Universidad Nacional Autónoma de México

Facultad de Ciencias

Bayesian Neural Ordinary Differential Equations for Epidemiological Forecasting

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

Físico

PRESENTA:

Aslan García Valadez

TUTOR

Dr. Isaac Pérez Castillo

Ciudad de México 2021

Contents

Chapter 1

Preliminaries

Machine learning has been an area of research with a huge boom in interest over the recent years. Many machine learning algorithms have been developed for a wide variety of applications with a spectacular performance on certain tasks, equalling that of a human and in some cases improving over human performance [9]. These algorithms have proven quite effective in many specific fields including image recognition [9], real time translation [2] and time series forecasting [10].

The main goal of machine learning is to use data to infer inherent patterns in it and use those patterns to shape the approach to treat unseen problems. In contrast to other modelling approaches, where intuition and previous knowledge are used to find an approach to the problem at hand, machine learning is focused on extracting the 'intelligence' from the data itself [3]. For instance, in image recognition the data is the image itself and from it, a label of the content in the image is produced by a "learned" rule from previously labelled images; in real time translation, a sentence or phrase in a given language is given as data and a translation to another language is produced. This approach leads to rather opaque models with little interpretative use but highly effective.

Machine learning has a wide variety of applications in different fields, however there are three main learning problems into which most use cases can be grouped into: *supervised learning*, *unsupervised learning* and *reinforcement learning*. In supervised learning we are given a set of n samples of data, these samples $x_\mu \in \mathbb{R}^m, \mu = 1,2,\dots,n$ are accompanied with a label $y_\mu \in \mathbb{R}^n$. The goal of supervised learning is to find a function f such that $f(x_\mu) = y_\mu, \mu = 1,2,\dots,n$ and when presented with a new sample x_{new} without its label, $f(x_{new})$ will approximate the label [3].

Unsupervised learning involves input data with no labels available, in stark contrast to supervised learning. The goal of unsupervised learning then is to recover the inherent structure of the data from the given dataset. Unsupervised learning is often used for *generative modelling*, where the objective is to find a probability distribution from which to generate data which is statistically similar to the given dataset. An arbitrary probability distribution in general can be modelled with unsupervised learning techniques [3].

Finally, reinforced learning does not directly involve a dataset, but rather, an artificial agent is placed in an environment. The actions taken by the agent are determined by obtaining information from the environment and calculating a possible reward, the actions change the environment with the intention of maximising reward. Learning occurs by interacting with the environment and the idea is to obtain reward-maximising strategies. Reinforced learning techniques have been successfully used to teach an artificial agent board games such as chess and Go [3].

In this work, we focus on a particular subset of machine learning techniques involving *neural networks*, these are commonly known as *Deep Learning*. We explore the possible application of

Deep Learning to forecasting of epidemics with a Bayesian approach. In this chapter we present a review of neural networks, in particular one of the earliest architectures designed: the *multilayer perceptron*. We focus on supervised learning with multilayer perceptrons from a frequentist perspective and expose several inherent shortcomings to it. As an alternative, we review Bayesian concepts and definitions and apply them to supervised learning. Finally, we discuss practical implementation of Bayesian methods to neural networks, with an emphasis on numerical approximation to the posterior via MCMC methods.

1.1 Basics of Neural Networks

Neural networks have been one of the most widely used tools for pattern recognition and machine learning in the modern era. The reasons for their wide usage are their remarkable performance with so little prior information, their tolerance to random noise within the data, and the high model flexibility it offers. While there are many different types of neural networks, we will discuss the most basic type of them: the multilayer perceptron, one of the earliest architectures developed. In this section we shall provide the basic definitions and elements of a multilayer perceptron and how it is usually applied to supervised learning problems.

1.1.1 Multilayer Perceptron as a Function Approximator

We start by defining a L-layered perceptron U_θ as a $\mathbb{R}^m \to \mathbb{R}^n$ function from a m-dimensional input space into a n-dimensional output space. The function U_θ depends upon its parameters $\theta = \{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \dots W^{(L)}, b^{(L)}\}$. For $l \in 1, \dots, L-1$, both $W^{(l)}$, the *weights*, and $b^{(l)}$ the *biases*, comprise the l-th *layer* of the neural network; this layer is characterised by its *width* r_l. As a convention we define $r_0 = m, r_L = n$; the width r_l determines the dimensions of the weights and biases: $W^{(l)}$ is a matrix of dimensions $r_l \times r_{l-1}$ and $b^{(l)}$ a vector of dimension r_l [3].

For the first layer we can take an input $x = (x_1, \dots, x_m)$ and define its *output* $a^{(1)}$, a vector of dimension r_1 given by

$$a^{(1)} = W^{(1)}x + b^{(1)}. \tag{1.1}$$

Given the output, we define the *activation* $z^{(1)}$, another r_1-dimensional vector, by applying element-wise an *activation function* $g^{(1)}$, a $\mathbb{R} \to \mathbb{R}$ mapping. The j-th component of the activation is given by

$$z_j^{(1)} = g^{(1)}(a_j^{(1)}). \tag{1.2}$$

We extend these definitions for every layer $1 < l \leq L$ by taking $x = z^{(0)}$ and the final n-dimensional output as $z^{(L)}$, so that we have a r_l-dimensional output vector $a^{(l)}$ given by

$$a^{(l)} = W^{(l)}z^{(l-1)} + b^{(l)}, \tag{1.3}$$

and a r_l-dimensional activation $z^{(l)}$ obtained by element-wise application of an activation function $g^{(l)}$, such that the j-th component of $z^{(l)}$ is

$$z_j^{(l)} = g^{(l)}(a_j^{(l)}). \tag{1.4}$$

Given an input x, the L-layered multilayer perceptron U_θ maps it by succesive forward application of the equations (1.3),(1.4).

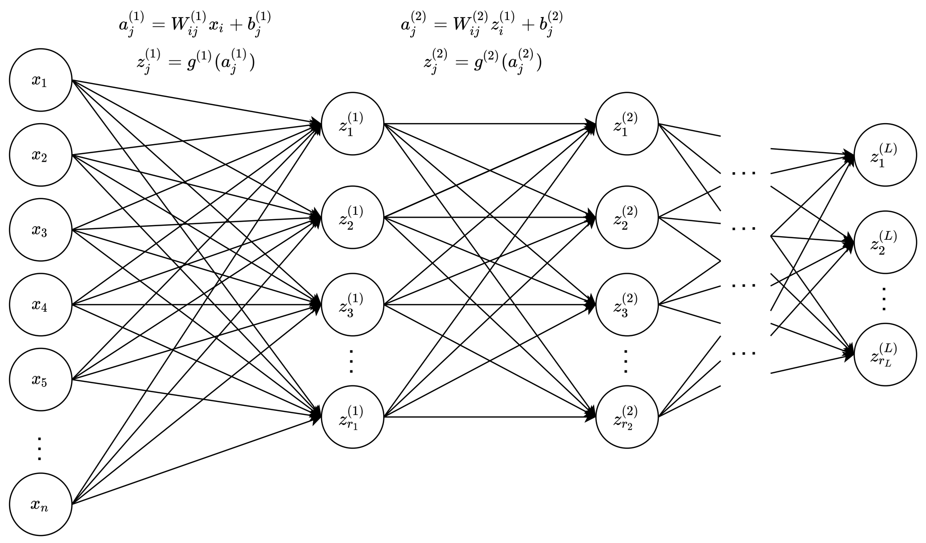

Figure 1.1: Graphical representation of a multilayer perceptron.

$$
\begin{aligned}
z^{(0)} &= x, && (1.5)\\
a^{(1)} &= W^{(1)} z^{(0)} + b^{(1)}, && (1.6)\\
z^{(1)} &= g^{(1)}(z^{(1)}), && (1.7)\\
&\vdots \\
a^{(l+1)} &= W^{(l+1)} z_{(l)} + b^{(l+1)}, && (1.8)\\
z^{(l+1)} &= g^{(l+1)}(z^{(l+1)}), && (1.9)\\
&\vdots \\
U_\theta(x) &= z^{(L)} = g^{(L)}(a^{(L)}). && (1.10)
\end{aligned}
$$

It is important to note that the activation functions $g^{(l)}$ must be non-linear functions in every layer, otherwise we reduce the perceptron into a simple linear transformation. Common activation functions are the ReLU (rectified linear unit), sigmoid σ and hyperbolic tangent tanh functions, presented here:

$$\mathrm{ReLU}(x) = \max(0, x); \qquad \sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}}; \qquad \tanh(x) = \frac{\mathrm{e}^{x} - \mathrm{e}^{-x}}{\mathrm{e}^{x} + \mathrm{e}^{-x}} \tag{1.11}$$

While the affine structure of every layer, and a clever choice of activation function are important from a computational point of view, the successive application of non-linear functions are the perceptron's greatest strength; this inherent nonlinearity is what gives the perceptron its very high model flexibility.

Indeed, there have been several results concerning the perceptron's ability to approximate any arbitrary function (although results have been proven for other types of architectures), these results are known as *Universal Approximation Theorems*. In [6] it is proven that a neural network with a single hidden layer is able to approximate any function for an arbitrary continuous, discriminatory, sigmoidal activation if no constraint is put on the number of neurons and size of the weights; [12] extends the result for non-continuous activations and relaxes the assumptions on the activation, only requiring the activation functions to be non-polynomical. It is to be noted that while these results provide a theoretical guarantee of approximation to any arbitrary function, there is no result for the number of nodes or layers required to provide an approximation. From a practical point of view, approximation might not be possible in some cases due to the high number of parameters, and therefore computation power, required.

1.1.2 The Supervised Learning Problem

Now that we have seen the potential of a neural network as an universal approximator, the immediate question is: How do we achieve this? We discussed briefly the three main machine learning problems, and for each one of them the approach is different. Here we focus on the supervised learning problem, where we wish to use a dataset and its labels to find a function that approximates the labels without the labels necessarily available.

We define a *training set* $\mathscr{D} = \{(x_\mu, y_\mu)\}_{\mu=1}^{n}$ comprised of the n samples, or *training examples*, $x_\mu \in \mathbb{R}^d$ and their respective labels $y_\mu \in \mathbb{R}^m$. The d components of each training sample are called the *features* of the data. To find our suitable function, we use a neural network U_θ with a fixed number of layers L, layer widths r_l and activation functions $g^{(l)}$ for $l = 1, 2, \ldots, L$. The neural network's output for a given input is completely determined by the parameters of each layer. Finding the mapping between features and labels is equivalent to finding the set of weights and biases for every layer such that for every sample in the training set x_μ, $U_\theta(x_\mu) \approx y_\mu$. The process to achieve this is commonly called *training* the neural network.

To train a neural network it is convenient to define a loss function $\mathscr{L}(\theta; \mathscr{D})$; a common way to construct such a loss function is to define a loss $\mathfrak{L}(\theta; x_\mu, y_\mu)$ for $x_\mu, y_\mu \in \mathscr{D}$ and define $\mathscr{L}$ as the mean of $\mathfrak{L}$ over the training set:

$$\mathscr{L}(\theta; \mathscr{D}) = \frac{1}{n} \sum_{\mu=1}^{n} \mathfrak{L}(\theta; x_\mu, y_\mu). \tag{1.12}$$

The loss function is a means of quantifying the *training error*, that is the error the neural network has with respect to the training labels.

By way of example, consider the case where the labels of the training set are a continuous, real variable. For every pair $x_\mu, y_\mu \in \mathscr{D}$ we define $\mathfrak{L}(\theta, x_\mu, y_\mu)$ as the *square error*

$$\mathfrak{L}(\theta, x_\mu, y_\mu) = \|U_\theta(x_\mu) - y_\mu\|^2. \tag{1.13}$$

Then the loss function is the *mean square error*

$$\mathscr{L}(\theta; \mathscr{D}) = \frac{1}{n} \sum_{\mu=1}^{n} \|U_\theta(x_\mu) - y_\mu\|^2. \tag{1.14}$$

The parameters θ of the neural network are adjusted to minimise the loss function, and we shall say that the optimal parameters $\theta^\star$ are those which minimise $\mathscr{L}(\theta; \mathscr{D})$, that is

$$\theta^\star \equiv \underset{\theta}{\operatorname{argmin}} \mathscr{L}(\theta; \mathscr{D}). \tag{1.15}$$

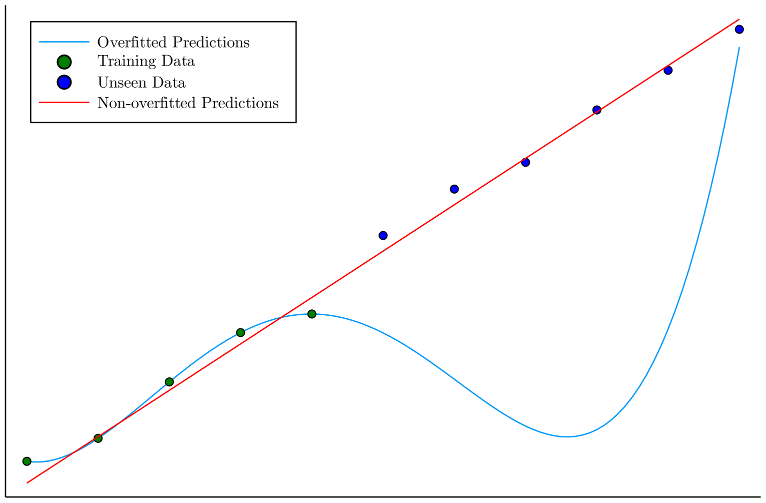

Figure 1.2: A case of overfit for a simple supervised learning problem with one feature and one dimensional label. Notice the blue curve has no training error, yet its predictions on unseen data are rather poor; the red curve has a slightly higher training error but much better predictions.

Thus, our supervised learning problem is effectively reduced into a mathematical optimisation one. Once we have solved this optimisation problem with the optimisation algorithm of our preference, we then expect the neural network not only to approximate with relative precision the labels in the training set, but to predict, or generalise, to inputs and labels previously not seen during training.

While $\theta^\star$ ensures a low error on the labels of the training set by virtue of being a minimiser of the loss function, there is no a priori guarantee it will have a low error on new examples which were not seen during training. This error, defined as the *generalisation error*, is most important in applications and measures the performance of the neural network. The standard way of estimating the generalisation error is picking a subset of the training set (at random) to evaluate the performance of the trained neural network on unseen samples, the rest of the dataset is used as a training set [3].

An important problem that often arises in supervised learning is the problem of *overfitting*. Overfitting refers to the fact that low training error doesn't guarantee good generalisation error. A common cause of overfitting is having a small training dataset, this will cause random nuances in the data to be encoded into the neural network. Even if a large amount of data is available, a neural network with too many parameters will also capture the random nuance in the data resulting in overfit [1].

Besides monitoring the generalisation error as described above, regularisation is a common technique to prevent overfit. As mentioned, a large number of parameters may cause overfitting; it is natural then to constrain the neural network to use fewer non-zero parameters. We define a *penalty* using the p-norm over the parameters θ as

$$\lambda \sum_{\theta} \|\theta\|^p \equiv \lambda \sum_{l=1}^{L} \left(\|W^{(l)}\|^p + \|b^{(l)}\|^p \right), \tag{1.16}$$

where the p-norm of a $n \times m$ matrix W is defined as

$$\|W\|^p = \left(\sum_{i=1}^{n} \sum_{j=1}^{m} |w_{ij}|^p \right)^{1/p}. \tag{1.17}$$

The parameter λ is a positive real number, denoted as the *regularisation parameter*, and it controls the strength of the penalty, usually λ is fine-tuned by hand. This penalty is added to the chosen cost function and a new cost function is obtained:

$$\mathscr{L}(\theta) = \frac{1}{n}\sum_{\mu=1}^{n} \mathfrak{L}(\theta; x_\mu, y_\mu) + \lambda \sum_{\theta} \|\theta\|^p . \tag{1.18}$$

Other techniques to prevent overfit include early stopping and trading width by depth. In *early stopping* the optimisation algorithm used to find $\theta^\star$ is stopped before obtaining a minimum. Both the training error and the generalisation error are monitored and while the training error will always decrease, generalisation error will increase when overfit starts to occur; at this point the optimisation is terminated. Trading width by depth is a less direct method to avoid generalisation. Networks with more layers, greater depth, often require less units on each layer, i.e. they require less depth on each layer. Increasing the depth acts as regularisation since the features on later layers are constrained by the structures of the earlier layers. On the other hand, too deep networks may take too long to converge to a minimum, so this must be taken into account [1].

1.2 Basics of Bayesian Inference

In the above section we discussed the basics of neural networks and gave a passing sketch of how they are commonly calibrated to match outputs from a training dataset. On this section, we focus in the shortcomings of that approach, emphasising the need of quantifying the trustworthiness and the uncertainty of the output of a neural network, specially when used to predict an output coming from inputs not used for training the neural network. After doing so, we provide the basic concepts and definitions of Bayesian inference, a probabilistic framework which provides a conceptually clear way of quantifying uncertainty in our predictions.

1.2.1 Understanding Uncertainty

As we mentioned earlier, one of the advantages of neural networks is their high modelling flexibility, the high number of parameters and non-linear activation functions make it possible to approximate any function with few previous assumptions or constraints. However the low interpretability of the results obtained from a neural network and the risk of overfitting are two important problems that arise in supervised learning. Another important aspect is quantifying the amount of information about the process we can obtain from the data, in other words, estimating the generalisation power of our neural network. Usually, whether a neural network makes accurate predictions or not is estimated by using data available but not used for training and measure the agreement between the data and the neural network's output. This approach gives no estimate of the probability of the output matching the correct one and therefore we have no way of knowing how robust our prediction actually is [4].

These aforementioned problems make it difficult to have trustworthy predictions from a neural network, and therefore their use for decision making, especially in areas of high risk, is limited. As we see, robust modelling using neural networks must then avoid the inherent overconfidence of the predictions coming from the training process. Any useful estimate provided by a neural network should not then be a simple output but rather the uncertainty of it must be completely characterised.

We can classify the uncertainty inherent to any kind of data-based predictions as either *aleatoric* or *epistemic*. Aleatoric uncertainty comes from the stochastic nature of the phenomenon we intend to model and the error made during the collection of observations, this kind of uncertainty is inherent to the data and is irreducible. On the other hand, epistemic uncertainty is that uncertainty due to lack of more knowledge; in a certain way, this kind of uncertainty encodes what we don't know about the process due to "missing pieces" in our knowledge [4].

1.2.2 Maximum Likelihood Estimation

To start with, let's formalise the intuition of the neural network as a parametric model approximating the true function governing the model. We treat the features and labels of the data x, y as random variables; we define a conditional probability $p(y|x)$ from which the training set $\mathscr{D}$ is a sample.

Let $p(y|x;\theta)$ be a parametric family of probabilistic distributions; given x we use $p(y|x;\theta)$ to map it to a number estimating the probability $p(y|x)$, this estimate depends on the parameters θ used. This definition of parametric family of distributions is a general one, although we will focus in the particular case in which a neural network U_θ is involved, in this case θ refers to the parameters of the neural network. The frequentist approach involves assuming there exists a true set of parameters for the probability distribution. This set of parameters is fixed but unknown.

The training set is used to estimate this true set of parameters. To do so, we start assuming every sample $(x_\mu, y_\mu) \in \mathscr{D}$ is independent and identically distributed. Then, we define the *likelihood* as

$$p(\mathscr{D}|\theta) \equiv \prod_{\mu=1}^{n} p(y_\mu|x_\mu;\theta), \tag{1.19}$$

we expect $p(y_\mu|x_\mu;\theta)$ to approximate $p(y_\mu|x_\mu)$ for every sample (x_μ, y_μ). To do so we define the *maximum likelihood estimator (MLE)* $\theta^\star$ as

$$\theta^\star_{MLE} \equiv \operatorname*{argmax}_\theta p(\mathscr{D}|\theta). \tag{1.20}$$

The MLE is often found by taking the negative logarithm of the likelihood, since it is more numerically stable; to express the likelihood as an expectation with respect to the training set, it is also often scaled by $\frac{1}{n}$, n being the number of samples in the training set. Since the logarithm preserves inequalities, the MLE $\theta^\star$ can be equivalently defined as[8]

$$\theta^\star_{MLE} = \operatorname*{argmin}_\theta \left[-\frac{1}{n} \sum_{\mu=1}^{n} \log p(y_\mu|x_\mu;\theta) \right]. \tag{1.21}$$

Notice that the definitions (1.21) and (1.15) coincide if $\log p(y_\mu|x_\mu;\theta) = \mathfrak{L}(\theta; x_\mu, y_\mu)$. In most cases this is true, and the loss function $\mathscr{L}(\theta;\mathscr{D})$ is revealed as the (negative) log likelihood of the probabilistic model we assign to our supervised learning problem.

As a practical example, consider the case where the labels y correspond to a real continuous variable. Let us further assume a multivariate Gaussian probability model such that

$$p(y|x) = \mathscr{N}(y; U_\theta(x), \sigma^2). \tag{1.22}$$

Here, $U_\theta(x)$ is predicting the mean of the Gaussian; we further assume that the covariance matrix is the identity matrix multiplied by a fixed, pre-chosen, real constant σ. The negative log likelihood of the model for the training set is then

$$\sum_{\mu=1}^{n} \log p(y_\mu|x_\mu;\theta) = m\log\sigma + \frac{m}{2}\log 2\pi + \sum_{\mu=1}^{n} \frac{\|U_\theta(x_\mu) - y_\mu\|^2}{2\sigma^2}, \tag{1.23}$$

removing all constants we obtain the MLE for this model as

$$\theta^{\star}_{MLE} = \underset{\theta}{\operatorname{argmin}} \sum_{\mu=1}^{n} \frac{\|U_\theta(x_\mu) - y_\mu\|^2}{2\sigma^2}. \tag{1.24}$$

The MLE obtained is equivalent to the minimum of the loss function in (1.14) since the loss function and the negative log likelihood only differ by a constant and thus have the same maxima and minima.

1.2.3 Bayesian Inference

The Bayesian approach involves not only treating the dataset $\mathscr{D}$ as a sample of $p(y|x)$. We propose a parametric family of distributions given by a fixed neural network with parameters θ, the parameters θ are themselves considered to be random variables. We define the *prior* of the parameters $p(\theta)$ as the distribution over θ; this distribution is chosen before taking into account the training set and its purpose is to encode previous assumptions about the model. By virtue of Bayes' theorem, we define the *posterior distribution* $p(\theta|\mathscr{D})$ as:

$$p(\theta|\mathscr{D}) = \frac{p(\mathscr{D}|\theta)p(\theta)}{p(\mathscr{D})}. \tag{1.25}$$

We can start to see the appeal of Bayesian inference: conceptually, given the data $\mathscr{D}$ and model depending on the parameters θ the posterior gives us a probability distribution for the parameters explaining the data. The prior $p(\theta)$ represents our knowledge of the event we are modelling without having seen any data, the likelihood encodes the knowledge we extract from the available data. The likelihood "updates" our prior knowledge and the result is the posterior.

An attractive aspect of Bayesian inference is the introduction of priors. The priors are useful for encoding beforehand knowledge into the problem; adding more knowledge into the problem helps us make more accurate estimations, improve generalisation power beyond the dataset and in the case of punctual estimation, faster convergence into a more reliable minimum [4]. Punctual estimation of the posterior is similar to maximum likelihood estimation, the difference lies in the optimisation objective: we seek to find a set of parameters which maximise the posterior, instead of the likelihood. The resulting set of parameters is defined as the *maximum a posteriori* estimator:

$$\theta^{\star}_{MAP} = \underset{\theta}{\operatorname{argmax}}\, p(\theta|\mathscr{D}). \tag{1.26}$$

To illustrate the utility of priors for the parameters of a neural network, let us consider again a probabilistic model with a Gaussian distribution $\mathscr{N}(y; U_\theta(x), \sigma)$ as discussed on the previous section. We use as prior a Gaussian distribution with mean 0 and as covariance matrix proportional to the identity matrix by a constant σ', over the parameters, assuming they are independent and identically distributed, that is

$$p(\theta) \propto \exp\left[-\frac{1}{2(\sigma')^2}\|\theta\|^2\right] \equiv \prod_{l=1}^{L} \exp\left[-\frac{1}{2(\sigma')^2}\left(\|W^{(l)}\|^2 + \|b^{(l)}\|^2\right)\right]. \tag{1.27}$$

Taking the negative logarithm of the posterior defined by the likelihood and the prior, and removing constants (particularly, the $p(\mathscr{D})$ term, since it doesn't have any dependence on θ), the MAP estimate is

$$\theta^{\star}_{MAP} = \underset{\theta}{\operatorname{argmin}} \frac{1}{n} \sum_{\mu=1}^{n} \|U_\theta(x_\mu) - y_\mu\|^2 + \lambda \sum_{\theta} \|\theta\|^2. \tag{1.28}$$

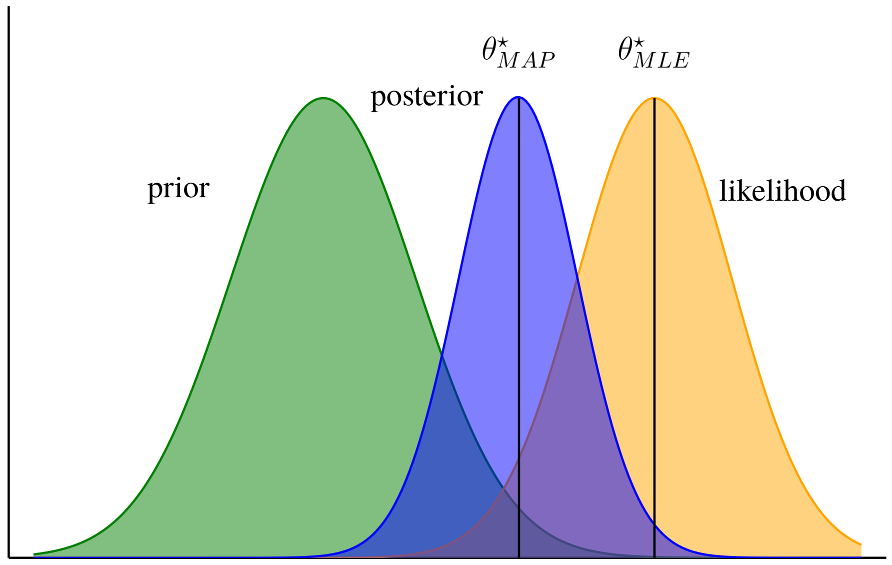

Figure 1.3: An example of a Bayesian posterior. Notice how point estimation (either MAP or MLE) reduces a distribution to a single point.

Notice that the negative log probability is in this case equivalent to (1.18) when $p = 2$; an analogous argument can be made for an arbitrary p, showing that regularisation is nothing more than adding priors and optimising with respect to the posterior. The Bayesian approach provides then a straightforward approach to design more complicated, yet interpretable, regularisation terms [8].

While point estimates form the basis of supervised learning from a probabilistic perspective, they are an attempt to summarise a distribution into a single point. As such, they throw away much of the information about the uncertainty of our predictions since a prediction with a point estimate assumes we have obtained the "true" set of parameters of the model and there is no possible uncertainty on the parameters. The main reason for using Bayesian inference in this work is the estimation of the epistemic uncertainty in the predictions of our model, particularly, we wish to obtain an uncertainty for the predictions y on unseen data x, given the knowledge we obtained from the training set; then, it is important to take into account the whole posterior over θ instead of its likeliest value. Through the posterior, we can do so by defining the *predictive posterior distribution*:

$$p(y|x,\mathscr{D}) = \int p(y|x,\theta)p(\theta\big|\mathscr{D})\;\mathrm{d}\theta. \tag{1.29}$$

1.2.4 Tempered posteriors

In recent times, the application of Bayesian methods to Deep Learning has resulted in the observation of improved model performance with the introduction of *tempered posteriors* [20]. Having chosen a likelihood function $p(\mathscr{D}\big|\theta)$ and a prior $p(\theta)$, we define the tempered posterior $p_T(\theta\big|\mathscr{D})$ by means of a "temperature" parameter T:

$$p_T(\theta\big|\mathscr{D}) = p(\mathscr{D}\big|\theta)^{1/T}p(\theta.) \tag{1.30}$$

The temperature parameter is tuned to obtain an acceptable predictive performance and it is empirically observed in most that $T < 1$ yields the best predictive, leading to "cold" posteriors [20].

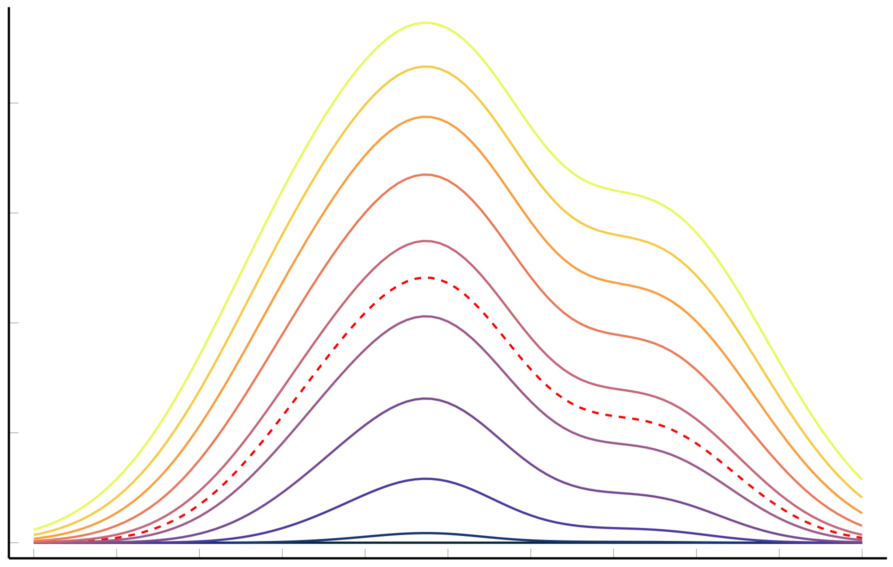

Figure 1.4: A visual representation of posterior tempering for different temperatures T; we include both $T > 1$ and $T < 1$. Notice that for $T < 1$, the distribution sharpens around its mode and other regions of interest of the posterior may be smoothed over.

The introduction of a cold temperature in the likelihood part of the posterior is equivalent to overcounting the data, and thus it smooths over several local modes of the posterior which might not provide either a good fit to the training data or a good generalisation to unseen data. In this work, we shall explore the effects of the temperature parameter for predictions and compare them to an untempered posterior.

1.3 Practical Implementation of Bayesian Inference

While the Bayesian framework is very appealing because of its conceptual simplicity and the offered ability to estimate uncertainty, it is only recently that it has started to take off in usage; the main reason for this is the high computational cost associated. The equation (1.25) is deceptively simple, since so far we haven't discussed the $p(\mathscr{D})$ term. We define $p(\mathscr{D})$, the *evidence*, as the posterior integrated over the whole set of possible parameters $\boldsymbol{\theta}$

$$p(\mathscr{D}) = \int p(\mathscr{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})\ \mathrm{d}\boldsymbol{\theta}. \tag{1.31}$$

This integral is a multidimensional integral and in general it is not analytically tractable, partly due to a likelihood which is not tractable analytically but mainly because of the high number of dimensions involved. Precisely this last reason makes the integral hard approximate numerically, since most numerical integration schemes lose precision on high dimensions. Thus, obtaining an analytical predictive posterior is only feasible for a small number of specific cases with few dimensions.

This difficulty can be circumvented by remembering the definition of the predictive posterior in (1.29). Since it is obtained by marginalising over the posterior, we may very well think of it as the expected value of a certain quantity. Then this integral can be approximated by taking samples from the posterior and calculating with them $p(y|x,\boldsymbol{\theta})$, finally averaging over all the samples:

$$\int p(y|x,\theta)p(\theta\big|\mathscr{D})\ \mathrm{d}\theta \approx \sum_{\theta\sim p(\theta\big|\mathscr{D})} p(y|x,\theta). \tag{1.32}$$

Thus, we have reduced the problem of calculating a possibly incalculable integral into obtaining samples from a certain distribution. While this is not by a any means a trivial problem, especially with the distributions that will appear later, it certainly is more tractable, with several algorithms at our disposition.

As discussed, this intractability of the posterior makes sampling methods one of the most useful and widely available methods for approximation. In particular, the most widely used sampling methods are *Markov Chain Monte Carlo* methods. These algorithms require the generation of a Markov Chain (a stochastic process where the next step of the process depends only on the current state of the chain) whose equilibrium distribution is the posterior.

1.3.1 Random Walk Metropolis-Hastings

The Metropolis-Hastings algorithm is one of the classical algorithms for posterior approximation. While computationally prohibitive for very large dimensional models, we shall review it here to establish some basic concepts common to the rest of the algorithms. The algorithm takes an unnormalised probability distribution $\pi(\theta)$ - in our specific case $\pi(\theta) = p(\mathscr{D}\big|\theta)p(\theta)$ -, a proposal distribution $g(\theta'\big|\theta)$, and number of iterations. This proposal distribution must be such that the *detailed balance* condition is fulfilled [17]:

$$\pi(\theta)g(\theta'\big|\theta) = \pi(\theta')g(\theta\big|\theta'). \tag{1.33}$$

the Markov chain itself must be *ergodic*, which means it must be *aperiodic* and *recurrent*. The Markov chain is essentially walking randomly along the parameter space, the acceptance step is critical to steer the chain into regions of high probability. It is fairly common to use a Gaussian distribution $\mathscr{N}(\theta^t,\varepsilon)$ as the proposal distribution. The variance ε is a specially important parameter to tune since a very small variance will not let the random walker explore the whole parameter space (without a large number of iterations) while a too large variance will not let the random walker be attracted into high probability regions easily [17].

```
Given $\theta^0, \pi(\theta), M$
for $t = 1$ to $M$ do
    $\theta' \sim g(\theta'\big|\theta^{t-1})$
    $A(\theta', \theta^{t-1}) = \min(1, \frac{\pi(\theta')}{\pi(\theta^{t-1})})$
    $u \sim \mathrm{Unif}(0,1)$
    if $u \leq A(\theta', \theta^{t-1})$ then
        $\theta^t = \theta'$
    else
        $\theta^t = \theta^{t-1}$
    end
end
```

Algorithm 1: Metropolis-Hastings algorithm [17].

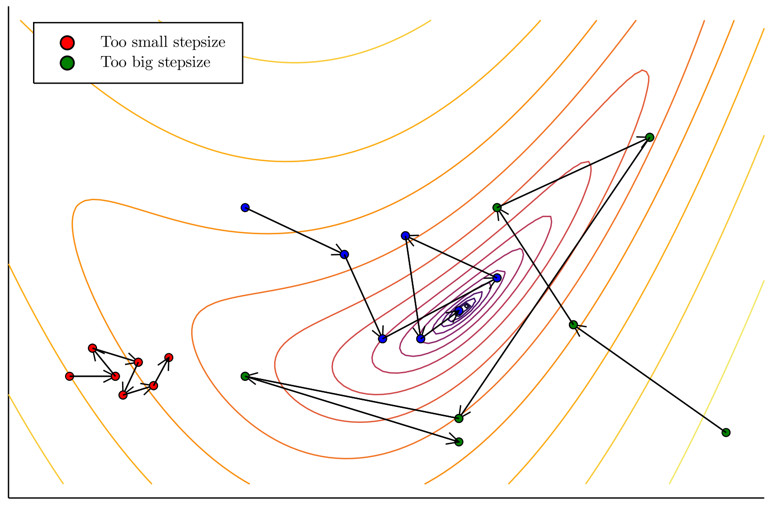

Figure 1.5: Three different realisations of Random Walk Metropolis-Hastings with different variance for the proposal function, notice the importance of choosing an adequate variance. Too small a variance will not sample efficiently since it doesn't traverse the whole posterior in a short number of iterations; a large variance will cause most of the proposals to be rejected and the chain will get stuck in a single part of the posterior. The contour plot represents the (unnormalised) density distribution we wish to sample from.

1.3.2 No U-Turn Sampling

This sampling algorithm is intended to be an improvement over Metropolis-Hastings by attempting to remove the random walking element. One of the deficiencies of random walk Metropolis-Hastings is the fact that it traverses the parameter space randomly. For high dimensional models, this random walk might take a very prohibitive time to converge. For continuous variables, as it is in this case, we can define an auxiliary set of momentum variables and implement Hamiltonian Monte-Carlo [11].

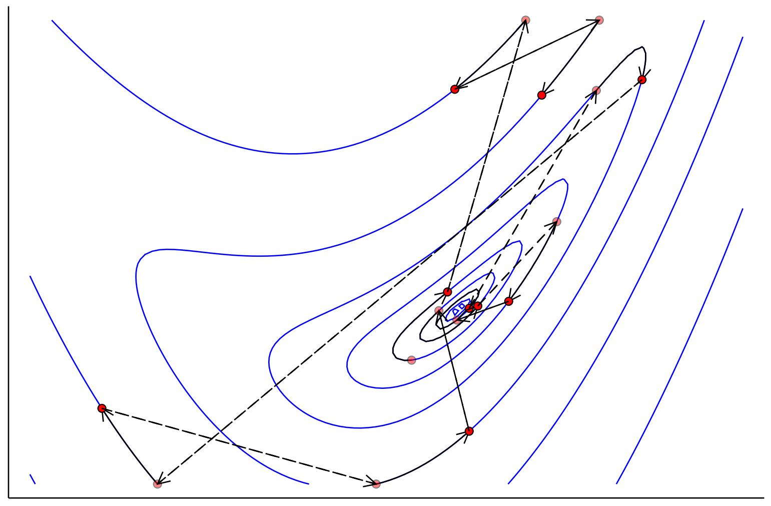

Figure 1.6: A visual representation of Hamiltonian Monte Carlo. Random sampling of momenta changes the level curve of the particle, which is then evolved in time according to Hamilton's equations with a Leapfrog numerical scheme. This allows for a greater exploration of the posterior.

For every variable θ_d, we define a momentum r_d. These momenta are usually taken to be independently drawn from a normal distribution. This results on the following joint distribution $p(\vec{\theta}, \vec{\mathbf{r}})$:

$$p(\vec{\theta}, \vec{\mathbf{r}}) = \exp(\mathscr{L}(\vec{\theta}) - \frac{1}{2}\vec{\mathbf{r}} \cdot \vec{\mathbf{r}}) \tag{1.34}$$

$\mathscr{L}$ is the logarithm of the joint density of the variables $\vec{\theta}$. This has a very natural interpretation as a Hamilton system where θ is a particle's position and $\vec{\mathbf{r}}$ is the momentum vector in a d-dimensional space. We can then simulate the time evolution with an economic numerical method: the Leapfrog algorithm which induces the updates:

$$r^{t+\varepsilon/2} = r^t + \frac{\varepsilon}{2}\nabla\mathscr{L}(\theta) \qquad \theta^{t+\varepsilon} = \theta^t + \varepsilon r^{t+\varepsilon/2} \qquad r^{t+\varepsilon} = r^{t+\varepsilon/2} + \frac{\varepsilon}{2}\nabla\mathscr{L}(\theta^{t+\varepsilon}) \tag{1.35}$$

We can evolve this for L steps using the stepsize ε. However, since this is an approximation, we incur in numerical error. Thus we can employ an acceptance-rejection step as in Metropolis to ensure that we are moving into lower energy (higher probability) regions.

Since the Leapfrog method functions as the proposal function of Metropolis-Hastings, the parameters L and ε need to be tuned in a manner similar to the variance for the proposal distribution. Usually ε is fixed, and the number of steps L is the one to be adjusted. The NUTS algorithm is an alternative to this, by using an adaptive L found by means of a recursive algorithm [11].

1.3.3 Stochastic Gradient Langevin Dynamics

This algorithm is specially suited for large scale datasets and intended to be computationally cheaper than the usual MCMC algorithms. In this algorithm, we implement Langevin dynamics. The parameter updates consist of gradient steps with an additional injection of Gaussian noise to avoid a collapse into a minimum. The resulting updates for a dataset with N data items x_i are

$$\theta^{t+1} = \theta^t - \Delta\theta_t, \tag{1.36}$$

$$\Delta\theta_t = \frac{\varepsilon_t}{2}\left(\nabla \log p(\theta^t) + \sum_{i=1}^{N} \log p(x_i \middle| \theta^t) + \eta_t\right), \tag{1.37}$$

$$\eta \sim \mathcal{N}(0,\varepsilon). \tag{1.38}$$

To ensure convergence to a local maximum, a major requirement is for the stepsizes ε_t to satisfy:

$$\sum_{t=1}^{\infty} \varepsilon_t = \infty, \qquad \sum_{t=1}^{\infty} \varepsilon_t^2 < \infty. \tag{1.39}$$

A usual choice for the stepsize is $\varepsilon_t = a(b+t)^{-\gamma}$ with $\gamma \in (0.5, 1]$ [19]. Note that this can be seen as using a gradient-based proposal function and while usually an acceptance step is required, it can be proven [19] that the decreasing schedule for ε_t makes it unnecessary.

1.3.4 Parallel Tempering

All of the methods mentioned above have a key fault. They will not sample the posterior accurately if the posterior has multiple modes since the chain is not likely to move between them. To overcome this, we use *parallel tempering*, also known as Metropolis Coupled Markov Chain Monte Carlo (or MC^3). This method is based on a simple statistical physics intuition: the system's energy distribution collapses to its most likely energy (the ground state) as the temperature decreases. Conversely, heating the system will make the distribution wider and the energy valleys will be less sharp.

```
Given $[\theta_1^{(0)}, \dots, \theta_n^{(0)}], \pi(\theta), [T_1, \dots, T_n], M$
for $t = 1$ to $M$ do
    for $i = 1$ to $n$ do
        $\theta_i' \sim g(\theta_i' | \theta_i^{t-1})$
        $A(\theta_i', \theta_i^{t-1}) = \min\left(1, \frac{\pi(\theta_i')^{T_i}}{\pi(\theta_i^{t-1})^{T_i}}\right)$
        $u \sim \text{Unif}(0,1)$
        if $u \leq A(\theta_i', \theta_i^{t-1})$ then
            $\theta_i^t = \theta_i'$
        else
            $\theta_i^t = \theta_i^{t-1}$
        end
    end
    $j = \text{Random}(1, \dots, n)$
    $P(i, i-1) = \min\left(1, \frac{\pi(\theta_{i-1}^t)^{T_i}\pi(\theta_i^t)^{T_{i-1}}}{\pi(\theta_i^t)^{T_i}\pi(\theta_{i-1}^t)^{T_{i-1}}}\right)$
    $u \sim \text{Unif}(0,1)$
    if $u \leq P(i, i-1)$ then
        $\theta_i^t = \theta_{i-1}^t$
        $\theta_{i-1}^t = \theta_i^t$
end
```

Algorithm 2: Parallel tempering algorithm [13].

The algorithm consists of running n Markov chains through the parameter space in parallel with an unnormalised posterior $\pi(\theta) = p(\mathscr{D} | \theta)p(\theta)$. We define a set of temperatures $T_1, T_2, \dots, T_n$ so that we have the following auxiliary distributions $\pi_i(\theta) = \pi(\theta)^{T_i}$ (occasionally, only the likelihood part $p(\mathscr{D}|\theta)$ is scaled, the prior is left untouched). In every iteration of the algorithm, a candidate for every chain is proposed and is accepted or rejected according to the usual Metropolis step. After the acceptance or rejection of the candidate for all chains a new Metropolis step happens, this time between chains. A common way to do it is to use adjacent chains, since acceptance will not be very likely for $T_i << T_j$, although occasionally random chains are used as well [13]. Notice that this algorithm can be implemented with any of the algorithms reviewed in this section, since parallel tempering does not specify the proposal function to be used for the generation of candidates in each chain.

1.3.5 Adaptive Parallel Tempering

Parallel tempering offers a way to sample a multimodal distribution, a case in which a traditional Metropolis-Hastings algorithm might get stuck in a single mode for too long and lead to a false convergence. However, it is non-trivial to determine the temperature ladder to be used for each particular case. Temperatures are often chosen empirically, guided by several trial runs and some general guidelines [14].

In an adaptive algorithm, the temperatures are not set *a priori*, but rather, in every iteration of the algorithm the temperatures themselves are updated along with the parameters to be sampled. We seek to find a ladder of temperatures in which the mean transition probability (the probability of switching parameters across chains) is uniform. To do so, we define our sequence of inverse temperatures $\{\beta_n\}_{n\geq 0} = \{\beta_n^{(1:N)}\}_{n\geq 0}$ to be parametrised by $\{\rho_n\}_{n\geq 0} = \{\rho_n^{(1:N)}\}_{n\geq 0}$ so that

$$\beta_n^{(1)} = 1, \tag{1.40}$$

$$\beta_n^{(m)} = \beta^{(m)}(\rho_n^{(1:m-1)}), \qquad m \in \{2, \dots, N\}, \tag{1.41}$$

$$\beta^{(m+1)}(\rho^{(1:m)}) = \beta^{(m)}(\rho^{(1:m-1)}) \exp\left(-\exp(\rho^{(m)})\right). \tag{1.42}$$

The adaptation of the temperatures occurs by updating $\{\rho_n\}_{n\geq 0}$ as follows:

$$\rho_n^{(l)} = \Pi_\rho\left(\rho_{n-1}^{(l)} + \gamma_{n,1} H^{(l)}\left(\rho_{n-1}^{1:l}, \theta_n\right)\right). \tag{1.43}$$

Π_ρ is a projection into a constrain set, and

$$H^{(l)}\left(\rho^{(1:l)}, \theta\right) = \min\left[1, \left(\frac{\pi(\theta^{l+1})}{\pi(\theta^{(l)})}\right)^{\Delta\beta^{(l)}(\rho^{(1:l)})}\right] - \alpha^\star, \tag{1.44}$$

$$\Delta\beta^{(l)}(\rho^{(1:l)}) = \beta^{(l)}(\rho^{(1:l-1)}) - \beta^{(l+1)}(\rho^{(1:l)}). \tag{1.45}$$

$\alpha^\star$ is the target acceptance rate between tempered chains. The algorithm is designed for the inverse temperatures to converge into a value which yields the desired target acceptance rate $\alpha^\star$, without any prior tuning of the temperatures [14].

Chapter 2

Neural Ordinary Differential Equations

On the previous chapter, we examined one of the most simple neural network models. Now, we shall combine that and ordinary differential equations to produce *Neural Ordinary Differential Equations*. First of all, we give a brief over view of ordinary differential equations; then we go on to explain exactly how do they fit into a deep learning framework; we also give a passing sketch of their diverse applications, with an emphasis on their modelling potential for time series.

2.1 Ordinary Differential Equations

Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ be a n-dimensional vector where each $x_i \in \mathbf{x}$ is a real-valued function; in particular we shall develop the case where

$$x_i : J \to \mathbb{R}, \;\; i = 1, 2, \ldots n, \tag{2.1}$$

$J = [t_0, t]$ is a continuous interval of $\mathbb{R}$. We define an k-th order *ordinary differential equation* (ODE) for $\mathbf{x}$ as

$$\mathbf{x}^{(k)} = \mathscr{F}\left(t, \mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(k-1)}\right). \tag{2.2}$$

Here, $\mathscr{F}$ is a mapping $J \times \mathbb{R}^n \to \mathbb{R}^n$ and $\mathbf{x}^j$ stands for the vector of j-th derivatives of the components of $\mathbf{x}$, that is

$$\mathbf{x}^{(j)} = \left(\frac{\mathrm{d}^j x_1}{\mathrm{d}t^j}, \frac{\mathrm{d}^j x_2}{\mathrm{d}t^j}, \ldots \frac{\mathrm{d}^j x_n}{\mathrm{d}t^j}\right). \tag{2.3}$$

A k-order differential equation can always be transformed into a system of k 1-st order differential equations by considering the vector $\mathbf{y} = (\mathbf{x}, \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(i)}, \ldots, \mathbf{x}^{(k-1)})$, then we obtain the system

$$\begin{aligned} y_1^{(1)} &= y_2 \\ y_2^{(1)} &= y_3 \\ &\vdots \\ y_k^{(1)} &= \mathscr{F}(t, \mathbf{y}) \end{aligned}$$

Furthermore, we can eliminate the explicit dependence on t of $\mathscr{F}$ by adding a new variable $y_1 = t$, then $\mathbf{y} = (y_1, \mathbf{x}, \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(i)}, \ldots, \mathbf{x}^{(k-1)})$ and

$$\begin{aligned} y_1^{(1)} &= 1 \\ y_2^{(2)} &= y_3 \\ &\vdots \\ y_{k+1}^{(1)} &= \mathscr{F}(\mathbf{y}) \end{aligned}$$

In light of this, we will from now on only discuss systems of 1-st order ordinary differential equations with no explicit time dependence.

We define an *initial value problem* for a system of ODEs by specifying the *initial conditions* $(t_0, y_1^0, \ldots, y_n^0) \in J \times \mathbb{R}^n$. A *solution* ϕ for the initial value problem is a vector of functions $\phi_1, \ldots, \phi_n : J \to \mathbb{R}^n$ that satisfy the system of ODEs and furthermore:

$$\begin{aligned} \phi_1(t_0) &= y_1^0 \\ \phi_2(t_0) &= y_2^0 \\ &\vdots \\ \phi_n(t_0) &= y_n^0 \end{aligned}$$

From the above given definitions, we see that a system of ODEs is a mapping of an n-dimensional space into itself. This n-dimensional space, which in our particular case is $\mathbb{R}^n$, is defined as the *phase space*. A solution ϕ of an initial value problem can be interpreted as follows: given an initial point $\mathbf{y}(t_0) = (y_1^0, \ldots, y_n^0)$ on the phase space with an initial time t_0, an ODE system $\mathbf{y}^{(1)} = \mathscr{F}(\mathbf{y})$ is a temporal evolution rule outputting a *trajectory* ϕ which moves in time through the phase space, representing the temporal evolution of $\mathbf{y}(t_0)$.

2.2 Combining ODEs and Neural Networks

The connection between neural networks and differential equations is made when we use a neural network U_θ as the mapping $\mathscr{F}$ of an ODE system, resulting in the *neural ordinary differential equation*

$$\mathbf{y}^{(1)} = U_\theta(\mathbf{y}). \tag{2.4}$$

In this situation U_θ gives the time evolution of its input $\mathbf{y}(t_0)$ through phase space, resulting in an output $\mathbf{y}(t)$. From a computational point of view, we use a black-box numerical library to use already-implemented numerical methods for approximation of differential equations; the specific library used throughout this work is `DiffEqFlux`, a Julia library specially designed for neural ordinary differential equations. [15]

Neural ODE can be integrated in a rather natural way into a deep learning framework. A clear example comes from *residual networks*, commonly used for image classification[9]. A residual network is constructed from a neural network U_θ mapping the input space $\mathbb{R}^n$ by mapping an input x into

$$x \to x + U_\theta(x). \tag{2.5}$$

We can define a sequence of outputs $x_t, t \in \{0, 1, \ldots, T-1\}$ given by

$$x_{t+1} = x_t + U_\theta(x_t) \tag{2.6}$$

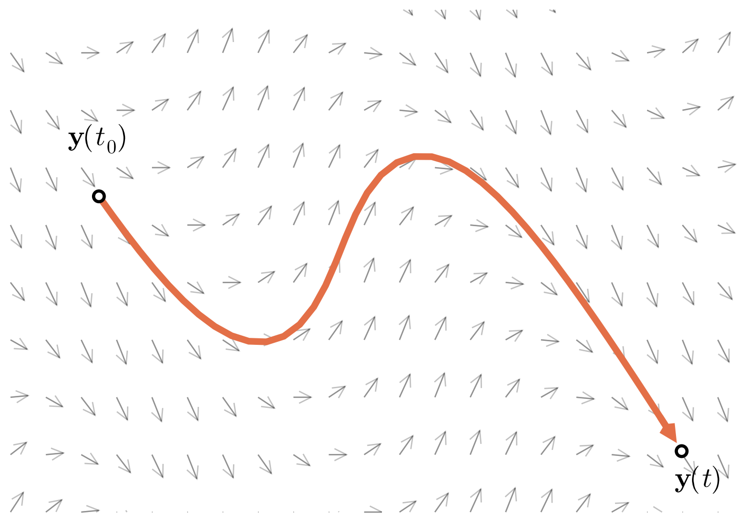

Figure 2.1: The trajectory through phase space from $\mathbf{y}(t_0)$ given by $\mathscr{F}(\mathbf{y})$, which is represented on the plot as the vector field over the phase space. Note that $\mathscr{F}(\mathbf{y})$ is always tangent to the trajectory and represents its velocity on phase space.

The final output x_T is composition of these equations. This is a discretisation of the equation (2.4), obtained from a first order Taylor expansion of U_θ around the point x_t, so a residual layer can be interpreted as a discrete NODE[5] or conversely, a NODE is a continuous residual neural network.

Although, as discussed, NODE layers have wide potential for different applications, in this work we shall focus on learning systems of differential equations from data. This is a supervised learning problem where our training set consists of a set of observation times $\{t_i\}$'s and observations $\{\mathbf{z}(t_i)\}$ of the system we wish to model. The idea is then to obtain a neural network U_θ capturing the hidden dynamics that govern the system. The neural network is expected not only to interpolate accurately between the observations in the training set, we wish to extrapolate from the training data to generate predictions from our differential equations model. Our fundamental assumption is that the system from which we are sampling the training data is generated by a differential equation so that

$$\frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}t} = \mathscr{F}(\mathbf{z}(t)) \tag{2.7}$$

then, we use a multilayer perceptron U_θ to approximate this function $\mathscr{F}$. As mentioned on the previous chapter, we would like to obtain an estimate of the epistemic uncertainty associated to the data and, through the posterior predictive distribution, give a probabilistic estimate of our predictions. As discussed on the first chapter, this is equivalent to inducing a probabilistic Bayesian model once we set the corresponding likelihood and priors for θ.

The assumption of a system being governed by a system of ordinary differential equations are widely used for modelling because they are a fairly natural way of expressing the continuous time dynamics of a certain set of variables z_i's with respect to time t. As we see, ODEs provide us with a natural language to discuss the time evolution of a system; then, in principle, the only remaining step is to find an adequate F which accurately models our system. However, using neural networks to learn differential equations is in stark contrast to the usual modelling approach with ODEs; usually the system of differential equations is derived not from data but from heuristic arguments or previous knowledge. This latter approach usually results in fairly interpretable models with little flexibility, neural networks are in the opposite end: the flexibility of a neural network provided by the universal approximation theorems is very high, although the high number of parameters and the non-linearities involved make for a model with little interpretability.

Chapter 3

Some Simple Examples

In this chapter we wish to demonstrate the Bayesian framework that has been already discussed. In particular, it is of interest to determine and discuss the performance of the sampling algorithms discussed at the end of the past chapter. To do so, we generate artificial data from two deterministic ODE systems and from a small part of the whole time series, attempt to forecast the rest of it along with a probabilistic distribution for the model's forecast.

3.1 Case Study 1: Reconstruction of a Neural ODE

In this case study, we define a $\mathbb{R}^2 \to \mathbb{R}^2$ mapping by using a neural network with one hidden layer of fifty neurons. The activation function for the hidden layer is a sigmoid and the activation function for the output layer is the identity. We initialise at random a fixed set of weights and biases to generate data for two variables x_1, x_2 for $t \in [0, 25]$ by numerically integrating with the initial condition $x_1(0) = 1, x_2(0) = 0$. We split this data in two and use the first half ($t \in [0, 12.4]$) to benchmark the four sampling algorithms previously discussed.

As likelihood we use a Gaussian distribution with constant variance centered on the model's predictions. We use a uniform prior over the parameters. This results on a measure $\pi(\theta|\mathscr{D})$ proportional to the posterior, given by

$$\pi(\theta|\mathscr{D}) = \prod_{t=0}^{12.4} \exp\left[-\frac{1}{2}\left(\frac{\text{NODE}_\theta(x_t) - x_t}{\sigma}\right)^2\right] \tag{3.1}$$

3.1.1 Random Walk Metropolis-Hastings

Metropolis-Hastings was implemented by using a random sample from a multivariate normal distribution $\mathcal{N}(0, 0.1I)$. The chain was run for 2000 steps with a variance $\varepsilon = 1.5$ for the proposal function (Gaussian proposal). Several trial chains were run varying both the variance ε and the number of steps for the chain. It was observed that the acceptance rate was higher than 90% for all ε proposals used (ε from 10^{-4} to 10.5).

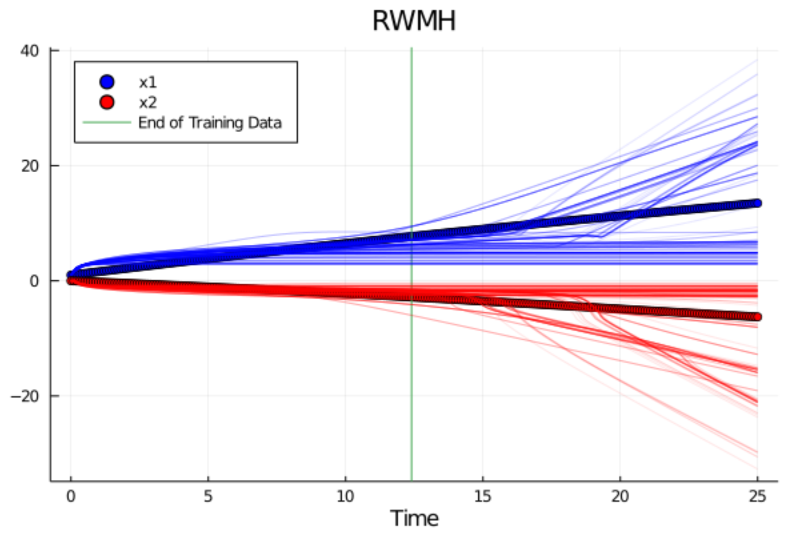

Figure 3.1: 250 samples from RWMH

3.1.2 No U-Turn Sampling

The algorithm was initialised with a starting point generated from a multivariate normal $\mathcal{N}(0, I)$. The identity matrix was used as mass matrix. We allowed for 300 samples, using 100 to allow the algorithm to warm-up, aiming for an acceptance rate of 0.65.

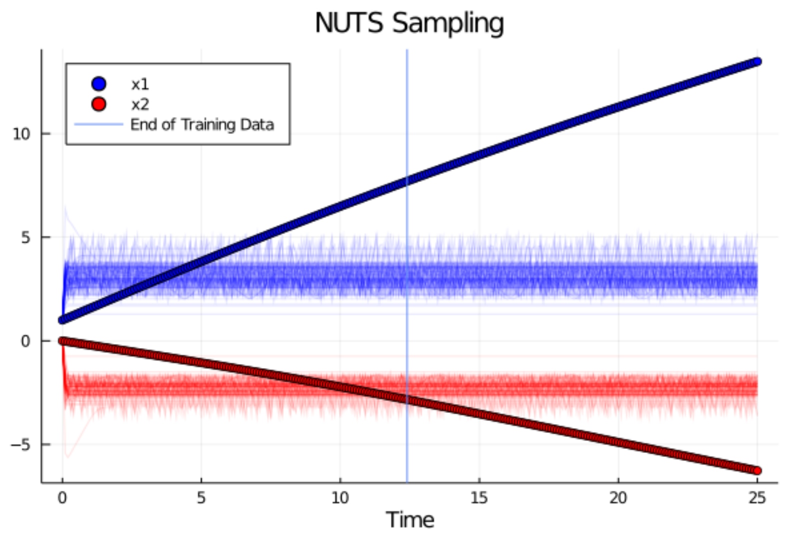

Figure 3.2: 200 samples from NUTS sampling

3.1.3 SGLD

The algorithm was again initialised with a variable from a Gaussian distribution. SGLD was run for 40000 iterations with parameters $a = 0.035, b = 0.25, \gamma = 0.65$; these parameters were chosen after tuning using several trial runs. The last 500 samples from the algorithm were used to approximate the posterior.

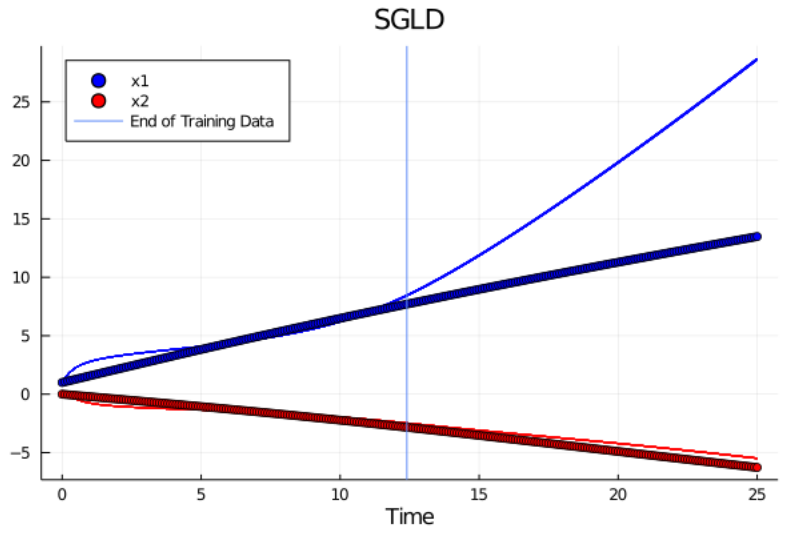

Figure 3.3: 500 samples from SGLD

3.1.4 Parallel Tempering

In this implementation of parallel tempering, we use a ladder of eight decreasing temperatures in a geometric sequence $\{T_i = 0.1^{i-1} | i = 1, \ldots, 8\}$. Each of the eight chains were initialised with a $\mathcal{N}(0, 0.5I)$ sample. The algorithm was run for 2500 steps with a variance $\varepsilon = 0.1$ for the proposal function. We use the last 500 samples from the warmest chain to approximate the posterior.

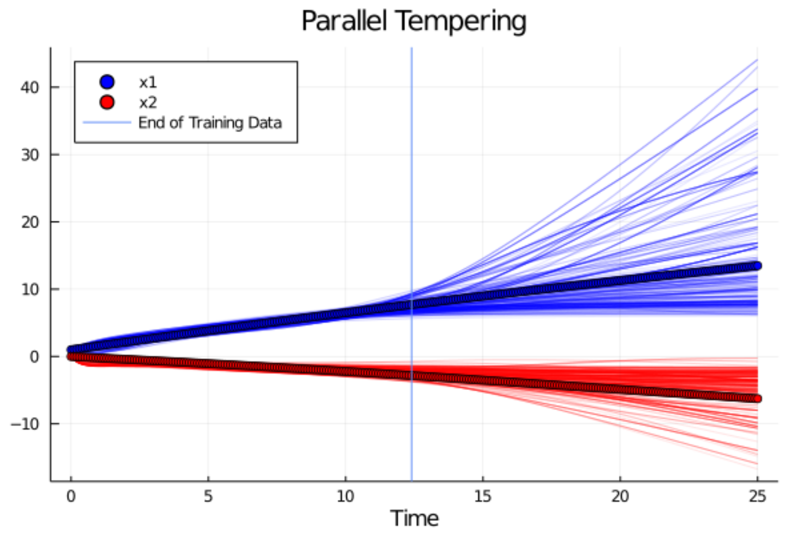

Figure 3.4: 500 samples from parallel tempering

As we can appreciate from the plots, the vanilla Metropolis-Hastings algorithm and NUTS result in worse estimates. Both SGLD and parallel tempering provide a better estimate, reaching into the posterior's regions of high probability. However, as appreciated on (3.3), the sampling process gets stuck into a single local minimum and doesn't explore the rest of the posterior. Parallel tempering clearly avoids this, traversing several minima, as we can see on (3.4). It is to be remarked, that unlike the rest of the sampling algorithms, NUTS wasn't fine tuned; allowing a greater number of warm-up samples or changing the target acceptance rate might improve its performance. However, in principle, the algorithm is self adapting, so the improvement might not be as large as required. The computational time for NUTS is also considerably higher than the other three methods, taking about eight hours to complete, compared to minutes for the rest of the algorithms.

3.2 Case Study 2: Lotka-Volterra Equations

In this section we apply the aforementioned sampling methods to the Lotka-Volterra system of equations. The Lotka-Volterra equations are two nonlinear differential equations, originally proposed to model interactions between a prey and a predator. The two variables x_1, x_2 evolve in time according to:

$$\frac{\mathrm{d}x_1}{\mathrm{d}t} = \alpha x_1 - \beta x_1 x_2 \tag{3.2}$$
$$\frac{\mathrm{d}x_2}{\mathrm{d}t} = \delta x_1 x_2 - \gamma x_2 \tag{3.3}$$

with $\alpha, \beta, \gamma, \delta$ real, positive parameters for the model.

We generate data with the initial condition $x_1(0) = 1, x_2(0) = 1$ and parameters $\alpha = 1.5, \beta = 1, \gamma = 3, \delta = 1$ for the time span $t \in [0, 14]$. The architecture proposed for our neural network is a simple two layered multilayered perceptron with fifty neurons on its hidden layer. The activation for the hidden layer is a sigmoid and the identity function for the output layer. As posterior, we proposed again an uniform prior on the parameters of the neural network and a Gaussian likelihood as used in the first case study.

3.2.1 Random Walk Metropolis-Hastings

The algorithm was implemented using a Gaussian distribution with variance $\varepsilon = 0.5$ as proposal function. The chain was run for 2000 steps and while several trial runs were used to determine optimal number steps and variance for proposal function, the acceptance rate consistently remained around 90%. The last 500 steps of the chain were used to simulate the posterior.

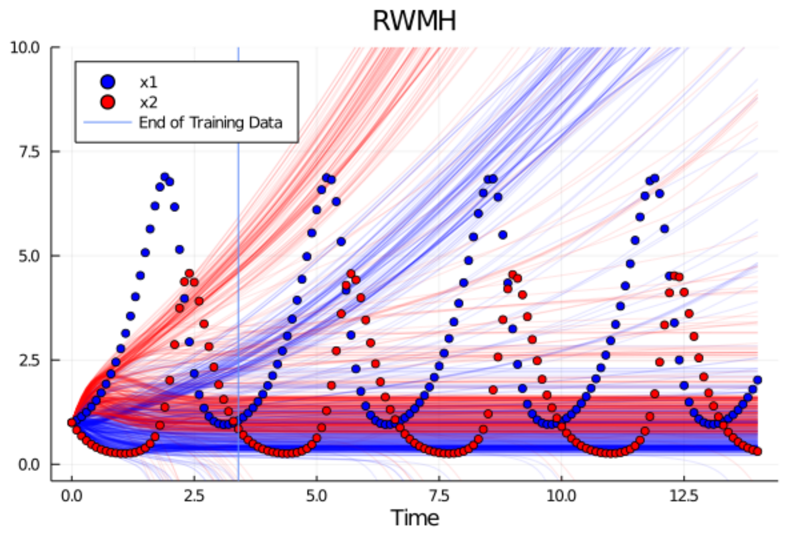

Figure 3.5: 500 samples from Random Walk Metropolis-Hastings.

3.2.2 No U-Turn Sampling

The algorithm was randomly initialised, with the identity matrix again as mass matrix. The algorithm was run for 500 steps, allowing the first 100 for warming up of the chain. The algorithm adapts the stepsize and mass matrix with a target acceptance rate of 0.65.

3.2.3 SGLD

As before, the algorithm was randomly initialised and run for 40,000 steps. As parameters, $a = 0.5, b = 0.05, \gamma = 0.5$ were chosen after several trial runs. The last 500 steps of the algorithm were used as samples.

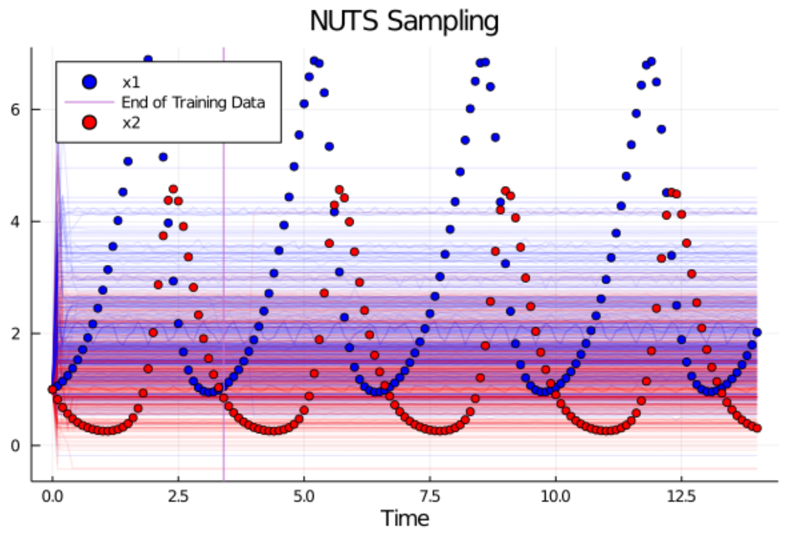

Figure 3.6: 400 samples from NUTS algorithm.

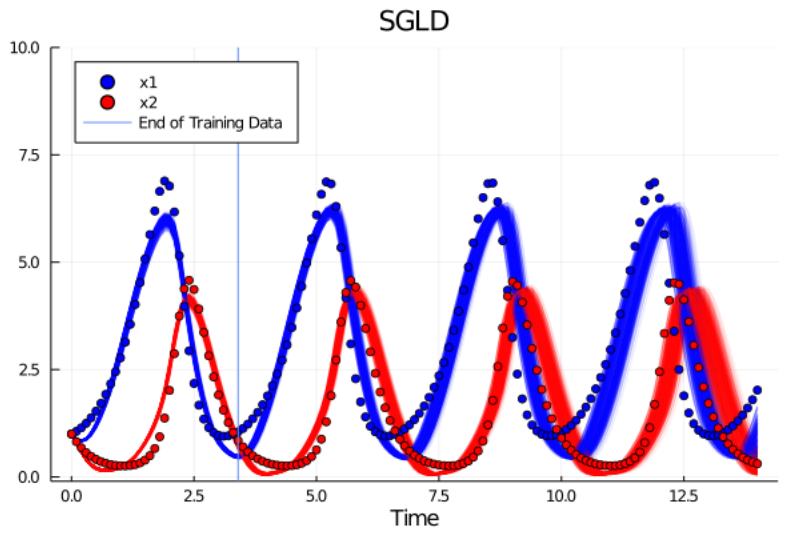

Figure 3.7: 500 samples from SGLD.

3.2.4 Parallel Tempering

After several trial runs fine tuning the number of steps, the temperature ladder and the variance for the proposal function in regular parallel tempering, we ran Adaptive Parallel Tempering with 10 temperature levels using an adaptive proposal function. The algorithm was run for for $350,000$ steps. The target acceptance rate between the temperature chain was 0.24. The posterior itself was tempered with an initial temperature of $T = 1 \times 10^{-4}$. The last 500 samples from the warmest chain were used to simulate from the posterior.

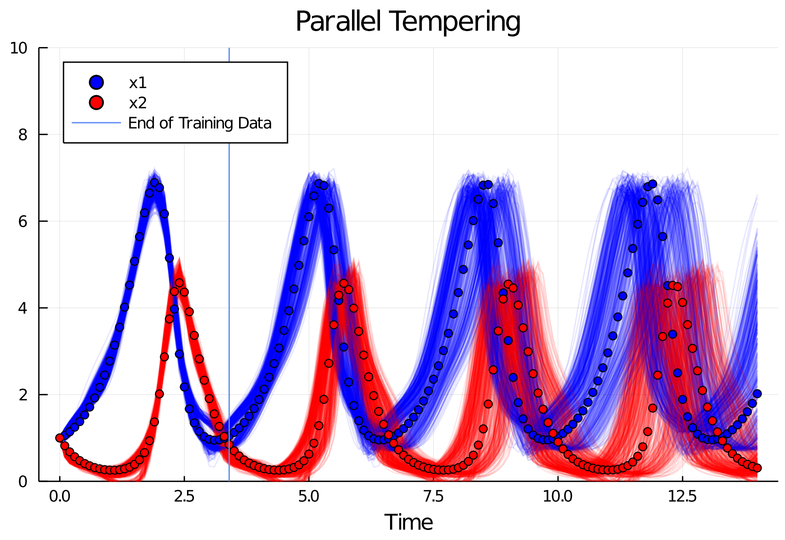

Figure 3.8: 500 samples from parallel tempering.

As shown in the previous figures, we see a similar trend for the sampling methods as in case study 1. Metropolis-Hastings and NUTS don't seem to converge into the high probability regions of the posterior. In that respect, both SGLD and parallel tempering seem to do better. Interestingly enough, the neural network captures the periodicity inherent to the system in a mostly accurate manner with a little amount of data. The initial temperature used in both parallel tempering cases seems to indicate that posterior tempering provides not only better fits to the training data but also better generalisations. Parallel tempering however, allows the exploration of multiple modes of the posterior instead of getting stuck in the MAP region.

Again, it is to be noted that SGLD converges to a single mode of the posterior, while parallel tempering, both in the adapting and non-adapting case, explores more of the posterior. The low temperatures required in both cases to reach the high probability regions seem to imply a posterior with several local minima, which would explain the poor performance of Metropolis-Hastings and NUTS, since it would be likely for them to get stuck on different local modes, impeding them from traversing the whole posterior.

This hypothesis seems reinforced by the fact that in both cases NUTS took a significantly higher execution time (about eight hours for NODE reconstruction and about four days for Lotka-Volterra). The high execution time is due to the recursive nature of the algorithm; it runs several trees independently to determine the best stepsizes to avoid get stuck on local minima. Several local minima would then increase the computation time. This also might be indicative of the fact that using more iterations of the algorithm might allow better performance, although computationally more expensive than parallel tempering.

Finally, it is worth noting that a change of prior might improve the performance of the algorithms, since it might help to smooth out some of the modes of the posterior. For both parallel tempering and random walk Metropolis-Hastings, special consideration must be taken on the number of parameters for the neural network. A high number of parameters might make the random walk about the parameter space rather inefficient and therefore the chains might take too long to converge to the posterior. This problem is often referred as the *curse of dimensionality*.[4]

Chapter 4

Epidemiological Forecasting

Now that we have reviewed the necessary Bayesian framework and we have discussed through two simple examples the pros and cons between different numerical approximations to the posterior, we are ready to focus on the main application of this thesis: epidemiological forecasting. As we have previously discussed, a Bayesian framework is more robust than a simple point estimation, considering that point estimation discards all possible forecasts as given by the data and it doesn't provide quantification of the quality of the estimation. Having a probability distribution for our epidemiological forecasts, the Bayesian predictive posterior, makes for a more trustworthy prediction system for decision making involving huge health and economic risks such as those posed by the implementation of restrictive measures.

In this chapter, we briefly discuss the workhorse for epidemiological modelling: compartmental models. We focus particularly on the SEIR model, and using generated data from this model, we apply our Bayesian NODE techniques for forecasting. Finally, we discuss possible applications and implementation details for real data.

4.1 Compartmental Epidemiological Models

We begin by considering a population that can be divided according to a certain set of criteria, in this case stages of disease, into n homogeneous compartments resulting on the set of variables $x = \{x_1, x_2, \dots, x_n\}$. These variables are the ones that define our epidemiological compartmental model. From those compartments, determine by epidemiological interpretation the subset of infectious compartments of x. For ease of discussion, let the first m compartments be the infectious ones; using this convention, we define the set $\mathbf{X}_s = \{x \geq 0 \big| x_i = 0, i = 1, \dots, m\}$ as the set of all disease free-states (no infected population). Defining $\mathscr{F}_i(x)$ as the rate of appearance of infected individuals on compartment i and $\mathscr{V}_i^+(x), \mathscr{V}_i^-(x)$ as the rate of transfer in and out of compartment i respectively by all other means, the compartments evolve from a non-negative set of initial conditions according to the system of equations

$$\frac{\mathrm{d}x_i}{\mathrm{d}t} = f_i(x) = \mathscr{F}_i(x) - \mathscr{V}_i(x) \tag{4.1}$$

where $\mathscr{V}_i(x) = \mathscr{V}_i^- - \mathscr{V}_i^+$.[7]

Note that both $\mathscr{F}_i$ and $\mathscr{V}_i$ denote a directed transfer between compartments, they must be non-negative. From this fact, the following set of properties follow:

$$\text{if } x \geq 0, \text{ then } \mathscr{F}_i, \mathscr{V}_i^+, \mathscr{V}_i^- \geq 0 \text{ for all } i = 1, \ldots, n \tag{4.2}$$

$$\text{if } x_i = 0, \text{ then } \mathscr{V}_i^- = 0, \text{ in particular, if } x \in \mathbf{X}_s \text{ then } \mathscr{V}_i^- = 0 \text{ for } i = 1, \ldots, m \tag{4.3}$$

$$\mathscr{F}_i = 0 \text{ for } i = 1, \ldots, m \tag{4.4}$$

Finally, to ensure a model consistent with observation, we must assume that a free of disease population will always remain so. This gives the property:

$$\text{if } x \in \mathbf{X}_s, \text{ then } \mathscr{F}_i(x) = 0, \text{ and } \mathscr{V}_i^+(x) = 0 \text{ for } i = 1, \ldots, m \tag{4.5}$$

From all the first two conditions, a highly important condition arises. If both conditions are fulfilled, then $x_i = 0$ automatically implies that $f_i(x) \geq 0$ and hence, the non-negative cone ($x_i \geq 0, i = 1, \ldots, n$) is a *forward-invariant* set of the system of equations. Forward invariance guarantees that the solutions to the compartmental model with non-negative initial conditions will always remain non- negative. [7]

4.2 NODEs for Compartmental Models

Our next objective then is to incorporate neural networks into this compartmental model framework. We do this by splitting the population into compartments according to epidemiological interpretation and/or availability of data and sample the fraction of population in each compartment for certain moments in time. Then, we use a neural network to predict the evolution of those compartment from the given data. It is somewhat inexact to say that we are replacing the compartmental model with a neural network since we are implicitly choosing a compartmental model when splitting the population. Rather, we are replacing the *dynamics* of the model, given by (4.1), with the neural network.

It is very important to remark on the conditions (4.2), (4.3), (4.4) given on the previous section. These conditions are rather strong yet absolutely necessary from an interpretability perspective. Forward invariance, guaranteed by the referenced conditions, assures us that whenever our initial conditions are "physical", our solutions will always be "physical" since negative compartments are senseless from a modelling point of view. Then, special care must be taken when designing the neural network architecture to be used for the approximation of the model's dynamics; we attempt to address this and show the capability of NODEs for the following case study on one of the simplest compartmental models.

4.3 Case Study: SEIR Model

The SEIR model assumes a constant population N divided into four compartments: susceptibles S, exposed E, infected I and recovered R, we normalise the population so that the compartments actually represent the *fraction* of the total population in each stage of the disease. The flow of the populations through the four stages is $S \to E \to I \to R$ in the following way: the population in S is not immune to the virus, after coming into contact with the infected populations it is infected with a probability β; the newly infected do not immediately turn infectious, rather, the disease is latent for those in the E compartment for a certain period of time; after a period of time, the infected population becomes infectious to others, flowing into the I compartment; finally, the infectious population eventually recovers after a certain time and flows into the R compartment. Thus, we can write the differential equations system for the four variables as

$$\frac{\mathrm{d}S}{\mathrm{d}t} = -\beta SI \tag{4.6}$$

$$\frac{\mathrm{d}E}{\mathrm{d}t} = \beta SI - \frac{1}{\tau_E}E \tag{4.7}$$

$$\frac{\mathrm{d}I}{\mathrm{d}t} = \frac{1}{\tau_E}E - \frac{1}{\tau_i}I \tag{4.8}$$

$$\frac{\mathrm{d}R}{\mathrm{d}t} = \frac{1}{\tau_i}I \tag{4.9}$$

the parameter β corresponds to the infection force of the disease, τ_E and τ_I correspond to the average time a person spends on the exposed E or infected I stage, respectively.

The system we want to model has two constraints that must be preserved by the predictions of the neural ODE: the aforementioned forward invariance and a constant population N. This last condition implies the sum of the derivatives for each compartment must be zero since

$$\frac{\mathrm{d}N}{\mathrm{d}t} = \frac{\mathrm{d}S}{\mathrm{d}t} + \frac{\mathrm{d}E}{\mathrm{d}t} + \frac{\mathrm{d}I}{\mathrm{d}t} + \frac{\mathrm{d}R}{\mathrm{d}t} = 0 \tag{4.10}$$

then, every solution for each compartment must be within the interval $[0,1]$. The second constraint is relatively simple to encode into the neural network: instead of outputting a 4-element vector corresponding to the derivative of the system, the neural network return a 3-element vector and the fourth element is determined by the fact that the four entries must sum to zero. The second constraint however is not as simple to encode directly, so the approach used was to make use of the fact that the solution must be between 0 and 1. Since the sigmoid mapping $\sigma(x) = \frac{1}{1+e^{-x}}$ is injective on the $(0,1)$ interval, we can apply the inverse transformation of the sigmoid $x = -\log(\frac{1}{\sigma} - 1)$ to the training data and solve the differential equation in that inverse space. The solution is taken to the original solution space by applying the sigmoid mapping. Note that since the inverse of the sigmoid doesn't preserve the zero-sum derivative condition, we can't implement both constraints at the same time.

For this case study we generate data from the SEIR model with normalised variables using the initial conditions $S(0) = 0.9, E(0) = 0.02, I(0) = 0.07, R(0) = 0.01$ using the parameters $\beta = 0.25, \tau_E = 50, \tau_I = 45$. The system of equations was numerically solved and then Gaussian noise with variance $\sigma = 1 \times 10^{-5}$ was added to the data. For our Bayesian model a neural network with two hidden layers was used; the first layer consisted of 12 neurons, the second layer used 5 neurons and finally the output layer with four output neurons. In both hidden layers, a σ activation was used, the output layer has an identity activation. We apply a Gaussian $\mathscr{N}(0,1)$ prior on the parameters θ of the neural network such that the posterior is written as

$$p(\theta|\mathscr{D}) \propto e^{-\mathscr{L}(\theta)}, \qquad \mathscr{L}(\theta) = \sum_{t_0}^{T} [x_t - \hat{x}_t(\theta)]^2 + \sum_{\theta} \theta^2, \tag{4.11}$$

with $\hat{x}_t(\theta)$ as usual, standing for the solution given by the NODE at time t with initial condition x_{t_0} and parameters θ.

To sample the posterior, adaptive parallel tempering was used with ten different temperature levels. We set an initial temperature of $T = 1 \times 10^{-4}$ for the posterior. The last $2,000$ samples from the warmest chain were used. In order to showcase the reduction of epistemic uncertainty when additional data is fed into the model, this process was repeated three times; starting with the first 100 days as training data, we increased the time window in steps of 50 days until reaching 200 days for training usage.

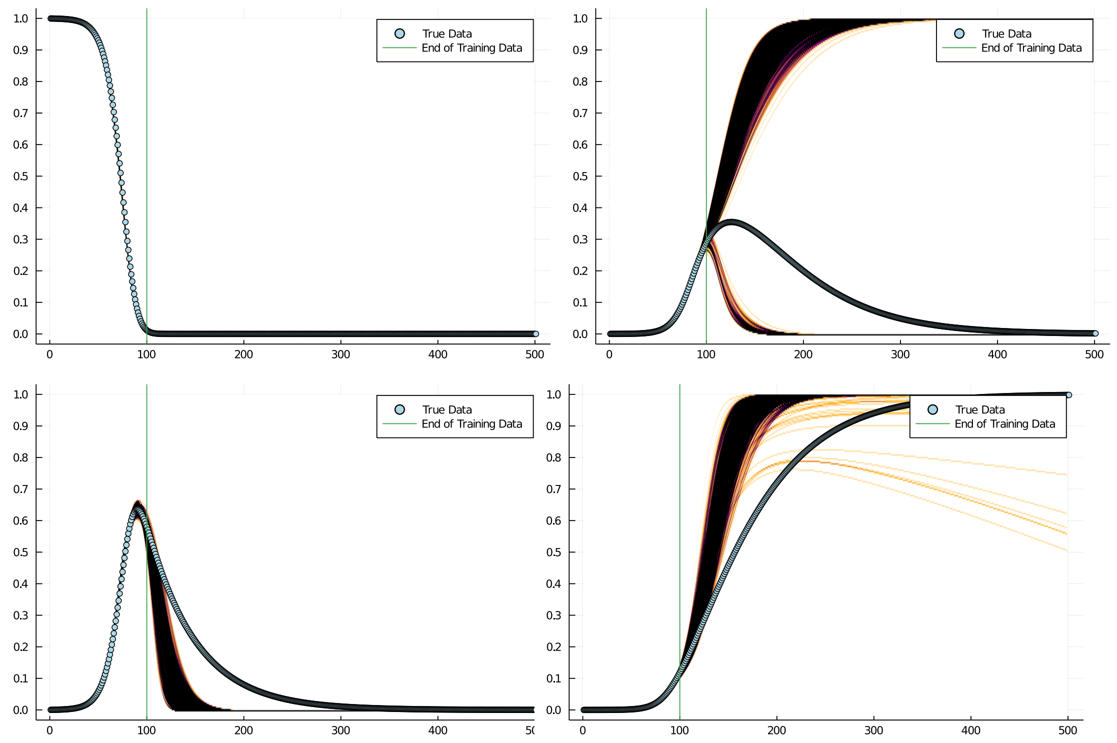

Figure 4.1: Predictive posterior for SEIR using $T = 100$

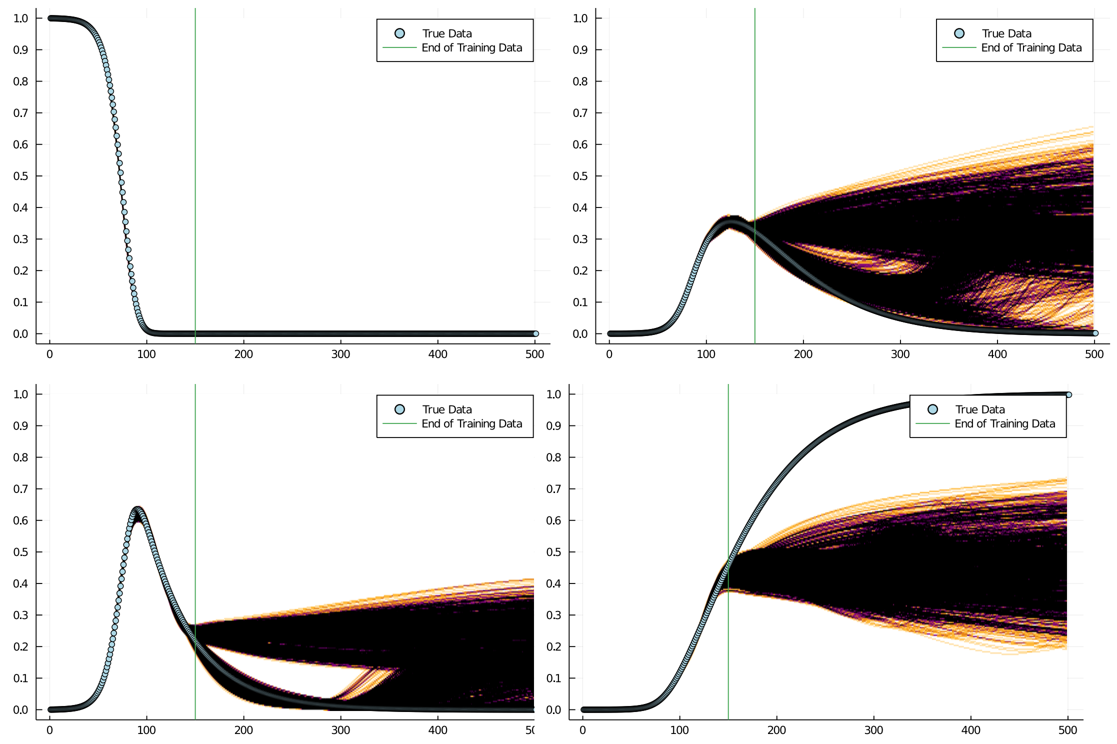

Figure 4.2: Predictive posterior for SEIR using $T = 150$

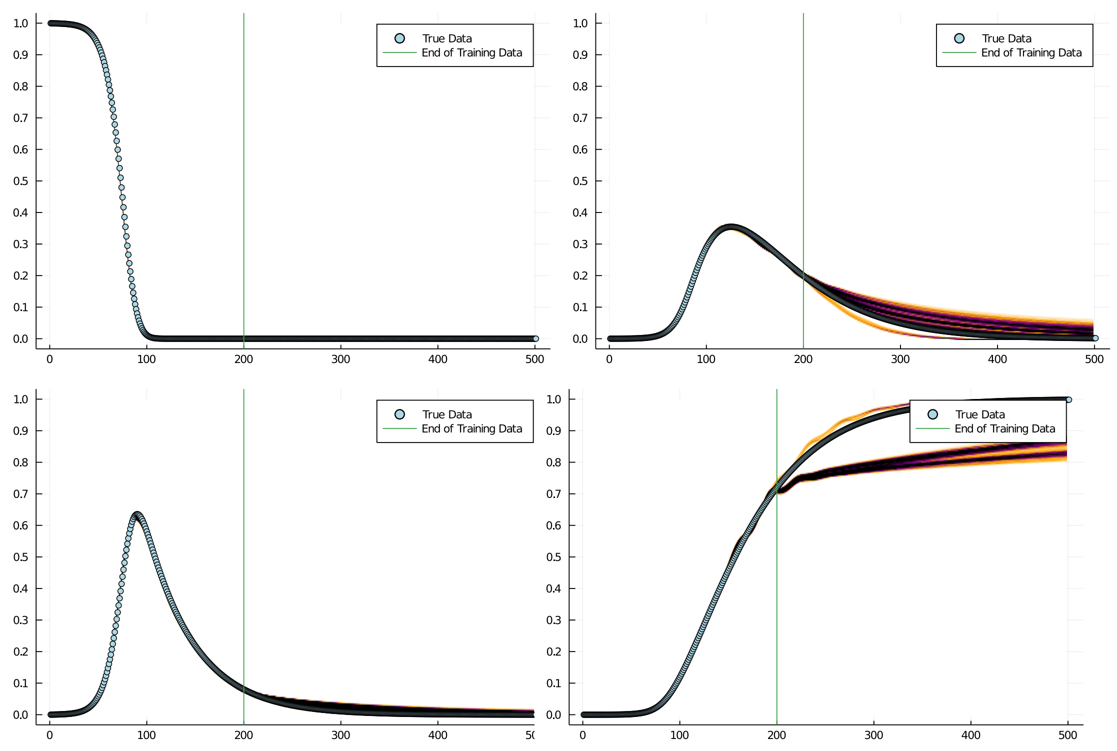

Figure 4.3: Predictive posterior for SEIR using $T = 200$

There are several important things to note from the above figures. In general, the neural network gives a fairly close fit to the data although, while we have our solutions constrained to the $(0,1)$ interval, there is no conservation of constant population for all of the solutions. This is to be expected since the sigmoid mapping trick only takes care of constraining the solution but it doesn't preserve the zero-sum condition for the derivative. This might be mitigated by adding more either more layers or neurons to the neural network in order to allow for greater complexity in the neural network, however, the addition of parameters increases the dimensions of the parameter space which worsens the performance of the sampling algorithms used.

Notice as well that adding more data improves the generalisation capabilities of the model, which again it is to be expected since when more data is added, the neural network gains more information about the system. As we also expected, the variance in the predictive posterior is reduced when more data is added. Overall we notice that in the regime of little training data, the neural network model seems to be more predisposed to overfitting; this is expected since it is common knowledge that neural networks tend to require a high amount of training data to generalise better.

Chapter 5

Conclusions and Future Work

Neural networks are a powerful tool for pattern recognition due to their capability of approximating any function given enough layers and parameters. Differential equations have been for centuries one of the most widely used tools for modelling due to their natural interpretation in many physical and natural processes. Recently, research effort has been directed in matching both tools into an even more powerful tool, combining the high model flexibility of neural networks with the very natural notion of evolving an input through time, notion very naturally expressed with differential equations. One of the immediate areas where both tools come together is in the forecasting of time series, where a neural network is used to recreate from training data, the dynamical equations governing the system.

A particularly important part of forecasting is assessing the trustworthiness of our predictions and quantifying the information we gain from the available data about the system. Neural networks as commonly used, with a frequentist framework, are prone to overfitting and throw away valuable information about the uncertainty of our forecasts. Bayesian inference provides a conceptually clear framework to avoid these issues. Treating the parameters of the neural network as random variables themselves, allows us to define a predictive posterior in which our knowledge about the epistemic uncertainty of our model is encoded.

A key part of Bayesian inference is an adequate selection of priors. Not only does a good prior encode our beliefs about the data better, but a poor prior may render a model useless. While we showed that a Gaussian prior on the neural network's parameters yields good results, there is still work to be done in figuring out better priors, since there's not much foundation for it, other than its ease of computation.

While very conceptually sound, the integrals required to calculate the predictive posterior present a very high barrier for the implementation of Bayesian methods in practise. A common way to go around the intractability of the required integrals are sampling methods, these allow for a sound approximation of the posterior although they usually are computationally expensive and somewhat difficult to evaluate. Bayesian neural networks is a growing field of study which until recently has not been hyped due to the high number of parameters involved, the high dimensionality of the parameter space makes even approximate methods prohibitively expensive to implement. In this work, we compared four different sampling methods, finding that parallel tempering with a descending temperature scale is the most effective one, both in performance and computation time. It is to be noted, though, that all the neural networks used are rather shallow and the number of parallel chains used was relatively low. Unfortunately parallel tempering using random walks as a proposal function doesn't scale very well as the number of parameters increases and loses much of its effectiveness due to the famous curse of dimensionality.

A particularly interesting area of application of Bayesian inference and Neural Ordinary Differential Equations is epidemiological forecasting. A frequentist approach in an area of very high risk, with huge health, economic and societal implications, such as epidemiology is rather useless and a Bayesian approach becomes a very appealing option. One of the most widely used tools for modelling epidemics is compartmental models. These models are governed by a system of differential equations describing the evolution of a pandemic. Here is where NODEs come into action, the effectivity of neural networks to detect patterns in data makes them very attractive for discovering the underlying governing equations of the compartmental model with few assumptions about it. It is important to note though, that compartmental models have very specific mathematical properties which ensure that the model produces interpretable solutions. These properties are not trivial to implement in a neural network and a future line of work is designing a suitable architecture in order to encode more prior knowledge about the model.

In spite of the previously stated, a case study using simulated data from the classic SEIR model shows promising evidence of the predictive power of NODEs and Bayesian methods. We showed that a shallow neural network is enough to approximate the governing equations of the SEIR model, although some of the solutions obtained as part of the predictive posterior are not interpretable, due to the flaws in the architecture that were mentioned above. As it was expected, the epistemic uncertainty of the model is reduced when more data is used, as it was showcased by showing the approximate predictive posterior for different time windows. It was noted that, as it is the case with neural networks, predictions with poor data tended to generalise poorly. A solution to this and the previous point is the use of Universal Ordinary Differential Equations, where only a part of the differential equations are replaced by a neural network[16]. This allows us to write in model structure in a more easy and straightforward manner. An added benefit is that less data is necessary to produce good predictions.

In many practical cases it is not possible to have a reliable sample of certain compartments of epidemiological models. This leaves us with an incomplete vector state for the model's differential equations making them not well defined. It is of interest then to determine the performance of NODEs with missing data; a particularly interesting approach is detailed on [18] where the variable space is artificially augmented to account for the missing variables.

As we can see, Neural Ordinary Differential Equations are a relatively new area of research but the fact that it is conceptually appealing for forecast prediction and the ease of implementation of Bayesian methods make it a very interesting area with very high potential.

Bibliography

[1] C. C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer International Publishing, 2018.

[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[3] G. Carleo, I. Cirac, K. Cranmer, L. Daudet, M. Schuld, N. Tishby, L. Vogt-Maranto, and L. Zdeborová. Machine learning and the physical sciences. *Reviews of Modern Physics*, 91(4), Dec 2019.

[4] T. Charnock, L. Perreault-Levasseur, and F. Lanusse. Bayesian neural networks, 2020.

[5] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud. Neural ordinary differential equations. In *Advances in neural information processing systems*, pages 6571–6583, 2018.

[6] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 5(4):455–455, 1992.

[7] P. V. D. Driessche and J. Watmough. Reproduction numbers and sub-threshold endemic equilibria for compartmental models of disease transmission. *Mathematical Biosciences*, 180(1-2):29–48, 2002.

[8] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT Press, 2016.

[9] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[10] H. Hewamalage, C. Bergmeir, and K. Bandara. Recurrent neural networks for time series forecasting: Current status and future directions. *International Journal of Forecasting*, 37(1):388–427, Jan 2021.

[11] M. D. Hoffman and A. Gelman. The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *J. Mach. Learn. Res.*, 15(1):1593–1623, 2014.

[12] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993.

[13] J. S. Liu. *Monte Carlo strategies in scientific computing*. Springer, 2008.

[14] B. Miasojedow, E. Moulines, and M. Vihola. Adaptive parallel tempering algorithm, 2012.

[15] C. Rackauckas, M. Innes, Y. Ma, J. Bettencourt, L. White, and V. Dixit. Diffeqflux.jl - a julia library for neural differential equations, 2019.

[16] C. Rackauckas, Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, A. Ramadhan, and A. Edelman. Universal differential equations for scientific machine learning, 2020.

[17] C. P. Robert and G. Casella. *Monte Carlo statistical methods*. Springer, 2005.

[18] R. Strauss. Augmenting neural differential equations to model unknown dynamical systems with incomplete state information, 2020.

[19] M. Welling and Y. W. Teh. Bayesian learning via stochastic gradient langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 681–688, 2011.

[20] F. Wenzel, K. Roth, B. S. Veeling, J. Świątkowski, L. Tran, S. Mandt, J. Snoek, T. Salimans, R. Jenatton, and S. Nowozin. How good is the bayes posterior in deep neural networks really?, 2020.