Summer 7-13-2019

A Deep Learning Approach to Uncertainty Quantification

Mst Afroja Akter
University of New mexico

Mst Afroja Akter

Candidate

Mathematics and Statistics

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Dr. Mohammad Motamed, Chairperson

Dr. Stephen Lau

Dr. Jacob B. Schroder

A Deep Learning Approach to Uncertainty Quantification

by

MST AFROJA AKTER

B.Sc., Mathematics, Jahangirnagar University, 2010
M.S., Mathematics, Jahangirnagar University, 2011

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Mathematics

The University of New Mexico
Albuquerque, New Mexico

July, 2019

Dedicated to my friends and family.

Acknowledgements

Thank you Dr. Mohammand Motamed, Dr. Jacob B. Schroder and Dr. Stephen Lau for your support and advice.

A Deep Learning Approach to Uncertainty Quantification

by

Mst Afroja Akter

B.Sc., Mathematics, Jahangirnagar University, 2010
M.S., Mathematics, Jahangirnagar University, 2011
M.S., Mathematics, University of New Mexico, 2019

ABSTRACT

In this thesis we consider ordinary differential equations (ODEs) with random parameters. We focus on Monte Carlo (MC) sampling for computing the statistics of some quantities of interest (QoIs) given by the solution of the ODE problems. We use the 4th order accurate Runge-Kutta (RK4) method as the deterministic ODE solver. We then develop a hybrid MC sampling method that combines RK4 with neural network models to efficiently compute the statistics of QoIs within a desired accuracy. We present several numerical examples to verify the accuracy and efficiency of the proposed hybrid method compared to classical MC sampling. The hybrid method that we develop can be applied to more complicated physical problems given by partial differential equations (PDEs).

Contents

List of Figures

List of Tables

List of Abbreviations

UQ	**U**ncertainty **Q**uantification
ODE	**O**rdinary **D**ifferential **E**quation
MC	**M**onte **C**arlo
ML	**M**achine **L**earning
NN	**N**eural **N**etwork
DNN	**D**eep **N**eural **N**etwork
BPNN	**B**ack **P**ropagational **N**eural **N**etwork

List of Symbols

l	Number of layers
η	Learning rate
σ	Activation function
w	Weight Matrix
b	Bias
C	Cost function
n^l	Number of neurons in layer l
w^l_{kj}	Weight connecting neuron K from layer l with neuron j to layer $(l-1)$
b^l	Bias vector of layer l
a^l_j	Output of neuron j from layer (l-1)
ϵ^l	Error vector in layer l

Chapter 1

Introduction

Many systems in science and engineering can be modelled using differential equations. Due to inaccuracies in the model or the presence of uncertainty, a modelled system can never be one hundred percent accurate. The goal of uncertainty quantification (UQ) is to understand how these unknowns affect our model outputs.

One of the most useful tools in uncertainty quantification is the MC method due to its simplicity and dimension independent convergence rate. Generally, the MC method work by averaging over a sufficiently large number of samples [Metropolis and Ulam 1949; Cunha Jr et al. 2014]. In our case, a sample will have associated with it a deterministic differential equation which must be solved using a numerical solver. Since MC takes many samples to converge, problems over long time periods or with many dimensions may be intractable if the DE solver is computationally too expensive.

In this thesis we consider ODEs with random parameters. We focus on MC sampling for computing the statistics of some QoIs given by the solution of the ODE problem. We use the RK4 method as the deterministic ODE solver. We then develop a hybrid MC sampling method that combines RK4 with neural network models to efficiently compute the statistics of QoIs within a desired accuracy. We present several numerical examples to verify the accuracy and efficiency of the proposed hybrid method compared to classical MC sampling. It is to be noted that the proposed strategy can be applied to more complicated models given by partial differential equations and use more advance UQ techniques, such as stochastic collocation [Mathelin, Hussaini, and Zang 2003].

The rest of the thesis is organized as follows. In Chapter 2 we discuss the prerequisites of feed-forward neural networks. In Chapter 3 we present a general MC method and an RK4 version of the MC method called RK4MC. In Chapter 4, we introduce a hybrid MC method named NNMC. Finally, we discuss the results of our numerical examples in Chapter 5 and the conclusion is in Chapter 6.

Chapter 2

Artificial Neural Networks

The term artificial neural networks(ANN) refers to a large group of algorithms inspired by the biological function of the human brain. Such algorithms have the ability to "learn" from data and make decisions or determine trends with little to no domain knowledge. This feature has led ANNs being applied to problems in many fields. In this chapter we will explain the core ANN algorithms used to generate our thesis result.

Regression

Most machine learning (ML) algorithms are based on supervised learning, that is when both training inputs and outputs are available. Supervised ML problems can be divided into regression and classification problems. In this thesis we are performing function approximation which falls in to the regression category. The most popular neural network (NN) architecture for solving regression problems is called feed-forward. One reason these neural networks is so popular is because they can theoretically approximate a large class of functions to arbitrary accuracy. This property is known as the universal approximation theorem for feed-forward networks [Hassoun 1995].

Feed-Forward Networks

In figure 2.1 a typical diagram of a neural network is shown. The action of the network is to take a vector of length 3, apply an affine transformation and feed the result through an activation function. A general feed-forward network will perform this operation multiple times with different affine transformations. These models are called feed-forward because the information among the neurons flows in a forward direction from the input layer to the output layer.

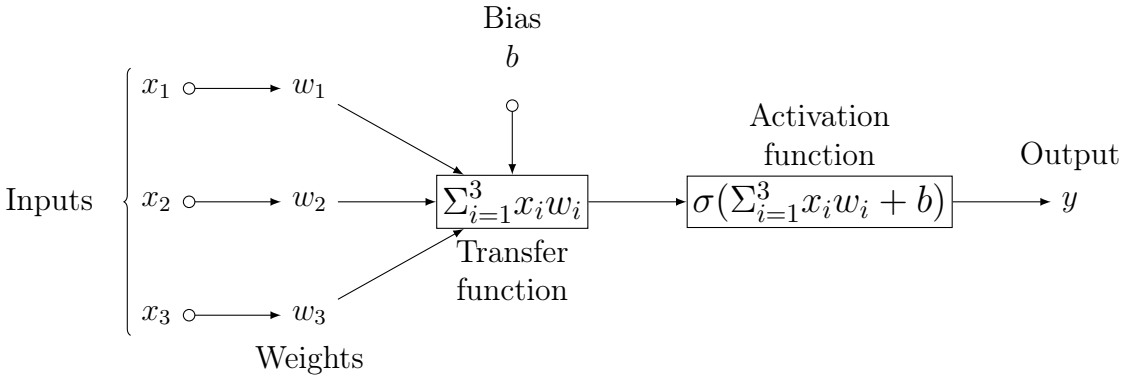


FIGURE 2.1: Simple feed-forward neural network

Figure 2.2 is an example of feed forward deep neural network with two hidden layers. Formally we can define a feed-forward network as a function $a : \mathbb{R}^{n^1} \to \mathbb{R}$ defined by

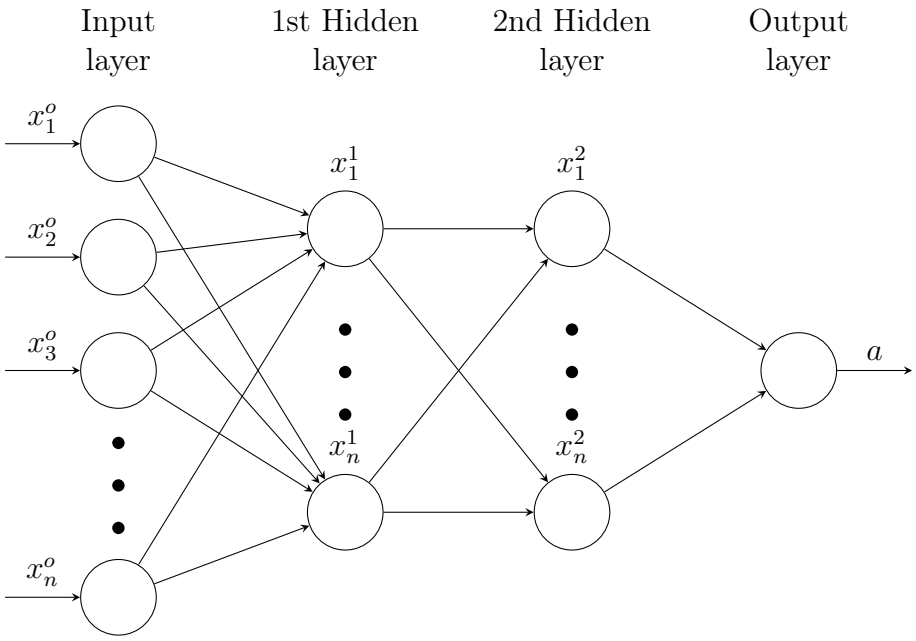


FIGURE 2.2: A Deep Neural Network

$$
\begin{aligned}
a(x) &= a^L \circ a^{L-1} \circ \cdots \circ a^2 \circ a^1(x) \\
a^j(x) &= \sigma^j(w^j x + b^j) \\
w^j &\in \mathbb{R}^{n^{j-1} \times n^j}, \\
b^j &\in \mathbb{R}^{n^j}, \\
\sigma^j &: \mathbb{R}^{n^j} \to \mathbb{R}^{n^j},
\end{aligned}
$$

where $\sigma_i^j = \sigma_j$ and $\sigma_j : \mathbb{R} \to \mathbb{R}$ is smooth and monotonically increasing. We refer to w^k, b^k, σ^k as the **weights**, **biases** and **activation function** of layer k. The elements of w^k, b^k are generally referred to as parameters of the network. Now we can discuss the universal approximation theorem.

Universal approximation theorem. Let σ be a smooth monotonically increasing function. Then given any continuous real-valued function f on a compact subset K of $\mathbb{R}^n$ and $\varepsilon > 0$, there exists vectors $w_1, \ldots, w_N, \alpha$, and b such that

$$|a(x, w, \alpha, b) - f(x)| < \epsilon \quad \text{for all } x \in K$$

where

$$a(x, w, \alpha, b) = \sum_{j=1}^{N} \alpha_j \sigma(w_j^T x + b_j)$$

and $w_j \in \mathbb{R}^n, \alpha_j, b_j \in \mathbb{R}$. [Hassoun 1995].

The remaining challenge is the actual determination of activation functions and parameters. This brings us to the process of training.

2.1 Training Process

We wish to find a neural network a which is close to f with respect to some metric which we call the cost function. The training process starts by choosing a cost function which can greatly affect the training process in practice. Once a cost function is chosen, it is minimized with respect to the weights and biases. The final weights and biases are then used by the network. One example of a cost function is the **Mean Squared Error**

$$C(w, b) = \frac{1}{2} \sum_{x^j} (f(x^j) - a(x^j))^2 \tag{2.1}$$

where $\{x_j\}$ is some finite subset of the domain.

To evaluate and validate the network during the training process we first generate or collect a representative dataset of f function evaluations with corresponding input values, which is typically split into three parts:

- Training: This data is used in the optimization step which may entail calculating gradients of the cost function in the case of gradient descent.
- Validation: This data provides an unbiased evaluation of a model on the training dataset while tuning hyperparameters. This dataset can be used to influence training and therefore cannot be used to test the final model without introducing bias.
- Testing: This data provides an unbiased evaluation of final model selection. After testing the model on this dataset we save the model for further use.

The most popular optimizers used in practice are variants of gradient descent. For neural networks, the gradient can be calculated efficiently using the feed-forward and back propagation algorithms [Nielsen 2015].

Algorithm 1 Gradient Descent(GD)

1. **Input x:** Assuming a^1 is the output of input layer.
2. **Feedforward:** For each layer, $l = 2, 3, 4, ..., L$ compute

$$z^l = w^l a^{l-1} + b^l \quad \text{and} \quad a^l = \sigma(z^l) = \sigma(w^l a^{l-1} + b^l)$$

   Where z^l is input and a^l is output vector of layer l.
3. **Output error:** Compute the error vector,

$$\epsilon^L = \nabla_a C \odot \sigma'(z^L).$$

   Where, $a \odot b$ is the element-wise product of two vectors a and b, sometimes called Hadamard product.
4. **Backpropagate the error:** For each layer, $l = L-1, L-2, ..., 2$ compute

$$\epsilon^l = ((w^{l+1})^T \epsilon^{l+1}) \odot \sigma'(z^l)$$

5. **Gradient descent:** The gradient of the cost function is given by

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \epsilon^l_j \quad \text{and} \quad \frac{\partial C}{\partial b^l_j} = \epsilon^l_j.$$

6. **Weight and bias update:** For each layer $l = L-1, L-2, ..., 2$ update the weights and biases according to

$$w^l \to w^l - \eta \nabla_{w^l} C = w^l - \eta \sum_x \epsilon^l (a^{l-1})^T,$$

and

$$b^l \to b^l - \eta \nabla_{b^l} C = b^l - \eta \sum_x \epsilon^l$$

where η is the learning rate.

Repeatedly applying the above algorithm, we can find a model with best fit of weights and biases that minimizes the cost function. Gradient descent is one of commonly used training algorithms. Discussion about other optimization algorithms are in section 2.2.5.

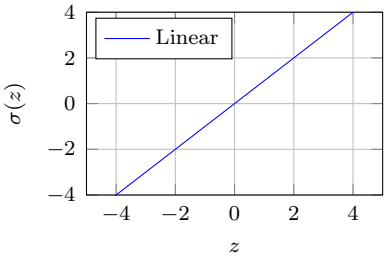
2.2 Hyperparameters and Tuning Technique

2.2.1 Activation Functions

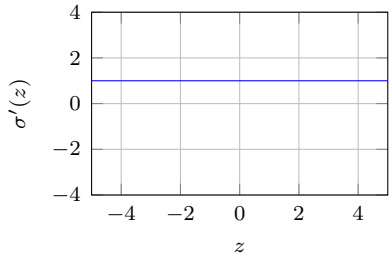
The human brain has billions of neurons and they pass electronic signals from one neuron to another through synapses.Whenever we see, hear or feel something a synapses is fired from one neuron to another. The activation function, denoted by σ, does the same work for ANN. An activation function decides whether the weighted input of the current neuron is going to pass to the next neuron or not.

$$a = \sigma(\sum z) = \sigma(\sum xw + b),$$

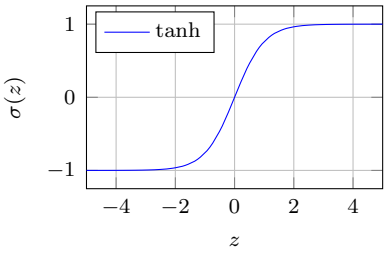
where a is the neuron output and z is the neuron input for a layer.
It introduces non-linearity to our network. If we do not apply the activation function then the neurons output would be linear to the input, which is a polynomial of degree one. That is, a ANN without an activation function is a simple linear regression model and it will fail to learn correlation between more complex and multidimensional nonlinear data.
There many activation functions for different kind of problems. Most NN learning is based on gradient-descent which requires derivative of activation functions. Figure 2.3 provides the activation functions and their derivatives that we used in our thesis [Géron 2018].



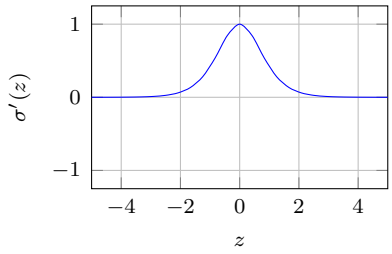
(a) Linear



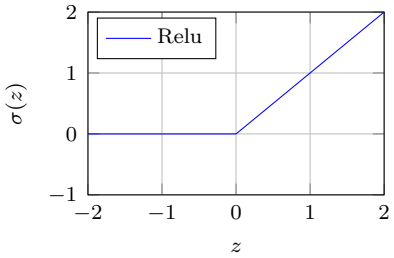
(b) Derivative of Linear activation function.



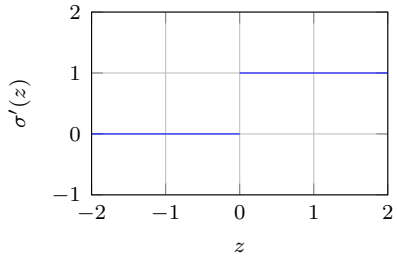
(c) Hyperbolic tangent



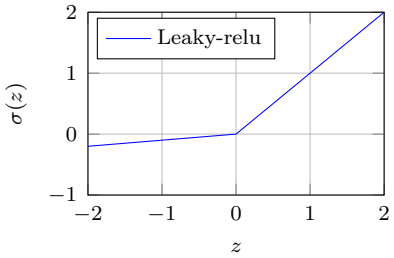
(d) Derivative of $\tanh(z)$.



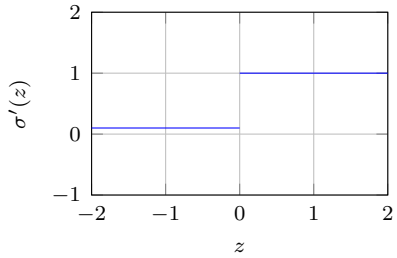
(e) Rectified linear unit(Relu)



(f) Derivative of Relu.



(g) Leaky-Rule



(h) Derivative of Leaky-Rule.

FIGURE 2.3: Commonly used activation functions and their derivatives

Linear

$$\sigma(z) = z$$

The linear activation function gives the same output as input. Since it's derivative is 1 everywhere, the gradient descent is constant and does not depend on the input data z. That is, changing the number of input data doesn't affect the model performance. Furthermore, if all layers have a linear activation function, then the final activation function of the last layer is equivalent to a linear function of the input of first layer. That means all layers can be replaced by a single layer.

Hyperbolic Tangent

$$\sigma(z) = \tanh(z) = \frac{2}{1 + e^{-2z}} - 1$$

This activation function is S-shaped, continuous, differentiable and it's output ranges from -1 to 1, which makes each layer output normalized at the beginning of the training. This often helps to speed up the training.

Rectified Linear Unit(Relu)

$$\sigma(z) = \max(0, z)$$

The relu is a popular method for function approximation. It is fast to compute and continuous and differentiable everywhere except at 0. It's derivative changes directly from 0 to 1, which makes gradient descent bounce around. Since the relu activation function vanishes all the negative inputs, neurons with negative values can not make any contribution to the results.

Leaky-ReLU

The Leaky-ReLU function is an improved version of the **ReLU** activation function. For relu, the gradient is 0 for $x < 0$, which made the neurons die for the activation in that region. Leaky-relu is introduced to address this problem. Which is defined as

$$\sigma(z) = \begin{cases} 0.01z & z < 0 \\ z & z \geq 0 \end{cases}$$

2.2.2 Learning Rate, η

During the training process we update the weights of the model using backpropagation algorithm. The size of the update during this process is controlled by the **step size**

or **learning rate**. Learning rate ranges from 0 to 1 is a hyperparameter that controls the rate at which the model can learn. We always have to be very careful to choose a leaning rate that is not too large or too small. Because, "When the learning rate is too large, gradient descent can inadvertently increase rather than decrease the training error. When the learning rate is too small, training is not only slower, but may become permanently stuck with a high training error" [Goodfellow, Bengio, and Courville 2016].

2.2.3 Number of Epochs

One epoch is a one complete cycle of feed-forward and backpropagation of the entire dataset through the NN. One epoch is never enough to train a NN. Choosing a number of epochs highly depends on the diversity of data, but not using an appropriate number of epochs would lead the model to overfit or underfit. Unfortunately, there is no way to choose one specific number but there are some regularization techniques that we can use to overcome this situation.

2.2.4 Batch Size

Applying the NN to a large dataset is usually too time and memory intensive. We can divide training dataset into smaller batches and use smaller samples to train the NN, which makes training faster and more memory efficient. **Batch size** is the total number of training examples present in a single batch.

2.2.5 NN Optimization

Optimization is a process of searching for parameters w, b that optimize a given function. For NN, we minimize a cost[1] function $C(w, b)$ and save best parameters of our model. Usually finding a minimum of a non-convex cost function is not easy. Often gradient descent gets stuck in the local minimum or saddle point[2], finding a way out these local minimum or saddle point is challenging. We will talk about some popular optimization functions and general techniques to handle this challenge.

Gradient Descent

In the gradient descent algorithm, the weight and bias updates are done by,

$$w \to w - \eta \nabla_w C \quad \text{and} \quad b \to b - \eta \nabla_b C,$$

[1]The choice of cost function is specifically related to the problem and the result we care about.
[2]Gradient is almost zero in all directions of this point, making it impossible to escape

where η is the learning rate. If η is very small, the training requires large number of iterations and time, whereas a very large step size might prevent the model from finding the minimum. If the gradient is zero in all direction, GD fails to escape from the saddle point. Moreover, GD uses the entire training dataset in each iteration. This is not a problem if we are using few thousands training data, but usually NNs work best with millions of training data. Working with the entire training dataset is time and memory consuming.

Mini-Batch GD

To overcome problems with GD, we apply the mini-batch GD algorithm where we divide the whole training dataset into smaller batches and train over each batch sequentially. Doing so makes algorithm faster and more memory efficient and gives us an intuition of GD before finishing the entire training dataset. For a mini-batch of m training examples, the cost function for one mini-batch (x^m, f^m) ,

$$\begin{aligned} C_m &= \frac{1}{m}\left[\frac{1}{2}\left|f^m(x^m) - a^{m,L}\right|^2\right] \\ &= \frac{1}{2m}\Sigma_{j=1}^{m}\left|f_j^m - a_j^{m,L}\right|^2 \end{aligned} \tag{2.2}$$

Algorithm 2 MBGD

1. **Input** a set of m training examples.
2. **For a mini-batch of m training examples:** Set the corresponding input activation $a^{m,1}$ and perform the following steps:
   - **Feed-forward:** For each $l = 2, 3, ..., L$ compute

     $$z^{m,l} = w^l a^{m,l-1} + b^l \quad \text{and} \quad a^{m,l} = \sigma(z^{m,l}).$$

   - **Output Error $\epsilon^{m,L}$:** Compute the error vector

     $$\epsilon^{m,l} = \nabla_a C_m \odot \sigma'(z^{m,l}).$$

   - **Backpropagate the error:** For each $l = L-1, L-2, ..., 2$ compute

     $$\epsilon^{m,l} = ((w^{l+1})^T \epsilon^{m,l+1}) \odot \sigma'(z^{m,l}).$$

   - **Weight and bias update:** For each $l = L-1, L-2, ..., 2$ update the weights and biases according to

     $$w^{m,l} \to w^{m,l} - \frac{\eta}{m}\Sigma_m \epsilon^{m,l}(a^{m,l-1})^T,$$

and

$$b^{m,l} \to b^{m,l} - \frac{\eta}{m}\Sigma_m \epsilon^{m,l}.$$

where η is the learning rate.

The path of gradient descent using the mini-batch GD algorithm is a little noisy than the GD algorithm [Ng 2017]. The possible reason of the noise could be because some mini-batches with misleading examples. There is no specific rule how to choose the mini-batch size. But if the mini-batch size is very large then it will behave like the batch GD and if it is very small(say 1) then each example will be a mini batch. Which is Stochastic gradient descent(SGD) with very large noise and we loose the speedup since we do the feed forward and back-propagation process for each training example.

GD With Momentum

In mini-batch GD we use a subset of training data to update the network parameters, which brings some oscillation to the gradient path towards the convergence. GD with momentum makes the path smooth by updating the parameter with a weighted average of gradient, V.
From now we will denote the derivatives $\frac{\partial C}{\partial W}$ as dW and $\frac{\partial C}{\partial b}$ as db for simplicity of writing.

Momentum is responsible for a smooth path towards convergence by taking previous gradients into account. We can apply it with batch GD, mini-batch GD or SGD.

Root Mean Squared Propagation

The root mean squared propagation (RMSProp) focus on reducing the oscillations of the gradient path towards the convergence, in a different way than the gradient descent with momentum. Unlike GDM algorithm, there is no need to adjust the learning rate for the RMSProp algorithm. The RMSProp optimizer adjusts the learning rate automatically by choosing a different learning rate for each parameter according to the equations in the following algorithm.

Denote the exponential average of squares of past gradients along W and b as S_{dW} and S_{db} respectively.

To avoid dividing by zero, we add a very small number ε in the denominator. Dividing the gradient by corresponding square root makes the learning rate reduce faster for the parameter where the gradient is large and slower for the parameter where the gradient is smaller. This prevents the noise in the convergence path.

Algorithm 3 GDM

On iteration t:

1. Compute the derivatives dW, db using current the mini-batch.
2. Denote V_{dW} and V_{db} as the exponential average of past gradients along the weight W and the bias b respectively, defined by:

$$V_{dW} = \beta V_{dW} + (1 - \beta) dW,$$

$$V_{db} = \beta V_{db} + (1 - \beta) db.$$

3. Update the weight and bias:

$$W = W - \eta V_{dW},$$

$$b = b - \eta V_{db}.$$

Where η is learning rate and β is another hyperparameter ranging from 0 to 1, called momentum [Ng 2017].

Algorithm 4 RMSProp

On iteration t:

1. Compute the derivatives dW, db using current mini-batch.
2. Compute:

$$S_{dW} = \beta S_{dW} + (1 - \beta) dW^2,$$

$$S_{db} = \beta S_{db} + (1 - \beta) db^2.$$

3. Update weight and bias:

$$W = W - \frac{\eta}{\sqrt{S_{dW}} + \varepsilon} S_{dW},$$

$$b = b - \frac{\eta}{\sqrt{S_{db}} + \varepsilon} S_{db}.$$

Where η is the learning rate and momentum β ranges from 0 to 1 [Ng 2017].

Adaptive Moment Estimation(Adam)

The GDM algorithm accelerates the search of gradient descent path towards the direction of minima, whereas in the RMSProp algorithm continue the search in the direction of oscillations. The Adam optimizer takes biggest pros of RMSProp and

combine them with idea known from momentum optimization. The following algorithm includes the update equations required for the Adam optimizer.

Algorithm 5 Adam

On iteration t:

1. Initialize $V_{dw} = 0, S_{dw} = 0, V_{db} = 0, S_{db} = 0,$
2. Compute the derivatives dW, db using current the mini-batch.
3. Compute V_{dw} and V_{db} like momentum:

$$V_{dW} = \beta_1 V_{dW} + (1 - \beta_1) dW$$

$$V_{db} = \beta_1 V_{db} + (1 - \beta_1) db.$$

   where V_{dW} and V_{db} are the exponential average of past gradients along W and b respectively.

4. Compute S_{dw} and S_{db} like RMSProp:

$$S_{dW} = \beta_2 S_{dW} + (1 - \beta_2) dW^2$$

$$S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2.$$

   where S_{dW} and S_{db} are the exponential average of squares of past gradients along W and b respectively.

5. Compute:

$$V_{dW}^{\text{corrected}} = \frac{V_{dW}}{(1 - \beta_1^t)},$$

$$V_{db}^{\text{corrected}} = \frac{V_{db}}{(1 - \beta_1^t)},$$

$$S_{dW}^{\text{corrected}} = \frac{S_{dW}}{(1 - \beta_2^t)},$$

$$S_{dW}^{\text{corrected}} = \frac{S_{db}}{(1 - \beta_2^t)}.$$

6. Update W, b:

$$W = W - \frac{\eta}{\sqrt{S_{dW}^{\text{corrected}}} + \varepsilon} V_{dW}^{\text{corrected}}$$

$$b = b - \frac{\eta}{\sqrt{S_{db}^{\text{corrected}}} + \varepsilon} V_{db}^{\text{corrected}}$$

   Where η is learning rate needs to be tuned during training and momentum β_1 and β_2 ranges from 0 to 1 and common choice for them is 0.9 and 0.999

respectively and $\varepsilon = 10^{-8}$ recommended by Adam paper [Kingma and Ba 2014].

2.3 Regularization Technique

There are many parameters that can cause overfitting of a model. Overfitting is a situation when the NN model memorize the training data but fails to perform on new data. Regularization is a techniques that makes some modifications to the leaning algorithm in order to generalize the model. There are some popular regularization technique. In our computation we use **Early stopping** to prevent the model from overfitting.

Early stopping

A large number of epochs may lead the model to overfit on training data, whereas too few may result to underfitting. Early stopping is a technique that allows us to give an arbitrary large number of training epochs to our model and stop training once the model performance on the validation data starts decreasing.

Chapter 3

Uncertainty Quantification for Differential Equations

Many physical and biological systems are modeled by differential equations. A major difficulty in the study of these systems arises from the presence of uncertainty, due to our limited knowledge about the system and/or the intrinsic variability of the system [Sullivan 2015].

Uncertainty Quantification is a process that enables us to identify and characterize uncertainty in the system and propagate it through the mathematical model to obtain output predictions.

Among different types of UQ, we are concerned with the forward propagation of uncertainty, where the uncertainty in the input model parameters is known and described by a set of random variables. Our goal is to obtain the uncertainty for output at some QoIs given by the solution of the underlaying mathematical models presented in 3.1.

3.1 Problem Statement

We consider ordinary differential equations (ODEs) of the following form:

$$\begin{aligned} u_t(t,Y) &= f(t,u,Y), \quad t \geq 0, \quad Y \in \Gamma \subset \mathbb{R}^N \\ u(0,Y) &= g(Y) \end{aligned} \tag{3.1}$$

where $u = (u_1, u_2, ..., u_r)^T \in \mathbb{R}^r$ is the vector of unknowns, $Y = (y_1, y_2, ..., y_P)^T \in \Gamma \subset \mathbb{R}^P$ is a vector of P random variables, $f(t,u,Y)$ is a given function and $g(Y)$ is a random function.

We want to evaluate **expected value**, $\mathbb{E}\Big[u(T,Y)\Big]$ of the ODE 3.1 at $t = T$.

3.2 Monte Carlo Sampling

MC is a simple and popular method for computing integrals and in particular expected values. A basic MC method to compute the expected value of $u(t, Y)$ with respect to Y is the following:

Algorithm 6 MC

1. Draw N samples $\{Y^{(n)}\}_{n=1}^{N}$ of a random distribution Y.
2. Compute approximations $\left\{\tilde{u}(T, Y^{(n)})\right\}_{n=1}^{N}$, which require solving N deterministic ODEs.
3. Evaluate the expected value $\mathbb{E} \approx \frac{1}{N} \sum_{n=1}^{N} \tilde{u}(T, Y^{(n)}) := \mathscr{A}_{MC}$.

3.2.1 MC Error Analysis

Let $\tilde{u}(T, Y^{(n)})$ be n approximate solutions of the ODE 3.1. We define the MC estimator,

$$\mathscr{A}_{MC} = \frac{1}{N} \sum \tilde{u}(T, Y^{(n)}) \tag{3.2}$$

There are two errors in $\mathscr{A}_{MC}$ [Motamed 2018]:

- Error in approximating the solution $u(T, Y^{(n)})$ by $\tilde{u}(T, Y^{(n)})$.
- Error in approximating the integral by sum.

Therefore, error in the approximated expected values is,

$$\begin{aligned} \epsilon_{MC} &= \left| \mathbb{E}\left[u(T, Y)\right] - \mathscr{A}_{MC} \right| \\ &= \left| \mathbb{E}\left[u(T, Y)\right] - \mathbb{E}\left[\tilde{u}(T, Y)\right] + \mathbb{E}\left[\tilde{u}(T, Y)\right] - \mathscr{A}_{MC} \right| \\ &\leq \left| \mathbb{E}\left[u(T, Y) - \tilde{u}(T, Y)\right] \right| + \left| \mathbb{E}\left[\tilde{u}(T, Y)\right] - \mathscr{A}_{MC} \right| \\ &= \epsilon_I + \epsilon_{II}. \end{aligned} \tag{3.3}$$

We call ϵ_I and ϵ_{II} the discretization error and statistical error respectively. We wish to find an upper bound for ϵ_{MC}; to do this we must assume that both u and the error in the approximation to u are bounded. That is, there exists $C, \varepsilon > 0$ such that for all T and Y, $|u| < C$ and $|u - \tilde{u}| < \varepsilon$.

We obtain an upper bound for the discretization error,

$$\epsilon_I = |\mathbb{E}\left[u - \tilde{u}\right]| \leq \mathbb{E}\left[|u - \tilde{u}|\right] \leq \varepsilon.$$

To bound the statistical error we use the following result,

$$\begin{aligned}
\mathbf{V}\text{ar}\left[\tilde{u}\right] &= \mathbb{E}\left[\tilde{u}^2\right] - \mathbb{E}\left[\tilde{u}\right]^2 \\
&\leq \mathbb{E}\left[\tilde{u}^2\right] \\
&= \mathbb{E}\left[(u - (u - \tilde{u}))^2\right] \\
&\leq \mathbb{E}\left[u^2\right] + \mathbb{E}\left[(u - \tilde{u})^2\right].
\end{aligned}$$

The first term of the last expression is bounded since u is bounded,

$$\mathbb{E}\left[u^2\right] = \int_\Gamma u^2 \pi(Y) dY \leq \int_\Gamma C^2 \pi(Y) dY < \infty.$$

where $\pi(Y)$ is the probability density function.
The other term is bounded since the approximation error is bounded,

$$\mathbb{E}\left[(u - \tilde{u})^2\right] = \int_\Gamma (u - \tilde{u})^2 \pi(Y) dY \leq \int_\Gamma \varepsilon^2 \pi(Y) = \varepsilon^2.$$

Therefore, $\mathbf{V}\text{ar}[\tilde{u}] < \infty$, by Central Limit Theorem, we have

$$\begin{aligned}
\epsilon_{II} &= |\mathbb{E}\left[\tilde{u}(T, Y)\right] - \mathscr{A}_{MC}| \\
&= \left| \mathbb{E}\left[\tilde{u}(T, Y)\right] - \frac{1}{N} \sum_{n=1}^{N} \tilde{u}(T, Y^{(n)}) \right| \\
&\lesssim C \frac{\sqrt{\mathbf{V}\text{ar}\left[\tilde{u}(T, Y)\right]}}{\sqrt{N}} \\
&= \mathcal{O}\Big(\frac{1}{\sqrt{N}}\Big).
\end{aligned} \tag{3.4}$$

Here, we use notation $\lesssim$ instead of $\leq$, because the left-hand-side is a random quantity (since it depends on $\left\{Y^{(n)}\right\}_{n=1}^{N}$), while the right-hand-side is a deterministic quantity.

Hence, the bound for the total MC error,

$$\begin{aligned} \epsilon_{MC} &\leq \epsilon_I + \epsilon_{II} \\ &\lesssim \varepsilon + \frac{C}{\sqrt{N}} \\ &= \mathcal{O}(\varepsilon) + \mathcal{O}\big(N^{-\frac{1}{2}}\big). \end{aligned} \tag{3.5}$$

3.2.2 Deterministic ODE Solver

Consider the ODE problem 3.1. We choose the RK4 method to evaluate $u(T, Y)$. The MC algorithm with RK4 ODE solver is,

Algorithm 7 RK4MC

1. Draw N samples $\{Y^{(n)}\}_{n=1}^{N}$ of a random vector Y.
2. Compute approximate solutions $\left\{u_h(T, Y^{(n)})\right\}_{n=1}^{N}$, of the ODE using RK4 method.
3. Evaluate the expected value, $\mathscr{A}_{RK4MC} := \frac{1}{N}\sum_{n=1}^{N} u_h(T, Y^{(n)})$.

RK4 Method

The RK4 method is based on the higher order terms of the Taylor series expansion of $u(\cdot, Y)$. For step-size $h > 0$, define

$$\begin{aligned} u_0 &= u(0, Y) \\ u_{n+1} &= u_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \\ t_{n+1} &= t_n + h \end{aligned} \tag{3.6}$$

for $n = 0, 1, 2, 3, \ldots$

$$\begin{aligned} k_1 &= f(t_n, u_n, Y), \\ k_2 &= f(t_n + \frac{h}{2}, u_n + \frac{k_1}{2}, Y), \\ k_3 &= f(t_n + \frac{h}{2}, u_n + \frac{k_2}{2}, Y), \\ k_4 &= f(t_n + h, u_n + k_3, Y). \end{aligned} \tag{3.7}$$

The value u_n approximates u at time $t = nh$.

3.2.3 MC complexity analysis with RK4 ODE solver

The cost to compute each $\tilde{u}(T, Y^{(n)}) = u_h(T, Y^{(n)})$ is $4C_f\left(\frac{T}{h}\right)$, where C_f is the computation cost of each slope k_i and $\frac{T}{h}$ is the total number of steps. Since C_f and T are constants the computation cost of RK4 method is proportional to h^{-1}. To evaluate $u_h(T, Y)$ for N realizations costs Nh^{-1}.

Therefore, the computational cost of MC estimator, $\mathscr{A}_{RK4MC} = \frac{1}{N}\sum_{n=1}^{N} u_h(T, Y^{(n)})$ is

$$W_{RK4MC} \propto Nh^{-1}. \tag{3.8}$$

Optimal selection of h and N

For RK4 method, the discretization error, generated from the ODE solver, $\epsilon_I = C_1 h^4$, makes the Monte-Carlo error bound,

$$\epsilon_{RK4MC} \lesssim C_1 h^4 + \frac{C}{\sqrt{N}} = \mathcal{O}(h^4) + \mathcal{O}(N^{-\frac{1}{2}}).$$

In addition, we desire that the minimum of W_{RK4MC} is subject to the MC error ϵ_{RK4MC} being less than or equal to some given tolerance ϵ_{tol}. Hence, we can find h_{opt} and N_{opt} by solving the following optimization problem

$$\begin{aligned} &\min_{h,N} W_{RK4MC} \quad \text{s.t} \quad \epsilon_{MCRk4} \leq \epsilon_{\text{tol}} \\ \text{i.e.} \quad &\min_{h,N}(Nh^{-1}) \qquad \text{s.t} \quad C_1 h^4 + \frac{C_2}{\sqrt{N}} = \epsilon_{\text{tol}} \end{aligned} \tag{3.9}$$

Let's introduce the Lagrangian to solve the minimization problem,

$$\mathcal{L}(h, N, \lambda) = Nh^{-1} + \lambda\Big(C_1 h^4 + \frac{C_2}{\sqrt{N}} - \epsilon_{\text{tol}}\Big)$$

Applying, $\partial_h \mathcal{L} = \partial_N \mathcal{L} = \partial_\lambda \mathcal{L} = 0$, yields three equations with three unknowns (h, N, λ).

$$\begin{aligned} &\partial_h \mathcal{L}(h, N, \lambda) = 0, \\ &-Nh^{-2} + 4\lambda C_1 h^3 = 0, \\ &\lambda = \frac{1}{4C_1} N h^{-5} \end{aligned} \tag{3.10}$$

Similarly,

$$\begin{aligned} &\partial_N \mathcal{L}(h, N, \lambda) = 0, \\ &h^{-1} - \frac{1}{2}\lambda C_2 N^{\frac{-3}{2}} = 0, \\ &\lambda = \frac{2}{c_2} N^{\frac{3}{2}} h^{-1} \end{aligned} \tag{3.11}$$

Equating both λ's,

$$\begin{aligned} &\frac{1}{4C_1} N h^{-5} = \frac{2}{C_2} N^{\frac{3}{2}} h^{-1} \\ &N^{-\frac{1}{2}} = \frac{8C_1}{C_2} h^4 \end{aligned} \tag{3.12}$$

Finally

$$\begin{aligned} &\partial_\lambda \mathcal{L}(h, N, \lambda) = 0 \\ &C_1 h^4 + C_2 N^{-\frac{1}{2}} - \epsilon_{\text{tol}} = 0 \end{aligned} \tag{3.13}$$

Substituting $N^{-\frac{1}{2}} = \left(\frac{8C_1}{C_2}\right) h^4$ gives,

$$\begin{aligned} &C_1 h^4 + C_2 \left(\frac{8C_1}{C_2}\right) h^4 = \epsilon_{\text{tol}} \\ &9C_1 h^4 = \epsilon_{\text{tol}} \\ &h^4 = \frac{\epsilon_{\text{tol}}}{9C_1} \\ &h_{\text{opt}} = \mathcal{O}(\epsilon_{\text{tol}}^{\frac{1}{4}}) \end{aligned} \tag{3.14}$$

Replacing h into N,

$$N^{-\frac{1}{2}} = \frac{8}{9C_2}\epsilon_{\text{tol}} \tag{3.15}$$

$$N_{\text{opt}} = \mathcal{O}(\epsilon_{\text{tol}}^{-2}) \tag{3.16}$$

Therefore, with the optimal choice of h and N and using equation 3.8, the optimal cost of MC becomes:

$$W_{opt} = \mathcal{O}\left(\epsilon_{tol}^{-2-\frac{1}{4}}\right) = \mathcal{O}\left(\epsilon_{tol}^{-2.25}\right) \tag{3.17}$$

In summary, MC method converges at a rate of $N^{-\frac{1}{2}}$, which means that the deterministic ODE solver needs to be evaluated for a large number of samples. In the RK4 method, the larger N couples with a small step size h, which makes the overall computation process expensive. In Chapter 4, we will discuss an algorithm which reduces the cost of solving ODE.

Chapter 4

A Hybrid Neural Network Monte Carlo Sampling Method

Monte Carlo converges at a rate of $\mathcal{O}(N^{-\frac{1}{2}})$, which requires a large number of samples for $\mathscr{A}_{MC}$ to approximate $\mathbb{E}$ well. In chapter 3, we discussed the numerical method RK4. Here, we will introduce another method which is a combination of RK4 and a NN. Our goal is to develop a model using a small training dataset which can find solutions with the same accuracy as RK4. Once the model is trained, we will use it to evaluate the approximate solutions $\left\{u(T, Y^{(n)})\right\}_{n=1}^{N}$ of the ODE 3.1 for a large set $\left\{Y^{(n)}\right\}_{n=1}^{N}$ of samples, where N is the number of samples defined in equation 3.15. The algorithm for this method is discussed below:

Algorithm 8 Hybrid NNMC

---

1. Draw M random samples $\{Y^{(m)}\}_{m=1}^{M}$, where $M \ll N$.
2. Solve the ODE M times with step size h using the RK4 method.
3. Train a NN using the dataset of M training examples.
4. Draw N random samples, $\{Y^{(n)}\}_{n=1}^{N}$.
5. Use trained NN to predict the approximate solutions $u_{h,NN}(T, Y^{(n)})$, of the ODE at $\{Y^{(n)}\}_{n=1}^{N}$.
6. Evaluate $\mathscr{A}_{\text{NNMC}} := \dfrac{1}{N}\displaystyle\sum_{n=1}^{N} u_{h,NN}(T, Y^{(n)})$.

4.1 NN Training Cost

As we see from the training algorithms in chapter 2, the training happens in three steps:

Forward propagation:

- Propagate the input to the first layer with $a^1 = w^1 x$ takes $n^1 n^0$ operations where n^0 is the number of neurons in the input layer and n^1 is the number of neurons in the first hidden layer.
- For each of the hidden layer that goes from $(l-1)$ to l, compute the output of layer l, $a^l = \sigma(z^l)$ and the input of layer l, $z^l = w^l a^{l-1} + b^l$. Computing z^l is the cost of a matrix-vector product and a vector addition. That is, the order of cost to compute z^l is $n^{l-1} n^l$.
- Similarly at the output layer L, compute $a^L = \sigma(z^L)$ and $z^L = w^L a^{L-1} + b^L$. This has the order of cost, $n^L n^{L-1}$.

Therefore, the order of the cost of one forward propagation is $\sum_{l=1}^{L} n^{l-1} n^l$.

Backpropagation:

Consider the MSE cost function, $C = \frac{1}{2}(a^L - f)^2$, where f is the target vector. For backpropagation we must differentiate the cost function with respect to the weights and biases. To do this we must first find the error vectors. For the output layer, the error vector is computed as

$$\epsilon^L = (a^L - f) \odot \sigma'(z^L),$$

and for layers $1 \leq l \leq L$ the error vector is

$$\epsilon^{l-1} = (\epsilon^l)^T w^l.$$

Then the gradients are calculated as

$$\frac{\partial C}{\partial w^l} = (\epsilon^l)^T a^{l-1}, \quad \frac{\partial C}{\partial b^l} = \epsilon^l.$$

To calculate ϵ^L, we perform a vector subtraction and one element-wise vector multiplication which is $\mathcal{O}(n^L)$ operations. Then calculating ϵ^{l-1} from ϵ^l we perform a matrix-vector multiplication which costs $\mathcal{O}(n^l n^{l-1})$.

Finally, calculating $\frac{\partial C}{\partial w^l}$ for each layer requires a vector outer product which leads to a total cost of $\mathcal{O}\Big(\sum_{j\geq l} n^j n^{j-1}\Big)$.

Parameter Updates:

To update the weights and biases we must do

$$w^l \leftarrow w^l - \eta \frac{\partial C}{\partial w^l},$$
$$b^l \leftarrow b^l - \eta \frac{\partial C}{\partial b^l}.$$

The computation of new weight is component-wise sum of $\mathcal{O}(n^l n^{l-1}) + \mathcal{O}\Big(\sum_{j\geq l} n^j n^{j-1}\Big) = \mathcal{O}\Big(\sum_{j\geq l} n^j n^{j-1}\Big)$. The computation of bias is dominated by the time complexity of weights.

Hence, the complexity of one backpropagation is $\mathcal{O}\Big(\sum_{l=1}^{L} n^l n^{l-1}\Big)$ which is equal to the number of weights in the network. Therefore we see backpropagation and forward propagation are both linear in the number of parameters in the network. The computational cost of one forward and backpropagation is

$$\mathcal{O}\Big(\sum_{l=1}^{L} n^l n^{l-1}\Big) + \mathcal{O}\Big(\sum_{l=1}^{L} n^l n^{l-1}\Big) = \mathcal{O}\Big(\sum_{l=1}^{L} n^l n^{l-1}\Big).$$

If we have M training examples and e epochs then the cost is, $\mathcal{O}\Big(Me\sum_{l=1}^{L} n^l n^{l-1}\Big)$.

4.2 MC complexity analysis using an NN ODE solver

For this hybrid algorithm we perform step 2 of the MC algorithm using a neural network and leave the remaining steps unchanged. The total cost can be broken up into three parts:

- The cost to generate M training data using RK4 method is $\mathcal{T}_1 = \mathcal{O}(Mh^{-1}), M \ll N$.
- The cost to train a NN is $\mathcal{T}_2 = \mathcal{O}\Big(Me\sum_{l=1}^{L} n^l n^{l-1}\Big)$.

  Here, M, e and n^l are the number of training examples, epochs and number of neurons in layer l respectively.

- The cost to predict $\left\{u_{h,NN}(T, Y^{(n)})\right\}_{n=1}^{N}$ using the trained NN is $\mathcal{T}_3 = \mathcal{O}\big(N \sum_{l=1}^{L} n^l n^{l-1}\big)$.

Combining these, the total computational cost of algorithm 8 is,

$$W_{NNMC} = \mathcal{T}_1 + \mathcal{T}_2 + \mathcal{T}_3 \tag{4.1}$$

where, $\mathcal{T}_1$ is small since $M << N$. The training time $\mathcal{T}_2$ could be large depending on the NN architecture. Furthermore, since $\mathcal{T}_1$ and $\mathcal{T}_2$ are a one time cost, once the model is trained it can be saved and reused. The prediction cost, $\mathcal{T}_3$ is the cost of N evaluations of the network, which scales linearly with N. In the next chapter we will present some numerical examples to see how the cost of the hybrid NNMC method behaves in practice compared to the cost of the classical RK4MC sampling method.

Chapter 5

Numerical Examples

In this chapter, we present numerical experiment on three ODEs of type 3.1. Our goal is to predict the expected value of the solutions of the ODE using RK4MC and NNMC algorithms. For each ODE we compare the relative error of the expected values, rate of convergence and run-time of both algorithms for three different step sizes h.

NN Data Preparation For each step size h we generate a dataset $\left\{\left(Y^{(m)}, u_h(T, Y^{(m)})\right)\right\}_{m=1}^{M}$ of M training examples using the RK4 method, where $\{Y^{(m)}\}$ is the NN input and $u_h(T, Y^{(m)})$ is the corresponding target output. Then we split the dataset of M examples into three parts, we use 70% for training, 20% for validation and 10% for testing respectively. However, we update the parameters and select hyperparameters by observing the model performance on validation data and finalize the model after checking the performance on test data. We follow the same procedure for all ODEs.

NN development choice: For each ODE we use a different NN architecture. However for each h we use the same architecture but train different models for the NNMC algorithm. The final architectures and models were determined heuristically and by trial and error. This is typical in developing a neural network. In this thesis we have compared three different size of models to select a final model with best performance on the validation and test data.

Error Function: For all examples we will use the following error definition. Denote $\mathbb{E}$ as the expected value of the exact solutions, we define the error function of the NNMC and RK4MC algorithms as

$$\epsilon_{RK4MC} = \frac{|\mathbb{E} - \mathscr{A}_{RK4MC}|}{|\mathbb{E}|}$$

and

$$\epsilon_{NNMC} = \frac{|\mathbb{E} - \mathscr{A}_{NNMC}|}{|\mathbb{E}|}.$$

5.1 Example 1

ODE: The first ODE we consider is,

$$\frac{du(t,Y)}{dt} + \frac{1}{2}u(t,Y) = 5(Y + \sin(2t)), \qquad 0 \le t \le T,\ Y \sim U[1,2], \tag{5.1}$$
$$u(0,Y) = 0.$$

where $T = 20$ and Y is a scalar uniform random variable ranging from 0 to 1. The exact solution at any t is,

$$u(t,Y) = 10Y(1 - e^{-\frac{t}{2}}) + \frac{20}{17}(0.5\sin(2t) - 2\cos(2t) + 2e^{-\frac{t}{2}}).$$

For $t = T$, the expected value of the exact solution is $\mathbb{E} = \int_1^2 u(T,Y)dY$. We wish to estimate the expected value of $u(T,Y)$ for various maximum relative error of $\epsilon_{tol} = 10^{-2}, 10^{-3}, 10^{-4}$ using both NNMC and RK4MC. Using the theory developed in section 3.2.3, the choices of step size and sample count which ensure this tolerance for the RK4MC algorithm are $h = 0.05, 0.025, 0.0125$ and $N = 7 \times 10^2, 5 \times 10^4, 5 \times 10^6$ respectively. Figure 5.1 shows the training dataset generated by RK4 method.

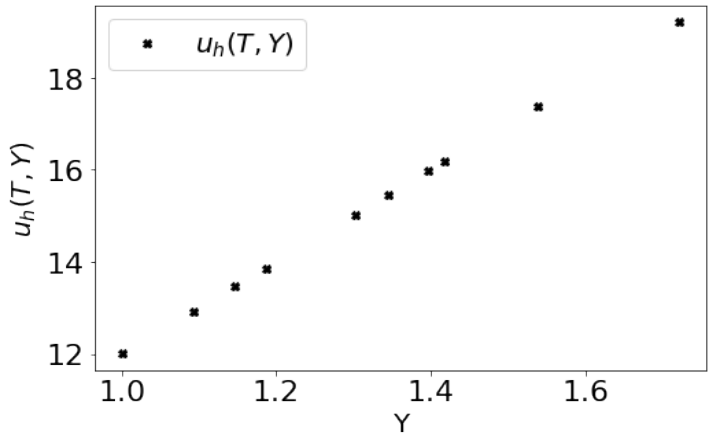


FIGURE 5.1: Training data generated by RK4 with step size, $h = 0.0125$.

NN Architecture: For each step size,h we will generate a training dataset using the RK4 method and train a corresponding NN model. We found that a small number of examples, $M = 10$ was sufficient to train the NN. This is to be expected since the solution u is linear in Y.

All three models for this ODE share the same architecture and same number of training examples, M. Each model has three layers, an input, output and a single hidden layer. Both input and output layers have one neuron each and the hidden

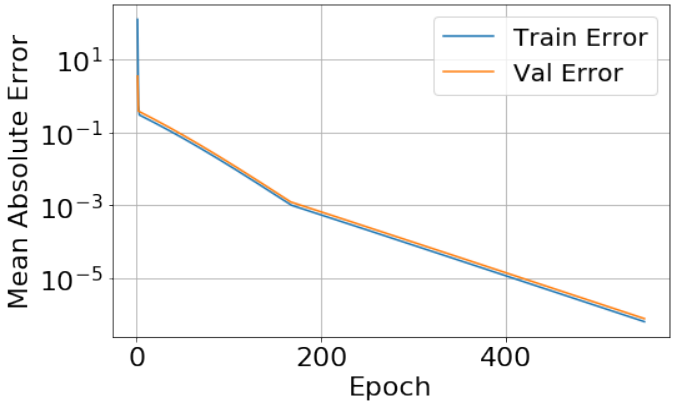


FIGURE 5.2: Semi-log plot of MSE loss with respect to epochs. The error plot is for the dataset with $h = 0.0125$ and $N = 5 \times 10^6$

layer has 10 neurons. There is no activation function for the input and output layer. For the hidden layer we used the Leaky-ReLU activation function. We used the **mean squared error** loss function and **stochastic gradient descent with momentum** as the optimizer. Figure 5.2 shows the training and validation loss corresponding to the number of epochs. In this figure, we see both the validation and training loss are decreasing, which is a sign that there is no overfitting.

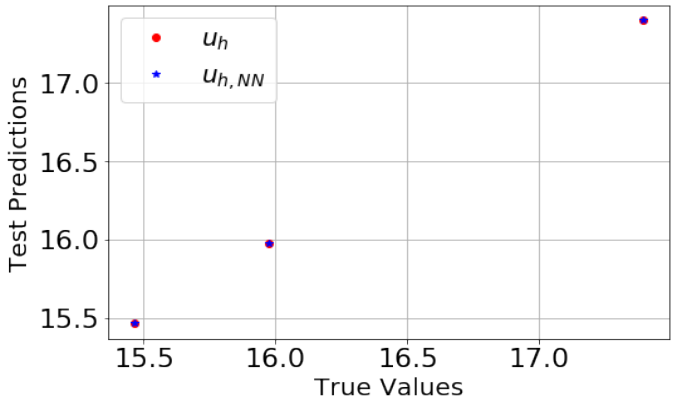


FIGURE 5.3: NN predicted solution verses the target solution on the test data for $h = 0.0125$.

Furthermore, figure 5.3 shows the true target solution against the NN predicted solution. We see that the model generates accurate predictions on the test data and completes the training process.

The next results will be generated by the performance of the large number of samples with the saved NN models. The right graph in figure 5.4, shows the NNMC method inherits the order $\mathcal{O}(h^4)$ from the RK4 method.

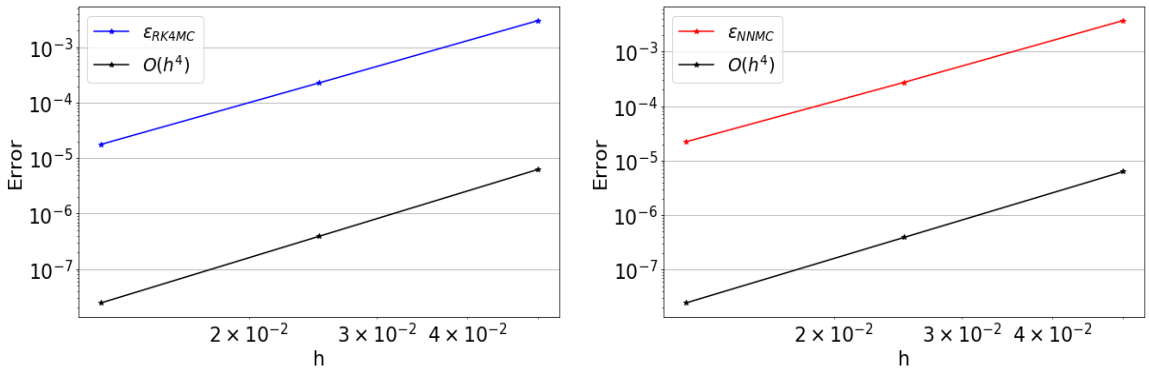


FIGURE 5.4: Convergence rate of the algorithm RK4MC [left] and the algorithm NNMC [right].

We will now compare the computation time of the NNMC and RK4MC algorithms. All the computations are done on CPU. Figure 5.5 (left) shows that NNMC prediction time, that is excluding the training time, is much less than the prediction time needed for the RK4MC method. The black line is proportional to the theoretical cost of RK4MC method.

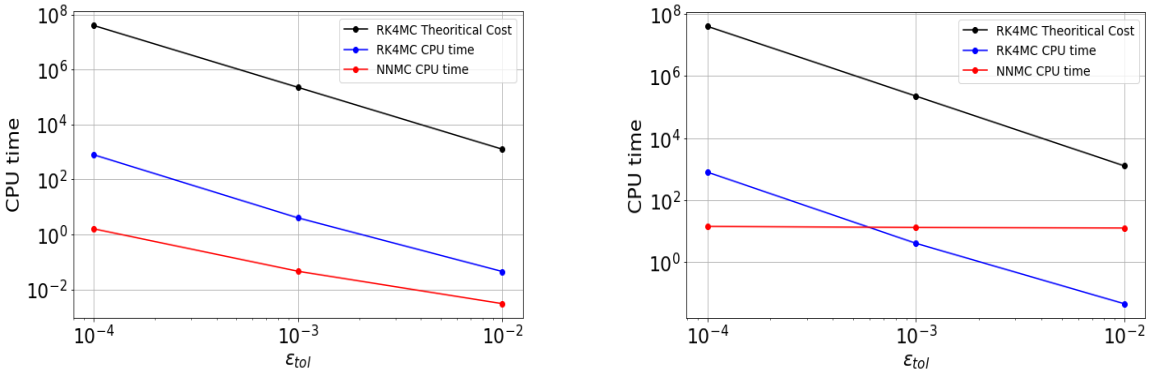


FIGURE 5.5: CPU time excluding training time[left] and including training time [right] of the algorithms RK4MC and NNMC.

Even including the costly NNMC training step, for smaller tolerances we see that the NNMC method takes less time than the RK4MC method in the right graph of figure 5.5. For larger tolerances the hybrid NNMC method is relatively costly but as the tolerance decreases there is almost no change in the NNMC method computation time unlike the RK4MC method.

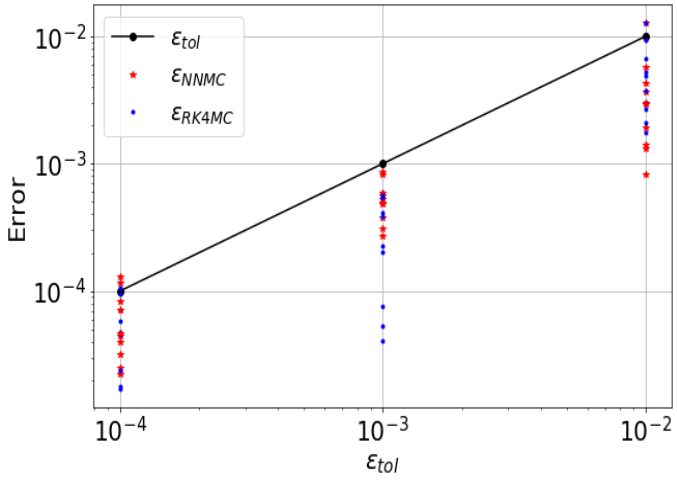


FIGURE 5.6: The RK4MC and NNMC errors in 10 different run

Figure 5.6 represents the relative error of the expected values predicted by both RK4MC and NNMC method over 10 different runs. We can see that the error goes above the tolerance level for some runs which is due to the statistical error.

5.2 Example 2

The second ODE we consider is,

$$\frac{du(t,Y)}{dt} + \frac{1}{2}u(t,Y) = 1 + \frac{y^2}{2} + y\cos(tY) + \frac{\sin(tY)}{2} - \frac{\sin(4t)}{Y} - \frac{\sin^2(2t)}{4Y},$$
$$u(0,Y) = 2 + Y^2,$$

where $0 \leq t \leq T$, $Y \sim U[1,2]$ and $T = 20$. The exact solution is,

$$u(t,Y) = 2 + Y^2 + \sin(tY) - \frac{\sin^2(2t)}{2Y}.$$

For $T = t$, the expected value of the exact solution is, $\mathbb{E} = \int_1^2 u(T, Y)dY$.
The setup for this example is the same as example 1. We choose the error tolerance, $\epsilon_{tol} = 10^{-1}, 10^{-2}, 10^{-3}$ and we find corresponding sufficient choices of $h = 0.1, 0.05, 0.025$ and $N = 5 \times 10^2, 6 \times 10^4, 5 \times 10^6$ following the same procedure as in example 5.1. Figure 5.7 shows the RK4 solutions of the ODE at the final time T for a set of training input Y.

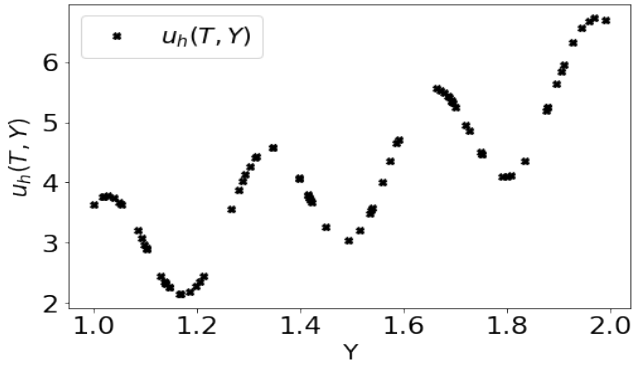


FIGURE 5.7: Training data generated by RK4 with step size $h = 0.025$.

NN Architecture: For each h we train a NN with $M = 80$ training examples. Each model has same architecture with 6 layers, one input, four hidden layers and one output. There is one input and one output neuron and each hidden layer has 32 neurons. The input and output layer has no activation function and each hidden layer uses the hyperbolic tangent activation function. We use the **Adam** optimizer with initial learning rate 0.03, the **mean absolute error** (MAE) loss function and **early stopping** to prevent overfitting.

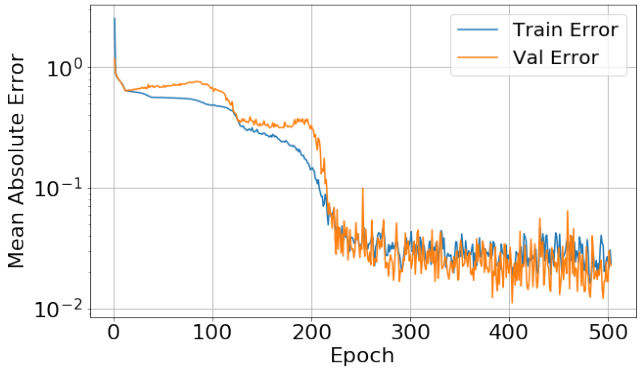


FIGURE 5.8: MAE loss function with respect to epochs. The error plot is for the dataset with $h = 0.025$ and $N = 5 \times 10^6$.

Figure 5.8 shows the model performance on training and validation data. In this figure, the decreasing validation and training error indicate that the model is not overfitting.

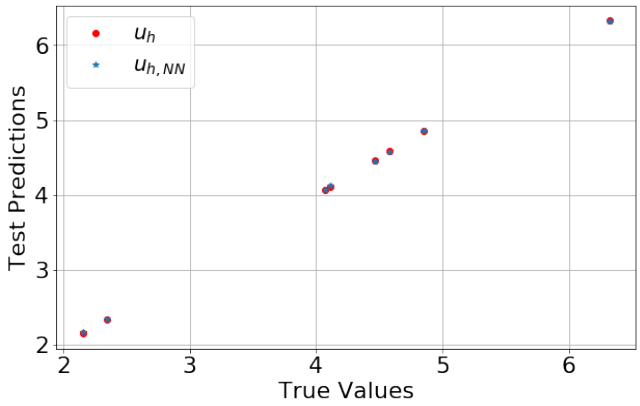


FIGURE 5.9: NN predicted solution verses the target solution on the test data for $h = 0.025$.

In figure 5.9 we see the NN predicted value verses the target values, Where u_h are the target values and $u_{h,\mathrm{NN}}$ are the corresponding model predictions. Since the model is not overfitting and has good performance on unseen test data, we can say this is a well trained model and this completes the training process.

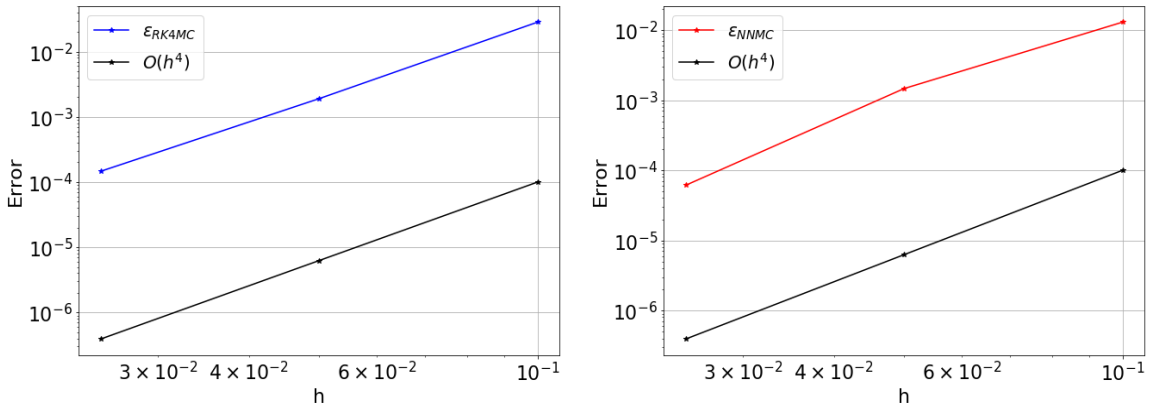


FIGURE 5.10: Convergence rate of the RK4MC [left] and NNMC [right] method.

Now we compare the relative error and performance of both algorithms for a large sample size, N. Similar to the previous example 5.1 we can see from figure 5.10 that the NNMC method inherits the order $\mathcal{O}(h^4)$ from the the RK4 method.

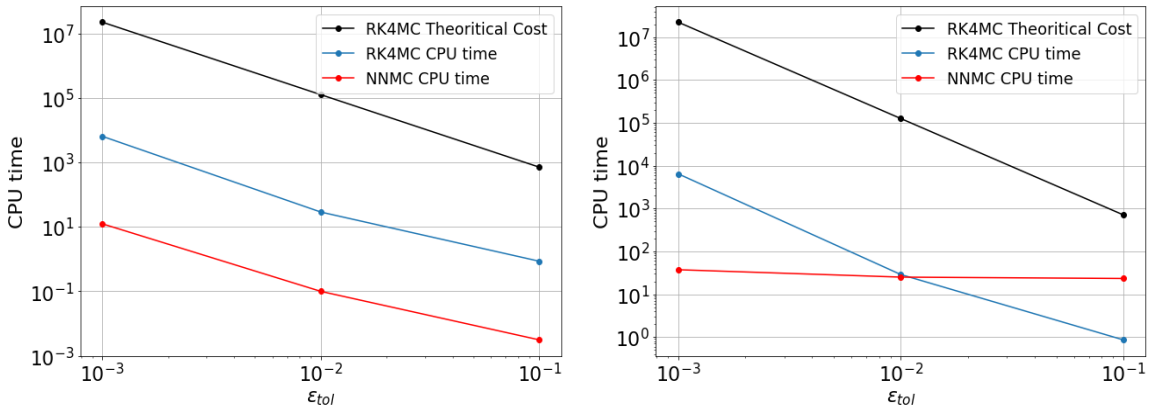


FIGURE 5.11: CPU time of RK4MC and NNMC method without training time [left] and with training time [right]. The black line is proportional to the theoretical cost of RK4MC method. All the computation is done in the CPU.

As in the first example, the left graph of figure 5.11 shows that NNMC prediction cost is again lower than the RK4MC method prediction cost. Furthermore, the right graph shows the training cost is almost independent of the error tolerance and for smaller error the hybrid NNMC method is faster than the classical RK4MC method.

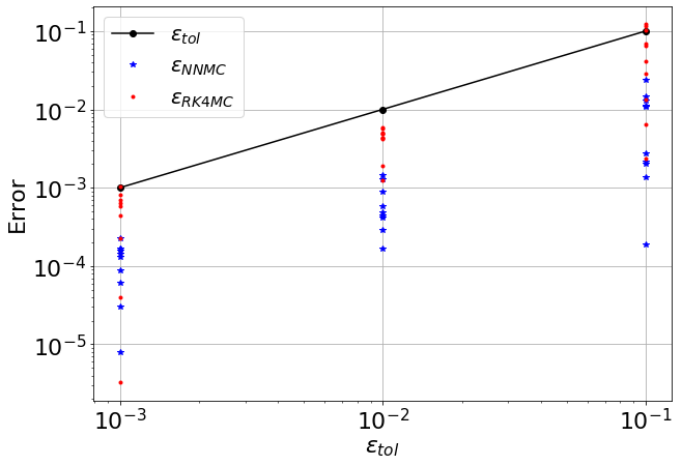


FIGURE 5.12: RK4MC error and NNMC error in 10 different runs.

Finally, figure 5.12 shows relative error for both methods in 10 different runs. The errors above the tolerance level for some runs is due to the statistical failure.

5.3 Example 3

ODE: The third and final ODE we consider is,

$$\frac{du(t,Y)}{dt} + \frac{1}{2}u(t,Y) = \left(\sum_{i=1}^{10} \frac{\frac{1}{16}y_i \cos(ty_i)}{1 + \frac{1}{16}\sin(ty_i) + 0.25y_i^2} + \frac{1}{2}\right) \prod_{i=1}^{10} \left(1 + \frac{1}{16}\sin(ty_i) + 0.25y_i^2\right),$$
$$u(0,Y) = \prod_{i=1}^{10} \left(1 + 0.25y_i^2\right),$$

where $0 \leq t \leq T$, $T = 20$, and $Y = (y_1, y_2, \dots y_{10})^T$ with $y_i \sim U[0,2]$ for $i = 1, 2, \dots, 10$ corresponds to the 10 input neurons. The exact solution is,

$$u(t,Y) = \prod_{i=1}^{10} \left(1 + \frac{1}{16}\sin(ty_i) + 0.25y_i^2\right).$$

At the final time T, the expected value of the exact solution of the ODE is,

$$\mathbb{E} = \frac{1}{2^{10}} \left(\prod_{i=1}^{10} \int_0^2 u_i(T, y_i) dy_i\right).$$

For this ODE we consider the error tolerance, $\epsilon_{tol} = 10^{-1}, 10^{-2}, 10^{-3}$. In the same way as the last two problems we choose $h = 0.5, 0.25, 0.125$ and $N = 3 \times 10^2, 5 \times 10^3, 10^6$. The training dataset has ten input variables $y_1, y_2, \dots y_{10}$ and one output variable u. Figure 5.13 represents the joint distribution of three columns y_1, y_{10} and u of training data, which gives some intuition of the training dataset. The histograms on the diagonal shows the probability distribution of a single variable while the scatter plots on the upper and lower triangles show the relationship between two valuables. For example, on the third row the right most plot shows the probability distribution of the solution u and the left most plot shows the scatter plot of the solution u with respect to the input variable y_1.

NN Architecture: Similar to the previous examples, for each h we will develop a NN model with $M = 7000$ training examples. The models with M training examples generated with step sizes $h = 0.5$ and $h = 0.125$ has exactly the same NN architecture: one input layer, four hidden layers and one output layer. The architecture was primarily determined by trial and error. Table 5.1 gives detailed information about the layers. Additionally, we use the Adam optimizer with initial learning rate $\eta = 0.005$ and the mean absolute error (MAE) loss function. We initiate the training with 200 epochs and use early stopping to prevent over fitting. The model with training data

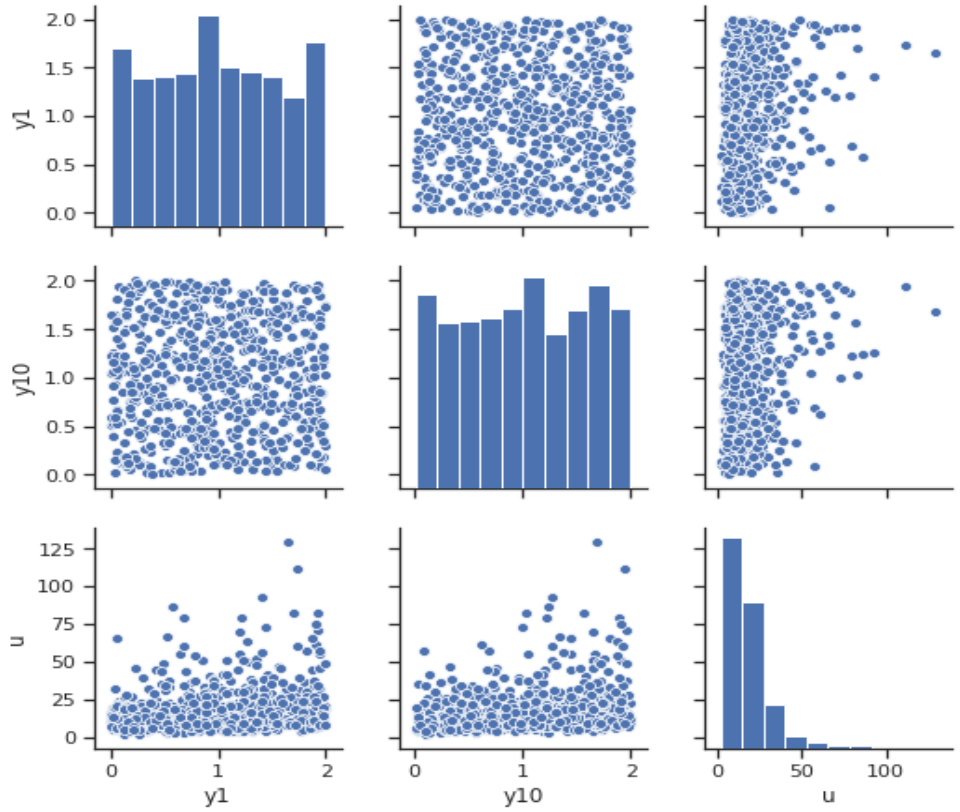


FIGURE 5.13: Joint distribution of few columns of the training data generated by RK4 method with step size, $h = 0.125$.

generated with step size $h = 0.25$ gives the desired performance with Leaky-ReLU activation function. The rest of the choice of hyperparameters of NN architecture is the same as 5.1.

M	Layers	Neurons	Activation function	Loss function	Optimizer
7000	Input Layer	10	None	MAE	Adam
	Hidden Layer 1	44	ReLU		
	Hidden Layer 2	44	ReLU		
	Hidden Layer 3	44	ReLU		
	Hidden Layer 4	44	ReLU		
	Hidden Layer 5	44	ReLU		
	Output Layer	1	None		

TABLE 5.1: NN architecture

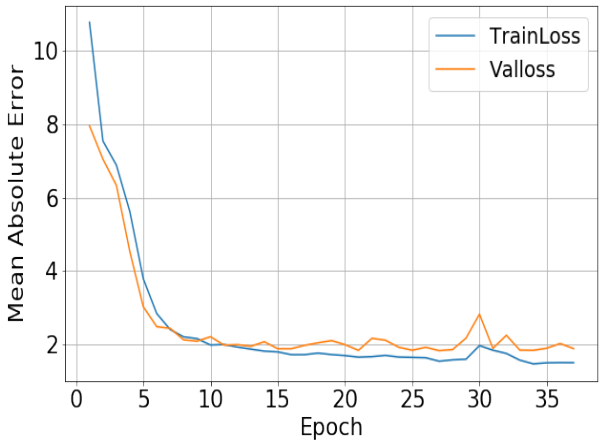


FIGURE 5.14: MAE error/loss function with respect to epochs. The error plot is for the dataset with $h = 0.125$ and $N = 10^6$.

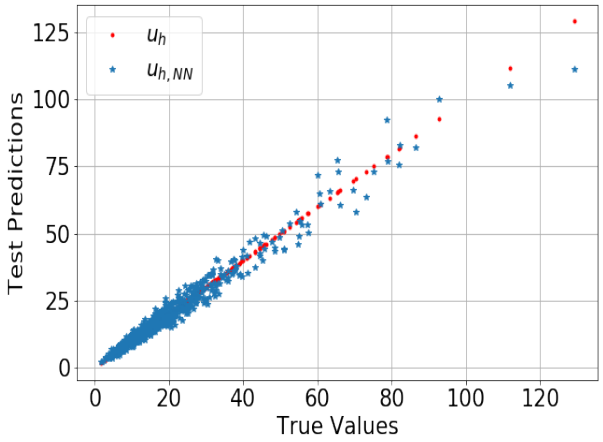


FIGURE 5.15: NN predicted solution verses the target solution on the test data for $h = 0.125$.

Figure 5.14 shows the MAE loss function over train and validation data and figure 5.15 shows the model performance on the test data. Where u_h are the target values and $u_{h,\mathrm{NN}}$ are the corresponding model predictions.

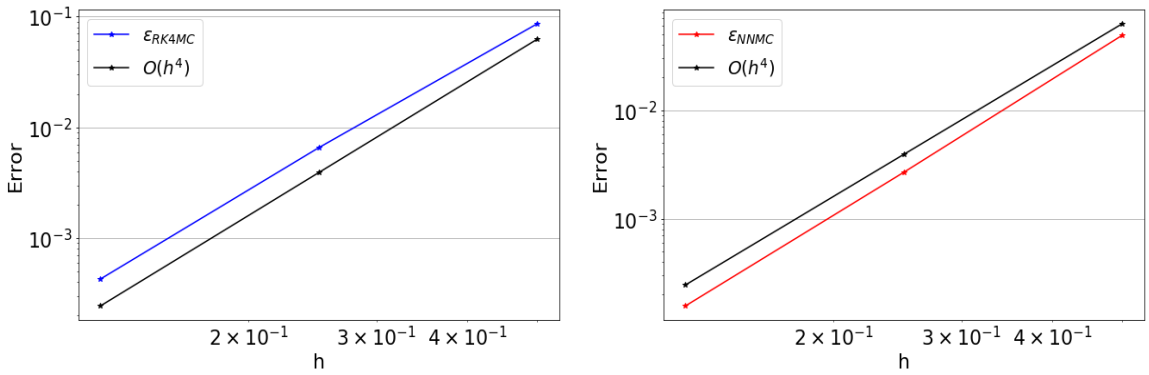


FIGURE 5.16: Convergence rate of the RK4MC [left] and NNMC [right] method.

Similar to the previous two examples, for this example we develop a NN model to obtain relative error on the same order as RK4MC for all h. We see in figure 5.16 that NNMC inherits the $\mathcal{O}(h^4)$ from RK4 method.

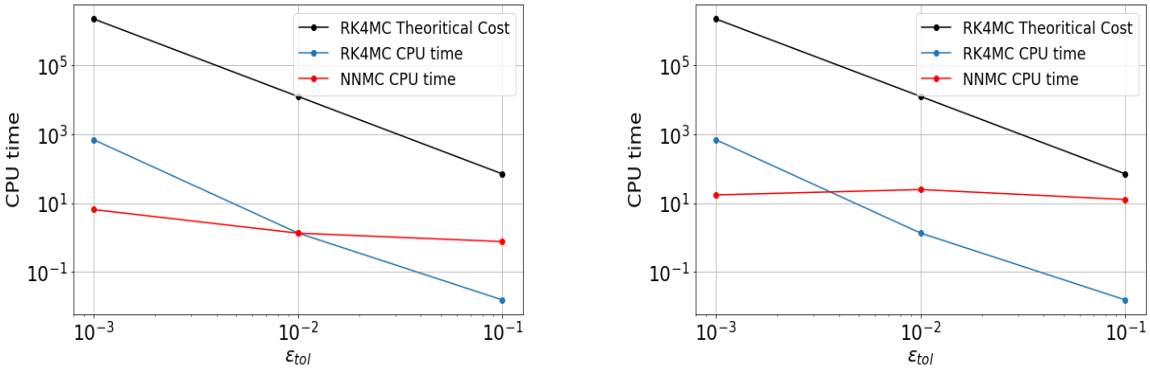


FIGURE 5.17: CPU time of RK4 and NN with training time [left] and without train time [right].Theoretical cost is proportional to the cost of the algorithm RK4MC. The coputation was done in the CPU.

Figure 5.17 shows the computational cost of both methods. Again, we observe that the prediction cost required by NNMC is smaller than the the RK4MC method for larger sample sizes. From the graph on the right we see that the training time is comparatively higher for this ODE. Furthermore, we notice that the training time is almost independent of the tolerance. Hence, if we use smaller ϵ_{tol}, then RK4MC has higher cost than the NNMC method. Due to computational and time limitations we had to set our error tolerance relatively larger than the previous two problems.

Figure 5.18 shows the relative error for both methods over 10 different runs. We see that the relative error of both methods is below the desired tolerance level for more than 95% of the runs. The remaining 5% are above the tolerance level due to statistical error.

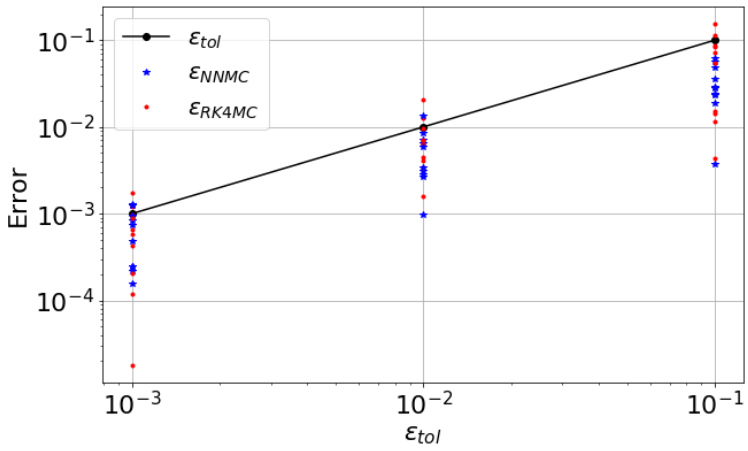


FIGURE 5.18: The RK4MC error and NNMC errors in 10 different runs.

Overall, We find that with a good NN architecture it is possible to generate the approximate solutions of ODEs for a large number of samples. In addition, we observe that the NN prediction time is significantly faster than the classical RK4 method. Furthermore, the NN training time is a one time cost and is independent of step size and error tolerance. This can be especially beneficial, when solving complex ODEs or PDEs for which slopes are expensive to compute or large final times are required. This will in turn enable the computation of the statistics of QoIs by the NNMC sampling much more efficiently compared to when we use the classical MC sampling, particularly when high accuracy within small tolerances are desired.

Chapter 6

Conclusion

In this thesis we have investigated the applicability of using NNs to find the approximate solutions of ODEs for use in MC integration. We observe though several numerical examples that a properly trained NN can produce solutions with the same accuracy as the RK4 method. Furthermore, the training and evaluation of a model does not depend on the step size. Whereas, when using the RK4 method with a very small step size to compute solutions for a large number of samples can be extremely time consuming. Therefore the MC algorithm can be significantly accelerated if a NN can be found which is cheap to evaluate and requires few training examples.

Our work involves simple ODEs with a short time interval ($T = 20$). A further study can be done with more complex ODEs and PDEs problems and using more advanced deterministic solvers. It can be extended to other UQ technique such as stochastic collocation.

Bibliography

[Cun+14] Americo Cunha Jr et al. "Uncertainty quantification through the Monte Carlo method in a cloud computing setting". In: *Computer Physics Communications* 185.5 (2014), pp. 1355–1363.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[Gér18] Aurélien Géron. *Neural networks and deep learning*. O'Reilly Media, Inc, Mar. 2018. ISBN: 9781492037354. URL: https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch01.html.

[Has95] Mohamad H. Hassoun. *Funcdamentals of Artificia Neural Networks*. Massachusetts Institute of Technology, Dec. 1995. ISBN: 978-0-262-51467-5.

[KB14] Diederik P. Kingma and Jimmy Lei Ba. *Adam: A method for Stochastic Optimization*. Dec. 2014. URL: https://arxiv.org/abs/1412.6980.

[MHZ03] Lionel Mathelin, M Yousuff Hussaini, and Thomas A Zang. "A stochastic collocation algorithm for uncertainty analysis". In: (2003).

[Mot18] Mohamed Motamed. *Lecture notes on Monte Carlo Sampling*. Fall 2018.

[MU49] Nicholas Metropolis and Stanislaw Ulam. "The monte carlo method". In: *Journal of the American statistical association* 44.247 (1949), pp. 335–341.

[Ng17] Andrew Ng. *Lecture notes on Optimization Algorithms*. Fall 2017.

[Nie15] Michael A. Nielsen. *Neural Netwoks and Deep Learning*. Determinaton Press, 2015. URL: http://neuralnetworksanddeeplearning.com/about.html.

[Sul15] Timothy John Sullivan. *Introduction to Uncertainty Quantification*. Springer International Publishing, Switzerland, Dec. 2015, pp. 1–10. ISBN: 978-3-319-23395-6. DOI: 10.1007/978-3-319-23395-6.