs.pyro.ai/en/dev/_modules/pyro/infer/predictive.html (Accessed: 12 May 2024).
- Kingma, D. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. Computer Science. [online] doi:<https://doi.org/10.48550/arXiv.1412.6980>.

16. Code

The code can be found on GitHub at:

https://github.com/Suzystar3/Uncertainty_Quantification/tree/main

This may be useful for easy review and download for anyone interested in replicating the results.

compare.ipynb

```
import torch
import make_data
import run_models

def get_target_function(x):
    # True function is  $x * \sin(x + 5)$ 
    return x*torch.sin(x+5)
x, y = make_data.make_initial_data(get_target_function)
x_test, y_test, y_perfect =
make_data.make_test_data(get_target_function)
```

```

reused_params = {"x": x, "y": y, "x_test": x_test, "y_test": y_test,
                 "y_perfect": y_perfect,
                 "function_type": "xsin(x+5)", "figsize": (4, 4),
                 "samples": 500}

param_set = [
    {"model_type": "bnn_variational_pyro", "dropout_prob": 0,
     "epochs": 20000, "samples": 500, "width": 100, "depth": 2},
]
for set in param_set:
    set.update(reused_params)
for set in param_set:
    run_models.run_model(**set)

```

make_data.py

```

import torch

def make_initial_data(get_target_function, start=-5, end=5,
                     noise_coeff=1):
    torch.manual_seed(42)

    x = torch.linspace(start, end, 10000)
    y = get_target_function(x) + noise_coeff*torch.normal(0, 1,
size=x.size())
    return x, y

def make_test_data(get_target_function, start=-5, end=5, noise_coeff=1):
    torch.manual_seed(66)
    x_test = torch.linspace(start, end, 10000)
    y_test = get_target_function(x_test) + noise_coeff*torch.normal(0,
1, size=x_test.size())
    y_perfect = get_target_function(x_test)

    x_test = torch.unsqueeze(x_test, dim=1)
    y_test = torch.unsqueeze(y_test, dim=1)
    y_perfect = torch.unsqueeze(y_perfect, dim=1)
    return x_test, y_test, y_perfect

```

run_models.py

```

import time
import gpytorch

```

```

import make_data
import eval_data
import bnn_bayesian
import pyro_bayesian
import nn_standard
import gaussian_process

def run_model(x, y, x_test, y_test, y_perfect,
model_type="bnn_variational", function_type=None,
            dropout_prob=0.1, epochs=100, samples=100, mu=0, sigma=0.1,
width=10, depth=2, reverse_plot=False, figsize=(10, 8)):
    filename =
(f"./Graphs/{model_type}_{function_type}_{width}_{depth}_{dropout_prob}_
{epochs}_{samples}_{mu}_{sigma}.png")
    title = f" ".join(model_type.split("_")).title() { "
".join(function_type.split("_")), Width: {width} Depth: {depth}
Dropout: {dropout_prob} Epochs: {epochs} Samples: {samples}\n"
    start = time.perf_counter()
    if model_type=="bnn_variational":
        model = bnn_bayesian.bnn_model(mu=mu, sigma=sigma,
dropout_prob=dropout_prob, width=width, depth=depth)
        bnn_bayesian.train_bnn_model(model, x, y, epochs=epochs)
        mean_values, std_values = bnn_bayesian.get_dist(model, x_test,
num_samples=samples)
    elif model_type=="bnn_MCMC":
        model = pyro_bayesian.BNN(hid_dim=width, n_hid_layers=depth)
        mcmc = pyro_bayesian.train_pyro_model(model, x, y,
num_samples=epochs)
        mean_values, std_values = pyro_bayesian.get_dist(model, mcmc,
x_test)
    elif model_type=="bnn_variational_pyro":
        model = pyro_bayesian.BNN(hid_dim=width, n_hid_layers=depth)
        model, mean_field_guide =
pyro_bayesian.train_pyro_mcmc_model(model, x, y, num_epochs=epochs)
        mean_values, std_values =
pyro_bayesian.get_dist_variational(model, x_test, mean_field_guide,
num_samples=samples)
    elif model_type=="standard_nn":
        model = nn_standard.NeuralNetwork(dropout_p=dropout_prob,
num_hidden_layers=depth, hidden_layer_size=width)
        nn_standard.train_nn_model(model, x, y, epochs=epochs)
        mean_values, std_values = nn_standard.get_dist(model, x_test,
num_samples=samples)
    elif model_type=="gaussian_process":
        likelihood = gpytorch.likelihoods.GaussianLikelihood()
        model = gaussian_process.ExactGPMModel(x, y, likelihood)

```

```

        # Epochs should be 100 as this model takes much longer and is very
        accurate
        gaussian_process.train_gp_model(model, x, y, likelihood,
epochs=epochs)
        mean_values, std_values = gaussian_process.get_dist(model, x_test,
likelihood)
        mean_values = mean_values.numpy()
        std_values = std_values.numpy()
        training_time = time.perf_counter() - start
        eval_data.plot_model(x, y, x_test, y_perfect, mean_values,
std_values, title=title, filename=filename, reverse_plot=reverse_plot,
figsize=figsize)
        mse = eval_data.mean_squared_error(y_perfect, mean_values)
        coverage = eval_data.coverage_95(y_test, mean_values, std_values)
        general_info = title + f"Training time: {training_time} MSE: {mse}
Coverage: {coverage}\n"
        with open("write_up.txt", "a") as writefile:
            writefile.write(general_info)

```

eval_data.py

```

import numpy as np
import matplotlib.pyplot as plt

def plot_data(x, y, figsize=(5, 4)):
    plt.figure(figsize=figsize)
    plt.scatter(x, y, s=5)
    plt.show()

def plot_model(x, y, x_test, y_perfect, mean_values, std_values,
title=None, filename=None, reverse_plot=False, figsize=(10, 8)):
    plt.figure(figsize=figsize)
    plt.scatter(x, y, alpha=0.8, label="Original data",
color="#00dca6ff")

    plt.fill_between(x_test.data.numpy().T[0],mean_values-2.0*std_values,mean
n_values+2.0*std_values,alpha=0.2,color='#003c3cff',
                    label='95% confidence interval')

    if reverse_plot == False:

plt.plot(x_test.data.numpy(),y_perfect.data.numpy(),'.',color="#FF00ACFF
", lw=2,label='Original function')

plt.plot(x_test.data.numpy(),mean_values,color='#003c3cff',lw=2,label='P
redicted Mean Model')

```

```

    else:

plt.plot(x_test.data.numpy(),mean_values,color='#003c3cff',lw=2,label='P
redicted Mean Model')

plt.plot(x_test.data.numpy(),y_perfect.data.numpy(),'.',color="#FF00ACFF
", lw=2,label='Original function')
    plt.legend()
    if title is not None:
        plt.title(title)
        plt.xlabel('x')
        plt.ylabel('y')
    if filename is not None:
        plt.savefig(filename)
        plt.show()

def mean_squared_error(y_test, mean_values):
    return np.sqrt(np.sum((np.array(y_test) - mean_values.reshape(-1,
1))**2))

def coverage_95(y_test, mean_values, std_values):
    upper = mean_values.reshape(-1, 1)+2*std_values.reshape(-1, 1)
    lower = mean_values.reshape(-1, 1)-2*std_values.reshape(-1, 1)
    return np.mean((lower<=np.array(y_test)) &
((np.array(y_test)<=upper)))

```

bnn_bayesian.py

```

import torch
import torch.nn as nn
import torch.optim as optim
import torchbnn as bnn
import numpy as np

def bnn_model(mu=0, sigma=0.1, dropout_prob=0, width=100, depth=2):
    # Creates the middle hidden layers for the model
    layers = [bnn.BayesLinear(prior_mu=mu, prior_sigma=sigma,
in_features=1, out_features=width),
nn.Tanh(),
nn.Dropout(p=dropout_prob)]
    # Adds the layers for the depth that we want
    for i in range(depth - 1):
        layers += [bnn.BayesLinear(prior_mu=mu, prior_sigma=sigma,
in_features=width, out_features=width),
nn.Tanh(), nn.Dropout(p=dropout_prob),]

```



```

        layers += [bnn.BayesLinear(prior_mu=mu, prior_sigma=sigma,
in_features=width, out_features=1)]
        model_nn = nn.Sequential(*layers)
        return model_nn

def train_bnn_model(model_nn, x, y, kl_weight=0.01, learning_rate=0.01,
epochs=4000):
    mse_loss = nn.MSELoss()
    kl_loss = bnn.BKLLoss(reduction='mean', last_layer_only=False)
    # kl weight started as 0.01
    kl_weight = kl_weight
    optimizer_nn = optim.Adam(model_nn.parameters(), lr=learning_rate)
    for step in range(epochs):
        pre = model_nn(torch.unsqueeze(x, dim=1))
        mse = mse_loss(pre, torch.unsqueeze(y, dim=1))
        kl = kl_loss(model_nn)
        cost = mse + kl_weight*kl
        optimizer_nn.zero_grad()
        cost.backward()
        optimizer_nn.step()
        if step%500 == 0:
            print(step, "/", epochs)

def get_dist(model, x_test, num_samples=10000):
    models_result = np.array([model(x_test).data.numpy() for k in
range(num_samples)])
    models_result = models_result[:, :, 0]
    models_result = models_result.T
    mean_values = np.array([models_result[i].mean() for i in
range(len(models_result))])
    std_values = np.array([models_result[i].std() for i in
range(len(models_result))])
    return mean_values, std_values

```

pyro_bayesian.py

```

import numpy as np
import torch
import torch.nn as nn
import pyro
import pyro.distributions as dist
from pyro.nn import PyroModule, PyroSample
from pyro.infer import Predictive, MCMC, NUTS, SVI, Trace_ELBO
from pyro.infer.autoguide import AutoDiagonalNormal

```

```

class BNN(PyroModule):
    def __init__(self, in_dim=1, out_dim=1, hid_dim=10,
n_hid_layers=5):
        super().__init__()

        self.activation = nn.Tanh() # Chosen activation function

        # Creates a list of the consecutive layer sizes
        self.layer_sizes = [in_dim] + n_hid_layers * [hid_dim] + [out_dim]
        # Creates a list of pyro modulkes the appropriate input and output
size
        layer_list = [PyroModule[nn.Linear](self.layer_sizes[idx - 1],
self.layer_sizes[idx]) for idx in
                        range(1, len(self.layer_sizes))]
        # We add all these layers into one collective pyro module list
        self.layers = PyroModule[torch.nn.ModuleList](layer_list)

        # Sets the weights and the bias for all the appropriate layers
        for layer_idx, layer in enumerate(self.layers):
            layer.weight = PyroSample(dist.Normal(0., 0.1).expand(
                [self.layer_sizes[layer_idx + 1],
self.layer_sizes[layer_idx]]).to_event(2))
            layer.bias = PyroSample(dist.Normal(0.,
0.1).expand([self.layer_sizes[layer_idx + 1])).to_event(1))

        def forward(self, x, y=None):
            x = x.reshape(-1, 1)
            x = self.activation(self.layers[0](x)) # input --> hidden
            for layer in self.layers[1:-1]:
                x = self.activation(layer(x)) # hidden --> hidden
            mu = self.layers[-1](x).squeeze() # hidden --> output
            sigma = pyro.sample("sigma", dist.Gamma(.5, 1)) # infer the
response noise

            # Run a for loop over x.shape[0] so over every value
            with pyro.plate("data", x.shape[0]):
                # Sample the observation given the mu we received and the
sigma we are using
                # we get some observation from the mu of what we get from
combining everything plus some sigma from the gamma we gave ourselves
                obs = pyro.sample("obs", dist.Normal(mu, sigma * sigma),
obs=y)
            return mu

def train_pyro_model(model, x, y, num_samples=50):

```

```

    # Defines the NUTS kernel to help with MCMC
    nuts_kernel = NUTS(model, jit_compile=False)
    # Define the MCMC sampler
    mcmc = MCMC(nuts_kernel, num_samples=num_samples,
warmup_steps=num_samples)
    # Run MCMC training on the data we have
    mcmc.run(x, y)
    return mcmc

def train_pyro_mcmc_model(model, x, y, num_epochs=200):
    # Defines the guide for the distribution of the parameters in the
model
    mean_field_guide = AutoDiagonalNormal(model)
    # Sets the optimizer as the Adam optimizer with a normal learning
rate
    optimizer = pyro.optim.Adam({"lr": 0.01})

    # Defines the stochastic variational interference with the type of
guide used,
    # the optimizer and the loss defined
    svi = SVI(model, mean_field_guide, optimizer, loss=Trace_ELBO())

    # Clears what we already have stored to save memory
    pyro.clear_param_store()

    # We do a stochastic variational interference training step for
each of the epochs
    for epoch in range(num_epochs):
        loss = svi.step(x, y)
    return model, mean_field_guide

def get_dist(model, mcmc, x_test):
    predictive = Predictive(model=model,
posterior_samples=mcmc.get_samples())
    preds = predictive(x_test)
    mean_values = preds['obs'].T.detach().numpy().mean(axis=1)
    std_values = preds['obs'].T.detach().numpy().std(axis=1)
    return mean_values, std_values

def get_dist_variational(model, x_test, mean_field_guide,
num_samples=50):
    predictive = Predictive(model=model, guide=mean_field_guide,
num_samples=num_samples)
    preds = predictive(x_test)

    mean_values = preds['obs'].T.detach().numpy().mean(axis=1)

```

```
std_values = preds['obs'].T.detach().numpy().std(axis=1)
return mean_values, std_values
```

nn_standard.py

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

class NeuralNetwork(nn.Module):
    def __init__(self, input_size=1, output_size=1,
hidden_layer_size=100, num_hidden_layers=2, dropout_p=0):
        super(NeuralNetwork, self).__init__()

        self.input_layer = nn.Linear(input_size, hidden_layer_size)
        self.hidden_layers = nn.ModuleList([nn.Linear(hidden_layer_size,
hidden_layer_size) for _ in range(num_hidden_layers)])
        self.output_layer = nn.Linear(hidden_layer_size, output_size)
        self.activation = nn.Tanh()
        self.dropout = nn.Dropout(dropout_p)

    def forward(self, x):
        x = self.activation(self.input_layer(x))
        for layer in self.hidden_layers:
            x = self.activation(layer(x))
            x = self.dropout(x)
        x = self.output_layer(x)
        return x

def train_nn_model(model, x, y, learning_rate=0.01, epochs=4000):
    x = x.reshape(-1, 1)
    y = y.reshape(-1, 1)
    model.train()
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    model.train()
    for epoch in range(epochs):
        optimizer.zero_grad()
        outputs = model(x)
        loss = criterion(outputs, y)
        loss.backward()
        optimizer.step()

def get_dist(model, x_test, num_samples=10000):
```

```

        # Prediction
        model.train() # Set to training mode to enable dropout during
inference
        with torch.no_grad():
            outputs = torch.cat([model(x_test) for _ in range(num_samples)],
dim=1)
            mean_prediction = outputs.mean(dim=1).numpy()
            std_prediction = outputs.std(dim=1).numpy()
            return mean_prediction, std_prediction

```

gaussian_process.py

```

import torch
import gpytorch

class ExactGPModel(gpytorch.models.ExactGP):
    def __init__(self, x, y, likelihood):
        super(ExactGPModel, self).__init__(x, y, likelihood)
        self.mean_module = gpytorch.means.ConstantMean()
        self.covar_module =
gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel())
    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)

def train_gp_model(model, x, y, likelihood, learning_rate=0.01,
epochs=100):
    model.train()
    likelihood.train()
    optimizer_gp = torch.optim.Adam(model.parameters(),
lr=learning_rate)
    # "Loss" for GPs - the marginal log likelihood
    mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)
    for epoch in range(epochs):
        optimizer_gp.zero_grad()
        output = model(x)
        loss = -mll(output, y)
        loss.backward()
        optimizer_gp.step()

def get_dist(model, x_test, likelihood):
    # Get into evaluation (predictive posterior) mode
    model.eval()
    likelihood.eval()

```

```
# Test points are regularly spaced along [0,1]
# Make predictions by feeding model through likelihood
with torch.no_grad(), gpytorch.settings.fast_pred_var():
    y_pred_gp = likelihood(model(x_test))
    return y_pred_gp.mean, y_pred_gp.variance
```

model_functions.py

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchbnn as bnn

def bnn_model(mu=0, sigma=0.1, width=100):
    model_nn = nn.Sequential(
        bnn.BayesLinear(prior_mu=mu, prior_sigma=sigma, in_features=1,
out_features=width),
        nn.ReLU(),
        nn.Dropout(p=dropout_prob),
        bnn.BayesLinear(prior_mu=mu, prior_sigma=sigma, in_features=width,
out_features=width),
        nn.Tanh(),
        nn.Dropout(p=dropout_prob),
        bnn.BayesLinear(prior_mu=mu, prior_sigma=sigma, in_features=width,
out_features=1),
    )
    return model_nn
```