Imperial College London

Department of Earth Science and Engineering

MSc Applied Computational Science and Engineering

Independent Research Project

Bayesian Machine Learning for Quantifying Uncertainty in Surrogate Modelling

by

Archie Luxton

Email: archie.luxton21@imperial.ac.uk
GitHub username: asce-aol21
Repository: https://github.com/ese-msc-2021/irp-aol21

In collaboration with Arup
8 Fitzroy Street, London, United Kingdom, W1T 4BJ

Supervisors:
Dr. Ramaseshan Kannan (Arup)
Dr. Gerard Gorman (Imperial College London)

2nd September 2022

Declaration

This report describes work carried out between June 2022 and September 2022 in the Department of Earth Science and Engineering at Imperial College London under the supervision of Ramaseshan Kannan.

This report is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and acknowledgements. This report has not been submitted in whole or in part for any other degree or diploma at this or any other university. This report does not exceed the word limit of 5,000 words or the limit of 10 Figures and Tables.

Archie Luxton

London, September 2022

Abstract

Increasingly, innovation in engineering is being lead by fast, data-driven decisions. Such fast decisions are needed for rapid design iteration, which is used for early stage design exploration in structural engineering. At present, solutions for rapid design iteration are basic and lack any ability to communicate uncertainty to engineers, which is crucial in mission-critical applications such as structural engineering. Applications like this are prime candidates for uncertainty-aware surrogate models: these are lightweight models that can approximate an expensive simulation or calculation wherever speed of simulation is prioritised over accuracy, and provide a confidence interval associated with each prediction.

This project develops Neural Network (NN)-based surrogate models that can approximate *Finite Element* simulations to determine the modal frequency of a 3D structure based on its input geometry. These surrogates have been designed using the principles of Bayesian Neural Network (BNN)s, using the *MC Dropout* and *SGLD* methods to stochastically train Neural Networks and then use them to perform inference - producing point estimates and confidence intervals for every prediction made.

The surrogates built are shown to be fast to train, taking approximately 70 seconds each, and can carry out inference very quickly. Predictions made with the developed surrogates are between 500-$11,000\times$ faster than the equivalent calculations carried out by industry-leading *Oasys GSA*. The predictions are also accurate, with RMS errors of 0.008, 0.007 and 2.034 for the baseline training, validation and test sets for *MC Dropout*, and 0.01, 0.01 and 1.285 respectively for the re-trained versions using a larger training domain.

To improve the usefulness of the surrogates, a custom interface has been designed and implemented in collaboration with Arup which allows an engineer to parametrically design a structure using sliders. The application then instantly draws the geometry, returns the associated modal frequency and colours the structure according to the uncertainty in the prediction. This allows the engineer to iterate countless designs orders of magnitude faster than using existing workflows.

The accuracy of point predictions is shown to increase significantly when re-training the system with additional data, improving the RMS error of predictions by approximately 37%. By using the visualisation tools that are developed and supplied in `duq`, the user can easily see which areas of the domain have the highest uncertainty, and accordingly produce more training data in that region to reduce the uncertainty and increase accuracy.

Acknowledgements

I would like to thank my supervisor Ramaseshan Kannan, for his support and guidance over the course of this project, and Gihan Weerasinghe for his value feedback and advice. I'd also like to thank the teaching staff of the MSc Applied Computational Science course at Imperial College London, for the hard work and passion they put into delivering a fantastic course despite the difficulties of remote learning.

Contents

Acronyms

BNN Bayesian Neural Network. ii, iv, 2–5, 14, 20, 23

FEA Finite Element Analysis. iv, 1

HPC High Performance Computing. iv

KL Kullback-Leibler. iv

ML machine learning. iv

NN Neural Network. ii, iv, 1–3, 5, 6, 9, 20

OOD out of distribution. iv, 1, 3, 12, 23

PCA Principal Component Analysis. iv, 18

RST reStructuredText. iv, 6

SGD Stochastic Gradient Descent. iv, 3

SRT stochastic regularisation technique. iv, 3

UQ uncertainty quantification. iv, 1, 2, 8, 9, 14, 20, 23

1 Introduction

Gone are the days of drawing boards and slide rules: engineering is becoming increasingly reliant on fast, data-driven decisions. To innovate, fast and informed decisions are necessary throughout the whole pipeline of a project; from early-stage design exploration through to the monitoring and maintenance of assets.

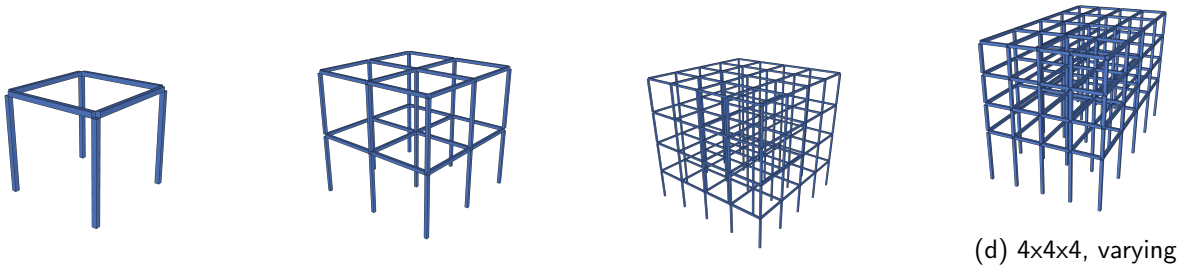
One process that requires these kind of fast decisions is rapid design iteration. However for structural engineering that requires Finite Element Analysis (FEA) simulations, iterating designs can become very expensive. *Surrogate models* are a potential solution to this. Surrogates are lightweight models that can approximate an expensive simulation or calculation, and are particularly useful when speed is prioritised over accuracy. Once an optimum design has been achieved using a surrogate, a full FEA simulation can be carried out to achieve the required accuracy of a final design. Neural Network (NN)s are a promising technology to use to construct surrogate models, as they can mimic complex, non-linear behaviour and can scale well to large amounts of high-dimensional data.

Another limitation to performing traditional simulations for rapid design iteration is the portability of legacy code and software: industry leading technology such as *Oasys GSA* [21] are compatible with Windows OS, which restricts their use on High Performance Computing (HPC) clusters or similar compute resources. Also, these traditional methods (e.g. FEA) scale in computational complexity with the number of nodes of geometry, whereas the computational complexity of a NN-based surrogate is agnostic of model size.

At present, solutions for rapid prototyping in this field are very basic, mostly depending on naive point sampling and interpolation methods. Moreover, and importantly, engineers have no indication about the uncertainty of the output from such methods, which is particularly crucial for mission critical applications in the field of structural engineering. Being able to communicate the limits of confidence, and particularly to be able to say "I don't know" when performing a prediction on out of distribution (OOD) data, should therefore form a key part of any solution to the rapid prototyping problem in this field.

In this project, we will explore a simple rapid-prototyping scenario: one of determining the modal frequencies of 3D structures, given the following information about their geometry: the number of bays in the x, y and z direction and the length of members in x, y and z (see Figure 1). These calculations can take anywhere between 19.4 to 4 seconds to compute using *Oasys GSA* [21]. Our goal is to solve this rapid prototyping problem by using a NN-based surrogate model to approximate the FE solution and then to quantify the uncertainty of this output. The Python package `duq` has been developed for this project. It is simple to use and can be applied to a range of surrogates using feed-forward networks to perform predictions and uncertainty quantification (UQ).

Though there will be some discussion about the design of network architecture and uncertainty quantification methods, complex and specialised methods are beyond the scope of this paper. Nor will the mathematical intricacies of each method be detailed in full; proofs and derivations can be found in the original authors' papers.



(a) 1x1x1 (b) 2x2x2 (c) 4x4x4 (d) 4x4x4, varying bay width

Figure 1: Examples of structure geometry to be explored

1.1 Applying Uncertainty to Neural Networks

Traditional NNs are incapable of giving an estimate of uncertainty with a prediction [1]. Applying Bayesian techniques to NNs is a common method to introduce UQ, with some popular methods being *MC (Monte-Carlo) Dropout* [10], *Stochastic Gradient Langevin Dynamics (SGLD)* [30] and *Bayes by Backprop (BBP)* [6].

The Bayesian paradigm centres around updating one's belief (the posterior) based on some evidence and prior knowledge (the prior) [7]. At the core of Bayesian inference is Bayes' Theorem which defines the posterior, which is the probability of an event occurring based on some prior knowledge. Different Bayesian Neural Network (BNN) methods approach the problem differently, though all first train the network (i.e. learn the model parameters), and then perform *inference* on an unseen point.

At training stage, the model parameters ($\boldsymbol{\theta}$) are learned based on the training dataset $\mathbf{X}$ and $\mathbf{Y}$. Applying Bayes' theorem [29]:

$$p(\boldsymbol{\theta}|\mathbf{X},\mathbf{Y}) = \frac{p(\mathbf{Y}|\mathbf{X},\boldsymbol{\theta})p(\boldsymbol{\theta})}{\int p(\mathbf{Y}|\mathbf{X},\boldsymbol{\theta})p(\boldsymbol{\theta})d\boldsymbol{\theta}} \tag{1}$$

Where $p(\boldsymbol{\theta}|\mathbf{X},\mathbf{Y})$ is the *posterior* distribution of model weights, $p(\mathbf{Y}|\mathbf{X},\boldsymbol{\theta})p(\boldsymbol{\theta})$ is the *likelihood* term, $p(\boldsymbol{\theta})$ is the *prior* and the numerator term is the *likelihood*, or *evidence*, term. Similarly once a model is trained, to determine the predictive distribution for $\mathbf{y}$ given an input $\mathbf{x}$, we apply Bayes' formula again [10]:

$$q(\mathbf{y}|\mathbf{x},\mathbf{X},\mathbf{Y}) = \int p(\mathbf{y}|\mathbf{x},\boldsymbol{\theta})p(\boldsymbol{\theta}|\mathbf{X},\mathbf{Y})d\boldsymbol{\theta} \tag{2}$$

The mean of this predictive distribution is then the most likely estimate for that target, and confidence intervals can be drawn out accordingly. Unfortunately, for a system with many parameters, the integrals in 1 and 2 are high-dimensional and generally intractable. They can only be solved analytically if the likelihood is *conjugate*[2] to the prior [27] [19]. Two primary methods of approximating these intractable integrals are documented: *Variational Inference (VI)*-based methods and *Monte-Carlo (MC)*-based methods.

In *MC*-based methods, the predictive distribution (Equation 2) can be approximated using sampling [10] [8]. Given T number of stochastic forward passes through a network, this predictive distribution is given by:

[1] Noting that the probability assigned to classifications from a softmax output layer shouldn't be confused with an uncertainty, as a model can be uncertain in its predictions even with a high softmax output [9].

[2] The likelihood is *conjugate* to the prior if it they are both in the same distribution family, e.g. if they are both Gaussian (normal). By extension, if the prior is conjugate to the likelihood, then the posterior is also in the same distribution family.

$$p(\mathbf{y}|\mathbf{x}) \approx \frac{1}{T}\sum_{t=1}^{T} p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}_t), \quad s.t. \quad \boldsymbol{\theta}_t \sim q(\boldsymbol{\theta}|\mathbf{X}, \mathbf{Y}). \tag{3}$$

In *Variational Inference (VI)*-based methods, the predictive distribution is approximated by minimising the *Kullback-Leibler (KL)* divergence, or maximising the Evidence Log Lower Bound (ELBO) between a new predictive distribution $q(\boldsymbol{\theta})$ (that has an easy to evaluate structure) and the final term in Equation 2 [8]:

$$q(\mathbf{y}|\mathbf{x}) = \int p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) q(\boldsymbol{\theta}) d\boldsymbol{\theta} \tag{4}$$

1.1.1 MC Dropout

MC Dropout is an *Monte-Carlo (MC)*-based method introduced by Gal and Ghahramani (2016) [10], and requires very little modification of existing training pipelines, making it very suitable for scalable surrogates. It applies a Bernoulli distribution mask to every neuron in the network apart from the output neurons, which gives each neuron a $p_{dropout}$ chance of having its output set to zero, where $p_{dropout}$ is the dropout probability applied to the model as a whole.

Dropout was originally outlined by Hinton et al., 2012 [14] as a stochastic regularisation technique (SRT), which reduces over-fitting and improves generalisation of neural networks. It remains a very popular method for regularisation, so is already implemented in many modern architectures. Typically the dropout mask is 'turned off' after training has completed, but MC Dropout performs multiple forward passes on a network with dropout still enabled to effectively sample the approximate posterior $q(\boldsymbol{\theta}|\mathbf{X}, \mathbf{Y})$ shown in Equation 3.

MC Dropout operates similarly to an ensemble method [9], whereby a large number of networks are trained independently, then the result of forward passes through all networks is averaged for a final result. Using ensembles of deterministic models is a bone-fide method for constructing BNNs, but it has a high memory footprint and the uncertainty it produces is inaccurate for OOD data [9] [22].

Gal and Ghahramani show that the predictive log-likelihood and RMSE of predictions made using MC Dropout are considerably improved over state-of-the-art methods at the time (Variational Inference [11] and Probabilistic Backpropegation [13], so it is a good candidate for easy to understand and efficient NN-based surrogate models.

1.1.2 Stochastic Gradient Langevin Dynamics (SGLD)

SGLD is method introduced by Welling et al., 2011 [30]. It combines the Stochastic Gradient Descent (SGD) optimisation method with Langevin Dynamics, an approach to modelling the dynamics of molecular systems (e.g. Brownian motion). By by adding the right amount of noise to a SGD algorithm and by annealing the learning rate, Welling et al. showed that the iterates converge to samples from the true posterior distribution. The Langevin dynamics portion of the method injects normally distributed noise (scaled according to the learning rate) into the weights of the network, resulting in the parameters converging to the full posterior instead of the maximum a posteriori mode [30].

In practice, the model's weights are saved at regular intervals after an initial burn-in period. The injection of Langevin noise adds stochastisity to the network, so each save state of the model forms an approximate predictive distribution.

If $\boldsymbol{\theta}_t$ are the model weights at time t, ϵ_t the learning rate at time t, N the total number of data points, n the number of datapoints in a subset or minibatch, x_{ti} the value of the ith value of x at

time t, η_t be normally distributed noise, $p(\boldsymbol{\theta_t})$ be a prior distribution and $p(x|\boldsymbol{\theta})$ be the probability of data item x given the model parameterised by $\boldsymbol{\theta}$, Welling et al. define the update step as:

$$\Delta\theta_t = \frac{\epsilon_t}{2}\left(\nabla logp(\theta_t) + \frac{N}{n}\sum_{i=1}^{n}\nabla logp(x_{ti}|\theta_t)\right) + \eta_t \tag{5a}$$

$$\eta_t \sim N(0, \epsilon_t) \tag{5b}$$

In practice [23] [24], this is usually implemented by adding a noise term into a custom SGD optimiser (e.g. *Adam* [18]), however in this project a method is constructed that can add the noise after any optimiser. If $\frac{\partial J(\theta_{t-1})}{\partial \boldsymbol{\theta}}$ is the gradient of the loss function at the previous iteration w.r.t. the model weights (calculated using backpropegation), then a simple SGLD update step becomes:

$$\Delta\theta_t = \epsilon_t \frac{1}{n}\sum_{i=1}^{n}\frac{\partial J(\theta_{t-1})}{\partial \boldsymbol{\theta}} + \eta_t \tag{6a}$$

$$\eta_t \sim N(0, \epsilon_t) \tag{6b}$$

The loss function used throughout this report is the Root Mean Squared Error (RMSE) loss, which is very commonly used in regression applications [9] [10] [13] [17]. If y_i is the true value of the ith datapoint (label) and $\hat{y}_i$ is the estimated value of the ith label, then *RMSE* is given by:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{N}} \tag{7}$$

Using SGLD, the learning rate ϵ should be annealed according to the Robbins-Monro [25] conditions, but in practice many authors do not do this [15], or they stop the annealing process once the learning rate becomes sufficiently small [2]. This annealing helps to reduce autocorrelation between samples, as shown by [30]. Warm restarts [28] can be used to help avoid a mode collapse problem, in which the wrong (or sub-optimum) parameterisations of $\boldsymbol{\theta}$ can fit the training set. Welling et al. used an annealing schedule based on $\epsilon_t = a(b+t)^{-\gamma}$, where $\gamma = 0.55$ and a and b are set such that ϵ_t decreases from 0.01 to 0.0001 over the duration of the run. The effect of this annealing schedule and learning rate is explored in the hyperparameter tuning (Section 2.4). Other versions of *SGLD* exist, such as Chunyuan Li et al. (2015)'s pre-conditioned SGLD (pSGLD) [20] and Ahn, Korattikara and Welling (2012)'s Stochastic Gradient Fisher Scoring (SGFS) [3] but these haven't been explored in depth, in an effort to keep the surrogate implementation as simple to implement as possible for engineers.

There are a range of Python packages available that can construct BNNs and perform inference, including PYMC [26], TensorFlow Probability [1], Pyro [5] and others online, however these aren't designed for surrogate models.

The focus of this project is to develop a framework, named `DeepUQ (duq)`, that can be applied to arbitrarily deep and wide feed-forward neural networks to make predictions, quantify the associated uncertainty and explain the process and results to engineers in a clear manner. The training and inference can be carried out using *MC Dropout* or *SGLD*.

2 Methodology

Over the course of this project, the Python package `DeepUQ (duq)` has been developed from scratch, leveraging existing packages and modules where required, such as PyTorch, NumPy, SciPy, Matplotlib, Seaborn, etc. The `.py` modules have been written using the *Spyder* IDE, using *Jupyter* Notebooks for running small chunks of code and plotting visualisations. The *Weights and Biases* API has been used to keep track of NN training and hyperparameter tuning, *Sphinx* has been used for automatic documentation generation, *Github* has been used for version control and automated testing and *GSA* and *GSAPy* have been used to generate the training and testing data.

2.1 Generating and Splitting Data

The main data corpus contains $1,978$ different structures that have been generated using *GSA* and *GSAPy*. Each configuration and the first 6 modal frequencies have been stored in `data/all_data.csv`. The input parameters for each structure are `num_x,y,z` (the number of bays in x, y, z), and `dim_x,y,z` (the length of elements in x, y, z). Each structure's inputs and 6 modal frequencies can be represented by a length 12 vector: [`num_x`, `num_y`, `num_z`, `dim_x`, `dim_y`, `dim_z`, `freq1`, `freq2`, `freq3`, `freq4`, `freq5`, `freq6`].

Due to stability issues in GSA, the main corpus is made up of many different subsets of generated data: most in realistic bounds but some in extreme bounds to test the performance of the BNNs. To keep the approach as generic and scalable as possible, data splitting routines have been developed to split this main corpus into subsets - see Appendix B and the Jupyter notebook `/notebooks/splitting_data.ipynb` for full details. Table 1 shows the domain of all the individual datasets that have been combined into the main corpus, and the domains of the datasets that have been split from this[3]. Since there is a very clear geometric meaning to the input parameters, the corpus has been split based given data domains: any inputs outside a given domain are split off into a test set, anything within that range becomes the training set (at 90%) and the validation set (at 10%).

Table 1: Domains of data subsets forming the full corpus.

Purpose	Dataset Name	Number of Points	num_x Range	num_y Range	num_z Range	dim_x Range	dim_y Range	dim_z Range
Data generated from GSA (and merged into one corpus)	A	1000	3-10	3-10	3-10	5-10	5-10	5-10
	B	30	1-30	1-30	1-50	0.5-25	0.5-25	0.5-25
	C	50	1-15	1-15	1-15	0.5-15	0.5-15	0.5-15
	D	400	1-15	1-15	1-30	0.5-15	0.5-15	0.5-15
	E	300	1-10	1-10	1-10	0.5-10	0.5-10	0.5-10
	F	200	1-5	1-5	1-5	0.5-10	0.5-10	0.5-10
Original Training	Training	929	3-10	3-10	3-10	5-10	5-10	5-10
	Validation	104	3-10	3-10	3-10	5-10	5-10	5-10
	Test	863	0-3, 10-25	0-3, 10-25	0-3, 10-30	0-5, 10-12	0-5, 10-12	0-5, 10-12
	Unused	82						
Retraining	Training	1,233	2-15	2-15	2-15	2-15	2-15	2-15
	Validation	138	2-15	2-15	2-15	2-15	2-15	2-15
	Test	607	0-2, 15-30	0-2, 15-30	0-2, 15-50	0-2, 15-25	0-2, 15-25	0-2, 15-25

[3]Note that all data must be normalised before being used for training or inference by the BNN.

2.2 Usage

2.2.1 Installation

For installation instructions, please refer to `README.md` in `irp-aol21`.

2.2.2 Documentation

Documentation has been generated using *Sphinx*. The html version can be found in `irp-aol21/docs/build/html/index.html`. Sphinx automatically populates the documentation with the information contained in each function's docstring, as well as manually written reStructured-Text (RST) code. There is also important information displayed in `README.md`.

2.2.3 Tracking Training with Weights and Biases

If `wandb_mode` is set to `True` when training the NNs, the user will be prompted to log in to *Weights and Biases* so the training can be tracked. Weights and Biases provides experiment tracking, dataset versioning and model management [4], and has been used to keep track of the model training over the course of this project. The user can access the training status of their models in real time, making it very useful for checking up on the training of large models.

2.2.4 Folder Structure

The `irp-aol21` directory (i.e. the root folder in the Github repository) has the following structure:

```
irp-aol21
├── data                                                CSV data from GSAPy
├── docs                                                      Documentation
│   ├── build
│   │   ├── html
│   │   │   ├── index.html                        Sphinx generated documentation
├── duq                                                   Main Python modules
│   ├── app.py                                                      Flask app
│   ├── mc_dropout.py
│   │   hfill MC Dropout module
│   ├── post.py                                          Post processing scripts
│   ├── pre.py                                        Data pre processing scripts
│   ├── sgld.py                                                    SGLD module
├── info                                                 Mandatory IRP documents
├── notebooks                                                  Jupyter notebooks
│   ├── changing_num_outputs                    Changing number of output params
│   ├── retraining                              Retraining models with more data
│   ├── reference_models               Models to give accurate point prediction
│   ├── splitting_data                 Investigation into splitting the datasets
│   ├── tuning                                          Hyperparameter tuning
│   ├── 1D_regression
│   │   ├── BLRPy.py                      Bayesian Linear Regression using MCMC
│   │   ├── BPRPy.py                   Bayesian Polynomial Regression using MCMC
│   │   ├── BR_GaussPy.py    Bayesian Regression (Gaussian basis) using MCMC
│   ├── others_implementations                  Others' BNN implementations
│   ├── SGLD.ipynb
│   ├── MC_Dropout.ipynb
├── reports
├── tests
├── trained_models
│   ├── for_prediction                  Pre trained models for predictions
│   │   ├── MC_Dropout
│   │   ├── SGLD
│   │   ├── MCDropout_reference
│   ├── for_retraining                Checkpoints used to continue training
├── environment.yml
├── LICENSE.txt
├── README.md
├── requirements.txt
├── setup.py
```

2.2.5 Underlying Algorithms

The two UQ methods implemented in `duq` are MC Dropout (Section 1.1.1) and SGLD (Section 1.1.2). The key algorithms underpinning their training are outlined in Algorithm 1 and Algorithm 2 below. Further details can be found in the code and documentation files.

Algorithm 1: MC Dropout Training Procedure

Data: Dataset $\mathbf{X}$, $\mathbf{Y}$ with **N** datapoints
Result: Learned model weights $\boldsymbol{\theta}$, training loss
Draw $\boldsymbol{\theta} \sim$ initial probability distribution
$J(\hat{\mathbf{y}}, \mathbf{y}) = RMSE(\hat{\mathbf{y}}, \mathbf{y})$
for $i = 0;\ i < num\_epochs;\ i = i + 1$ **do**
  $Loss \leftarrow 0$;
  for *minibatch* $\mathbf{x}$, $\mathbf{y}$ *of* $\mathbf{X}$, $\mathbf{Y}$ **do**
    Sample random activation coefficients: $z_{i,j} \sim Bernoulli(p_{dropout})$
    Apply mask to model weights: $\boldsymbol{\theta_i^{'}} \leftarrow \boldsymbol{\theta_i} \cdot diag(z_i)$
    $\hat{\mathbf{y}} \leftarrow$ model.forward$(\mathbf{x}, \boldsymbol{\theta}^{'})$
    $Loss \leftarrow Loss + J(\hat{\mathbf{y}}, \mathbf{y})$
    Calculate $\frac{\partial J}{\partial \boldsymbol{\theta}}$ via backprop
    Update model weights with chosen optimiser
  end
  Training loss $\leftarrow \frac{J}{N}$
end

Algorithm 2: SGLD Training Procedure

Data: Dataset $\mathbf{X}$, $\mathbf{Y}$ with **N** datapoints
Result: Learned model weights $\boldsymbol{\theta}$, training loss
Draw $\boldsymbol{\theta} \sim$ initial probability distribution
$J(\hat{\mathbf{y}}, \mathbf{y}) = RMSE(\hat{\mathbf{y}}, \mathbf{y})$
for $i = 0;\ i < num\_epochs;\ i = i + 1$ **do**
  $Loss \leftarrow 0$;
  for *minibatch* $\mathbf{x}$, $\mathbf{y}$ *of* $\mathbf{X}$, $\mathbf{Y}$ **do**
    $\hat{\mathbf{y}} \leftarrow$ model.forward$(\mathbf{x}, \boldsymbol{\theta}^{'})$
    $Loss \leftarrow Loss + J(\hat{\mathbf{y}}, \mathbf{y})$
    Calculate $\frac{\partial J}{\partial \boldsymbol{\theta}}$ via backprop
    Update model weights with chosen optimiser
    for $\theta_{i,j}$ *in* $\boldsymbol{\theta}$ **do**
      if $\theta_{i,j}$ *requires grad* **then**
        $\theta_{i,j} \leftarrow \theta_{i,j} + N(0, lr)$
      end
    end
  end
  Training loss $\leftarrow \frac{J}{N}$
end

2.2.6 Training NNs

Once the training, test and validation datasets have been split, or imported separately from elsewhere, the NNs can be trained. Jupyter notebooks outlining this training process can be found in `notebooks/MC_Dropout` and `notebooks/SGLD` for MC Dropout and SGLD respectively.

The `MC_Dropout` or `SGLD` model classes must be instantiated using the `params` `dict`, *normalised* `train_data`, and the corresponding `data_mean` and `data_std` that initially normalised the data. The means and stds ideally should be Pandas Series objects that each have a single value per column of the dataset - in our example it should be length 12. The user can optionally pass `wandb_mode`, which controls whether to log to *Weights and Biases* (which is `False` by default), and also `validation_data` which should have been normalised according to `data_mean` and `data_std`.

The `dict` controls every aspect of the model (including depth and width of network), and contains slightly different information depending on which UQ method is being used. Table 2 outlines the requirements of this `dict`, with the parameters specific to the *MC Dropout* and *SGLD* methods, and the other parameters that are agnostic of model choice. The parameters not shown here in the `params` dict are optional, and are only referenced by the Weights and Biases API and some plotting functionality later.

Table 2: Parameters dict (`params`) used for instantiating model classes

Category	Parameter	Description
MC Dropout Only	`drop_prob`	Dropout rate
	`num_samples`	Number of forward passes per datapoint
SGLD Only	`noise_multiplier`	Langevin scale factor
	`anneal_gamma`	Learning rate decay rate [4]
	`burnin_epochs`	Burnin epochs
	`num_networks`	Number of saved networks
Generic Hyperparameters	`num_epochs`	Number of epochs
	`batch_size`	Minibatch size
	`lr`	Learning rate
	`weight_decay`	L2 regularisation multiplier
Model Architecture	`input_dim`	No. input neurons
	`output_dim`	No. output neurons
	`num_units`	No. neurons per hidden layer
	`num_layers`	Total no. layers
Data	`y_cols`	Col num(s) of dependent vars
	`x_cols`	Col num(s) of independent vars

Once the `model` class has been instantiated, the user can call the `train_model` method from it. This can take multiple optional arguments: `LLP` (bool) controls whether or not to display the *LiveLossPlot* showing the loss curves[5], `checkpoint` (bool), whether or not to save training so it can be continued later, and `checkpoint_path`, which is the full path and name for the checkpoint.

The `train_model` method returns the `net` (the `nn.Module` class), the final `training_loss` and the final `validation_loss` [6] after training has completed.

[4] LR annealing according to `torch.optim.lr_scheduler.ExponentialLR`. See PyTorch documentation for more details.

[5] LLP should be set to False if running training from a terminal/command line

[6] `train_model` returns validation loss only if the user has specified `val_data` when instantianting the model class.

2.2.7 Performing Inference

Once the `model` class has been trained, inference [7] can be performed. The list below outlines a number of ways the user can generate predictions from a trained model:

1. *Performing prediction in Grasshopper / Rhino*: Grasshopper can query RESTful APIs to instantly visualise models and data in Rhino. A local port should be opened using *Flask*, which the user can do by navigating to `irp-aol21/duq` and running `flask run` in a terminal window. The user can then load up the Grasshopper script `DUQ_Test.gh`, adjust the parameters and see the result. See Section 4.1 for an outline of this process.

2. *Testing the model against reference data*: The user should call the `run_sampling` method, passing in the inputs ($\mathbf{X}$ and true values $\mathbf{Y}$. For `MC_Dropout` this runs `num_samples` number of stochastic forward passes through the network, and for `SGLD` this runs forward passes through `num_networks` saved networks [8]. `run_sampling` returns `means` (the mean prediction for each point), `stds` (the standard deviation of each point), `y_np` (the true reference values in NumPy format) and `samples`, the individual samples that have been pooled to produce the means and standard deviations.

3. *Performing a prediction on an arbitrary point*: The user should call `model.make_prediction`, passing in a single $\mathbf{x}$ value and optionally the `bool`s `verbose` and `plots`. This method also returns `means`, `stds` and `samples`.

4. *Peforming a range of predictions within a given domain*: The user can call `duq.post.generate_3d_samples()`, passing in the `model` class, `params` and `num_samples` if using `MC_Dropout`. `params` is a list of length 6 that contains the extents the user wants to explore. The user is able to choose a range of extents for any 3 of the 6 input dimensions, and the remaining 3 dimensions will be fixed at a value of choice. The script will then automatically generate the given number of predictions and uncertainties within the chosen domain, which can then be visualised using `duq.post.generate_3d_plot()`. See Section 4.2.3 for further details.

5. *Performing prediction using loaded model (in Python)*: Once a trained model has been saved (using `torch.save(model, savepath)`), it can then be loaded using `model = torch.load(savepath)`, then predictions can be made as above. See `irp-aol21/notebooks/loading_models.ipynb` for an example.

[7] Inference here refers to the machine learning definition (i.e. performing a prediction) instead of in the Bayesian modelling sense, where inference refers to integration over model parameters [9].

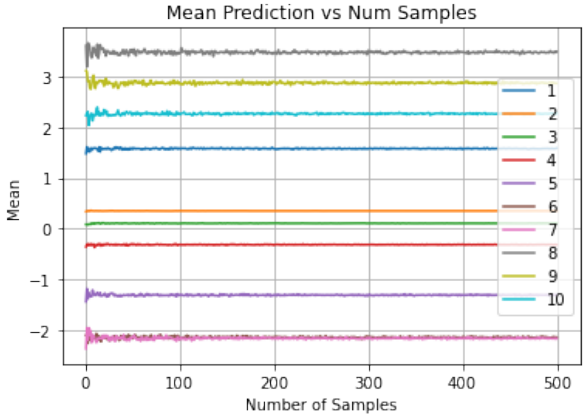
[8] Note that `num_samples` can be changed after model training, however `num_networks` cannot. See Section 2.3 for further analysis.

2.3 Number of Samples Drawn versus Prediction Accuracy and Uncertainty

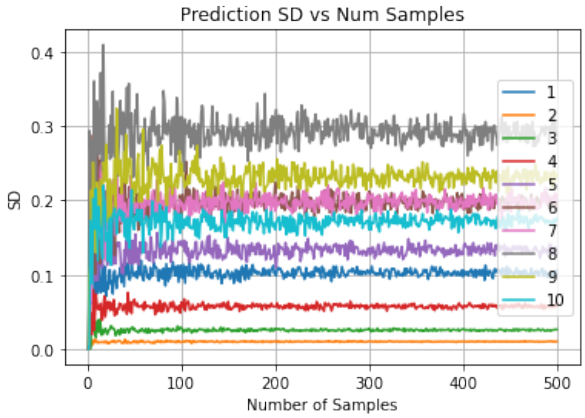
Since both MC Dropout and SGLD both depend on samples being drawn to generate a point prediction and uncertainty, an appropriate sample size should be determined for both methods.

MC Dropout and SGLD were trained using the data split discussed in Appendix B, then inference is carried out using a range of samples on 10 different datapoints. These datapoints span betwen $[1, 1, 1, 1, 1, 1]$ and $[40, 40, 50, 30, 30, 30]$.

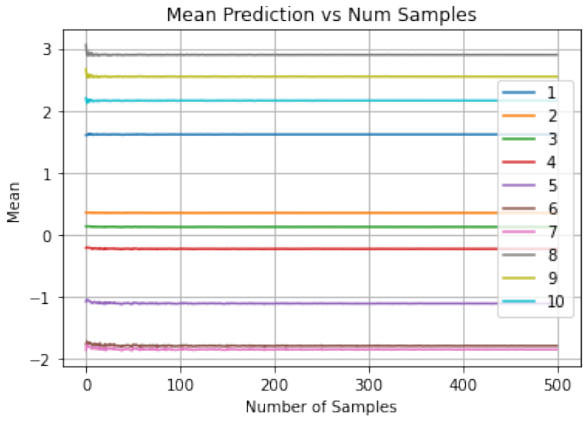
For MC Dropout, the number of stochastic forward passes through the network is varied between $(1, 500)$, and for SGLD the number of saved networks to perform forward passes on is varied between $(1, 500)$. Figure 2(a) and 2(b) shows the MC Dropout results and Figure 2(c) and 2(d) shows the results of SGLD [9]. For both methods, it appears that $100 - 200$ samples is ample to achieve a relatively stable uncertainty figure. The datapoints chosen for evaluation are a mixture of in-domain and out of domain. Full results, including tests of other scenarios and hyperparameters, are found in `irp-aol21/notebooks/num_sample_investigation`.



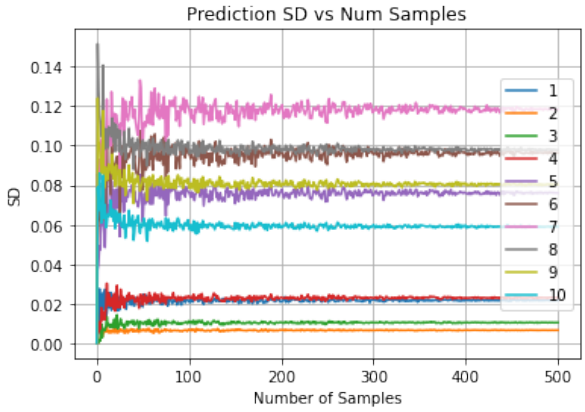
(a) MC Dropout: Mean Predictions



(b) MC Dropout: Std. Dev.



(c) SGLD: Mean Predictions



(d) SGLD: Std. Dev.

Figure 2: Varying number of samples drawn using MC Dropout and SGLD

[9] For MC Dropout, a dropout rate of 20% is used. For SGLD, the noise multiplier is set to 1.0.

2.4 Tuning the Bayesian Neural Networks

The surrogate models have their hyperparameters tuned to give optimal calibration of uncertainty - i.e. ensuring that the network doesn't have excessive confidence in OOD regions and has reasonable confidence where the accuracy of the predictions is high. A total of 65 models were trained using the parameters shown in Table 3. In most runs, only one parameter is modified at a time; the others remain at baseline. The full list of model selections is shown in `model_tracker.xlsx` in the `irp-aol21` repository. Jupyter notebooks were used to track model simulation and results - see `irp-aol21/notebooks/tuning/` for full details.

Table 3: Hyperparameters explored, and final values used for inference

Model	Parameter	Search Range *value (baseline)*	Final Value
MC Dropout	Epochs	1000-3000 (1000)	1000
	Drop prob	0.1-0.7 (0.2)	0.3
	Hidden layers	1-3 (1)	3
	Neurons per hidden layer	50-500 (100)	100
	No. samples	150	150
	Batch size	50-929 (50)	50
	LR	5×10^{-6}-5×10^{-1} (5×10^{-4})	5×10^{-4}
SGLD	Epochs	1000-2000 (2000)	1000
	Burnin epochs	250-500 (250)	250
	Weight decay	0-0.05 (0)	0
	Weight annealing	0.99-None (None)	None
	Noise multiplier	1-3 (1)	1
	Hidden layers	1-3 (1)	2
	Neurons per hidden layer	50-500 (100)	100
	No. samples	150	150
	Batch size	50-929 (50)	50
	LR	5×10^{-5}-5×10^{-2} (5×10^{-4})	1×10^{-3}

A model is trained for each hyperparameter configuration using the training dataset, then forward passes are carried out on the train, test and validation datasets. If the number of *untrustworthy* predictions is over 20% in the training set[10] or if the number of *untrustworthy* predictions is 100% in the validation set then the model is excluded. After this initial filtering, a range of metrics are calculated for the test, train and validation datasets: the number of *untrustworthy* predictions; the error between predictions and the true values; the relative uncertainty $(100 \times \sigma/\mu)$ of each prediction.

Each model is then tested against a range of manually inputted points, ranging from $[1, 1, 1, 1, 1, 1]$ through to $[40, 40, 50, 30, 30, 30]$. Based on all the analysis thus far and this final test, the best performing model is chosen and brought forward. Please see the corresponding notebooks in `irp-aol21/notebooks/tuning` for the full results of this hyperparameter selection. The results of this analysis are also logged to *Weights and Biases* - see https://wandb.ai/archieluxton/projects.

[10] A prediction has been classified *trustworthy* if its confidence interval captures the true value. Note this does not relate to the accuracy of the prediction itself. The mean prediction could be imprecise, but a wide confidence interval could still mean it scores high in this metric.

3 Code Metadata

Table 4: Key information about `duq`.

Field	Value	Comments
Package Name	DeepUQ (duq)	
Version Number	0.1	
Github Repo	https://github.com/ese-msc-2021/irp-aol21	
Documentation Location	irp-aol21/docs/_build/html/index.html	
Installation Guidance	See Readme, or Section 2.2.1	
Dependencies	Python 3.X	
	arviz	
	livelossplot	
	matplotlib	
	jupyter	
	numpy	
	pandas	
	scikit_learn	
	scipy	
	seaborn	
	torch	
Optional Dependencies	Grasshopper + Rhino	Windows only. Requires Flask
	joypy	For plots only
	wandb	For model tracking
	Flask	For RESTful API

4 Results

This section outlines the result of applying `duq` to construct a surrogate model that can predict modal frequencies of structures given details about their geometry as outlined in Section 2.1

Both *MC Dropout* and *SGLD* have been used to construct the surrogates, however only the results of *MC Dropout* will be presented here due to space limitations, as it is shown to have better uncertainty characterisation on OOD data when being re-trained on new data. Full details and analysis can be found in the Jupyter notebooks in `irp-aol21/notebooks/`. The model is trained on the original dataset split and then training is continued with the expanded dataset, as outlined in Section 2.1 and Appendix B. The expanded dataset adds 304 points to the original training set.

4.1 Visualising Results in Grasshopper / Rhino

Figure 3 shows the Grasshopper interface that has been successfully developed in collaboration with Kristjan Nielsen at Arup[11]. The application allows a user to parametrically design a simple structure (using sliders) that's instantly visualised in Rhino. Grasshopper serves as a rapid prototyping environment, where an engineer can very quickly perform preliminary design of structures.

For every configuration drawn, Grasshopper queries a RESTful API [12] to carry out forward passes of the trained surrogate that leverages*MC Dropout* or *SGLD* and returns a frequency prediction and an uncertainty. The relative uncertainty ($100 \times \sigma/\mu$) is used to colour the structure; green indicates a high certainty, and red indicates a low certainty[13]. The script also queries a reference surrogate that has been trained on the full corpus for many epochs (without UQ) to get an approximation to a *ground-truth*.

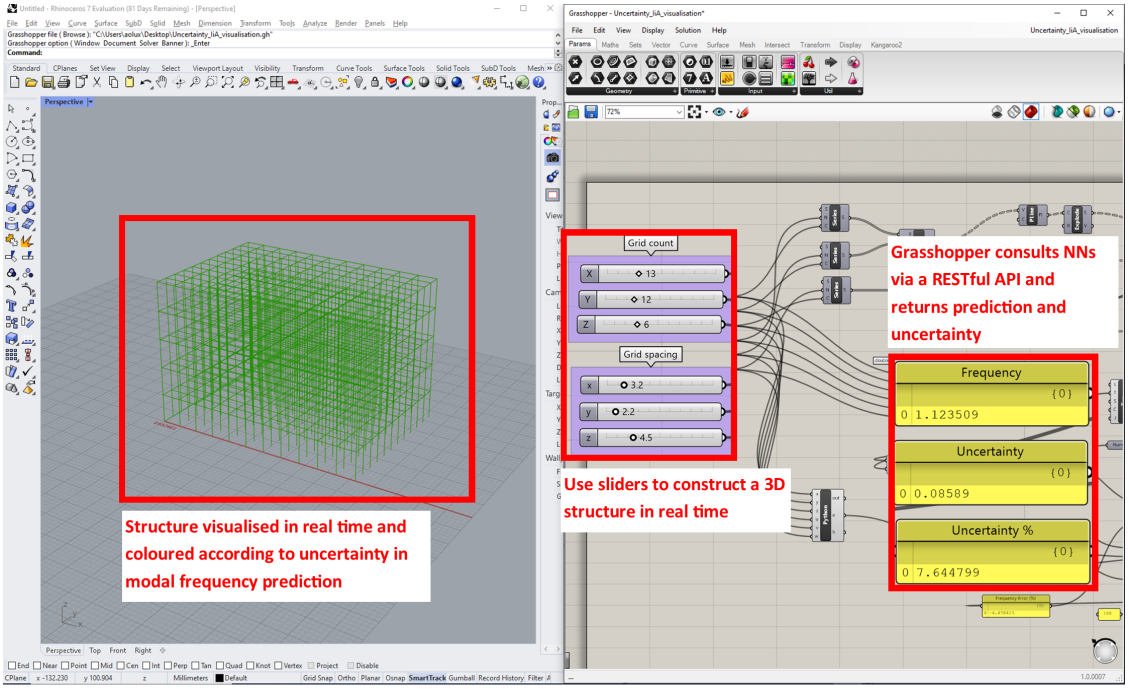
This interface is ideal for rapid prototyping of designs, allowing engineers to make informed decisions about a wide range of designs. The forward passes of the BNNs are made very fast: the MC Dropout-based model, the SGLD-based model and the reference model all take on average between 35.8ms and 38.6ms to return a prediction and standard deviation from the API[14]. An analysis on the accuracy and uncertainty of the surrogate before and after retraining with more data is explored in the further sections.

[11]The API interface in Grasshopper is entirely original work

[12]The API is opened locally using Flask as described in method 1 of Section 2.2.7

[13]Note that the scale of the colour grading can be adjusted. These pictures show green=0% and red=50% relative uncertainty

[14]Timing as a result of 7 runs of 200 loops for each method. Noting that 150 forward passes are carried out on the MC Dropout and SGLD networks, and only one is carried out on the reference network.



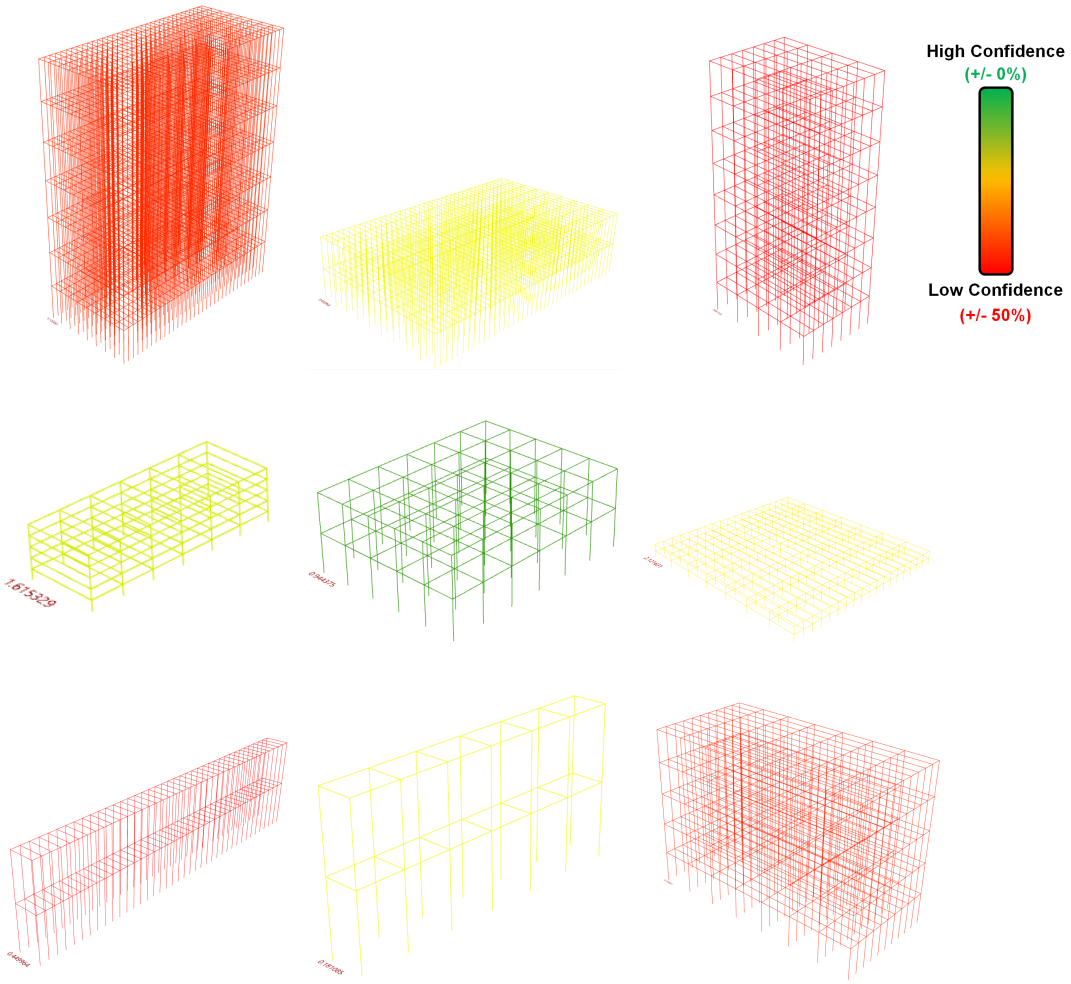
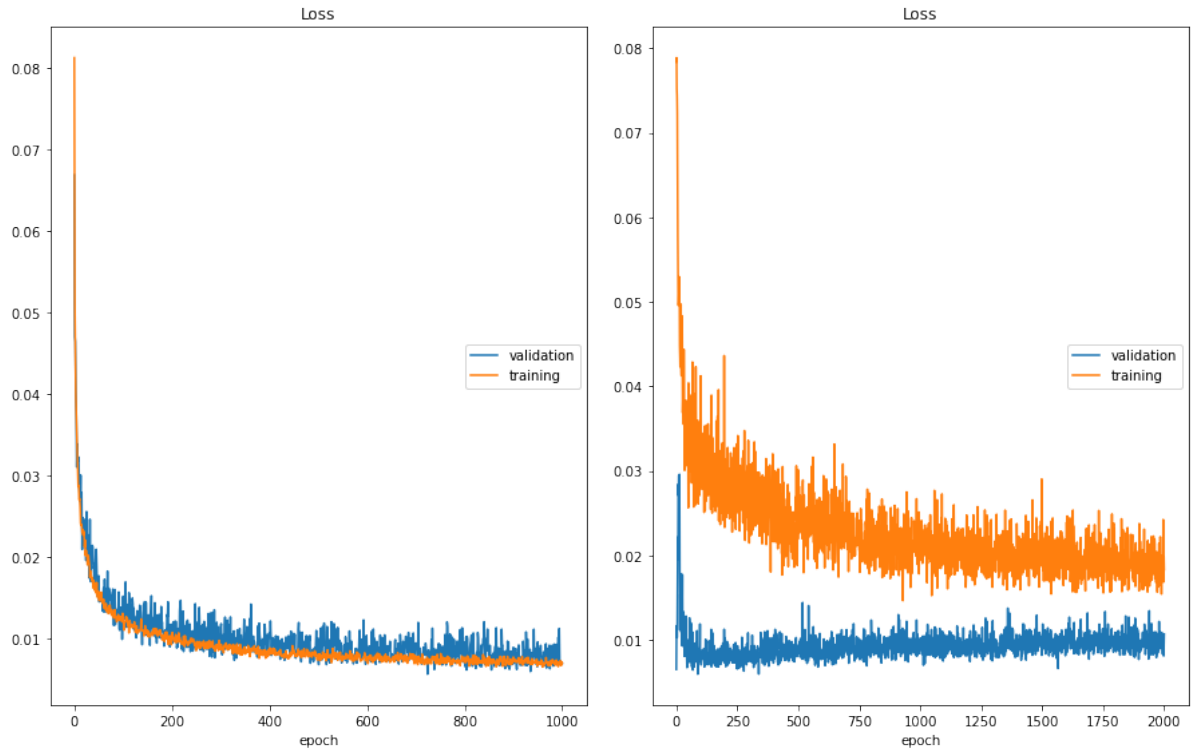


Figure 3: Grasshopper/Rhino UI and a variety of configurations coloured by uncertainty %

4.2 Loss Plots

The below plots show the process of training the tuned *MC Dropout* network on the datasets outlined in Section 2.1 and B.

The first round of training completed in 71 seconds, and the second round completed in 73 seconds. The first reached a minimum training loss of 0.007 and a minimum validation loss of 0.006, the second reached a minimum training loss of 0.015 and a minimum validation loss of 0.006. In practice, this re-training could have a different learning rate or batch size to improve convergence. Figure 4 shows the loss plots for the training and validation sets for both training rounds:



(a) MC Dropout loss plot - Original Training

(b) MC Dropout loss plot - Retraining

Figure 4: Loss plots for first and second rounds of training for MC Dropout

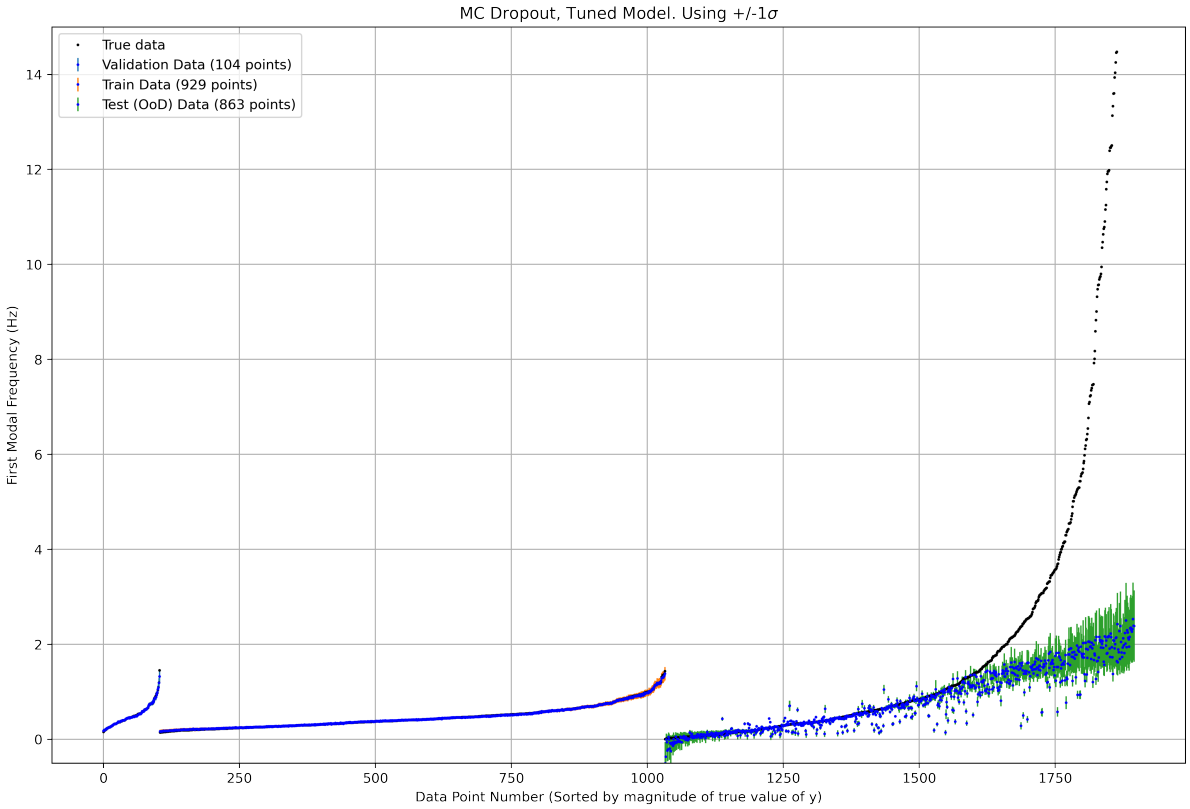
4.2.1 Error Bar Plots

Figure 5 shows every prediction and confidence interval (using $\pm 1\sigma$) made by the networks over the three datasets [15]. The y axis has been truncated, and limited only to the region containing the predictions and confidence intervals. Each datapoint has been sorted w.r.t. the scale of the true value, so here the x axis simply represents an arbitrary datapoint.

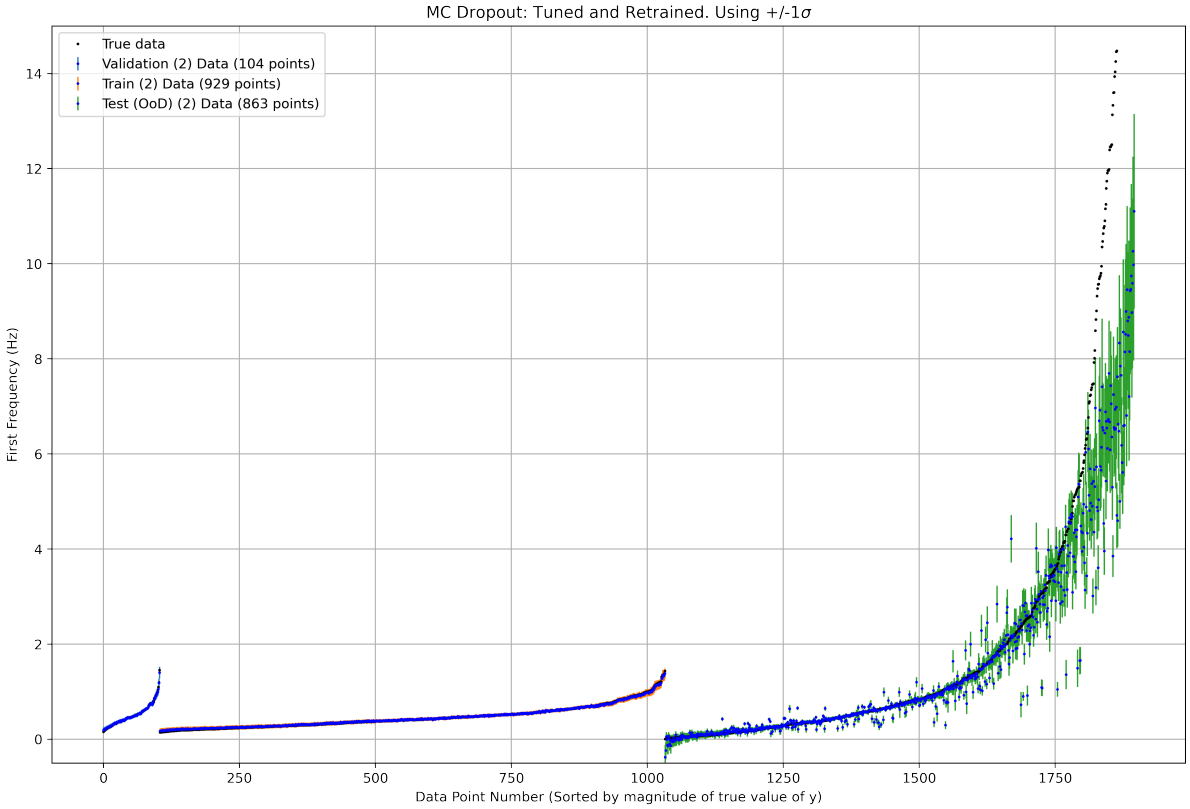
These two plots clearly show the impact that re-training has on the networks: when a wider domain of data is made available, the mean error between prediction and true value decreases from -2.014 to -1.216 [16].

[15] Note that for all models and methods, `duq` can produce confidence intervals using standard deviation or HDI

[16] See Appendix C for further analysis



(a) MC Dropout predictions and confidence intervals - Original training



(b) MC Dropout predictions and confidence intervals - Retrained

Figure 5: Predictions and confidence intervals ($\pm 1\sigma$) for MC Dropout. Figure shows the result of using the originally trained model and a re-trained version.

4.2.2 Distribution of Errors and Uncertainty

See Appendix C for a range of plots produced that summarise the prediction accuracy and the uncertainty of a trained and re-trained *MC Dropout* network. The results are summarised here:

4.2.3 Visualising Uncertainty in 3D

Figure 6 demonstrates the different methods for visualising uncertainty in 3D using `duq`.

The top row of figures shows $\mathbf{X}$ from every dataset being reduced from $\mathbb{R}^6$ to $\mathbb{R}^3$ using Principal Component Analysis (PCA). The points have been coloured according to the dataset, and sized based on the uncertainty in the resulting predictions from the trained networks detailed above. This clearly shows the higher uncertainty in the test set compared to the training/validation sets.

The second row of figures shows the uncertainty using method 3 outlined in Section 2.2.7: performing predictions on a wide range of points over a given domain, and returning their confidence interval (using $\pm 1\sigma$). The third row of figures shows the uncertainty using method 3 from Section 2.2.7, but scaling each point according to the relative uncertainty, i.e. $100 \times \sigma / \mu$. The length of members in x, y and z is fixed at $5m$ in the second and third row of plots in Figure 6.

Please refer to notebooks `notebooks/MC_Dropout.ipynb` and `notebooks/retraining/MC_Dropout.ipynb` for further details, and refer to the documentation for more commands and options in generating these plots. Animated 3D versions have been included in `irp-aol21/reports/final_report_images_videos`.

These plots show that the uncertainty of the frequency is most correlated with the number of bays in the z direction. This visualisation of uncertainty could be used in practice to decrease the uncertainty in a particular domain without needing much additional training data: the results can quickly be obtained and these 3D plots be examined for differences in uncertainty.

4.3 Adjusting Number of Output Predictions

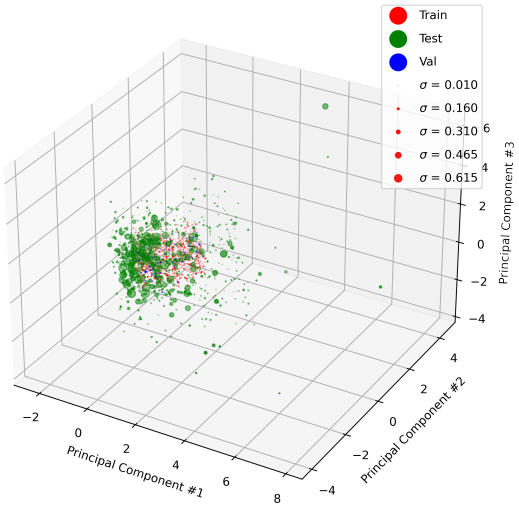
Both *SGLD* and *MC Dropout* were trained to predict between 1 and 6 modal frequencies. The convergence according to the loss plots was as expected; the higher the number of outputs, the slower the convergence and the higher resulting loss value, however the differences in loss were not great.

For SGLD, the validation loss after training was between 0.0027 (for a single output) and 0.0065 (for 6 outputs), and the time taken was relatively constant at between 434s (for a single output) to 441s (for 6 outputs).

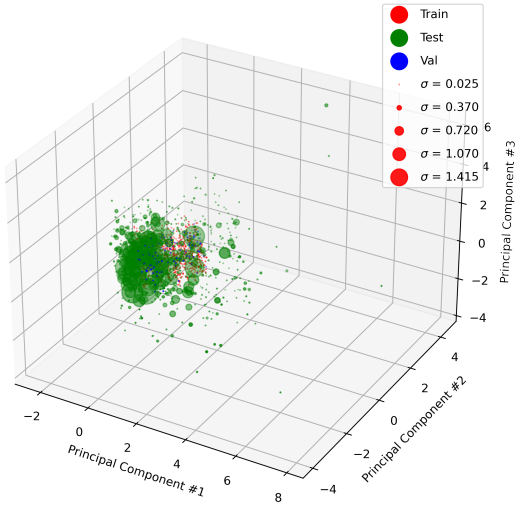
For MC Dropout it spanned between 0.0069 (for one output) and 0.0129 (for 6 outputs). The time taken was between 190s (for a single output) to 220s (for 6 outputs)

4.4 1D Regression

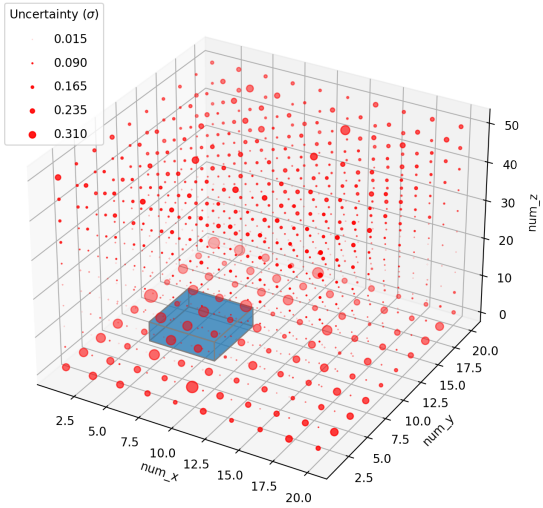
Though this project is focused on multi-dimensional inputs and outputs, please see Appendix A for a brief study into using a number of methods (including *MC Dropout*, *SGLD*, *VI+MC*, *Bayes by Backprop* and *Metropolis-Hastings MCMC*) to perform 1D regression on a damped sinusoid.



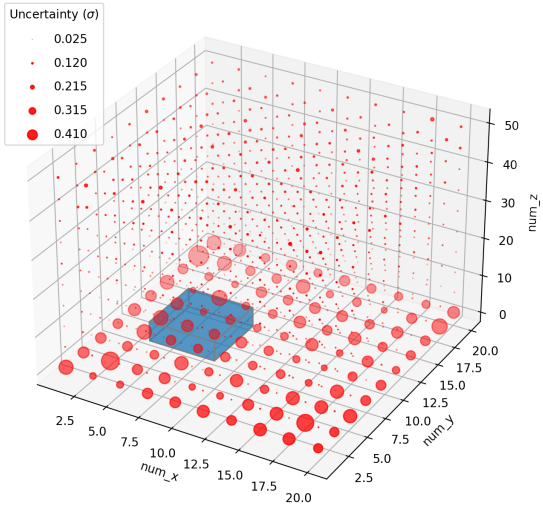
(a) MC Dropout uncertainty: test/train/val



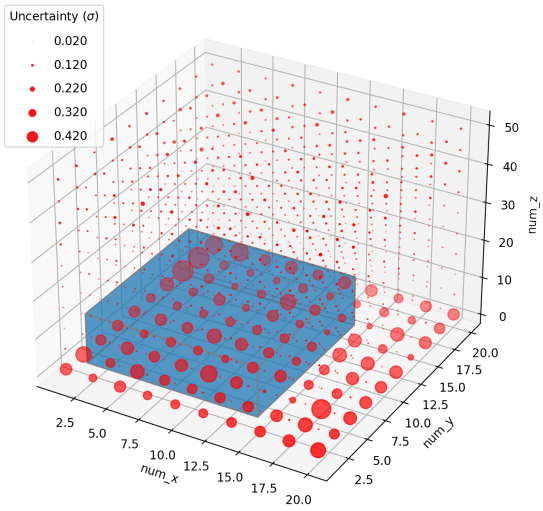
(b) MC Dropout (retrained) uncertainty: test/train/val



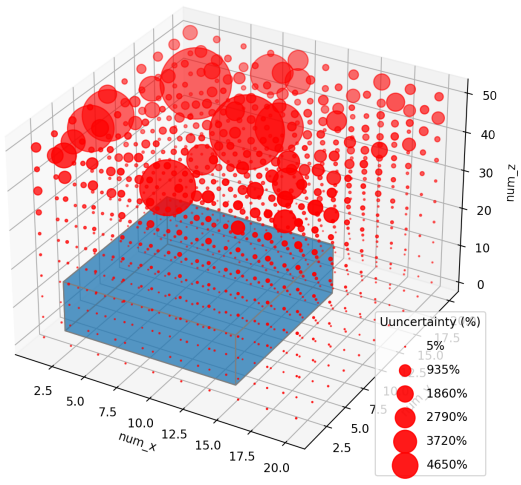
(c) MC Dropout uncertainty: synthetic data



(d) MC Dropout (retrained) uncertainty: synthetic data



(e) MC Dropout: relative uncertainty (synthetic data)



(f) MC Dropout (retrained): relative uncertainty (synthetic data)

Figure 6: Analysis of original and re-trained MC Dropout across multiple domains.

5 Discussion and Conclusions

It's critical for surrogate models to be able to convey uncertainty for them to be useful in professional engineering applications, particularly in mission-critical applications such as structural engineering.

The Python package `duq` has been developed that can very easily construct surrogates using *MC Dropout* and *SGLD* BNNs using feed-forward NN architecture and both *MC Dropout* and *SGLD* UQ techniques. These methods are shown to be fast; the surrogates designed here train in approximately $70s$ and can perform a prediction in approximately $36ms$ using API calls or from Python directly. The time taken to perform a prediction is an impressive $500\times$ to $11,000\times$ faster than performing the same simulation in *GSA*. Since `pytorch` is being used, there is inherent support for GPU acceleration using `CUDA`. This speed allows the user to iterate over countless designs using the purpose-built Grasshopper interface, which shows both the point prediction and uncertainty. Grasshopper also colours the generated structure according to its uncertainty; if the uncertainty in the design is too high, there is instant visual feedback prompting the user to explore other designs or re-train the surrogate.

The surrogate designed in `duq` is easy to use, requiring no construction of priors or fine tuning[17], and it is also accurate: the mean error between predictions and true values in a network trained in the domain $[(3, 10), (3, 10), (3, 10), (5, 10), (5, 10), (5, 10)]$ is 0 on the test and validation sets and 2.014 on the test set, and training it on an extended domain of data ($[(2, 15), (2, 15), (2, 15), (2, 15), (2, 15), (2, 15)]$) reduces the test set error to 1.216.

As the methods explored here give useful results to a simple engineering surrogate, future work may wish to explore the application of these methods to more complex surrogates, for example in predicting the transient response of structures to seismic forces. A web interface could also be developed to replace the Grasshopper/Rhino visualisation environment - this would improve portability and allow cross-platform support.

[17] Although tuning is shown to improve the calibration of uncertainty

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Sungjin Ahn. *Stochastic Gradient MCMC: Algorithms and Applications*. University of California, Irvine, 2015.

[3] Sungjin Ahn, Anoop Korattikara, and Max Welling. Bayesian posterior sampling via stochastic gradient fisher scoring. *arXiv preprint arXiv:1206.6380*, 2012.

[4] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[5] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*, 2018.

[6] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural network. In *International conference on machine learning*, pages 1613–1622. PMLR, 2015.

[7] Cameron Davidson-Pilon. *Bayesian methods for hackers: probabilistic programming and Bayesian inference*. Addison-Wesley Professional, 2015.

[8] Josiah et al. Davis. Quantifying uncertainty in deep learning systems. *AWS Professional Services*, 2020.

[9] Yarin Gal. *Uncertainty in Deep Learning*. PhD thesis, University of Cambridge, 2016.

[10] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059. PMLR, 2016.

[11] Alex Graves. Practical variational inference for neural networks. *Advances in neural information processing systems*, 24, 2011.

[12] Harry24k. bayesian-neural-network-pytorch. `https://github.com/Harry24k/bayesian-neural-network-pytorch`, 2019. Accessed on 25/07/2022.

[13] José Miguel Hernández-Lobato and Ryan Adams. Probabilistic backpropagation for scalable learning of bayesian neural networks. In *International conference on machine learning*, pages 1861–1869. PMLR, 2015.

[14] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[15] Machine Learning PhD student at University of Cambridge Javier Antoran. Bayesian-neural-networks. `https://github.com/JavierAntoran/Bayesian-Neural-Networks`, 2019. Accessed on 08/06/2022.

[16] Laurent Jospin. Bayesianmnist. https://github.com/french-paragon/BayesianMnist/tree/e0fb84de363ecda46217a3d8ac813e904f17a0ca, 2021. Accessed on 21/06/2022.

[17] Laurent Valentin Jospin, Wray L. Buntine, Farid Boussaïd, Hamid Laga, and Mohammed Bennamoun. Hands-on bayesian neural networks - a tutorial for deep learning users. *CoRR*, abs/2007.06823, 2020.

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

[19] Will Koehrsen. Introduction to bayesian linear regression. https://towardsdatascience.com/introduction-to-bayesian-linear-regression-e66e60791ea7, 2018. Accessed on 06/05/2022.

[20] Chunyuan Li, Changyou Chen, David Carlson, and Lawrence Carin. Preconditioned stochastic gradient langevin dynamics for deep neural networks. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[21] Oasys Limited. Oasys gsa. http://www.oasys-software.com/gsa. Accessed on 06/07/2022.

[22] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped dqn. *Advances in neural information processing systems*, 29, 2016.

[23] Henri Palacci. A look at sgd from a physicists's perspective - part 3, langevin dynamics and applications. https://henripal.github.io/blog/langevin, 2018. Accessed on 21/08/2022.

[24] PYSGMCMC. Stochastic gradient markov chain monte carlo sampling. https://pysgmcmc.readthedocs.io/en/pytorch/_modules/pysgmcmc/optimizers/sgld.html, 2017. Accessed on 25/07/2022.

[25] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[26] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. Probabilistic programming in python using pymc3. *PeerJ Computer Science*, 2:e55, 2016.

[27] Robert Schlaifer and Howard Raiffa. *Applied statistical decision theory*. Harvard University and MIT Press, 1961.

[28] Nabeel Seedat and Christopher Kanan. Towards calibrated and scalable uncertainty representations for neural networks. *arXiv preprint arXiv:1911.00104*, 2019.

[29] Dmitry Vetrov. Bayesian framework, deepbayes. https://github.com/bayesgroup/deepbayes-2019/blob/master/lectures/day1/1.%20Dmitry%20Vetrov%20-%20Bayesian%20framework.pdf, 2019. Accessed on 01/09/2022.

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

Appendix A 1D Regression

The plots shown in Figure 7 show the results of applying a range of methods to a 1-dimensional regression problem (a damped sinusoid). The following BNNs were used:

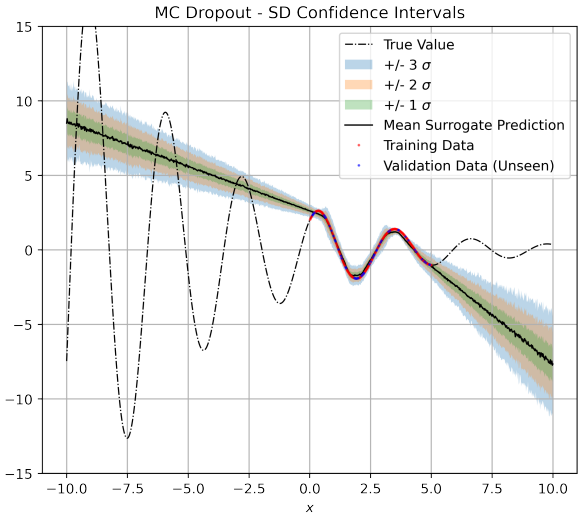
- MC Dropout (see Section 1.1.1)
- SGLD (see Section 1.1.2)
- VI + MC, adapted from the code available from [17] [16].
- Bayes by Backprop (BBP), adapted from the code available from [15].
  - TorchBNN, which is a specific implementation of *BBP*. Adapted from example provided by [12].

An MCMC-based method was also implemented from scratch and applied to this problem, utilising Gaussian basis functions and the Metropolis Hastings MCMC algorithm (see Figure 7(f)). 4 gaussians were used in this implementation, each with an associated scaling factor mean and standard deviation, for a total of 13 model parameters (including a term for aleatoric noise). Unfortunately the confidence interval in OOD regions is very small, making this unsuitable for carrying forward to the further experiments. See `irp-aol21/reports/final_report_images/1d_regression/gaussian_sine` for an animation visualising the MCMC process.

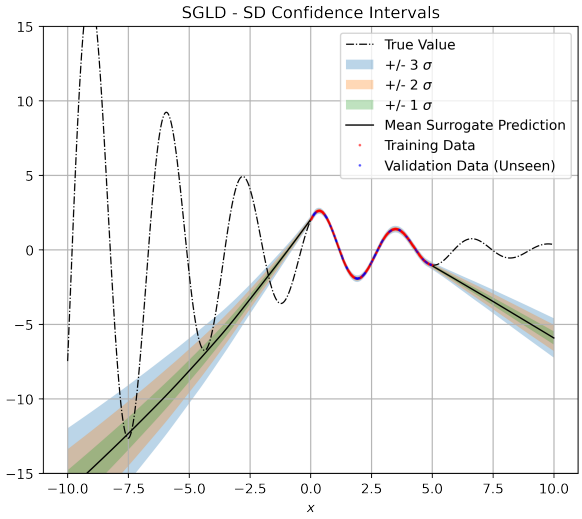
For all BNN methods, 200 training points were generated. All BNNs were trained with similar parameters, being trained for 2000 epochs with 300 neurons in a single hidden layer. The methods completed the calculations in the following times: *VI+MC*: 423s, *BBP*: 203s, *SGLD*: 185s, *MC Dropout*: 82s, *Gaussian basis function MCMC*: 59s, *torchBNN*: 19s. For further details of the simulations, please refer to the notebooks found in `irp-aol21/notebooks/1D_regression/`.

Of all the methods shown in Figure 7, *MC Dropout* and *SGLD* appear to have the highest accuracy in the training domain, and show the confidence interval becoming wider as x extends either side of the training domain.

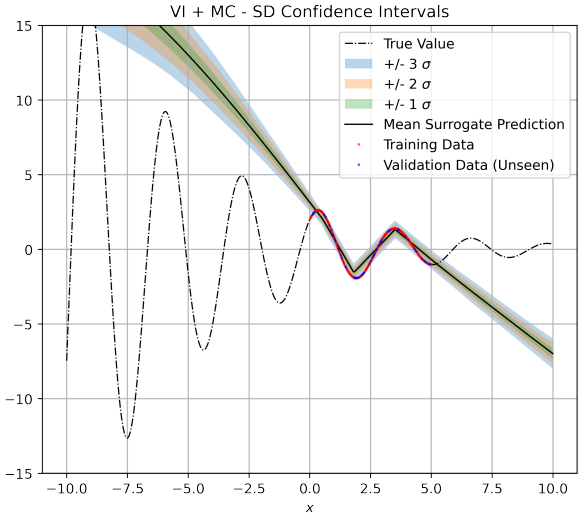
From this experiment, there is further confidence in choosing MC Dropout and SGLD as suitable UQ methods: they have demonstrated that they are simple to implement, predict in-domain data accurately, and can effectively increase the uncertainty in OOD predictions.



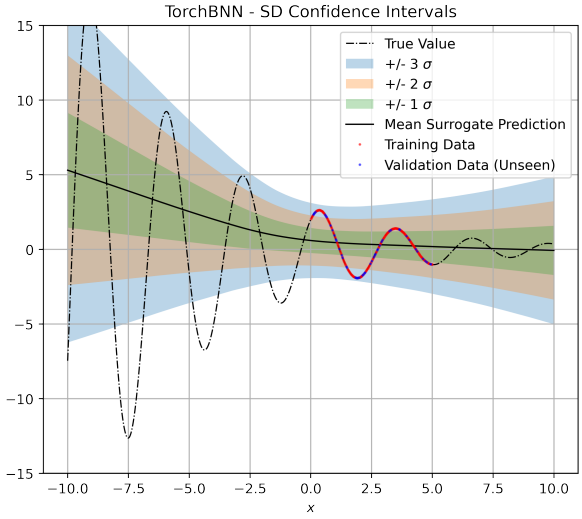
(a) MC Dropout: *82s run time*



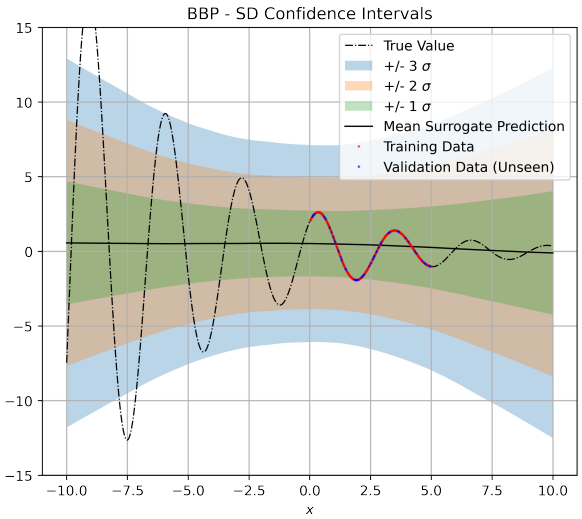
(b) SGLD: *185s run time*



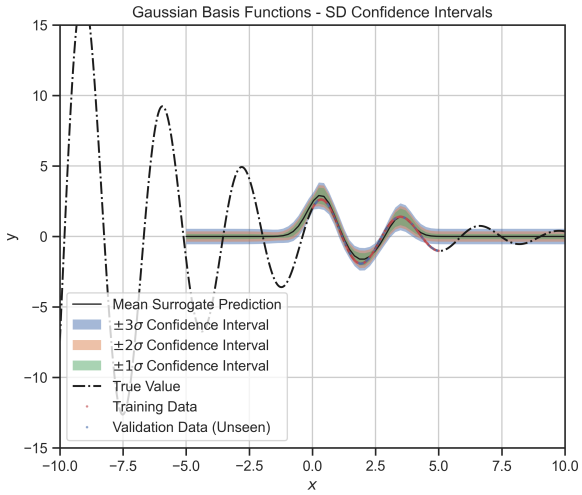
(c) VI + MC: *423s run time*



(d) TorchBNN: *19s run time*



(e) BBP: *203s run time*



(f) Gauss basis functions *59s run time*

Figure 7: Results of 1D regression using neural networks (a to e) and MCMC (f)

Appendix B Splitting Data

Since the input data in this application have very clear geometric meaning, i.e. all inputs are physical parameters that have real-life constraints and common ranges, it's possible to separate a training set from the main corpus using reasonable bounds on each input parameter. Since the input data is in a high-dimensional space ($\mathbb{R}^6$), PCA has been used to reduce the dimensionality to $\mathbb{R}^3$ for visualisation. Figure 8(a) shows the full corpus that was generated from Oasys GSA (1978 points total), and Figure 8(f) shows the result of the above process, splitting the full corpus into a test, train and validation dataset. To make the validation set, 10% of the test set is randomly split off the training set.

The datapoints in the training and validation sets are all within the domain of the largest dataset, *A*: $[(3, 10), (3, 10), (3, 10), (5, 10), (5, 10)(5, 10)]$. A datapoint is then assigned to the test set if it's either within $[(0, 3), (0, 3), (0, 3), (0, 5), (0, 5), (0, 5)]$ or $[(10, 25), (10, 25), (10, 25), (10, 12), (10, 12), (10, 12)]$. For further details and clarifications, see the `duq` documentation. The training set contains 929 points, the test set contains 863 points and the validation set contains 104. There are 82 unused points.

In an effort to keep the package flexible, other methods of splitting data have been explored below. Small-scale models were trained on data that was split using the methods below, and various metrics are compared.

a): Euclidean distance from geometric mean: The geometric mean of the input data is calculated, then the distance of each data point from that mean is calculated and stored. The dataset is sorted w.r.t. this calculated distance, then a portion is skimmed off the top and bottom of the dataset; in the example below, 5% of the highest and lowest L2 distances are skimmed off to form a test set. Then a portion of the remaining training set (in the example below, 10%) is randomly removed to form the validation set. See Figure 8(b).

During training, the training and validation loss are very dissimilar, with the validation loss being consistently higher and noisier than the training loss. It's clear that the training set encompasses some extreme inputs as some of the predicted frequencies are very high. Performance on the test set is very good as it isn't effectively OOD. It's clear from these tests that this method is not appropriate for consistently separating extreme datapoints from average ones.

b): Euclidean distance from arbitrary point: Since it's not unreasonable to expect to know at least one *average* datapoint, the Euclidean (L2) distance from a manually selected point is calculated for every datapoint and the process above is repeated again. See Figure 8(c).

During training, the results are very similar to when the data is split w.r.t. the geometric mean as above. The training and validation loss curves appear very different, and the resulting predictions in the training set exhibit characteristics of extreme geometric configurations. Performance on the test set is very good as it isn't effectively OOD. As above, this method is not appropriate for consistently separating extreme datapoints from average ones.

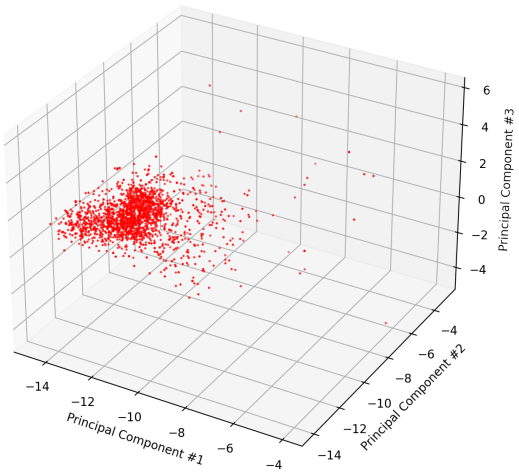
c): Split by scale of y: Similar to the two techniques above, the data is sorted w.r.t. the magnitude of a dependent variable (in the example below, the first modal frequency), then the top and bottom are skimmed off for the test set, then a portion randomly removed from the training set to form a validation set. See Figure 8(d).

The training and validation loss curves are much more similar than when using the methods above, and it's clear that the training set now does not encompass particularly extreme geometric configurations. The predictive performance on the test set is poor as would be expected. This appears to be a good method of splitting out extreme data.

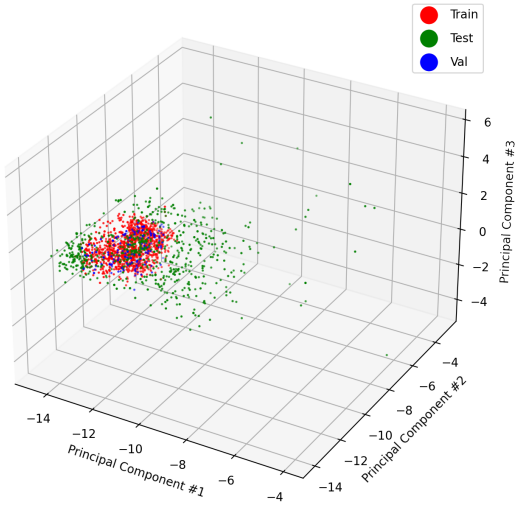
d): Euclidean distance from the geometric mean in reduced dimension space (PCA): Firstly, the whole corpus is reduced using PCA, then the geometric mean of this new data is calculated. A distance

from the geometric mean is specified, whereby any data outside the resulting sphere is put into a test set, and any data within is the training set. A validation set is extracted as before by splitting off a random portion of the training set, with the example below using 10%. See Figure 8(e).

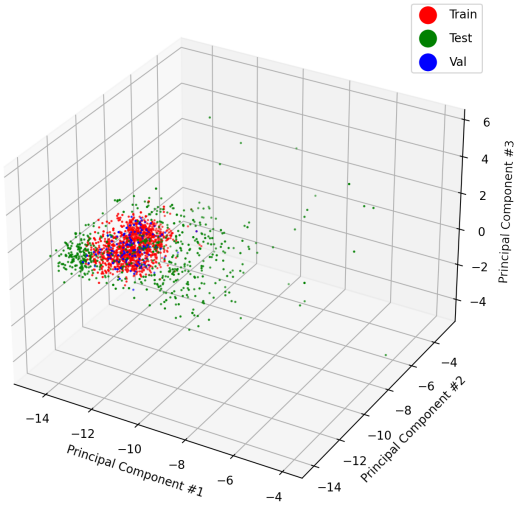
The training and validation loss curves are very similar using this method, though the validation curve exhibits a fair amount more noise than the training curve. Some extreme datapoints have been included in the training set, but it's clear that the method has separated the test set fairly well. Performance on the test set is poor as expected.



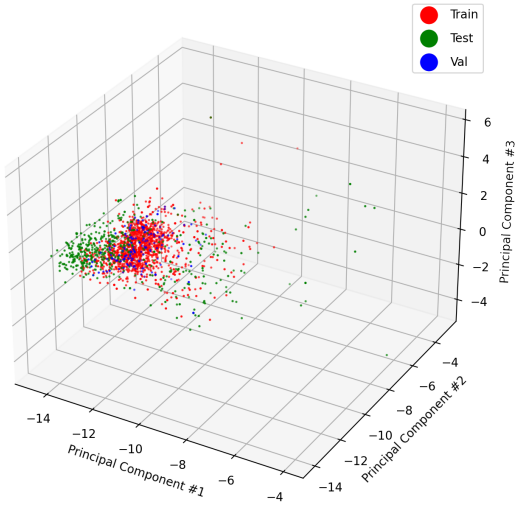
(a) Full dataset



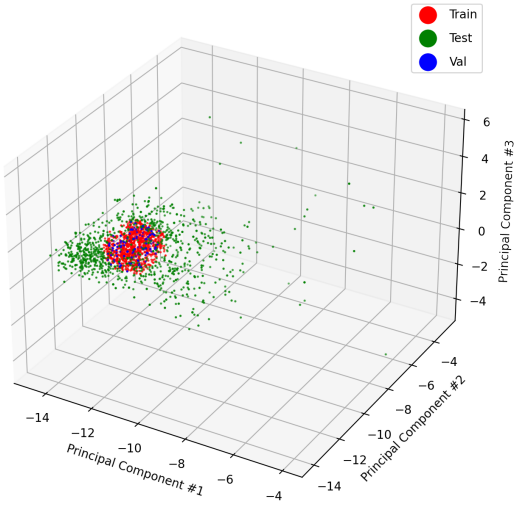
(b) Euclid dist from geometric mean



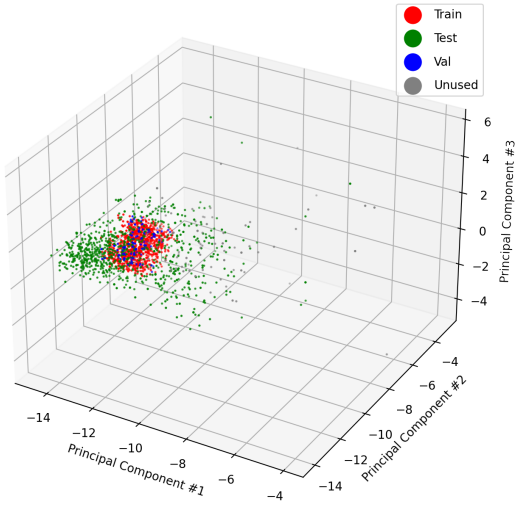
(c) Euclid dist from arbitrary point



(d) Scale of y



(e) Euclid dist from geometric mean, PCA



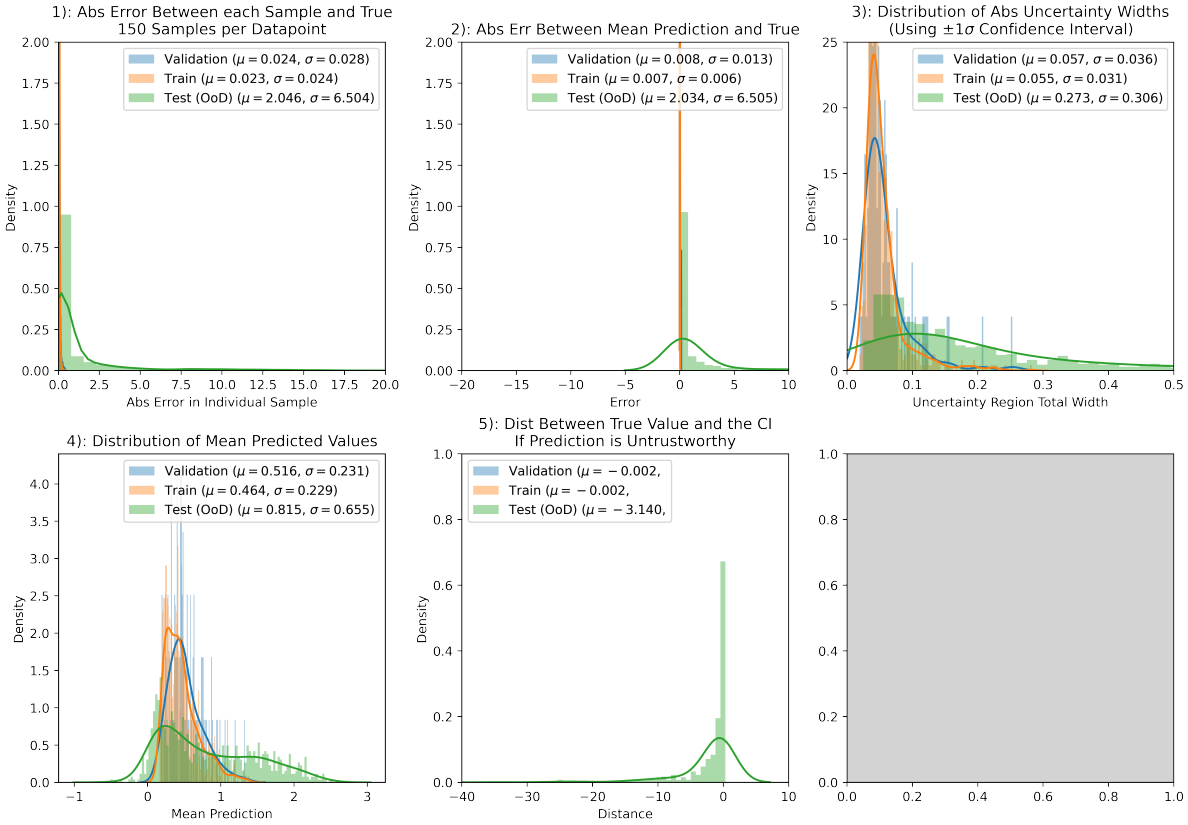
(f) Manual bounds

Figure 8: Splitting main corpus using different methods

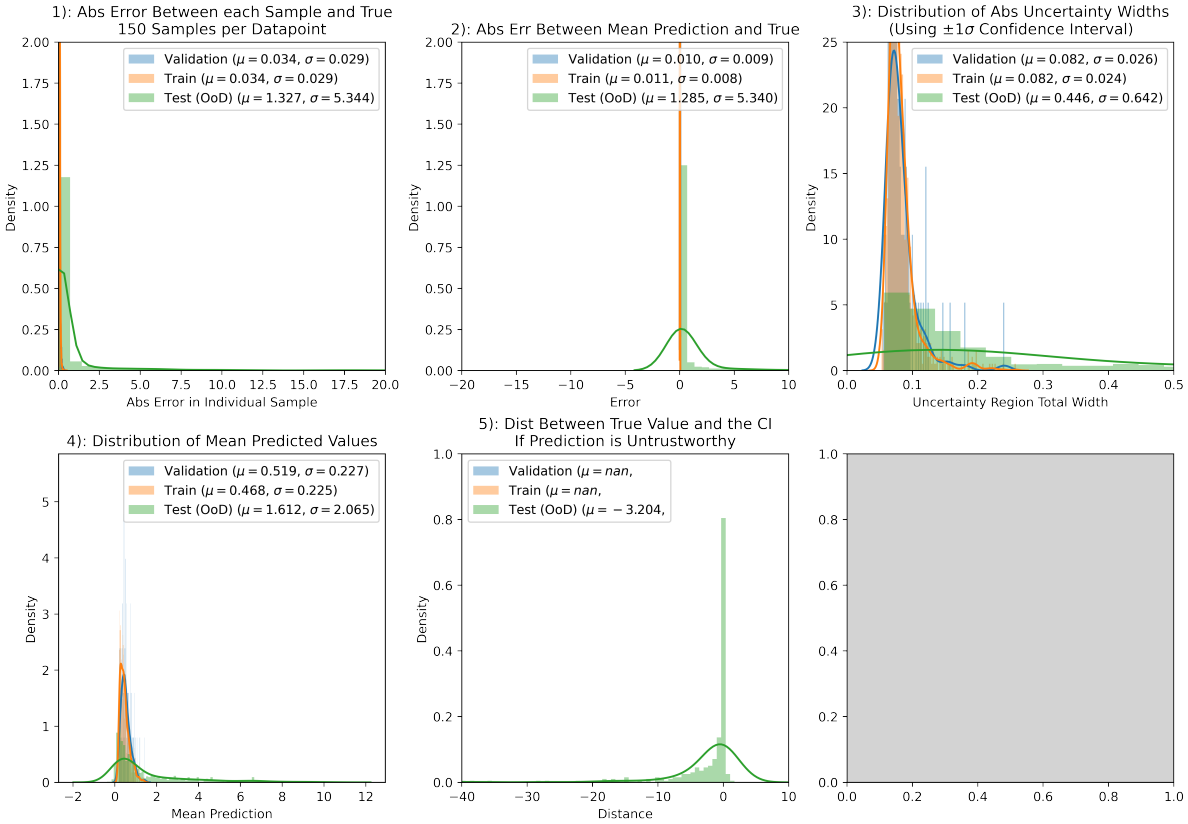
Appendix C Summary Plots Showing Accuracy and Uncertainty of Methods

Figure 9 shows a variety of analyses. Considering Figure 9(a), each subplot represents:

1. The error between each prediction sample and the true value. This outlines the accuracy of the predictions, and the spread of the samples.
2. Very similar to Plot 1, except looking at the mean of all samples drawn for each datapoint. This outlines the accuracy of the predictions.
3. Shows the absolute width of the confidence interval of every data point in each set. This outlines the uncertainty of the predictions.
4. Shows the distribution of the predicted values in each dataset. This outlines the performance of the model as well as describing the dataset.
5. If a confidence interval does not capture the true y value, then how far outside the confidence interval is the y value?



(a) MC Dropout - original training



(b) MC Dropout - retraining

Figure 9: Analysis of MC Dropout after first and second rounds of training. Comparing on the same datasets in both instances.