Learning Distributions with Neural Networks

Student number: 18086609

April 14, 2022

1 Abstract

Neural networks applied in a frequentist context have revolutionized domains like computer vision, natural language processing, and reinforcement learning. However, we show that a Bayesian neural network outperforms a standard neural network in both calibration and point estimates across two simple datasets. A Bayesian neural network can be created by defining a prior distribution over its parameters, applying Bayes' rule, and working with the network's posterior predictive distribution. The main limitation of a Bayesian neural network is that its predictive distribution has no analytical solution and so it must be approximated. This can be multiple orders of magnitude more computationally expensive than training a standard neural network. We explore the accuracy and cost of different methods to obtain this approximation.

Word count: 13,717

Contents

2 Frequentist vs Bayesian Neural Network

Neural networks are commonly used for prediction (regression or classification). In this setting, they map input predictors x to a response y. So we can think of a neural network simply as a function that maps $x \to y$, *conditioned* on some specific parameters of the neural network. In technical terms, the neural network is a likelihood function, $p(y|x,\theta)$.

Let's assume we observe some new predictors $\tilde{x}$ and we want to predict the response variable $\tilde{y}$ corresponding to these predictors. For example, an autonomous vehicle observes a new object and needs to classify it, or it observes a pedestrian and needs to predict her movement. In this context, we are interested in the quantity $p(\tilde{y}|\tilde{x})$, but a standard neural network only gives us $p(\tilde{y}|\tilde{x},\theta)$. The solution to this problem is to *train* the network on some observed data (x, y) by learning the distribution $p(\theta|x, y)$ and then marginalizing over this distribution during prediction, obtaining $p(\tilde{y}|\tilde{x}, x, y)$:

$$p(\tilde{y}|\tilde{x}, x, y) = \int_{-\infty}^{\infty} p(\tilde{y}, \theta|\tilde{x}, x, y) d\theta = \int_{-\infty}^{\infty} p(\tilde{y}|\tilde{x}, \theta) p(\theta|x, y) d\theta \tag{1}$$

There are two distinct approaches to learning $p(\theta|x, y)$, based on *frequentist* and *Bayesian* principles. In the frequentist context, we will estimate the value of θ and assume that the true value of θ is equal to the estimated value with 100% probability. Clearly, this is a naive assumption and no one actually believes this, but it is an assumption that we have to make in order to marginalize over θ. Perhaps we might avoid claiming that we know the true value of θ with certainty, but then we cannot marginalize over θ, i.e. we cannot really make any predictions of the form $p(\tilde{y}|\tilde{x}, x, y)$. Since our goal is to obtain predictions, we will assume that frequentist inference is equivalent to assuming that we know the true value of θ with certainty, i.e. we will assume a Dirac delta distribution over θ.

In contrast, in Bayesian inference, we must assume some prior distribution over θ before observing any data, and then update this belief given the observed data (x, y). This is achieved using Bayes' rule:

$$\begin{aligned}
p(\theta, y|x) &= p(\theta, y|x) \\
p(\theta|x, y)p(y|x) &= p(y|x, \theta)p(\theta|x) \\
p(\theta|x, y) &= \frac{p(y|x, \theta)p(\theta|x)}{p(y|x)} \\
p(\theta|x, y) &= \frac{p(y|x, \theta)p(\theta)}{\int_{-\infty}^{\infty} p(\theta, y|x) d\theta} \\
p(\theta|x, y) &= \frac{p(y|x, \theta)p(\theta)}{\int_{-\infty}^{\infty} p(y|x, \theta)p(\theta) d\theta} \\
p(\theta|x, y) &\propto p(y|x, \theta)p(\theta)
\end{aligned} \tag{2}$$

In Bayesian inference, the choice of the prior distribution over θ is subjective, since we must make this choice *before* observing any data. The resulting posterior distribution is typically very complex and has no closed-form solution. Instead, it must be approximated – this is discussed in section 4.

3 Neural network

A neural network can act as a universal function approximator. Hence, we can train a neural network to fit our observed data. For example, we can design a neural network that will output a single number, which will represent a point estimate for a given observation. Alternatively, we can assume that response y, conditional on the predictors x, is distributed according to a distribution belonging to a given family (e.g. normal). Then, given a set of predictors, the network can output a vector corresponding to the parameters of the given distribution. In other words, the network's output will be a distribution over y, rather than a single point estimate.

3.1 Multilayer perceptron

The multilayer perceptron (MLP) is arguably the simplest neural network architecture. We can think of it as a series of layers, where each layer performs a non-linear transformation of its input. These layers are sequentially chained so that the output of one layer is the input to the next layer.

Let's denote the input to the MLP as x_1 (this is a vector of predictors for a single observation in a dataset). This input will be passed to the first layer, which will perform a non-linear transformation of x_1 and return a transformed output, denoted x_2. Next, x_2 will be passed to the second layer, which will transform it and output x_3. This process will be repeated until we reach x_n, the output layer. So we can think of the MLP as the following series of transformations, where each arrow corresponds to one layer of the network:

$$x_1 \to x_2 \to x_3 \to \dots x_{n_1} \to x_n \tag{3}$$

The layer x_1 is called the *input layer*, x_n is called the *output layer*, and all the layers in between are called *hidden layers*. Each layer consists of a single matrix multiplication and a nonlinearity:

$$x_i = \sigma(W_i x_{i-1} + b_i) \tag{4}$$

Essentially, the input to the layer x_i is multiplied by a matrix W_i, shifted by a *bias* vector b_i, and transformed through a nonlinear function denoted σ. The layers are called *fully-connected* since each component of the input x_{i-1}^j is connected to each component of the output x_i^k through some matrix weight W_{jk}.

Another way to visualize the MLP is as a graph of connected nodes that represent *artificial neurons* (Fig. 1). In every layer, every node is connected to every node in the previous layer and every node in the following layer (hence the name *fully-connected*). We can think of the connections between the neurons as the matrix parameters W_{jk}, corresponding to the strength of each connection. At a very high level, this architecture is inspired by the human brain, although it has to be noted that the analogy is loose.

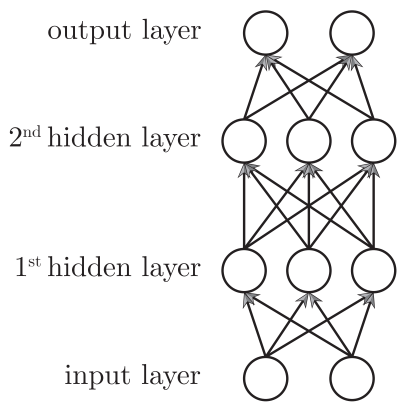


Figure 1: An illustration of a multilayer perceptron (MLP) with two hidden layers consisting of three neurons each. The input is passed to the bottom input layer and flows upward through the network to the top output layer. The nodes in the graph represent artificial neurons and each edge has a learnable weight.

The role of the nonlinearity σ (also called an *activation function*) is essential. If there was no nonlinearity in the network, each layer would only perform some specific linear transformation. However, the composition of linear transformations is still just a linear transformation. So without the nonlinearity, increasing the network's number of layers would have essentially no effect. In contrast, *with* the nonlinearity, the more layers are used, the more complex a function the neural network can express. Historically, it was common to apply the *sigmoid* activation function, which monotonically maps all real numbers to the interval $(-1, 1)$:

$$\sigma_{\text{sigmoid}}(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

While this is a perfectly reasonable activation function to use, in practice, the ReLU activation (*rectified linear activation function*) seems to work just as well and it is easier to compute:

$$\sigma_{\text{ReLU}}(x) = \max(0, x) \tag{6}$$

3.2 Loss function

Neural networks are typically trained using gradient descent. The idea is to define a loss function to evaluate the discrepancy between the neural network's desired output and its actual output, and then minimize this discrepancy (loss). A low value of the loss function implies that the output of the neural network is close to the desired output.

For example, when the neural network is designed to output point estimates, we can use the sum of squared residuals as the loss function: $\sum_i (\hat{y}_i - y_i)^2$. Conversely, when the neural network's output is a probability distribution, we can maximize the likelihood of the observed data: $\prod_i \hat{p}(y_i)$. However, often, the likelihood for a single observation is a very small number, much smaller than zero. Taking a product over thousands of very small numbers results in a number that is extremely small, easily smaller than 10^{-1000}. When computers store numbers, they must do so with a fixed precision, so that the numbers can be encoded by a fixed number of bits. Typically, this is no higher than 64 bits, split up between the number's *mantissa* (value) and *exponent* (order of magnitude). Out of the 64 bits used to store the number, only 11 are used to store its exponent, meaning the smallest number that can be natively stored on modern computers is $2^{-2^{10}} = 2^{-1023} \approx 10^{-308}$. While this is a very small number, it is not small enough to store some likelihood values. As a result, any likelihood smaller than roughly 10^{-308} would get rounded down to zero. This is an error that must be avoided, since it would prevent us from taking derivatives of the likelihood. Typically, this problem is avoided by working with the (negative) logarithm of likelihood (NLL), rather than the likelihood itself: $\text{NLL} = -\log(\prod_i \hat{p}) = -\sum_i \log \hat{p}(y_i)$.

3.3 Gradient descent

A neural network is defined by its architecture (e.g. number and type of layers) and its parameters, denoted θ. Typically, the architecture is defined by a human and fixed for the duration of training. Training is achieved only by modifying the parameters of the neural network. The most common way to do this is to iteratively differentiate the loss function w.r.t. the parameters and then update the parameters in the direction of decreasing loss. This is called *gradient descent*. More specifically, the parameters of the neural network at step $i+1$ are equal to the parameters at the previous step (i) plus a step in the direction of decreasing loss:

$$\theta_{i+1} \leftarrow \theta_i - \gamma \frac{\partial L}{\partial \theta_i} \tag{7}$$

In the above equation, γ is called the *learning rate* and it dictates how far along the direction of decreasing loss we move. The issue is that the gradient is only a linear approximation of the loss function, so stepping too far could cause the training procedure to become unstable. Conversely, setting the learning rate too small will increase the number of steps required to reach convergence, resulting in an unnecessary increase in computational cost.

In practice, modern machine learning libraries implement *automatic differentiation*, meaning they can analytically differentiate any function that is composed of simpler differentiable functions. A neural network is technically just a function composed of thousands (or millions, possibly billions) of parameters and simple differentiable operations like addition and multiplication. So, in practice, it suffices to define a neural network architecture and let the library compute its derivates. However, for the sake of understanding, it can still be helpful to derive the derivative of loss w.r.t. parameters by hand. This helps provide intuition behind the inner workings of modern machine learning libraries.

Let's assume that we are using an MLP where each layer has only a single parameter. Then, we can apply the chain rule to expand the derivative of the loss function w.r.t. the neural network's parameters W_i:

$$\frac{\partial L}{\partial W_i} = \frac{\partial L}{\partial x_n} \frac{\partial x_{n-1}}{\partial x_{n-2}} \cdots \frac{\partial x_{i+2}}{\partial x_{i+1}} \frac{\partial x_{i+1}}{\partial W_i} \tag{8}$$

Inside the above expression, there is a repeating term of the form $\frac{\partial x_i}{\partial x_{i-1}}$. In order to compute it, it is helpful to define $z_i := W_i x_{i-1} + b_i$. Then:

$$\frac{\partial x_i}{\partial x_{i-1}} = \frac{\partial \sigma(z_i)}{\partial x_{i-1}} = \sigma'(z_i) \frac{\partial z_i}{\partial x_{i-1}} = \sigma'(W_i x_{i-1} + b_i) W_{i-1} \tag{9}$$

A similar expression holds for the bias terms b_i:

$$\frac{\partial x_i}{\partial x_{i-1}} = \sigma'(W_i x_{i-1} + b_i) \tag{10}$$

Then, if we know the derivative of the loss function L, we know the derivatives of all the parameters of the network and we can iteratively apply gradient descent to train the network.

In reality, the parameters of each layer are vectors, rather than scalars. Surprisingly, the same equations still hold. The only difference is that instead of computing scalar derivates, we need to compute the Jacobians of each transformation, which have the same form as the scalar expression. The chain rule also applies in a similar way, except that instead of multiplying scalars, we matrix-multiply the Jacobians of each transformation.

3.4 Stochastic gradient descent

When large neural networks are trained on large datasets (e.g. millions of images), it would be extremely expensive to evaluate each gradient update on the whole dataset. In order to speed up the training, the dataset is typically split-up into several smaller *mini-batches*, and the gradient is computed independently on each of these mini-batches. This method is called *stochastic gradient descent* (SGD).

The term *epoch* is commonly used as a unit to measure how many dataset samples a neural network was trained on. For example, one epoch means that the model saw each instance from the dataset exactly once. Two epochs mean that the network saw the whole dataset twice... and so on. In full-batch gradient descent, one epoch corresponds only to a single update of the model parameters. In contrast, if the dataset is split into 100 mini-batches, one epoch represents 100 gradient updates. While each of these updates is noisy (it is only an approximation of the true gradient, based on the whole dataset), it turns out that many noisy gradient updates (i.e. SGD) typically result in faster training compared to full-batch gradient descent.

Another advantage of SGD is rooted in hardware. Neural networks are typically trained on hardware like graphics processing units (*GPUs*) or tensor processing units (*TPUs*). These devices are very fast at parallel computing, such as matrix multiplication and other element-wise operations. However, they also have limited memory available: typically in the range of 8 – 80 GB. During backpropagation, this is enough memory only for a few images, but in low-dimensional regression, it can actually be enough to store the whole dataset.

Within this essay, the terms *gradient-descent* and *SGD* are used interchangeably, even though full batches are always used. Since all experiments use a small dataset, using mini-batches is not necessary. However, if a larger dataset were used, using mini-batches would be preferable. For the purposes of this essay, there is no difference between gradient descent and stochastic gradient descent.

3.5 Learning rate schedule

In Eq. 7, we see that the SGD update depends on the learning rate γ. But what is a good value of γ? When the learning rate is too high, θ may fail to converge; when the learning rate is too low, training will take too long. One common solution is to use use a *learning rate schedule*, rather than a constant learning rate.

All experiments in this project use an *exponentially decaying* learning rate. The idea is that in the early phase of training, we can get away with a high learning rate and achieve fast convergence, while as we approach a local minimum, the learning rate needs to be decreased such that we can carefully descent into that minimum. This is implemented by multiplying the learning rate after each update by a constant rate r that is very slightly smaller than one:

$$\gamma_{i+1} \leftarrow r\gamma_i \tag{11}$$

3.6 Loss landscape

Whether a neural network is trained to maximize the log-likelihood (i.e. find the *maximum likelihood / ML* solution) or the log-posterior (i.e. find the *maximum a posteriori / MAP* solution), the geometry of its loss as a function of its parameters is interesting to study.

Let's assume we observe the data in Fig. 2 and want to train a neural network to model the conditional distribution $p(y|x)$.

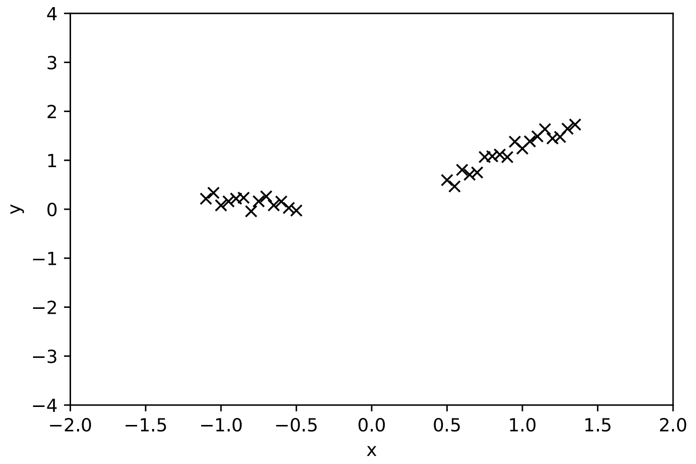


Figure 2: A toy 1D regression dataset. It only consists of the 31 observations displayed.

We use an MLP with a single hidden layer consisting of 50 nodes and a $N(0, 1)$ prior over the parameters θ. The network outputs the tuple (μ, σ^2) for each observation and we assume that $y \sim N(y, \sigma^2)$. We initialize six networks like this, each with parameters drawn independently from a $N(0, 0.22^2)$ distribution.[1] Afterward, we train each network using SGD until convergence. We set the loss function proportional to the logarithm of the posterior distribution. Fig. 3 shows the predictions of each of these networks. It appears that each networks has learned the same function, even though the parameters of each network are different. This is evident from Fig. 4, which shows the L_1 distance between the parameters of the first network and the other five during training. At initialization, the L_1 norms of the parameters of each network are between $0.18 - 0.25$ and they remain in this range after training. At initialization, the distances are all between $0.2 - 0.3$ and they only change very little during training. This implies that each of the 6 networks has converged to a different solution in the parameter space since the distances are roughly constant and roughly equal to the norms of the parameters.

[1]Using a larger variance to initialize the networks would cause problems with numerical stability.

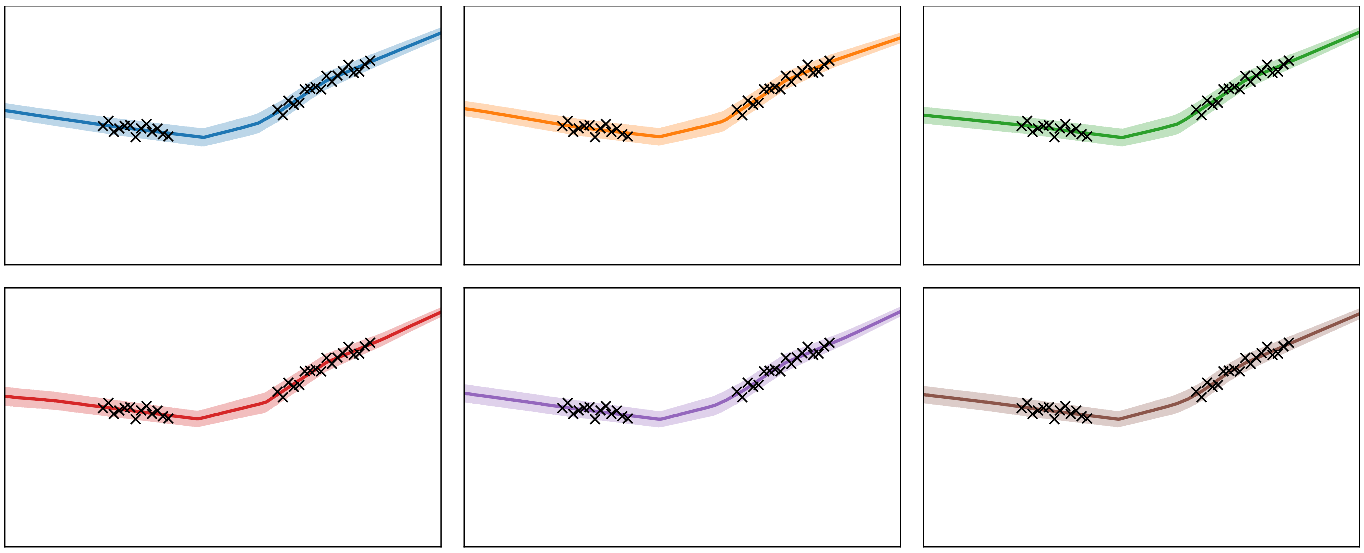


Figure 3: Each plot shows the predictions of an independently initialized and trained MLP. Even though each network has converged to a different solution in the parameter space, their predictions appear identical.

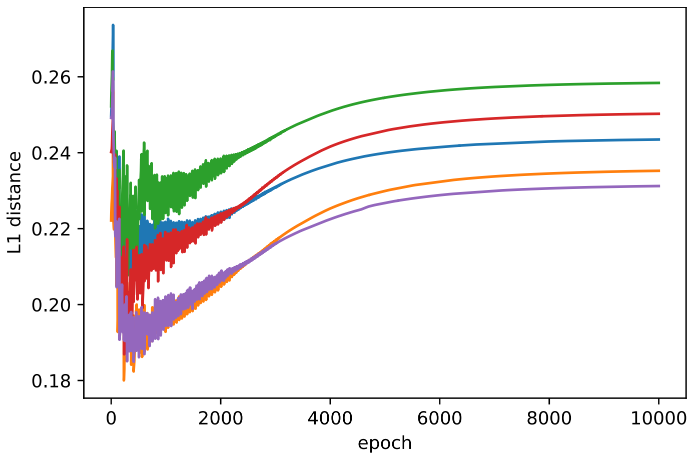


Figure 4: L_1 distance of five independently-trained MLPs to the a sixth independently-trained MLP during training.

Part of the reason behind this is that modern neural networks are typically *over-parametrized*, meaning that there are many different sets of parameters θ that fit the observed data equally well [1, 2, 3]. Each of the networks in this example was initialized differently and SGD tends to converge to a solution that is close to the initialization [4]. Hence, each of the different initializations resulted in a different SGD solution.

A natural question that arises is: *what does the geometry of the loss function look like?* One way to peek at this high-dimensional landscape is to plot the plane described by three independent SGD solutions. Let's denote the parameters of these three networks $\theta_0, \theta_1, \theta_2$. Next, let's define a new coordinate system under which the coordinates of these solutions are $(-1, 0), (1, 0), (1, 0)$. This can be achieved by defining the following auxiliary variables:

$$\begin{aligned} \theta_m &= \frac{\theta_0 + \theta_1}{2} \\ d_x &= \theta_1 - \theta_m \\ d_y &= \theta_2 - \theta_m \end{aligned} \tag{12}$$

Then, any point R along the surface described by $\theta_0, \theta_1, \theta_2$ can be mapped to the $2D$-coordinates (x, y) as follows:

$$R = \theta_m + x d_x + y d_y \tag{13}$$

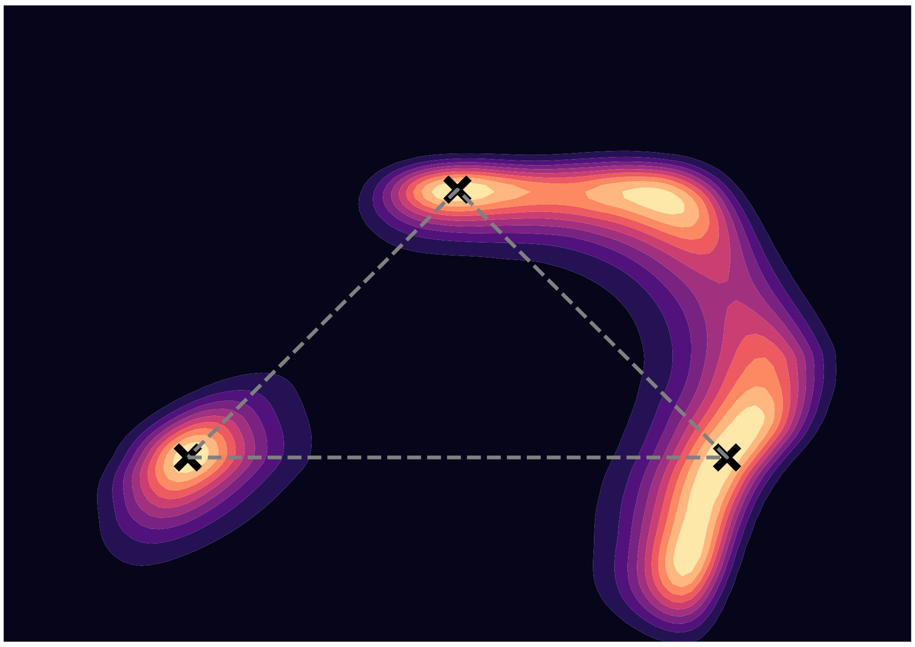


Figure 5: A contour plot of the loss function of an MLP with a single hidden layer. The three crosses connected by dashed grey lines represent three independent SGD solutions. They define the plane along which the loss is evaluated, displayed by the contours.

Fig. 5 shows the plane described by the three solutions $\theta_0, \theta_1, \theta_2$. Notice that θ_1 and θ_2 (the middle and right solutions) appear to be connected by a low-loss region, while θ_0 appears to be disconnected from the other two solutions.

In general, but especially in larger networks [3], a linear path between two SGD solutions will contain a region of very high loss, comparable to a randomly-initialized network. At the same time, deviating from an SGD solution in a random direction will also cause the loss to increase rapidly [5]. However, for any two SGD solutions, there typically exists a low-loss tunnel connecting them. The loss along this tunnel is approximately equal to loss at the two SGD solutions [3]. Fig. 5 hints at this phenomenon, although the loss does decrease slightly along the low-loss path from the middle solution to the right solution.

Garipov et al. [3] devise a simple method to find a low-loss tunnel between any two SGD solutions. Consider two arbitrary SGD solutions θ_0 and θ_1. Then, for any proposed path connecting θ_0 and θ_1, we will uniformly sample points from this path and evaluate the loss along these points. The goal is to find a path where the sum of these losses is the lowest. They obtained good results searching for a path θ_t defined by a quadratic Bézier curve:

$$\theta_t = (1-t)^2\theta_0 + 2t(1-t)\theta^* + t^2\theta_1, \quad t \in [0,1] \tag{14}$$

The above Bézier curve has a single free parameter, θ^*. SGD can be used to find θ^* given θ_0 and θ_1. For example, consider the left and middle solutions from Fig. 5, θ_0 and θ_1. Even though they appear to be disconnected, we can find a low-loss tunnel defined by a quadratic Bézier curve connecting them, as shown in Fig. 6.

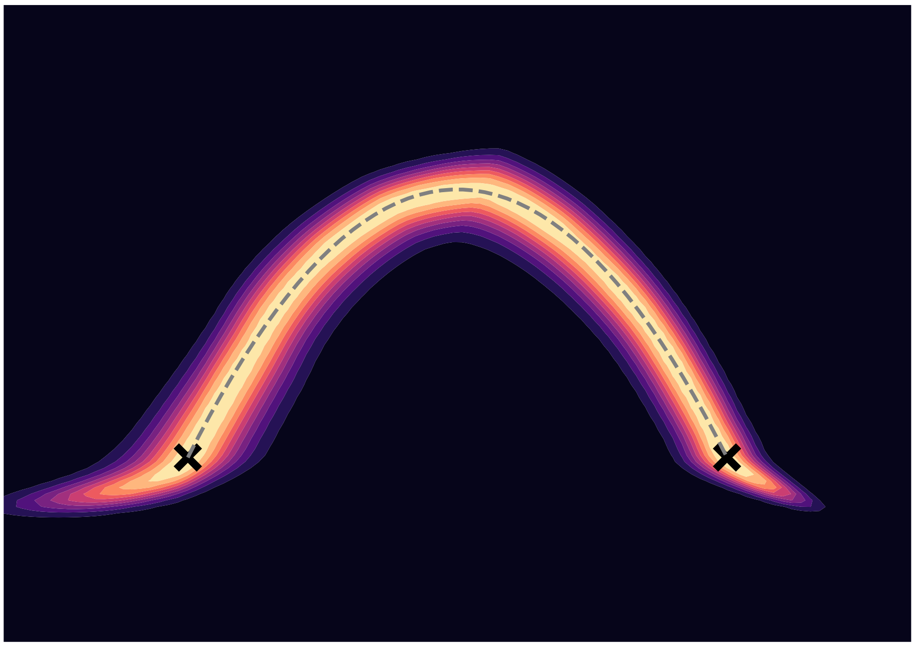


Figure 6: A contour plot of the loss function of an MLP with a single hidden layer. The two crosses represent two independent SGD solutions. The dashed line is a low-loss tunnel that connects these two solutions.

Even more surprisingly, the loss landscape of deep neural networks contains planes with arbitrary patterns, such as images of a bat or a cow [6]. The complexity of the loss landscape has important consequences for Bayesian neural networks that are discussed in section 6.

4 Bayesian neural network

As described in section 2, a neural network can be used as a likelihood function to represent $p(\tilde{y}|\tilde{x}, \theta)$. In a Bayesian context, we define a prior distribution (belief) over θ before observing any data, $p(\theta)$, and use this belief together with the likelihood function over a training dataset to obtain a posterior distribution over θ, $p(\theta|x, y)$, as described in Eq. 2. Given this posterior distribution over θ, we can obtain the *posterior predictive distribution* $p(\tilde{y}|\tilde{x}, x, y)$ as described in Eq. 1. Typically, the posterior predictive distribution is the distribution that we are most interested in. It allows us to perform prediction without conditioning on any particular value of θ.

4.1 Model uncertainty

When making a prediction, the total uncertainty in our estimate is a combination of *data uncertainty* and *model uncertainty*. Data uncertainty is the uncertainty over the output variable in any given likelihood function. For example, if we assume that $y \sim N(0, 1)$, the data uncertainty is equal to the variance of the normal distribution. However, in practice, we often don't know what the true distribution generating the data is. For example, it might be $N(0, 1)$, but it could also be $N(0.2, 1.1)$ or $N(0.06, 0.82)$. This is what the term *model uncertainty* refers to. It is the uncertainty in our predictions caused by the fact that we don't know what the true model is.

Let's return to the dataset in Fig. 2. Say we observe the plotted data (x, y), and we are interested in predicting $\tilde{y}$ given a new observation of $\tilde{x}$ (the accent on x and y is used to simply distinguish the new observation from the plotted data). We will use an MLP with a single hidden layer of 50 nodes and a $N(0, 1)$ prior over θ. As before, the network outputs the tuple (μ, σ^2) for each observation and we assume that $y \sim N(y, \sigma^2)$.

The frequentist approach would be to find the MAP solution for θ and use it to model $p(\tilde{y}|\tilde{x})$. However, there is a crucial problem with this approach. While the obtained model is certainly going to fit the observed data well, it does not take into consideration the fact that different models might also fit the data similarly well. Hence, it fails to take into consideration any kind of model uncertainty. As a result, its predictions will tend to be *overconfident*, meaning the model will output smaller confidence intervals than it should. For example, a 95% confidence interval generated by this model will contain *less* than 95% of the true values, since the interval is too narrow.

The Bayesian approach takes into consideration model uncertainty. When training a BNN, the output is a distribution over θ, rather than just a single value. Hence, the distribution over θ is a distribution over different models that might describe the data, and the density of the distribution describes our belief about how likely a specific model is.

Fig. 7 displays the concept of model uncertainty using a standard neural network and a Bayesian neural network. A NN trained using SGD is a single model corresponding to a single value of θ. On the other hand, a BNN comprises a distribution over possible models. The right plot in Fig. 7 shows the different likelihoods for y obtained by sampling θ from the posterior. The BNN does what we would intuitively like it to do: it considers that a set of different models might generate the data, and it takes into consideration that all of them might be true with some probability. As we move farther away from observed data along the x-axis, the BNN's uncertainty increases, since we are gradually less confident about the behavior of the true model there. Conversely, the NN's confidence *increases* as it moves farther away from the observed data, which is undesirable. There is no reason to assume that observation far away from the observed data will continue to follow the same linear trend that it predicts.

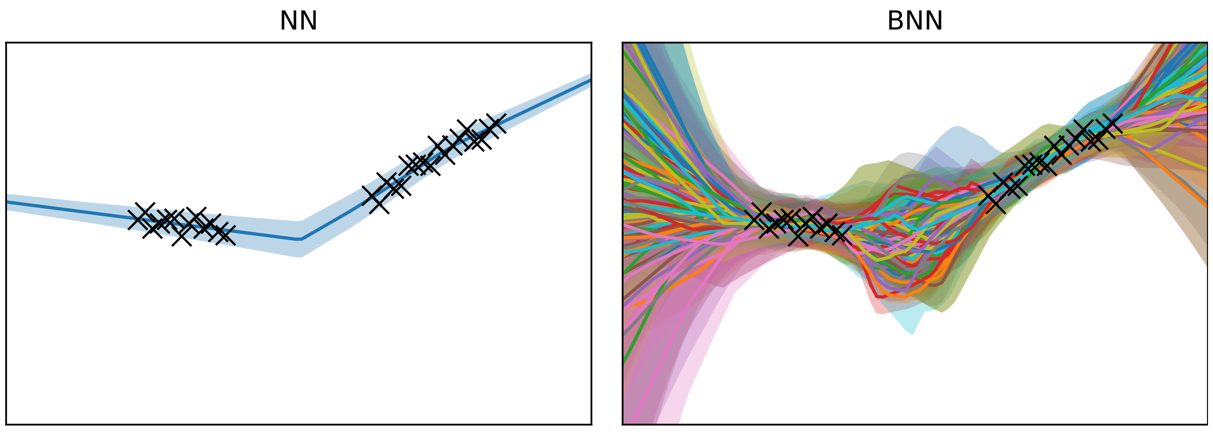


Figure 7: Left: predictions of a NN trained using SGD. Right: samples from the predictive distribution of a BNN.

A more accurate way to visualize the predictions of a BNN is to plot its probability density function (PDF), which requires marginalizing over θ. In general, this cannot be done exactly, but a good approximation is to draw samples from the posterior and average the predictions over these samples. This technique is further described in section 4.3.

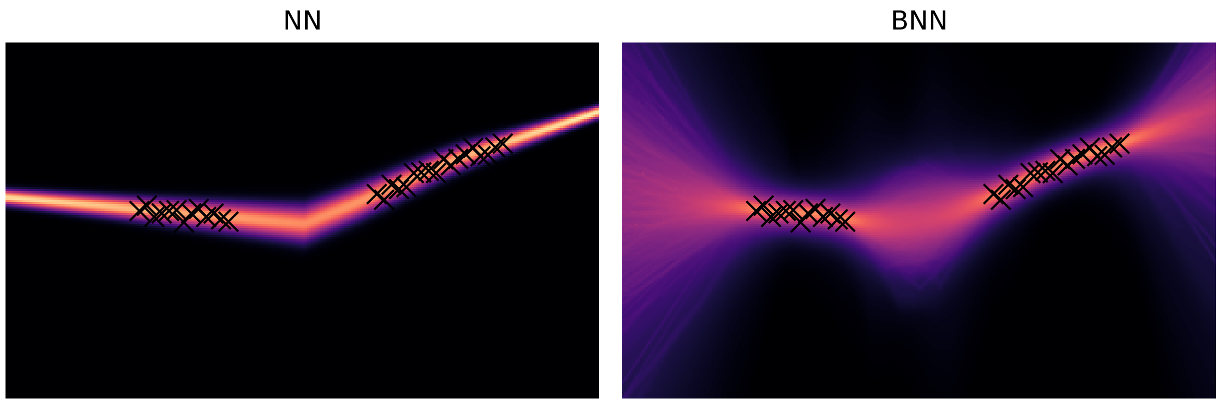


Figure 8: Density function of the predictive distribution of a NN and a BNN visualized using a heat map.

A simpler and more common way to visualize the predictions is to plot a confidence interval and a mean estimate. One way this can be achieved is to sample θ_i from the posterior and then sample y from each θ_i. The empirical quantiles of the sampled values of y can then be used to obtain approximate confidence intervals and an approximate mean.

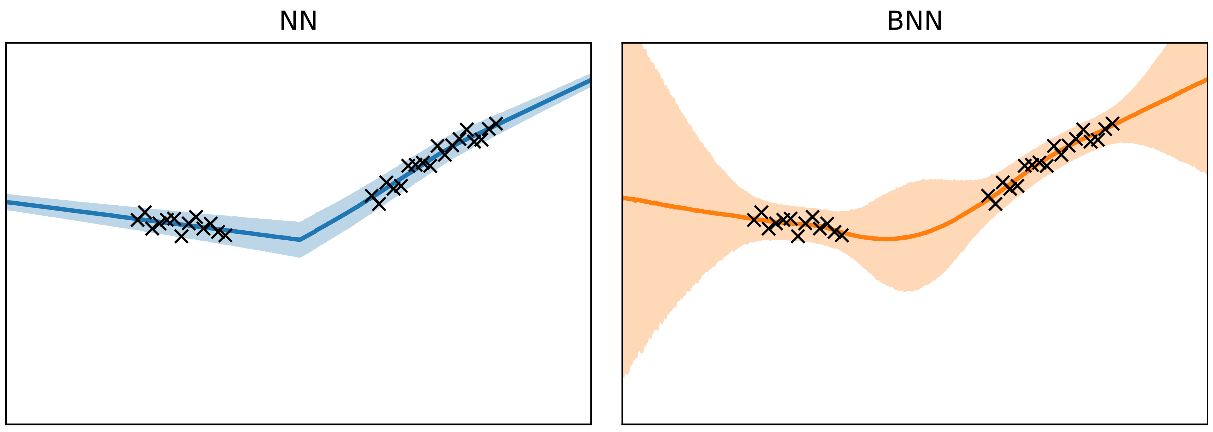


Figure 9: 95% confidence interval of the predictive distribution of a NN and a BNN

4.2 Prior selection

All of the prior examples used the prior distribution $\theta \sim N(0,1)$, but this choice is arbitrary. The prior acts as a regularizer, assigning more weight to some models and less weight to others. The goal of a normal prior centered around zero, with a low variance is to favor models with parameters that are small in magnitude. The smaller the magnitude of the parameters, the smoother and simpler we expect the likelihood function to be. Conversely, when a neural network has parameters that are large in magnitude, it is able to represent functions that are very rough.

Fig. 10 shows the posteriors obtained by varying the standard deviation of the prior. We see that a small standard deviation (i.e. an informative prior) acts as strong regularization. A value of 0.1 seemingly fails to capture the underlying trend in the observed data, but values $0.2, 0.5, 1, 3$ all seem reasonable. We might favor values 0.2 and 0.5 when we expect the underlying data generating process to be simple – when we expect that any observed trend in the data would continue to hold even for unobserved values of x. Values 1 and 3, on the other hand, represent an increased model uncertainty – even though the observed values of x have a certain trend, there might be a different trend for unobserved values of x. A standard deviation of 10 takes this to the extreme, representing the belief that we cannot say almost anything at all about the trend for unobserved values of x. This behavior of the posterior predictive distribution in a BNN is analogous to Gaussian processes. In fact, in the infinite-width limit, some BNNs become a Gaussian process [7].

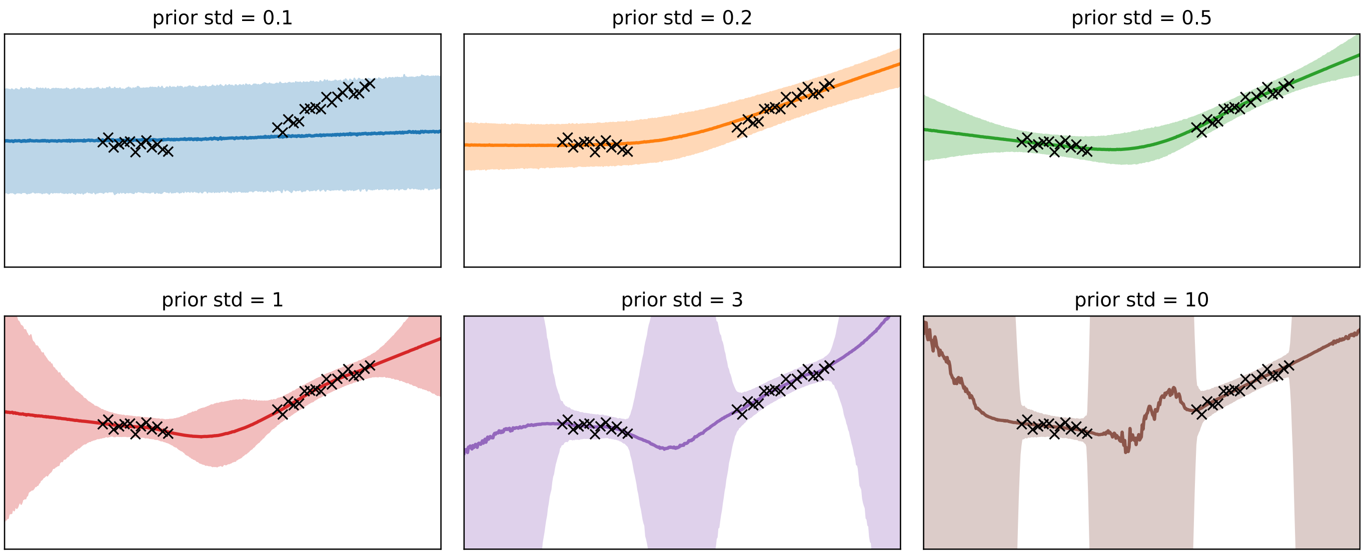


Figure 10: Predictive distribution of a BNN using various scales of a normal prior.

4.3 Prediction

The distribution over the response variable $\tilde{y}$, given a new observation $\tilde{x}$ and training data (x, y) is given by the *posterior predictive distribution* $p(\tilde{y}|\tilde{x}, x, y)$, derived in Eq. 1. This equation involves integrating over θ, which has the same number of dimensions as the number of parameters of the neural network, meaning possibly millions or billions. In general, the integral doesn't have a closed-form solution and numerical methods must be used to approximate it.

In low dimensions, the *trapezoidal rule* is an intuitive method to numerically compute an integral:

$$\int_a^b f(x)dx \approx \frac{1}{2}(b-a)(f(a)+f(b)) \tag{15}$$

However, let's say that we need to use n trapezoids to approximate a $1D$-integral. Then, as the number of dimensions d grows, the number of trapezoids required grows like n^d, i.e. exponentially. With anything but a tiny neural network, this method is infeasible.

Monte Carlo simulation is a better approach. We express the integral as an expectation over some probability distribution $p(x)$, draw samples $X_i \sim p(x)$, and compute the empirical mean of $f(x)$ over the samples:

$$\int f(x)p(x)dx = E_x[f(x)] \approx \hat{f}_n := \frac{1}{n}\sum_{i=1}^{n} f(X_i) \tag{16}$$

Using the strong law of large numbers and assuming x_i are i.i.d.:

$$\mathbb{P}\left(\lim_{n\to\infty} \hat{f}_n = \mathbb{E}_\pi[f]\right) = 1 \tag{17}$$

In other words, we can use n i.i.d. samples from $p(x)$ to approximate $\int f(x)p(x)dx$, denoting the estimand $\hat{f}_n$. As $n \to \infty$, the estimand will converge almost surely to the true value of the integral. Even better, by using the central limit theorem, we get that the estimand converges as $1/\sqrt{n}$, i.e., its convergence rate does not depend on the number of dimensions. This is a crucial property to work in high dimensions.

More realistically, we might only have access to samples from $p(x)$ that are serially dependent – the reason behind this will become clear in section 4.5. In this case, the variance of the estimand will depend on the autocovariance of the samples X_i, as derived in Eq. 18. While it is true that a negative autocovariance of $f(X_i)$ will reduce the total variance of $\hat{f}_n$, the variance of a transformation of f might increase (e.g., if X_i has a negative autocovariance, X_i^2 has positive autocovariance). For these reasons, it is preferable to get the autocorrelation of X_i as close to zero as possible, so that any transformation of X_i will also hopefully have autocorrelation close to zero.

$$\begin{aligned}
\operatorname{Var}\left(\tilde{f}_n\right) &= \frac{1}{n^2}\operatorname{Var}\left(\sum_{i=1}^{n} f(X_i)\right) \\
&= \frac{1}{n^2}\operatorname{Cov}\left(\sum_{i=1}^{n} f\left(X_i\right), \sum_{i=1}^{n} f(X_i)\right) \\
&= \frac{1}{n^2}\left(\sum_{i=1}^{n} \operatorname{Var}(f(X_i)) + 2\sum_{i<j}\operatorname{Cov}(f(X_i), f(X_j))\right)
\end{aligned} \tag{18}$$

We can apply this technique to the posterior predictive distribution of a BNN by taking the expectation of the likelihood $p(\tilde{y}|\tilde{x},\theta)$ over the posterior $p(\theta|x,y)$. However, we must be careful about how this simulation is implemented. As described in section 3.2, computers struggle to work with small numbers, which is why working with the logarithms of probabilities is more accurate. Unfortunately, we cannot average over $\log p(\tilde{y})$ directly, rather we must average over $p(\tilde{y})$. The naive solution would be to simply exponentiate $\log p(\tilde{y})$, but since $p(\tilde{y})$ may be close to zero, this would result in underflow. A better solution is to use the LogSumExp function, which is a stable way to compute the logarithm of a sum of exponents. The resulting expression is derived below:

$$\begin{aligned}
p(\tilde{y}) &= \mathbb{E}_\theta[p(\tilde{y},\theta)] \\
p(\tilde{y}) &\approx \sum p(\tilde{y}|\theta_i)p(\theta_i) \\
p(\tilde{y}) &\approx \sum p(\tilde{y}|\theta_i)\frac{1}{n} \\
p(\tilde{y}) &\approx \sum \exp\log p(\tilde{y}|\theta_i))\frac{1}{n} \\
\log p(\tilde{y}) &\approx \log(\sum \exp\log p(\tilde{y}|\theta_i)\frac{1}{n}) \\
\log p(\tilde{y}) &\approx \operatorname{LogSumExp}(\log p(\tilde{y}|\theta_i)) - \log(n)
\end{aligned} \tag{19}$$

Hence, all we need is to find an efficient method to sample the posterior. When the likelihood function has a simple form, and with a bit of luck, there may exist a *conjugate prior* distribution such that the posterior distribution belongs to the same family of distributions as the prior. This is a useful trick because it allows us to obtain the exact posterior distribution with a negligible computational expense. However, neural networks are so complex that they do not have any known useful conjugate priors. Instead, to obtain the posterior, we must choose one of two approaches: *variational inference* or *MCMC-based sampling*. Both of these methods only provide approximations to the posterior distribution and come with their own unique tradeoffs. These are discussed below in sections 4.4 and 4.5.

4.4 Variational inference

First, we could approximate the true posterior distribution using a simpler distribution from which we can easily draw samples. While the approximation may be inaccurate, we can often fit the simpler distribution with little computational expense. Hence, this is a good approach if we are willing to trade off accuracy for compute. It is called *variational inference.*

4.4.1 SWAG

Stochastic Weight Averaging Gaussian (SWAG) is an efficient way to compute a Gaussian approximation around the maximum a posteriori (MAP) estimate [5]. It introduces little computational overhead over conventional SGD training and improves both accuracy and calibration [5].

If we consider the geometry of the loss function around the MAP solution to be approximately Gaussian, we can exploit the SGD trajectory to fit this Gaussian approximation. The idea is that as SGD converges to the MAP solution, it will bounce around the minimum. We can take the sample of parameters after each SGD training epoch to fit a Gaussian approximation of the loss function. Since the loss function is proportional to the log-posterior distribution, the fitted Gaussian distribution is a local Gaussian approximation of the posterior around the MAP solution.

The main limitation of SWAG is that it only describes a single mode of log-posterior distribution. As shown in section 3.6, the posterior distribution of neural networks is multi-modal. In deep neural networks, the distributions described by each mode tend to differ in function space, so a local approximation around a single mode fails to capture the functional diversity of the posterior [2].

4.4.2 Deep ensembles

One model that takes into account the multi-modality of the posterior is a *deep ensemble.* It consists of training a set of independent neural networks using SGD, where each network is initialized with different parameters and hence converges to a different mode of the posterior, and then averaging their predictions. Another way of describing this model is that the posterior over θ is a uniform categorical distribution over the independent SGD solutions.

In section 3.6, Fig. 3 shows the predictions of each network in a six-network ensemble trained on a tiny regression dataset. In the figure, each network appears identical in function space, even though the parameters of each network are different. However, deeper neural networks have different behavior. In deep NNs, each mode of the posterior tends to differ from the other modes in function space [2].

Empirically, a deep ensemble is a more faithful approximation to the true posterior of a BNN than SWAG [8]. Point estimates of multiple modes capture more of the functional diversity of the posterior than a local approximation around a single mode. Deep ensembles are very popular in practice, as they are simple and provide a substantial increase in accuracy compared to SGD or SWAG [9].

4.5 Markov chain Monte Carlo

In variational inference, we work with the posterior distribution by approximating it using a simpler distribution. In order to avoid the approximation, we must directly draw samples from the true posterior distribution, as described in Eq. 2.

Markov chain Monte Carlo (MCMC) is a family of universal methods for drawing samples from a probability distribution. When applied to the posterior of a BNN, they allow us to draw (correlated) samples from the true posterior distribution, without having to approximate it using a simpler distribution.

The idea behind MCMC is to create a *Markov chain* that has an equilibrium distribution equal to the posterior distribution that we want to draw samples from – we will call this distribution the *target distribution*.

A Markov chain is any sequence of random variables θ_i that satisfies the Markov property: the probability of moving to the next state $(t+1)$ depends only on the current state (t) and not on the previous states $(1 \dots t)$. Eq. 20 defines this property formally.

$$p(\theta_{t+1}|\theta_1, \dots \theta_t) = p(\theta_{t+1}|\theta_t) \tag{20}$$

In MCMC, we define a process that satisfies the Markov property, and thus when we draw samples from it, we get a Markov chain. We define the process by its *transition kernel*, which is the probability distribution over the next state given the current state, $p(\theta_{t+1}|\theta_t)$. We will only focus on time-homogeneous chains, where the transition kernel doesn't change over time.

In order to draw samples $\theta_0 \dots \theta_{n-1}$ from a Markov chain defined by the transition kernel $p(\theta_{t+1}|\theta_t)$, we use the following algorithm:

Algorithm 1 Generating samples from a Markov chain

```
$\theta_0 \leftarrow \theta_{\text{init}}$
for $i \leftarrow 1 \dots n_{\text{samples}}$ do
    resample $\theta_i \sim p(\theta_i|\theta_{i-1})$
end for
```

The goal is to define a process whose unique *equilibrium distribution* is equal to the target distribution $\pi(\theta)$ that we want to draw samples from. This means that for any θ_0, as $n \to \infty$, $p(\theta_n|\theta_0) \xrightarrow{d} \pi(\theta)$. This property will be met iff the chain is *π-irreducible*, *aperiodic*, and *π-invariant*.

A chain is π-irreducible iff any state x that has a non-zero probability in the target distribution $\pi(\theta) > 0$ can be reached from any other state of the chain θ_0 in a finite number of steps n:

$$\pi(\theta) > 0 \Rightarrow \exists\ n,\ p(\theta_n = \theta|\theta_0) > 0 \tag{21}$$

A chain is periodic if it has a state θ s.t. the return to this state can only happen in a multiple of k steps, where $k > 1$. In all of the Markov chains that we consider, it is possible to transition from any state to any other state in a single step. This implies that the chains are *both* π-irreducible *and* aperiodic.

The last property that the Markov chain needs to meet is π-invariance. It means that once we reach the equilibrium state of the chain, the chain will remain in the equilibrium:

$$\theta_t \sim \pi(\theta) \Rightarrow \theta_{t+1} \sim \pi(\theta) \tag{22}$$

Let's denote our target posterior distribution as $\pi(\theta)$. Then, once we define a π-irreducible, aperiodic, and π-invariant chain, running algorithm 1 will yield samples that are asymptotically distributed as the posterior.

4.5.1 Metropolis-Hastings

The simplest and most common MCMC algorithm is *Metropolis-Hastings.* It is a general family of algorithms that ensure π-invariance. Two out of the three MCMC algorithms discussed in this essay belong to this family.

First, let's denote two arbitrary states in $\pi(\theta)$ as θ and θ'. Next, let's define a *proposal distribution* $Q(\theta'|\theta)$ and *acceptance probability* $A(\theta'|\theta)$. When $\theta' \neq \theta$, the transition *density* is equal to the product of the proposal density and the acceptance probability, as described in Eq. 23. The event $\theta' = \theta$ happens with *probability mass* $1 - \int_{-\infty}^{\infty} Q(\theta'|\theta)A(\theta'|\theta)d\theta'$.

$$p(\theta'|\theta) = Q(\theta'|\theta)A(\theta'|\theta) \tag{23}$$

One way to satisfy π-invariance is to make the probability of observing state θ followed by state θ' is the same as θ' followed by θ. This condition is called *detailed balance / reversibility* and is defined in Eq. 24.

$$\begin{aligned} p(\theta'|\theta)\pi(\theta) &= p(\theta|\theta')\pi(\theta') \\ \frac{p(\theta'|\theta)}{p(\theta|\theta')} &= \frac{\pi(\theta')}{\pi(\theta)} \\ \frac{Q(\theta'|\theta)A(\theta'|\theta)}{Q(\theta|\theta')A(\theta|\theta')} &= \frac{\pi(\theta')}{\pi(\theta)} \\ \frac{A(\theta'|\theta)}{A(\theta|\theta')} &= \frac{Q(\theta|\theta')\pi(\theta')}{Q(\theta'|\theta)\pi(\theta)} \end{aligned} \tag{24}$$

By setting $A(\theta'|\theta)$ to the value defined in Eq. 25, we ensure that detailed balance (Eq. 24) always holds, hence our chain is π-invariant.

$$A(\theta'|\theta) := \min\left(1, \frac{Q(\theta|\theta')\pi(\theta')}{Q(\theta'|\theta)\pi(\theta)}\right) \tag{25}$$

The different flavors of Metropolis-Hastings differ only in their proposal distribution $Q(\theta'|\theta)$. The acceptance probability is always the one defined in Eq. 25. A very useful property of this acceptance probability is that $\pi(\theta)$ needs to be known only up to a multiplicative constant. The posterior, as defined in Eq. 2 involves an intractable integral over θ, which has potentially millions of dimensions. Fortunately, the value of this integral is constant w.r.t. θ, so it cancels out in Eq. 25. Hence, when working with $\pi(\theta)$, we can evaluate it as a product of the likelihood multiplied by the prior, without having to compute the evidence.

The samples drawn by MCMC are dependent, by the nature of a Markov chain. However, in order to get a good approximation of the posterior predictive distribution, as described in section 4.3, we would ideally like the samples to be uncorrelated. The reason behind this is that a high autocorrelation of samples implies that the chain explores the posterior slowly. If the chain is currently in θ, subsequent samples from the chain will still be close to θ, rather than exploring the full distribution. A different view on this is that we would prefer the chain to have a large average step size between each of its samples. In order to achieve this, we must design a proposal distribution that proposes large steps that have a high acceptance probability.

4.5.2 Random-walk Metropolis-Hastings

The simplest Metropolis-Hastings algorithm is the *Random-walk Metropolis-Hastings* (RWMH). In RWMH, the proposal distribution over θ' is a normal distribution centered around x: $Q(\theta'|\theta) \sim N(\theta, \sigma^2)$. A useful property of the normal distribution is that its density at θ only depends on the distance of θ from the mean of the distribution. As a result, $Q(\theta'|\theta) = q(\theta|\theta')$, so Eq. 25 simplifies to Eq. 26. The left side of Fig. 11 illustrates the RWMH algorithm.

$$A(\theta'|\theta) = \min\left(1, \frac{\pi(\theta')}{\pi(\theta)}\right) \tag{26}$$

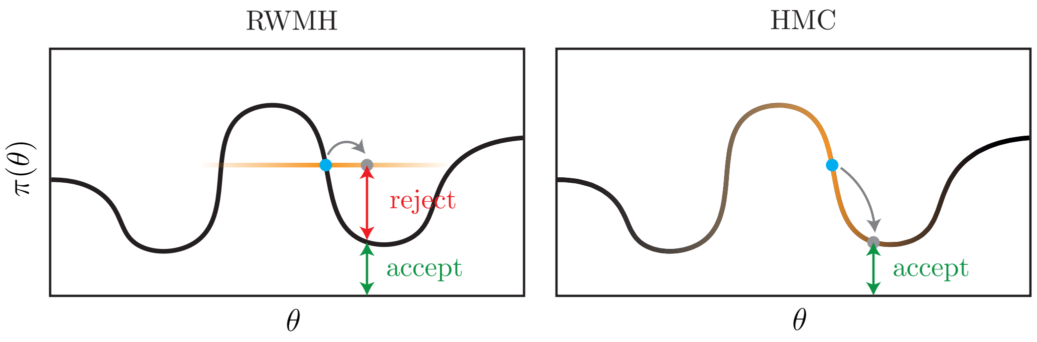


Figure 11: An illustration of Random-walk Metropolis-Hastings (RWMH) on the left and Hamiltonian Monte Carlo (HMC) on the right. The current state is represented by the blue point and the proposal distribution is illustrated by the yellow line. In RWMH, the proposal's acceptance probability depends on the height difference between the proposal and the target distribution. If the proposal is substantially higher than the target distribution, it will likely get rejected. If it is lower, it will always get accepted. Meanwhile, HMC proposes samples that lie exactly on the target distribution, so they always get accepted.

The only hyperparameter of the RWMH algorithm is the covariance matrix of the proposal distribution, σ^2. The simplest choice for the covariance matrix is a constant multiplied by the identity matrix, meaning the proposals for each component are independent and have the same variance. While under some conditions, there might exist better choices of σ^2, in different conditions, these might be inferior to the simple independent proposals with constant variance. Hence, for simplicity, we will only focus on independent proposals with constant variance. Hence, the only hyperparameter that remains is a single number that represents the standard deviation of each dimension of the proposal distribution – this is called the *step size*.

Setting the step size too low will result in tiny steps that always get accepted. Setting it too high will result in large steps that always get rejected. So there always exists an optimum step size that offers the best tradeoff between these qualities. In high dimensions, assuming a Gaussian target distribution, the optimum step size is the one that results in roughly 23% acceptance rate [10]. After optimal tuning, the computational complexity of RWMH scales linearly with the number of dimensions [11].

The complete RWMH algorithm is described below:

Algorithm 2 RWMH

```
for $i \leftarrow 1 \ldots n_{\text{samples}}$ do                      ▷ generate $n_{\text{samples}}$ from target distribution
    resample $\theta' \sim N(\theta, \sigma^2)$                           ▷ propose new parameters
    with prob. $\min(1, \frac{\pi(\theta')}{\pi(\theta)})$, set $\theta \leftarrow \theta'$       ▷ accept / reject the proposal
    $\theta_i \leftarrow \theta$                                          ▷ save new value
end for
```

4.5.3 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC) exploits the geometry of the target distribution to propose arbitrarily large steps with acceptance probability exactly equal to one [12]. This is illustrated on the right side of Fig. 11. In contrast to RWMH, the complexity of HMC scales like $d^{0.25}$ with the number of dimensions.

The idea is that instead of drawing samples directly from our target distribution $\pi(\theta)$, we define a new variable v, called the *momentum*, and we draw samples from the joint distribution $\pi(\theta, v) := \pi(\theta)\pi(v)$. Once we have the joint samples (θ, v), we can discard the momentum v and we are left with samples from our target distribution θ.

HMC alternates between updating the momentum v on its own and jointly updating the position together with the momentum (θ, v). Both of these updates are performed in a Metropolis-Hastings fashion. We will denote the current values as θ, v and the proposed values as θ', v'.

When the momentum is updated on its own, it is simply drawn from a normal distribution centered around zero, $v' \sim N(0, \sigma^2)$. Also, by design, the proposal distribution of v is equal to the target distribution of v: $Q(v) \overset{d}{=} \pi(v)$. Using the formula for acceptance probability from Eq. 25, we see that the proposal always gets accepted:

$$\begin{aligned}
A(\theta, v'|\theta, v) &= \min\left(1, \frac{Q(\theta, v|\theta, v')\pi(\theta, v')}{Q(\theta, v'|\theta, v)\pi(\theta, v)}\right) \\
&= \min\left(1, \frac{Q(v)\pi(\theta)\pi(v')}{Q(v')\pi(\theta)\pi(v)}\right) \\
&= \min\left(1, \frac{Q(v)\pi(v')}{Q(v')\pi(v')}\right) \\
&= 1
\end{aligned} \tag{27}$$

The joint proposal for (θ, v) is more complicated. In order to understand it, it helps to think of (θ, v) as the state of a physical particle rolling along a hill. θ is the particle's position, v is its momentum, and the shape of the hills is described by the target distribution. When we updated the particle's momentum, it was like flicking it in a random direction. In order to update its position and momentum together, we will simply let the particle roll for a while and then we will record its new position and momentum. The whole system is designed in such a way that no matter where the particle ends up, its new position and momentum will get accepted will probability one.

First, we need to define some quantities: $U(\theta)$ – the *potential energy* of the particle, $K(v)$ – the *kinetic energy* of the particle, and $H(\theta, v)$ – the *Hamiltonian* of the particle:

$$\begin{aligned}
U(\theta) &= -\log \pi(\theta) \\
K(v) &= \frac{1}{2}||v||^2 \\
H(\theta, v) &= U(\theta) + K(v)
\end{aligned} \tag{28}$$

Following the above equations, we can derive the relationship between the Hamiltonian and the target distribution:

$$\begin{aligned}
H(\theta, v) &= -\log \pi(\theta) + \frac{1}{2}||v||^2 \\
\pi(\theta, v) &= \exp -H(\theta, v) \\
&= \exp\left(\log \pi(\theta) - \frac{1}{2}||v||^2\right) \\
&= \pi(\theta)\frac{1}{2}||v||^2 \\
&\propto \pi(\theta)\pi(v)
\end{aligned} \tag{29}$$

When particle's state evolves through *Hamiltonian dynamics*, the particle's Hamiltonian is conserved, meaning that the density of the target distribution is conserved. Hamiltonian dynamics are defined by the following equations:

$$\begin{aligned}
\frac{d\theta}{dt} &= \frac{\partial H}{\partial v} \\
\frac{dv}{dt} &= -\frac{\partial H}{\partial \theta}
\end{aligned} \tag{30}$$

Additionally, Hamiltonian dynamics are time-reversible, so $Q(\theta', v'|\theta, v) = Q(\theta, v|\theta', v')$ and they preserve volume, so the ratio of probabilities before and after the transformation is the same as the ratio of probability densities . These properties, combined with the fact that the Hamiltonian is conserved, imply that the acceptance rate of the new state (θ', v') is exactly one:

$$\begin{aligned} A(\theta', v'|\theta, v) &= \min\left(1, \frac{Q(\theta, v|\theta', v')\pi(\theta', v')}{Q(\theta', v'|\theta, v)\pi(\theta, v)}\right) \\ &= 1 \end{aligned} \tag{31}$$

Unfortunately, Eq. 30 are differential equations that cannot be computed exactly – they must be numerically approximated. As a result, the Hamiltonian is not perfectly conserved, hence the true acceptance probability is close to one, but not exactly one:

$$\begin{aligned} A(\theta', v'|\theta, v) &= \min\left(1, \frac{Q(\theta, v|\theta', v')\pi(\theta', v')}{Q(\theta', v'|\theta, v)\pi(\theta, v)}\right) \\ &= \min\left(1, \frac{\pi(\theta', v')}{\pi(\theta, v)}\right) \\ &= \min\left(1, \frac{\exp -H(\theta', v')}{\exp -H(\theta, v)}\right) \\ &= \min\left(1, \exp\left(-H(\theta', v') + H(\theta, v)\right)\right) \\ &= \min\left(1, \exp\left(\log\pi(\theta') - \frac{1}{2}||v'||^2 - \log\pi(\theta) + \frac{1}{2}||v||^2\right)\right) \\ &= \min\left(1, \frac{\pi(\theta')}{\pi(\theta)}\exp\left(\frac{1}{2}||v||^2 - ||v'||^2\right)\right) \end{aligned} \tag{32}$$

In order to get the acceptance probability as close to one as possible, we must use an efficient way to simulate Hamiltonian dynamics, as described in Eq. 30. A standard way to simulate differential equations is Euler's method:

$$\begin{aligned} v_{t+\varepsilon} &\leftarrow v_t + \varepsilon\frac{dv_t}{dt}(t) = v_t - \varepsilon\frac{\partial U}{\partial\theta}(\theta_t) \\ \theta_{t+\varepsilon} &\leftarrow \theta_t + \varepsilon\frac{d\theta_t}{dt}(t) = \theta_t - \varepsilon v_t \end{aligned} \tag{33}$$

However, there exists a more accurate method to simulate Hamiltonian dynamics: the leapfrog algorithm [12]. First, we do a half-step update of the momentum, then a full update of the position (using the new momentum value), and finally another half-step update of the momentum (using the new position).

$$\begin{aligned} v_{t+\varepsilon/2} &\leftarrow v_t - \frac{\varepsilon}{2}\frac{\partial U}{\partial\theta}(\theta_t) \\ \theta_{t+\varepsilon} &\leftarrow \theta_t + \varepsilon v_{t+\varepsilon/2} \\ v_{t+\varepsilon} &\leftarrow v_{t+\varepsilon/2} - \frac{\varepsilon}{2}\frac{\partial U}{\partial\theta}(\theta_{t+\varepsilon}) \end{aligned} \tag{34}$$

Conveniently, each transformation in Eq. 34 is a shear transformation, so the leapfrog algorithm preserves volume exactly. Also, the algorithm is time-reversible by negating v, running it for the same number of steps, and negating v again. Both of these properties are required for Eq. 32 to hold [12].

In general, HMC has three hyperparameters: step size σ, number of leapfrog steps n, and *mass matrix* m. For simplicity, we only consider the use of a unit mass matrix, meaning we don't have to include it in any equations. Under some conditions, using a non-unit mass matrix might be beneficial, as it means that the momentum is updated at different rates for each parameter. However, there is no universally optimum value of the mass matrix.

The product of step size with the number of steps is called the *trajectory length.* It dictates the average arc length that the particle travels during Hamiltonian dynamics. If the trajectory length is too small, the particle will make tiny steps, resulting in highly correlated samples of θ. On the other hand, if the trajectory length is too large, the particle might repeatedly oscillate around its origin, as if it were rolling around the trough of a deep valley. This idea is further discussed in section 4.5.4. In practice, we might set the trajectory length roughly similar to what we would expect the standard deviation of the posterior to be [8]. This is simple to estimate when the prior dominates the likelihood, but harder when the likelihood dominates the prior.

The computational complexity of HMC scales linearly with the number of steps. Hence, given a fixed trajectory length, we would prefer the step size to be as large as possible. However, as the step size increases, the leapfrog approximation error increases, which reduces the acceptance rate. The solution is to use a step size as large as possible while maintaining a good acceptance rate – e.g. 80% [8].

A pseudocode implementation of HMC and the leapfrog method are provided in Algorithms 3 and 4.

Algorithm 3 HMC

```
for $i \leftarrow 1 \dots n_{\text{samples}}$ do        ▷ generate $n_{\text{samples}}$ from target distribution
    resample $v \sim N(0, \sigma^2)$        ▷ resample momentum
    $\theta', v' \leftarrow$ Leapfrog$(\theta, v)$        ▷ propose new parameters
    with prob. $\min\left(1, \frac{\pi(\theta')}{\pi(\theta)} \exp\left(\frac{1}{2}||v||^2 - ||v'||^2\right)\right)$, set $\theta \leftarrow \theta'$        ▷ accept / reject proposal
    $\theta_i \leftarrow \theta$        ▷ save new value
end for
```

Algorithm 4 Leapfrog

```
for $i \leftarrow 1 \dots n_{\text{steps}}$ do        ▷ do $n_{\text{steps}}$ steps
    $v \leftarrow v - \frac{\varepsilon}{2} \frac{\partial U}{\partial \theta}$        ▷ do a half-step update of momentum
    $\theta \leftarrow \theta + v$        ▷ do a full-step update of parameters
    $v \leftarrow v - \frac{\varepsilon}{2} \frac{\partial U}{\partial \theta}$        ▷ do a half-step update of momentum
end for
```

4.5.4 No-U-Turn Sampler

The *No-U-Turn Sampler* (NUTS) is a modification of HMC that adapts the trajectory length (by adapting the number of steps) depending on the local geometry of the target distribution [13]. The idea is to always run Hamiltonian dynamics *just long enough* – until the point when running the simulation for any longer would result in the particle moving *closer* to its origin, rather than away from it. Let (θ, v) denote the state of the particle at the start of Hamiltonian dynamics and (θ', v') denote the current state of the particle under Hamiltonian dynamics.

We can measure the distance of θ' to θ as $||\theta' - \theta||^2$, so the stopping condition is $\frac{d}{dt}||\theta' - \theta||^2 < 0$. When the stopping condition is reached, we say that the particle makes a *U-turn*. The stopping condition can be expressed in terms of θ', θ, v' as derived below:

$$\begin{aligned} \frac{d}{dt}||\theta' - \theta||^2 &< 0 \\ \frac{d}{dt}\left((\theta' - \theta) \cdot (\theta' - \theta)\right) &< 0 \\ 2(\theta' - \theta) \cdot \frac{d}{dt}(\theta' - \theta) &< 0 \\ (\theta' - \theta) \cdot v' &< 0 \end{aligned} \tag{35}$$

It would be convenient to simply leapfrog until the particle makes a U-turn and then use the last leapfrog step as the proposal for (θ, v). However, this could create a scenario where a particle traveling forward in time stops at a different location than a particle traveling backward in time, thus breaking detailed balance. Fig. 12 illustrates one such scenario. In order to satisfy detailed balance, we must make sure that the probability of proposing $(\theta, v) \rightarrow (\theta', v')$ is the same as $(\theta', v') \rightarrow (\theta, v)$. In HMC, this was satisfied by running the (time-reversible) leapfrog algorithm for a fixed number of steps. In order for NUTS to stop when it detects a U-turn *and* preserve detailed balance, it needs to operate in a more complicated fashion.

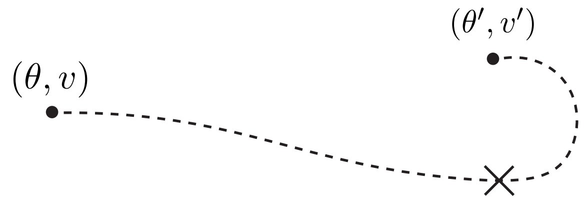


Figure 12: If a particle travels from (θ, v) along the dashed leapfrog trajectory, it will make a U-turn at the point (θ', v'). However, if the particle started at (θ', v') and went backward in time, it would make a U-turn at the point denoted by a large cross, not at the original starting point (θ, v).

NUTS works by recursively growing a balanced binary tree. First, we start with the initial position, pick a random direction (forward or backward) and leapfrog 1 step. Then, we again choose a random direction (forward or backward) and leapfrog 2 steps. Then, we choose a random direction and leapfrog 4 steps... in general, at iteration i, we chose a random direction and leapfrog 2^i steps. This process continues until we reach a U-turn.

This process implicitly grows a binary tree, where each leaf represents a single state of the particle (θ', v'). At each iteration of NUTS, the size of the tree is doubled: it either grows forward or backward by 2^i leaves. When the tree is grown forward, we leapfrog forward in time starting from the rightmost state $(\theta_{\text{right}}, v_{\text{right}})$ and append the new states to the right side of the tree. When the tree is grown backward, we leapfrog backward in time, starting from the leftmost state $(\theta_{\text{left}}, v_{\text{left}})$ and sequentially append the new states to the left side of the tree. The leaves of the binary tree are always indexed as $0 \ldots (n-1)$, from left to right, where n is the number of leaves in the tree. This means that the index of a given leaf can change as the tree grows since the leaf's distance from the leftmost leaf can change. Even though we alternate between growing the binary tree forward and backward, the resulting tree will be a consistent leapfrog path, i.e. running the leapfrog algorithm from $(\theta_{\text{left}}, v_{\text{left}})$ will always reach $(\theta_{\text{right}}, v_{\text{right}})$. Fig. 13 illustrates the binary tree created after the fifth iteration of NUTS.

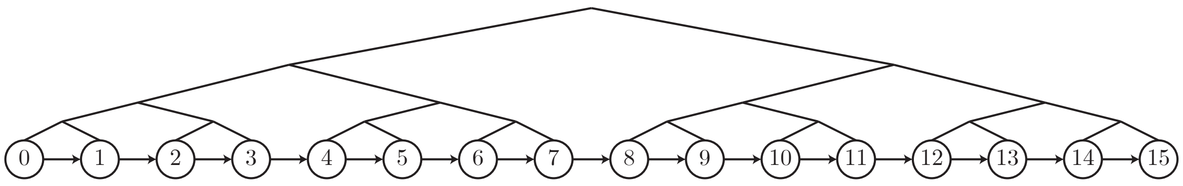


Figure 13: A binary tree that results from running NUTS for 5 iterations. It has $2^5 = 16$ leaves. The leaves are connected by arrows to indicate that the path along the leaves is a valid leapfrog trajectory: the ith leaf ($i \in \{0 \ldots 15\}$) is the result of running the leapfrog algorithm for i steps from leaf 0.

In each iteration of NUTS, in addition to growing the tree, we check for U-turns. More specifically, we check the U-turn condition for the leftmost and rightmost leaves of all balanced subtrees. Fig. 14 illustrates each pair of leaves that need to be checked (in a tree of height 5) by a colored line. When checking for a U-turn, we check whether continuing the Hamiltonian dynamics in *either* direction would result in the distance between the leaves decreasing. When checking the pair of leaves $(\theta_{\text{left}}, v_{\text{left}})$ and $(\theta_{\text{right}}, v_{\text{right}})$, this is derived in Eq. 36.

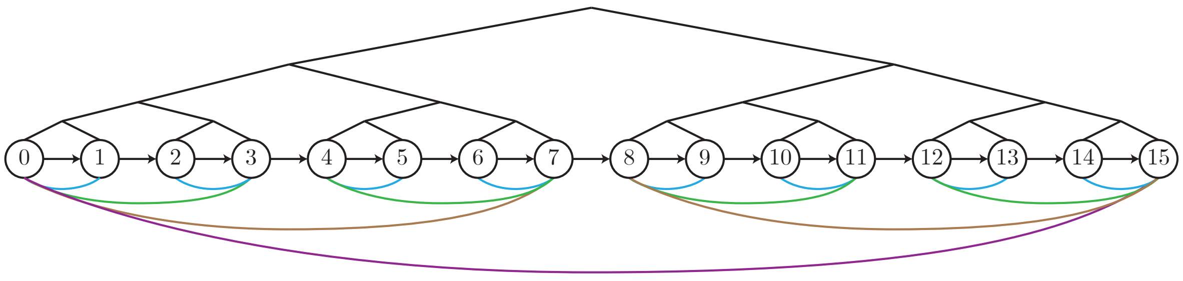


Figure 14: A binary tree that results from running NUTS for 5 iterations. Colored lines connect each pair of leaves that need to be checked for a U-turn in a tree of height 5.

$$(\theta_{\text{right}} - \theta_{\text{left}}) \cdot v_{\text{left}} < 0 \quad \text{or} \quad (\theta_{\text{right}} - \theta_{\text{left}}) \cdot v_{\text{right}} < 0 \tag{36}$$

In order to preserve detailed balance, once we detect a U-turn (or a different termination criterion is reached, as discussed later), the tree stops growing and NUTS proposes a new state sampled from all *candidate states* $\mathcal{C}$ along the leapfrog trajectory. The set of candidates states is a subset of all leaves in the tree $\mathcal{B}$. There is a deterministic process for mapping the set of all leaves to the set of candidate states $\mathcal{B} \to \mathcal{C}$. In Fig. 12, for example, if the initial state was (θ, v) and the particle leapfrogged until it reached (θ', v'), the part of the trajectory between the cross and (θ', v') would not be considered part of valid samples $\mathcal{C}$. The reason is that if a particle started at (θ', v') and traveled toward (θ, v), it would *always* stop at the cross, hence it would *never* reach (θ, v). This scenario would break detailed balance, hence it must be avoided. Recall that NUTS checks for a U-turn between the leftmost and rightmost leaves of all balanced subtrees each time the full tree doubles. If the U-turn is detected between the leftmost and rightmost leaves of the full tree, then any state along the leapfrog trajectory may be proposed without breaking detailed balance. However, if a U-turn is detected between anywhere *within* the leaves that were just added to the full tree, *only* those leaves may be proposed that existed *before* the tree doubling.

While in HMC the mapping $(\theta, v) \rightarrow (\theta', v')$ is deterministic, in NUTS, it is random. Still, in both cases $Q(\theta', v'|\theta, v) = Q(\theta, v|\theta', v')$. In NUTS, this is achieved by sampling (θ', v') uniformly from $\mathcal{C}$ and designing $p(\mathcal{B}, \mathcal{C} \mid \theta, v, u)$ such that if $(\theta, r) \in \mathcal{C}$ and $(\theta', r') \in \mathcal{C}$, then for any $\mathcal{B}$, $p(\mathcal{B}, \mathcal{C} \mid \theta, v) = p(\mathcal{B}, \mathcal{C} \mid \theta', v')$. In other words, any two states in $\mathcal{C}$ have the same probability of generating the full tree $\mathcal{B}$, and (θ', v') is sampled uniformly from $\mathcal{C}$.

Even though NUTS is closely related to HMC, it does not use Metropolis-Hastings sampling. Instead, it uses *slice sampling* [14]. Slice sampling is motivated by the observation that in order to draw samples from a distribution, we can take uniform samples from the area under the curve of its density function. This is achieved by alternately sampling vertically and horizontally from this area, as illustrated in Fig. 15. When sampling horizontally, we map $(\theta, v) \rightarrow (\theta', v')$. In order to sample vertically, we define a *slice variable* u, which is a scalar variable representing the height. The joint distribution of (θ, v, u) is uniform under the curve $\pi(\theta, v)$:

$$p(\theta, v, u) \propto \mathbb{I}\left(u \in [0, \pi(\theta, v)]\right) \tag{37}$$

When sampling vertically under the curve, we need to draw samples of the slice variable u conditional on (θ, v). Since the joint distribution (θ, v, u) is distributed uniformly under $\pi(\theta, v)$, $u|(\theta, v)$ is also distributed uniformly under $\pi(\theta, v)$: $u|(\theta, v) \sim \text{Uniform}[0, \pi(\theta, v)]$. Similarly, in order to sample horizontally under the curve, we draw samples of (θ, v) conditional on the slice variable u. $(\theta, v)|u$ is distributed uniformly along the region where the slice variable falls under the curve: $(\theta, v)|u \sim \text{Uniform}\{(\theta, v), \pi(\theta, v) \geq u\}$. However, to improve numerical stability, is is better to work with $\log u$ instead of u. The distribution of $\log u$ is derived in Eq. 38.

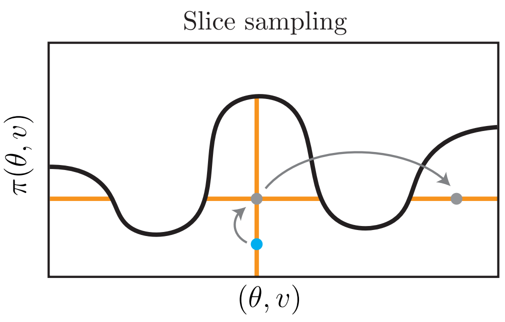


Figure 15: During slice sampling, we alternately sample vertically and horizontally from the are under the target density function. For example, starting at the blue point, we would sample vertically along the vertical orange line and arrive at the first grey point. Then we would sample horizontally along the horizontal orange line, arriving at the second grey point. This alternating cycle repeats indefinitely.

$$\begin{aligned} u &\sim \text{Unif}[0, \pi(\theta, v)] \\ \text{let } x := u/\pi(\theta, v) &\Rightarrow x \sim \text{Unif}[0, 1],\; -\log(x) \sim \text{Exp}(1) \\ \log(u) &= \log(\pi(\theta, v) \times x) = \log \pi(\theta, v) - (-\log(x)) \\ \log(u) &\sim \log \pi(\theta, v) - \text{Exp}(1) \end{aligned} \tag{38}$$

Technically, in NUTS, we don't sample (θ, v, u) directly. Instead, we performs *Gibbs sampling* over the joint distribution $(\theta, v, u, \mathcal{B}, \mathcal{C})$. Gibbs sampling simply means that we iteratively sample each variable from its *full-conditional* distribution, i.e. its distribution conditional on all the other variables. This is performed as described in Algorithm 5. All of the steps are valid Gibbs updates because they resample each variable (or set of variables) from their full-conditional distribution. In step 3, we first build the leapfrog trajectory $\mathcal{B}$. Then, when mapping $\mathcal{B} \to \mathcal{C}$, we only consider those states (θ', v') that satisfy detailed balance *and* constitute a valid update under slice sampling, i.e. only those where $u \leq \pi(\theta', v')$.

Algorithm 5 NUTS as Gibbs sampling over $(\theta, v, u, \mathcal{B}, \mathcal{C})$

```
for $i \leftarrow 1 \dots n_{\text{samples}}$ do
    resample $v \sim \mathcal{N}(0, \sigma^2)$
    resample $u \sim \text{Uniform}[0, \pi(\theta, v)]$
    resample $\mathcal{B}, \mathcal{C} \sim p(\mathcal{B}, \mathcal{C} \mid \theta, v, u)$
    resample $\theta, v \sim \text{Uniform}[\mathcal{C}]$
    set $\theta_i, v_i, u_i, \mathcal{B}_i, \mathcal{C}_i \leftarrow \theta, v, u, \mathcal{B}, \mathcal{C}$
end for
```

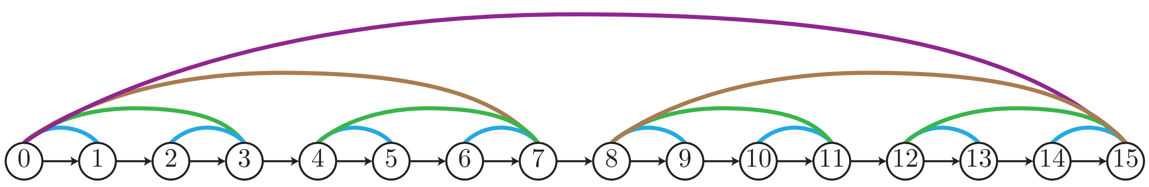
Also, in addition to U-turns, we add two additional stopping criteria. First, we define a maximum tree depth, which is equivalent to defining the maximum number of leapfrog steps. This is to ensure that the program terminates even in absence of U-turns and we don't run out of memory. Second, we define a maximum leapfrog error $\Delta_{\max}$. However, imposing a maximum error *does* violate detailed balance. Hence $\Delta_{\max}$ needs to be made very large, such that it is only reached if the simulation is poorly set up, i.e. by setting the step size too large.

4.5.5 Iterative NUTs

In the original NUTS algorithm, the process for building a tree, including all stopping conditions, is recursive. However, it is difficult to implement a recursive algorithm efficiently. The developers behind NumPyro (a probabilistic programming library) solved this problem by developing *Iterative NUTS*, which is an iterative formulation of NUTS, meaning it can be implemented much more efficiently [15].

The process of building a tree is described through a nested for loop. The outer loop iterates through tree height, $h \in \{1, 2, 4, 8, 16 \ldots\}$. Each height h corresponds to 2^h leaves (leapfrog steps). The inner loop iterates through each leaf, first generating it using the leapfrog algorithm, then checking for a U-turn. The outer loop is responsible for setting a forward / backward direction and checking whether any stopping condition was reached within the inner loop.

The main challenge in formulating NUTS as an iterative algorithm is to find an efficient way of checking U-turns. For example, imagine that we want to check for U-turns in a tree of height 5, as illustrated in Fig. 16. First, notice in the figure that each pair of leaves that we check for a U-turn consists of one leaf with an odd index and one leaf with an even index. This means that as we iterate through the leaves, we can use even nodes to store temporary state and odd nodes to check for U-turns against these previously-generated states. For example, at leaf 0, we would store its value and at leaf 1, we would check for a U-turn between leaf 0 and leaf 1. Notice that we do not need to store the value of leaf 1 because leaf 1 is not compared to any leaf with an index greater than 1.



base-10 index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
binary index:	0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1	1 0 0 0	1 0 0 1	1 0 1 0	1 0 1 1	1 1 0 0	1 1 0 1	1 1 1 0	1 1 1 1
bit count:	0	1	1	2	1	2	2	3	1	2	2	3	2	3	3	4
trailing bits:	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4

Figure 16: The 16 nodes correspond to the leaves of a height-5 tree. Colored lines indicate which pairs of leaves need to be checked for a U-turn. Each leaf is indexed in base 10 and base 2. The last two rows show the bit count and number of trailing bits of the binary representation of each index.

Since NUTS is based on a balanced binary tree, it is natural to index each leaf of the tree using binary numbers – the binary index of each leaf corresponds closely to the position of the leaf within the tree. For example, observe that the number of trailing bits of each even leaf dictates how many past leaves we need to check for a U-turn. For instance, leaf 1 has binary representation 01, hence its number of trailing bits is 1. This is because it is the rightmost leaf only within a subtree of height 1. On the other hand, leaf 7 has binary representation 111 and 3 trailing bits, because is the rightmost leaf in subtrees of height 1, 2, and 3. Hence, it must be compared to 3 other leaves when checking for U-turns.

Also, observe that we do not need to store the value of each even leaf. Instead, for a tree of height h, we can get away with storing just $\log_2 h$ leaves. We can use the bit count of each even leaf to decide where in this array it should be stored. We will store leaf 0 in the 0th element of the array. We must keep this leaf indefinitely and no other leaf has bitcount 0, so this value will never get overwritten. We will store each leaf with bitcount 1 in the first element of the array. There are multiple leaves with bitcount 1, so the first element of the array will get overwritten multiple times, but each time this happens, we no longer need to keep the previous values. In general, we will store *each* odd leaf as the bitcount(i)-th element in the array.

Lastly, we need to consider how to grow the tree backward, not just forward. When growing the tree forward, we leapfrog forward in time from the rightmost leaf of the tree. When growing the tree backward, we need to leapfrog backward in time from the leftmost leaf. Leapfrogging backward in time is equivalent to leapfrogging forward in time but setting a negative step size. Hence, to grow the tree in either direction, it suffices to always store only the leftmost and rightmost leaf. The direction of growth can only change when the tree doubles, in which case the only relevant leaf to store for checking U-turns is the leftmost / rightmost one. A simple way to implement this is to resample the direction each time the tree doubles and if the direction changes, simply switch the leftmost and rightmost leaves and negate the step size.

Putting all this together, Algorithm 6 describes a valid (but simplified) implementation of *Iterative NUTS*.

Algorithm 6 Iterative NUTS

```
for $i \leftarrow 1 \dots n_{\text{samples}}$ do    ▷ generate $n_{\text{samples}}$ from target distribution
    $\theta^+, \theta^-, \theta_{\text{out}} \leftarrow \theta$    ▷ initialize leftmost (−) and rightmost (+) leaf
    $v^+, v^-, v_{\text{out}} \leftarrow v$
    $n_{\text{valid leaves}} \leftarrow 1$    ▷ size of $\mathcal{C}$
    resample $\log u \sim \log \pi(\theta, v) - \text{Exp}(1)$    ▷ resample log of slice variable
    stop ← False    ▷ becomes true if any termination condition is reached
    resample $v \sim N(0, \sigma^2)$    ▷ resample momentum
    for $j \leftarrow 1 \dots 2^{\text{MaxTreeHeight}}$ do    ▷ loop through leaves
        if IsPowerOfTwo($j$) then    ▷ tree is about to double
            $\theta_{\text{out}}, v_{\text{out}} \leftarrow \theta_{\text{subtree out}}, v_{\text{subtree out}}$    ▷ leaves are in $\mathcal{C}$ only if there were no U-turns
            resample direction$_{\text{new}} \sim$ Uniform$[\{-1, 1\}]$    ▷ decide whether to leapfrog forward or backward
            if direction$_{\text{new}} \neq$ direction then    ▷ when direction changes, flip the tree
                $\theta^+, \theta^- \leftarrow \theta^-, \theta^+$
                $v^+, v^- \leftarrow v^-, v^+$
            end if
            direction ← direction$_{\text{new}}$
            checkpoints[0] ← $(\theta^-, v^-)$    ▷ checkpoint leftmost leaf
        end if
        $\theta^+, v^+ \leftarrow$ Leapfrog$(\theta^+, v^+, \text{direction})$    ▷ leapfrog a single step (forward or backward in time)
        if $j$ is even then    ▷ update checkpoints
            checkpoints[BitCount($j$)] ← $(\theta, v)$
        else if $j$ is odd then    ▷ check U-turns
            for $k \leftarrow$ BitCount($j$) − NumTrailingBits($j$) ... NumTrailingBits($j$) do
                $\theta^*, v^* \leftarrow$ checkpoints[$k$]
                stop ← stop or $(\theta^+ - \theta^*) \cdot v^* < 0$ or $(\theta^+ - \theta^*) \cdot v^+ < 0$
            end for
        end if
        stop ← stop or $\Delta_{\max} < \log u - \log \pi(\theta^+, v^+)$    ▷ check maximum leapfrog error
        if stop then
            $\theta_i, v_i \leftarrow \theta_{\text{out}}, v_{\text{out}}$    ▷ $\theta_{\text{out}}, v_{\text{out}}$ were sampled uniformly from the valid leaves
            break
        else
            if $\log u \leq \log \pi(\theta^+, v^+)$ then    ▷ slice sampling
                with prob. $1/n_{\text{valid leaves}}$, set $\theta_{\text{subtree out}}, v_{\text{subtree out}} \leftarrow \theta^+, v^+$    ▷ resample output leaf
                $n_{\text{valid leaves}} \leftarrow n_{\text{valid leaves}} + 1$
            end if
        end if
    end for
end for
```

5 Implementation

One of the strongest limitations of MCMC-based Bayesian neural networks (compared to standard neural networks) is that they are orders of magnitude computationally more expensive to train and run inference on [8]. For this reason, it is essential that any implementation of a BNN be as computationally-efficient as possible. The implementation behind this project utilizes several related techniques to achieve this (JIT, TPUs, SPMD), which are described below. All computations are implemented in *JAX* [16], an open-source machine learning library developed by Google.

JAX is designed to work with *pure* functions: functions whose only input is input parameters and the only output is the returned value. A pure function cannot have any side effects, i.e. it cannot modify any input or global variables. In order to implement pure functions, JAX relies on a pure pseudorandom number generator (PRNG). Instead of maintaining a global seed, the PRNG must be passed a *key* each time we use it to draw a random number. The function for drawing random numbers is also pure: given the same key, it always produces the same output. In order to draw a new random number, the key must be *split* into a new key. This greatly simplifies parallelizing code across multiple devices.

In order to reduce computation time, the computations are split across 8 Tensor Processing Unit (TPU) cores. A TPU is an application-specific integrated circuit (ASIC) developed by Google optimized to work with neural networks. Unlike Graphics Processing Units (GPUs), TPUs are deterministic, making experiments reproducible and parallelization simple. The computations are split up across the cores using *single program, multiple data* (SPMD) parallelism. Namely, the whole dataset is split into eight batches, and each batch is processed separately by a single core, using the same program – hence the name 'single program, multiple data'. The only instance when the cores need to be synchronized is when computing the likelihood – each core uses a different dataset batch, so it arrives at a different value. Once each core computes *its* likelihood, the values are synchronized across all cores and each core continues to work independently. Using all eight cores in this fashion results in more than a four-fold speedup over using a single core.

Lastly, JAX can Just-In-Time (JIT) compile Python functions in order to reduce overhead. However, using Python control flow is not allowed – JAX's custom control flow must be used instead. Recursion is not allowed at all. Thus, for example, to JIT compile the No-U-Turn Sampler, it must be implemented without recursion, as an iterative algorithm.

Running JIT-compiled code on a single TPU core provides roughly a 100-fold speedup over the 'state-of-the-art platform for high-performance statistical computation' *Stan* [17]. Using SPMD parallelism across 8 TPU cores provides roughly an additional 4-fold speedup.

6 Results

Until now, we have only discussed the theory behind NNs and BNNs and applied them to a toy regression dataset. Here, we use a real-world dataset: the *UCI Condition Based Maintenance of Naval Propulsion Plants Data Set* [18]. It is a regression dataset with 14 predictors and a single real-valued response variable. We use 10,740 'train' observations to fit each model and 1,194 'test' observations to assess model performance. The train and test observations are separated to obtain an unbiased estimate of model performance and discourage overfitting.

The goal is to benchmark the quality of predictions between a NN and a BNN and compare different approximations of a BNN posterior. Namely, we compare a NN trained using SGD, a deep ensemble (which is one of the best performing variational models [8]), and a BNN sampled using RWMH, HMC, and NUTS.

We use an MLP with 3 hidden layers, each consisting of 50 nodes, and ReLU activation. The network outputs the mean and standard deviation of the likelihood, which is set to be normal. The BNN has a $N(0, 1)$ prior over parameters.

6.1 Comparison of MCMC methods

For MCMC, we generated 5 chains of length 100 using each method. Afterward, the first 20 samples from each were discarded, since these depend heavily on the initialization rather than the target distribution – this technique is called *burn in*. In HMC, we set the step size to obtain an acceptance rate of roughly 80% and the number of steps such that generating the 100 samples would take roughly 4 minutes. For NUTS, we used the same step size and set the maximum number of steps to again obtain a running time of roughly 4 minutes. For RWMH, the step size was to obtain roughly a 23% acceptance rate. In order to use the same computation budget as HMC and NUTS, we sampled 3,000,000 points from the posterior and only kept 100 of them, i.e. 0.003%. In each method, the 5 chains had the same starting parameters, obtained by running SGD 5 times starting from a different random initialization.

For reference, the training time using SGD was just 15 seconds – less than a tenth of the compute budget spent on MCMC. In higher dimensions, HMC (one of the most efficient MCMC methods) is roughly 2 – 3 orders of magnitude more expensive than SGD [8].

In section 3, we established that the ideal MCMC algorithm would generate samples from the target distribution (the posterior of a BNN) with autocorrelation as close to zero as possible. Fig. 17 shows the actual autocorrelation that we obtained using RWMH, HMC, and NUTS.

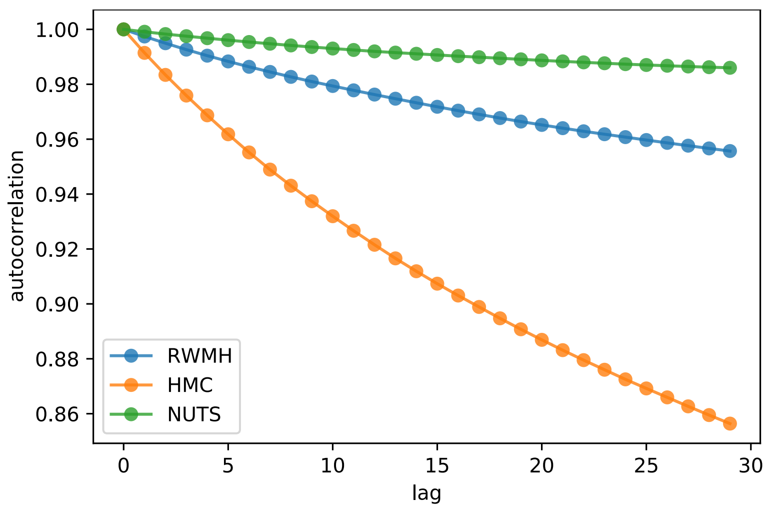


Figure 17: Sample autocorrelation. For each algorithm, the autocorrelation was estimated for each parameter of 5 independent chains and the estimates were then averaged across parameters and chains. The mean of each parameter was computed using all 5 chains, not independently for each chain as would be typically done.

Since the posterior has $d = 5952$ dimensions, it is natural the HMC outperforms RWMH, given that HMC scales like $d^{0.25}$ and RWMH scales like d [11]. However, the relatively poor performance of NUTS is surprising. The main reason is that NUTS never detected a single U-turn. Most likely, the posterior is so high dimensional that reaching a U-turn is very difficult. In order to reach a U-turn, we would need to use more than $2^{13} = 8192$ steps, which is both computationally intensive and causes the leapfrog error to grow. Given the same number of steps, NUTS is substantially slower than HMC because NUTS has a large overhead. In addition to leapfrogging, NUTS must check for U-turns using complex control flow. Moreover, in NUTS, most of the leapfrog steps are wasted: we leapfrog in both directions and then sample θ from a subset of the visited states. The way θ is sampled from $\mathcal{C}$ in our implementation is not as efficient as it could be [13]. A more efficient implementation of NUTS would get closer to HMC performance, but as long as there are no U-turns, it will always perform worse.

Another way to visualize the relative performance of these algorithms is by looking at the history of a single parameter from the target distribution. In Fig. 18, we see that HMC explores the target distribution much faster than RWMH, which is still faster than NUTS.

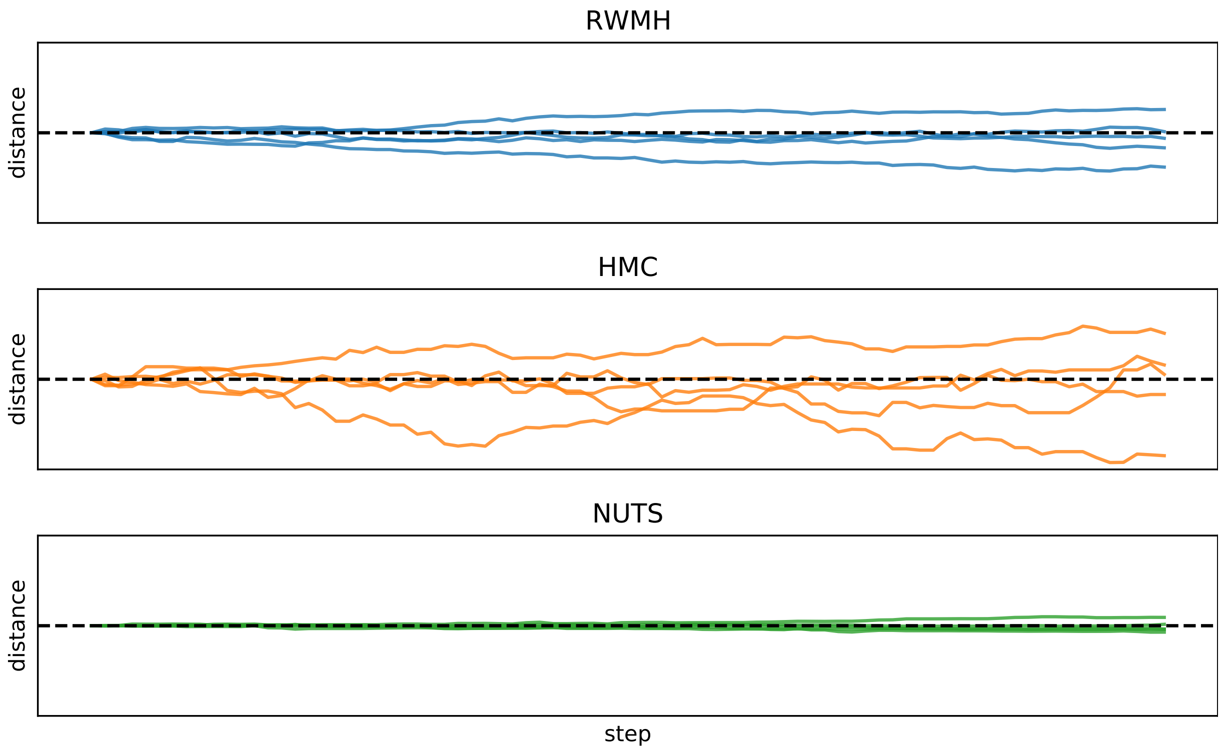


Figure 18: History of a single parameter from the target distribution. For each algorithm, 5 independent chains are displayed and each is centered using its first value. Each chain had a different initialization, so the curves only show the relative distance of each parameter from its initial position.

Additionally, Fig. 19 shows the distribution of the sampled values from the first chain using each algorithm. The values from each algorithm are close to each other since they all used SGD initialization. HMC has the largest spread since it was able to explore the largest area of the target distribution.

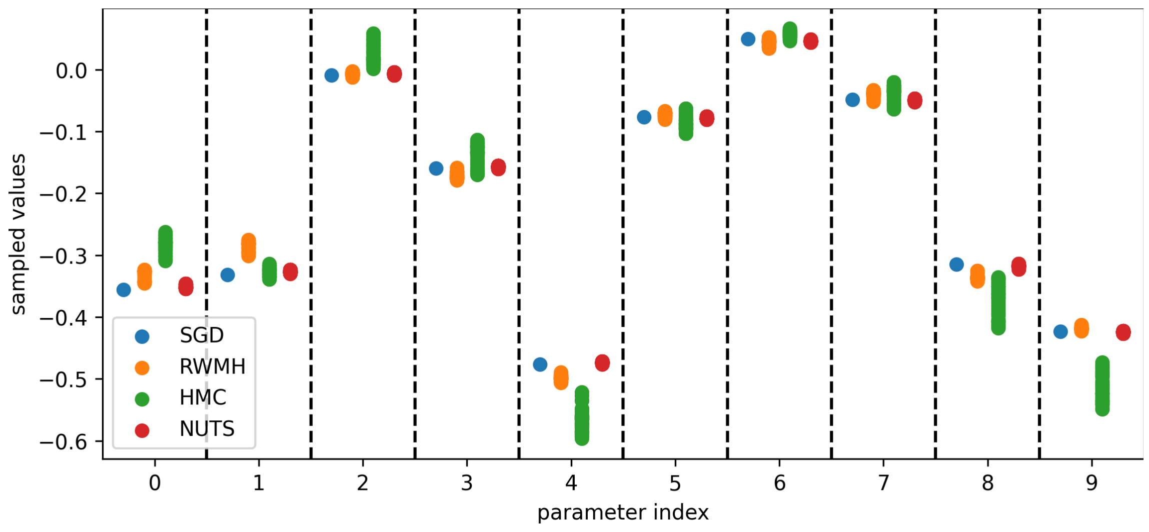


Figure 19: All sampled values of 10 parameters from the first chain of each algorithm. Dashed vertical lines separate parameters while colors separate algorithms. The sampled values are plotted vertically as points, with positions based on values.

A common way to estimate how well a single MCMC chain has explored its target distribution is the Gelman-Rubin $\hat{R}$ [19] diagnostic. The idea is to compare the within-chain variance W to the between-chain variance B of some function of the parameters $\psi(\theta)$. The between-chain variance must be estimated by running multiple chains – in our case, five. Let $m \in \{1 \dots M\}$ index each chain and $n \in \{1 \dots N\}$ index the nodes of each chain. And let $\bar{\psi}_m$ be the average of ψ for chain m across all of its nodes and $\bar{\psi}$ be the average of ψ across all nodes and chains. Then, $\hat{R}$ can be defined as the (square root of the) ratio of the target distribution's variance to the within-chain variance. If the chain explores the target distribution well, $\hat{R}$ will approach one:

$$\bar{\psi}_m := \frac{1}{N} \sum_n \psi_{mn} \tag{39}$$

$$\bar{\psi} := \frac{1}{MN} \sum_{m,n} \psi_{mn} \tag{40}$$

$$B := \frac{N}{M-1} \sum_m (\bar{\psi}_m - \bar{\psi})^2 \tag{41}$$

$$W := \frac{1}{M(N-1)} \sum_{m,n} (\psi_{mn} - \bar{\psi}_m)^2 \tag{42}$$

$$\hat{R} = \sqrt{\frac{\frac{N-1}{N} W + \frac{1}{N} B}{W}} \tag{43}$$

Table 1 shows the $\hat{R}$ computed for both parameters and predictions for each MCMC method. The $\hat{R}$ of parameters is very large: for reference, a value of less than 1.1 is typically desired. However, the $\hat{R}$ of predictions is much more reasonable, especially for HMC. Surprisingly, even though the Markov chain has only explored a small region of the posterior (i.e. parameters), it has explored a reasonably larger region of the posterior predictive distribution applied to the test dataset. MCMC methods tend to struggle with multi-modal distributions but fortunately, the modes of a BNN posterior tend to be connected by low-loss tunnels, as explored in section 3.6. This means that if we were to run the MCMC sampling for longer, we would expect the $\hat{R}$ to get much closer to 1. This is consistent with the results of Izmailov et al. [8].

Model	Parameter space	Prediction space
HMC	9.2	1.4
RWMH	30.9	2.6
NUTS	102.4	5.8

Table 1: Gelman-Rubin $\hat{R}$ computed for both parameters and predictions for each MCMC method. The values are computed using 5 chains and averaged across all parameters / predictions.

6.2 Comparison of predictive distributions

We compare the quality of the predictive distributions obtained from each model using three different metrics: point estimate accuracy, calibration, and likelihood (which combines the previous two metrics into a single value).

In order to assess the quality of point estimates, we can simply look at the mean squared error (MSE) of each model, as summarized in Table 2. As expected, a single NN trained using SGD has the worst predictions. A deep ensemble provides an improvement over SGD, although HMC wins by a large margin. Given an infinite computational budget, HMC, RWMH, and NUTS would all converge to the same equilibrium distribution. The only reason they provide different results is that they differ in their rate of convergence toward this distribution.

Model	Mean squared error	St. dev.
HMC	0.55	0.27
RWMH	3.12	1.27
ensemble	3.77	0.24
NUTS	4.99	1.47
SGD	5.20	1.43

Table 2: Mean squared error (MSE) of each model on test data. The standard deviation of the values is obtained by comparing 5 independent chains from each model.

Another important metric is calibration. Ideally, the uncertainty that a model predicts should match its true uncertainty about the estimated response variable. If a model consistently claims low uncertainty even though its residuals are large, it is *overconfident*. Conversely, if a model has small residuals but claims high uncertainty, it is *underconfident*. Ideally, the distribution of observed values matches the model's predictive distribution exactly. If this were true then the p-values of the response variable, measured using the model's predictive distribution, would be distributed uniformly between 0 and 1. In contrast, an overconfident model would have p-values disproportionally concentrated around 0 and 1. Fig. 20 shows these p-values for each model. As explained in section 4.1, a standard NN trained using SGD fails to incorporate model uncertainty. Using a variational model (e.g. deep ensemble) helps but only slightly. HMC, which offers the best approximation of the true BNN posterior, has an almost perfect calibration. RWMH and NUTS are not as good as HMC, but are still substantially better than SGD and deep ensembles.

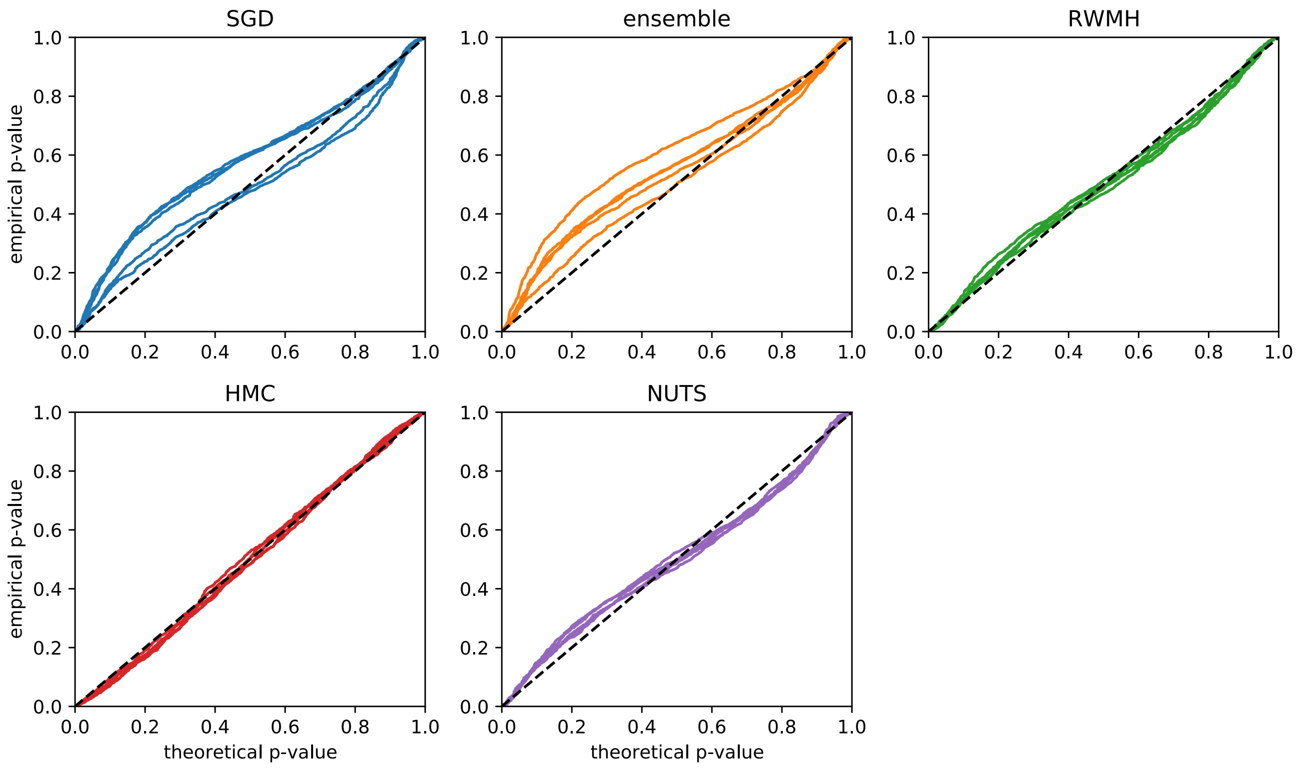


Figure 20: Calibration of each model. The p-value of the true response values is measured using the predictive distribution of each model. The empirical CDF of these values is plotted for each model. If the predictive distribution matches the true distribution, the p-values will be distributed uniformly, meaning they will lie on the dashed line.

Lastly, the likelihood of the test data is a *proper scoring rule*: it is maximized by reporting the true probability distribution. In order to obtain a high likelihood, the predictions must be calibrated and the residuals must be small. Table 6.2 summarizes the log-likelihood of the test data for each model.

Model	Log-likelihood	St. dev.
HMC	5857	83
RWMH	4216	85
NUTS	3539	69
ensemble	3456	23
SGD	3409	53

Table 3: Log-likelihood of each model on test data. The standard deviation of the values is obtained by comparing 5 independent chains from each model.

7 Conclusion

We compared a standard neural network to a Bayesian neural network approximated using several methods. We explored the theory behind each method and compared empirical results on a real-world regression dataset. We found the Bayesian neural network to outperform the standard neural network across all benchmarks. We obtained the most faithful approximation of the Bayesian neural network using Hamiltonian Monte Carlo (HMC). We also explored the No-U-Turn Sampler (NUTS), a popular modification of HMC. Despite the popularity of NUTS, we did not manage to achieve a single U-turn during Hamiltonian dynamics. Perhaps a larger dataset would need to be used, in order to obtain more stable gradients and thus allow for longer leapfrog trajectories – this would be an interesting topic for further research. Similarly, we did not experiment with the prior variance. However, as long as the dataset is large enough (hence the likelihood dominates the prior), the prior is not very important [8]. Bayesian neural networks are a promising alternative to standard neural networks, although more efficient methods are needed to reduce their computational complexity.

References

[1] Alexander D'Amour, Katherine Heller, Dan Moldovan, Ben Adlam, Babak Alipanahi, Alex Beutel, Christina Chen, Jonathan Deaton, Jacob Eisenstein, Matthew D Hoffman, et al. Underspecification presents challenges for credibility in modern machine learning. *arXiv preprint arXiv:2011.03395*, 2020.

[2] Stanislav Fort, Huiyi Hu, and Balaji Lakshminarayanan. Deep ensembles: A loss landscape perspective. *arXiv preprint arXiv:1912.02757*, 2019.

[3] Timur Garipov, Pavel Izmailov, Dmitrii Podoprikhin, Dmitry P Vetrov, and Andrew G Wilson. Loss surfaces, mode connectivity, and fast ensembling of dnns. *Advances in neural information processing systems*, 31, 2018.

[4] Samuel Stanton, Pavel Izmailov, Polina Kirichenko, Alexander A Alemi, and Andrew G Wilson. Does knowledge distillation really work? *Advances in Neural Information Processing Systems*, 34, 2021.

[5] Wesley J Maddox, Pavel Izmailov, Timur Garipov, Dmitry P Vetrov, and Andrew Gordon Wilson. A simple baseline for bayesian uncertainty in deep learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[6] Ivan Skorokhodov and Mikhail Burtsev. Loss landscape sightseeing with multi-point optimization. *arXiv preprint arXiv:1910.03867*, 2019.

[7] Roman Novak, Lechao Xiao, Jiri Hron, Jaehoon Lee, Alexander A Alemi, Jascha Sohl-Dickstein, and Samuel S Schoenholz. Neural tangents: Fast and easy infinite neural networks in python. *arXiv preprint arXiv:1912.02803*, 2019.

[8] Pavel Izmailov, Sharad Vikram, Matthew D Hoffman, and Andrew Gordon Gordon Wilson. What are bayesian neural network posteriors really like? In *International Conference on Machine Learning*, pages 4629–4640. PMLR, 2021.

[9] Andrew G Wilson and Pavel Izmailov. Bayesian deep learning and a probabilistic perspective of generalization. *Advances in neural information processing systems*, 33:4697–4708, 2020.

[10] Andrew Gelman, Walter R Gilks, and Gareth O Roberts. Weak convergence and optimal scaling of random walk metropolis algorithms. *The annals of applied probability*, 7(1):110–120, 1997.

[11] Oren Mangoubi, Natesh S Pillai, and Aaron Smith. Does hamiltonian monte carlo mix faster than a random walk on multimodal densities? *arXiv preprint arXiv:1808.03230*, 2018.

[12] Radford M Neal et al. Mcmc using hamiltonian dynamics. *Handbook of markov chain monte carlo*, 2(11):2, 2011.

[13] Matthew D Hoffman, Andrew Gelman, et al. The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *J. Mach. Learn. Res.*, 15(1):1593–1623, 2014.

[14] Radford M Neal. Slice sampling. *The annals of statistics*, 31(3):705–767, 2003.

[15] Du Phan, Neeraj Pradhan, and Martin Jankowiak. Composable effects for flexible and accelerated probabilistic programming in numpyro. *arXiv preprint arXiv:1912.11554*, 2019.

[16] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, pages 23–24, 2018.

[17] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

[18] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[19] Andrew Gelman and Donald B Rubin. Inference from iterative simulation using multiple sequences. *Statistical science*, 7(4):457–472, 1992.