Master's thesis

NTNU
Norwegian University of Science and Technology

Adrian Thomas Bruland

Manual Hyperparameter Tuning for Optimal Regression Uncertainty Estimates in Bayesian Deep Learning

Master's thesis in Physics and mathematics
Supervisor: Jo Eidsvik
April 2021

NTNU
Norwegian University of
Science and Technology

Adrian Thomas Bruland

Manual Hyperparameter Tuning for Optimal Regression Uncertainty Estimates in Bayesian Deep Learning

Master's thesis in Physics and mathematics
Supervisor: Jo Eidsvik
April 2021

Norwegian University of Science and Technology

NTNU
Norwegian University of
Science and Technology

Abstract

Artificial neural network models have been popular in diverse applications lately. The predictive abilities of these approaches have been extremely promising. Yet, it has not been easy to construct reliable uncertainty statements related to these network model results. There is hence ongoing work in formulating artificial neural network models in a Bayesian context, where the posterior distribution would enable coherent uncertainty quantification.

In this thesis, a Bayesian neural network model called Bayes-by-Backdrop is studied. The model differs from standard feedforward neural networks in that point estimates for a neuron's weights and biases are replaced by a full prior distribution, allowing for the application Bayesian methods. The implementation uses spike-and-slab priors, a Gaussian mixture with the same mean for each component distribution. Variation Inference is used to arrive at a posterior distribution of the network's predictions. In an experimental setup for tuning different hyperparameters, the implementation of the Bayes-by-Backprop algorithm is tested and investigated on a univariate nonlinear regression task including a sinusoidal signal and Gaussian noise. This class of algorithms utilizes Bayesian formalism to create a systematic and theoretically well-founded way of estimating data variability in the prediction. However, the uncertainty estimates of the algorithm is here shown to be very sensitive to hyperparameter settings, as well as the sheer number of data points.

The number of hyperparameters in an ANN is usually large. Doing a grid search of all hyperparameters is prohibitive in most cases. Instead, deep learning practitioners often manually try out one hyperparameter at a time, see what it does, and then select a small subset that seem promising for further tuning. However, tuning the centre and spread (mean and variance) of the variational posterior cannot be entirely decoupled in the Bayes-by-Backprop algorithm.

Even so, the basic strategy hyperparameter tuning strategy applied in this thesis tries to simplify the process. Find a good setting for the posterior mean, then tune the hyperparameters of posterior spread. For posterior variance tuning, the neural network width is shown to be a more stable dial than the variance of weight priors.

The thesis briefly goes over theory and background, then presents and discusses experiments. Finally, results are summarised and contextualised.

Oppsummering

Kunstige nevrale nettverksmodeller har vært populære i forskjellige applikasjoner i det siste. De prediktive evnene til disse tilnærmingene har vært ekstremt lovende. Likevel har det ikke vært lett å lage pålitelige usikkerhetsestimater knyttet til disse nettverksmodellresultatene. Det arbeides derfor kontinuerlig med å formulere kunstige nevrale nettverksmodeller i en Bayesisk kontekst, der posteriorfordelingen vil muliggjøre tolkbar usikkerhetskvantifisering.

I denne oppgaven studeres en Bayesiansk nevralnettverksmodell kalt Bayes-by-Backdrop. Modellen skiller seg fra standard "feedforward" nevrale nettverk ved at punktestimater for et nevrons vekter erstattes av en fullstendig prior-distribusjon, slik at Bayesianske metoder kan brukes. Implementeringen bruker spike-and-slab priors, en gaussisk blanding med samme gjennomsnitt for hver komponentfordeling. Variasjonell inferens brukes til å komme til en posterior-fordeling av nettverkets prediksjoner. I et eksperimentelt oppsett for tuning av forskjellige hyperparametere blir implementeringen av Bayes-by-Backprop-algoritmen testet og undersøkt på en univariat ikke-lineær regresjonsoppgave med et sinusformet signal og Gaussisk støy. Denne klassen av algoritmer bruker Bayesiansk formalisme for å skape en systematisk og teoretisk velbegrunnet måte å estimere datavariabilitet i spådommen. Imidlertid er usikkerhetsestimatene for algoritmen her vist å være veldig følsomme for innstillinger for hyperparameter, så vel som det store antallet datapunkter.

Antall hyperparametere i et ANN er vanligvis stort. Å gjøre et rutesøk av alle hyperparametere er i de fleste tilfeller uoverkommelig. I stedet prøver man ofte manuelt et hyperparameter om gangen, ser hva det gjør, og velger deretter en lite delmengde som virker lovende for videre innstilling. Imidlertid kan innstilling av sentrum og spredning (gjennomsnitt og varians) av den variasjonelle posterior ikke helt frakobles i Bayes-by-Backprop-algoritmen.

Likevel prøves den grunnleggende strategien for hyperparameterjustering som er brukt i denne oppgaven å forenkle prosessen. Finn en god setting for posterior-gjennomsnittet, og still deretter inn hyperparametrene for posterior-spredningen. For posterior variansjustering er nevrale nettverksbredder vist å være en mer stabil hyperparameter for tuning enn variasjonen av parameterenes priorfordeling.

Oppgaven går kort over teori og bakgrunn, og presenterer og diskuterer eksperimenter. Til slutt blir resultatene oppsummert og kontekstualisert.

Preface

This thesis concludes my years as a student with NTNU - an era coloured by broadening horizons, challenges overcome, friendships gained and the joy of learning. I have become part of community of so many talented an intellectually curious people, and I feel thankful to have had the experience.

In writing this thesis, the guidance and encouragement I have received from my supervisor, Jo Eidsvik, have been irreplaceable. Without him, this thesis would likely have never been completed. My greatest thanks to Jo for all his help along the way, and for many an engaging conversation.

The support from my family, and from my girlfriend, Yvonne, I could not have been without. I feel so grateful to have them in my life.

Contents

1 Introduction **8**

2 Artificial neural networks **10**
2.1 Background on regression 10
2.2 Neural Networks Models 12
2.3 Optimization, backpropagation and inference methods 15
2.4 Hyperparameter Tuning 18

3 Bayesian neural networks **19**
3.1 Bayesian View 19
3.2 Variational Bayesian inference in neural networks 21
3.3 Practical elements on implementation 22
3.4 Metrics 24

4 Procedure/experiments **25**
4.1 Background 25
4.2 Univariate nonlinear regression 28
4.3 Uncertainty estimation 45

5 Closing remarks **46**

Bibliography **48**

1 Introduction

The fields of deep learning and machine learning (ML) have seen a lot of development over the last decade, with many new applications in technology and science, as well as impressive advances in high-profile fields, such as a superhuman performance in the game of chess, contributing largely to the hype around the term artificial intelligence (AI). Such advances easily capture the attention and imagination of the public, in part because they have an easy-to-measure metric of whether a system/algorithm is superhuman or not - chess only has one winner and one loser, there are no in-betweens, no fuzzy logic, no qubit-like uncertainty in the state of the outcome (although draws can occur in chess).

However, there are many "fuzzy" tasks where we would like to be assisted by computers, where the answer we are looking for is not merely "yes or no". This is certainly true for some of the most financially and socially profitable applications of AI: tumour detection and classification in cancer research and medicine now performs at a superhuman level. Strides have been made towards self-driving cars, as one can now let cars drive on auto-pilot on highways. AI algorithms are being given an increasingly greater degree of autonomy over their own work, along with more influence over the lives of millions of people, whether through Netflix and Amazon recommender systems, Google advertisements tailored to each user, and even Chinese surveillance infrastructures used to monitor and calculate the value of each citizen in the eyes of the authorities. Some believe that AI will have an increasingly greater social impact in the coming decades. Deep learning is now a billion dollar industry, yet the theory underlying it is somewhat underdeveloped. The urgency in inventing ever new methods for solving increasingly complex challenges has pushed research communities into a largely trial-and-error based methodology, where the focus is on producing good results in applications at any cost. Developing a solid and far-reaching theoretical foundation that tell us exactly what ML and AI can and cannot do under various constraints has fallen to the wayside, and may be impossible.

The main workhorse of modern, high-profile AI is the Artificial Neural Network (ANN), a form of ML algorithm which can infer a layered representation of increasingly complex or abstract features in a process called deep learning. The word "deep" refers to the number of representational layers in a network; if it has more than two, the network is said to be "deep". Being provided a data set or an interactive environment, this form of AI can "learn", or infer, patterns that are too complex for humans to be able to hard-code into an algorithm. This self-inferring trait is part of what gives ANNs their flexibility and appeal - unlike earlier forms of AI, it does not rely on meticulous, time consuming attempts to convert human domain knowledge into hard-coded features. Instead, data is fed into the input part of the algorithm, and out comes impressive results in a much shorter time. However, because we do not easily see what happens to the data inside the algorithm, it is called a "black box" model, and this trait is a substantial challenge to model interpretability and accountability. The ANN simply attunes itself to any pattern

that exists in the data set, with no regards for whether human biases are coded into it. For this any many other reasons, the predictions of an ANN should be met with healthy scepticism.

If the ANN is provided with a data set with covariates and responses, it is said to perform supervised learning, while a data set with only covariates requires unsupervised learning. If instead the ANN is given the task to maximise some benefit in an interactive environment, it performs reinforcement learning. Such an algorithm could play chess, control a robot, or even monitor and tweak the traffic lights system in a city in order to optimize traffic flow. There are many forms of ML models besides ANNs, such as trees, support vector machines and K-nearest-neighbor, which tend to perform at least as well as ANNs on many simpler tasks, and they are usually much faster and less memory costly to fit, but ANNs tend to have advantages in certain situations. Among the most popular deep learning methods, ANNs are regularly used in several practical problems related to regression and classification tasks, see e.g. overview papers by [1] and [2]. In this work, the focus will be on regression problems, where the goal is to predict a response on the real line from covariates. The standard linear regression model is often too simplistic in complex data types, and it is rarely possible to directly extract transformed features in this context. With the abundant software that exist today, it is relatively easy to use ANNs to conduct non-linear regression tasks in such cases.

Getting uncertainty estimates for the results of an ANN is a central goal in numerous applications. One of the most important AI applications may be medicine, since many decisions in the health sector can be made in a better way when given improved probability estimates of different outcomes. This is true for each individual doctor all the way up to central authorities making decisions on public health for an entire country. Take tumour identification and classification as an example: Suppose a patient has an MRI/CT scan to reveal any tumours. An ANN looks at the scan, and gives a ”yes/no” answer as to whether there is a tumour in the image. If it says ”yes”, what is then the probability that the patient has a tumour? Is it 51% or 99%? What if it says ”no”; is there a 49% or a 1% probability of a tumour being present? Clearly, having a detailed probability interface would be a lot more informative for the medical practitioner, rather than being given only a ”yes/no” answer - this is likely true for many questions that show up in a treatment process. Access to accurate, reliable uncertainty estimates may have a large impact on major decision making in patient care.

Another example, in a reinforcement network setting: Let us say that AlphaGo finds 2 potential moves, labelled I and II. Move I has a higher expected increase in probability of a win, but also higher variance - i.e. it is a high risk, high reward move. A regular ANN can only say something quantitative and substantial about the probability of winning by looking at its past training history, which may involve playing against itself thousands or millions of times.

Other interesting applications where uncertainty statements are critical include self-driving cars, city planning, finance, corporate decisions, artistic/aesthetic uses, fluid mechanical simulations that rival state-of-the-art numerical methods, and so

much more. Some of these applications are "high-profile" in the sense of being the subject of media hype, yet other, less known uses may still prove to have a greater substantial impact on society than the hype cases. Arguably, uncertainty estimates could prove important in any ANN application that has a bearing on decision making with consequences for things humans value.

As it turns out, there are several ways of including uncertainty/stochasticity in a "standard" ANN setup, but it has not been straightforward to understand the properties of these methods. See e.g. the overview by [3]. Fast approaches use stochastic inclusion or deletion (dropout) of components in a standard architecture, random initialization or components in the stochastic gradient-based optimization. One naïve way of doing ANN uncertainty is then simply to treat each fit as a simulation, and use the set of simulations as an empirical distribution, whence can be found some uncertainty measures. In a regression setting, finding standard deviation and confidence intervals are natural candidates. Including a probabilistic model for the weights in the network structure, can facilitate the uncertainty quantification, moving beyond simple drop-out approaches. In a formal Bayesian setting with a prior model for the weights, one can address the posterior distribution of the weights in the network. Albeit easily done in theory, it is not obvious how to analyze this posterior distribution in practice, and several approximations have been suggested. This thesis will investigate the Bayes-by-Backprop (BBB) algorithm, which is a version of the variational inference (VI) approach. The algorithm used in the experiments is an implementation of BBB, as described in [4].

The thesis is structured as follows: In Chapter 2 we introduce the key components of the ANN model in the regression setting. This chapter hence sets the notation and terminology used in the later chapters. We further explain elements related to the network training, which involves the estimation of parameters in the ANN model as well as hyperparameter and architecture tuning. In Chapter 3 we present the Bayesian view of ANNs. After a brief historical background and motivation, we describe the core elements of the BBB methodology relying on VI methods for fitting the posterior distribution of the ANN weights. The criteria we use for comparison of different setting of the BBB approach includes the Mean square error (MSE) and rank statistics with tests such as the Kolmogorov-Smirnov (KS) statistic. In Chapter 4 we have simulation studies in a regression problem with a sine function, where the particular focus is on tuning the hyperparameters of the BBB reliably. In Chapter 5 we conclude with a summary of the main findings of this work and point to further work.

2 Artificial neural networks

2.1 Background on regression

Regression is a method for predicting a random continuous response variable $y_i \in \mathcal{R}$ as a function of explanatory variable vector $\boldsymbol{x}_i = (x_{i1}, \ldots, x_{ip}) \in \mathcal{R}^p$, where $i = 1, \ldots, n$ denotes the index of observations and p denotes the number

of covariates. Sometimes the explanatory variables are alternatively referred to in statistics as 'factors' or 'covariates', or in ML literature as 'features'. In a regression model, we represent the response as being generated by deterministic function of the explanatory variables $\boldsymbol{x}_i$ with some additive noise;

$$Y_i = f_{\boldsymbol{w}}(\boldsymbol{x}_i) + \varepsilon_i, \quad i = 1, \ldots, n, \tag{1}$$

where $\boldsymbol{w}$ denotes unknown model parameters. In most settings, the noise terms, also known as the errors of the response variable, are considered to be independent and identically distributed Gaussian variables with zero mean and constant variance σ_ε^2 so that

$$\varepsilon_i \sim N(0, \sigma_\varepsilon^2), \quad i = 1, \ldots, n, \quad \text{independent.} \tag{2}$$

In ML, inference is commonly referred to as "training" a model. The data used in the training is:

$$\mathcal{D}_{\text{train}} = \{(y_1, \boldsymbol{x}_1), (y_2, \boldsymbol{x}_2), \ldots, (y_n, \boldsymbol{x}_n)\}.$$

The goal is in most settings to train the functional relation in equation (1), given this data. When testing the approach, one often uses a hold-out or test data set. To understand the properties of the methods. This validation set will be denoted by $\mathcal{D}_{\text{valid}}$. Given new explanatory variables $\boldsymbol{x}_0$ (in the validation set) one can then predict the associated response through the trained functional relation:

$$E(Y|\boldsymbol{x}_0) = \begin{cases} \hat{f}(\boldsymbol{x}_0), \\ f_{\hat{\boldsymbol{w}}}(\boldsymbol{x}_0). \end{cases}$$

Here, the training works on the function form f directly (top row), or on parameters $\boldsymbol{w}$ in a pre-determined functional form for f, defining a family of mappings of which $f_{\hat{\boldsymbol{w}}}(\cdot)$ is an instantiation. Other goals may be finding the median or other quantiles of the response for the specified covariate, as is done in quantile regression [5]. Yet another goal is to characterize the uncertainty of the prediction to a reasonable level as we will discuss further in the next section.

The main question in regression analysis is the selection of the function f in equation (1). A common choice is that of linear multiple regression analysis where one can have

$$f_{\boldsymbol{w}}(\boldsymbol{x}_i) = w_0 + x_{i1}w_1 + \ldots + x_{ip}w_p. \tag{3}$$

This can be rather flexible even though it has a simple linear form. In particular, there are many possibilities in using derived features or transformed covariates as the explanatory variables $\boldsymbol{x}_i$, such as products or logarithms of x_{ij} with i constant. Also, the estimation of parameters is trivially done by the method of least squares, assuming the number of observations n is larger than the number of model parameters $p + 1$, that is the number of features along with the constant term w_0.

Despite its abundant use in practice, the linear regression model in equation (3) has limited ability to train relations in complex multivariable settings, and this

has motivated the search for more complex and nuanced functions such as neural networks. In particular, it is only natural to assume that any given dataset may have been created by a different, more complex process than what is assumed by a linear regression model. In this one is in a case of imposing assumptions onto a data set which are untrue, but which still serves the purpose of simplifying analysis and providing insights about the data. As we shall see, ANNs can be viewed as iterated polynomial regression. The fact that ANNs in principle have no known limits to what functions it can estimate, or how accurately, shows that regression has a great deal of flexibility in what models it can represent.

As touched upon, statisticians and computer scientists have somewhat different nomenclatures in reference to mathematical/probabilistic modelling. Covariate and response variables are known in the ML literature simply as "inputs and outputs", or even "features and labels", respectively. The ML language is a bit less rigorous. E.g. it does not distinguish between random variables and their values - both are referred to as "labels". This is not a huge problem, since the ML literature is so focused on applications - one usually knows what a researcher is referring to from the context.

2.2 Neural Networks Models

Neural networks in the human brain inspired the ANN term to describe computer-generated algorithms where quantitative data are flowing along network edges in a system with connected layers.

The ANN model can be regarded as a regression model where the features are derived by non-linear functions of linear combinations of covariates. This can occur via many layers of variables and with different number of variables per layer, see Figure 1. There exists myriads of versions of ANNs. In a simple setting with one hidden layer, consider feature variables $\boldsymbol{z}_j$, $j = 1, \ldots, K$ as an intermediate variable. Each of these features are activated by linear combinations of input covariates. Then the features take part in a linear predictor for the expected response. In mathematical terms, we then have:

$$\begin{aligned} z_{ij} &= \sigma(w^1_{0,j} + \boldsymbol{w}^1 \boldsymbol{x}_i), \quad j = 1, \ldots, K, \\ g(\boldsymbol{z}_j) &= w^2_0 + \boldsymbol{w}^2 \boldsymbol{z}_j, \end{aligned} \tag{4}$$

where the weights at level l are denoted by $\boldsymbol{w}^l_\cdot$ and the activation function $\sigma(r)$ can be selected. A common choice is the sigmoid function $\sigma(r) = 1/(1 + e^{-r})$. The final output function $f(\cdot)$ is now a combination of functions (σ and g in this simple example) going between the different layers, where one function is input to the next.

The term "weights" of one layer l refers to all parameters w^l_i that are multiplied by the output from the previous layer, while "bias" refers to the constant term w_0. (Confusingly, "weights" may also refer to both of these - we will refrain from this usage and distinguish between weights and biases.) The unknown nonlinear

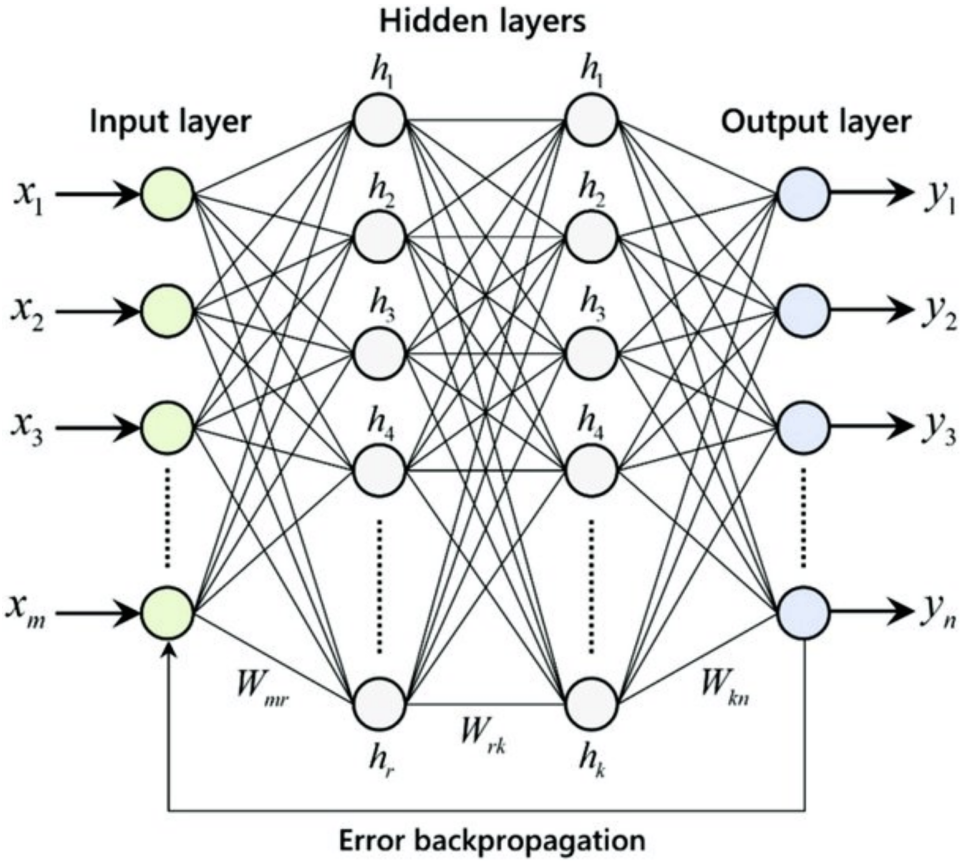


Figure 1: The Feedforward Neural Network. The system of edges going out from layer h_i multiplied with weight matrix $W_{i,i+1}$, added to bias vector $\boldsymbol{b}$, and finally passed to the next layer h_{i+1}. Biases are excluded from this graphical setup. From [6].

regression parameters are here then $\boldsymbol{w} = (\boldsymbol{w}^1, \boldsymbol{w}^2)$ in this simple illustration of an ANN model. With many layers in the ANN, there are plenty parameters that must be specified in this representation, and one often needs much training data to estimate them all. Note that the special case of multivariate regression is a type of feedforward neural network with activation equal to identity, $\sigma(r) = r$, in all layers. The response is a linear combination of the covariates. In general, the output could be multivariate as well. In this work, however, the scope is limited to having one output response, which is the simplest special case of the rightmost part of Figure 1.

Even though the scope is limited in this work where the focusing is on studying the robustness of parameterization in the ANN models for a nonlinear function prediction, it is worthwhile just touching the large-scale situation. How many covariates and responses can an ANN process? Figure 3 provides an example using

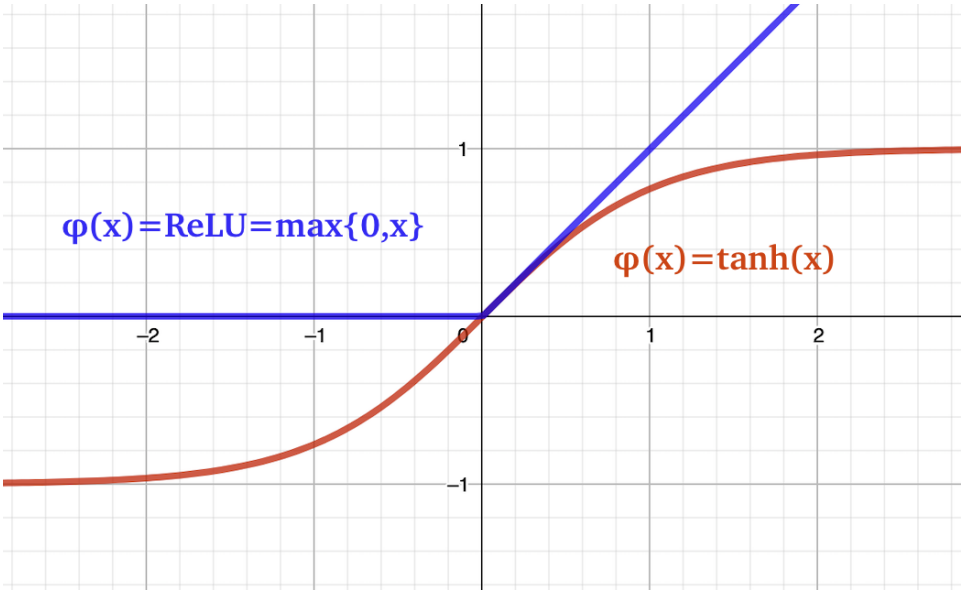


Figure 2: Commonly used activation functions: ReLU and the tangens hyperbolicus $\tanh(\cdot)$. The first shoots off to infinity with increasing input, while $\tanh(\cdot)$ has horizontal asymptotes in $y \in \{-1, 1\}$.

the VGG-16 CNN, which can classify images into upto 1000 classes. VGG-16 accepts a 224x224 image, which is turned into a 224x224x3 tensor. The tensor depth of 3 corresponds to the image's three color channels. This tensor is run through convolutional and max pooling layers, the latter of which reduces the tensor height and breadth. The last max pool returns a 7x7x512 tensor, which can be viewed as $7 \cdot 512 = 3584$ vectors, each with a length of 7. The flatten operation concatenates all of these vectors, producing one long vector with dimensions 1x1x25088. This forms the input of a fully connected feedforward neural network sporting 2 ReLU layers, each with 4096 neurons, finally producing 1000 output layer activations, the largest of which is the predicted class. Hence, the point of this digression: in theory, ANN's can handle arbitrarily large input and output vectors, enabling their state-of-the-art performance on high-dimensional problems involving complex statistical relationships.

One downside of ANNs is the difficulty in the interpretation of results. Unlike the linear regression model or other generalized linear models, there is no direct explicit way of relating a weight to the factor influence on the response. For linear regression one could look directly at the slope estimates in equation (3), while common logistic regression models use odds ratios to interpret estimated effects. However, for ANNs, the response is a highly complex relation of the inputs, which may be connected in inherently hard-to-interpret ways. Recent approaches attempt to look at ideas from sensitivity analysis to address these challenges of interpretability [8], but this remains a challenge for deep learning methods overall.

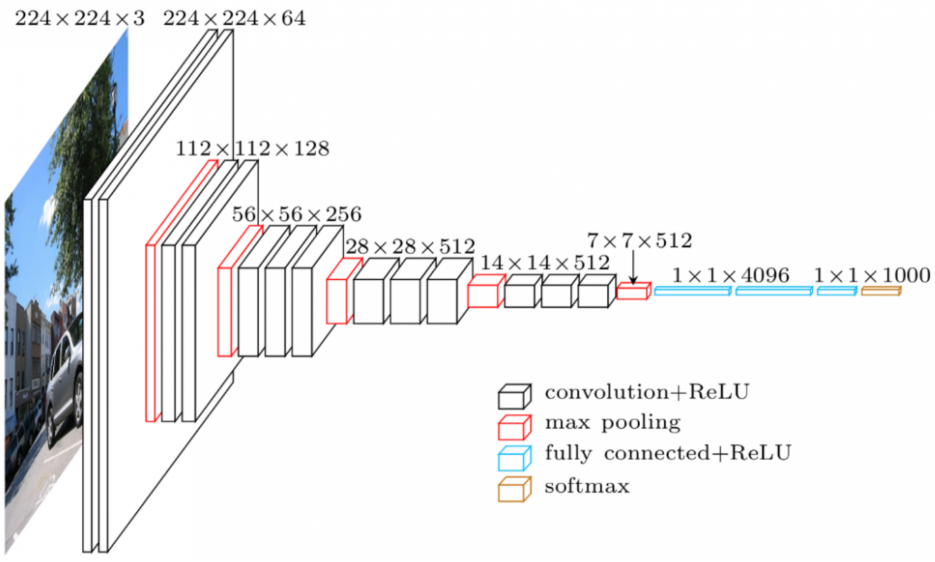


Figure 3: Model architecture of the VGG-16 CNN, an image classification ANN. Figure from [7]

2.3 Optimization, backpropagation and inference methods

While the coefficients in a multiple linear regression model can be estimated directly using the method of least squares, the challenge of specifying the weights in an ANN is not that straightforward. A common conception about ANNs is that it learns parameters from data through ”trial and error”. This statement can lead to the misconception that an ML algorithm tests out, say, M randomly selected hypotheses, and then represents the best hypothesis. Such a procedure would be much too inefficient - ANNs can have as many as 150 billion parameters, each of which can hold any real value in principle. It stands to reason that a search policy of the space of all hypotheses based solely on random selection will be extremely inefficient, and would be prohibitive in projects such as learning to play chess at a superhuman level, which takes millions of self-play sessions even with state-of-the-art optimisation techniques.

Instead of trying out parameter permutations randomly, the estimation procedures associate with ANNs strategically use what is learned from the current hypothesis to select the next candidate hypothesis, so as to arrive at an optimal solution as fast as possible. This sounds impressive, but is simple to express in programming and mathematical terms: choose a loss function $L(\boldsymbol{w})$ (also known as ’objective function’, ’cost function’) and an optimization function, and the algorithm will ”walk its way”, i.e. iteratively move, through the parameter space according to the optimizer’s policy, and try to fit a good model, i.e. a model with a minimal loss value between the fitted model’s predictions and the labels.

As such, the goal of the hypothesis search can be stated as finding parameter

vector $\boldsymbol{w}$ such that the loss is minimized:

$$\hat{\boldsymbol{w}} \in \underset{\boldsymbol{w}}{\operatorname{argmin}}\, L(\boldsymbol{w}). \tag{5}$$

The objective function $L(\boldsymbol{w})$ used in the current work is that of a negative log likelihood, assuming Gaussian error terms. Notably, this will later be modified when we introduce the Bayesian setting with a prior density function on the weights, which can be regarded as a way of regularizing the objective function. Alternative loss functions include cross-entropy and others [9].

This problem statement is easy to follow in theory, but is a central challenge in ML which has attracted a large portion of current and past literature. Parameter optimization in ANNs is usually non-convex, and it can be very high-dimensional. It also varies a lot from one problem to the next - a small change in the loss function or the network structure can lead to very different parameter estimates. The objective function is optimized iteratively, where one for each epoch strategically selects each new hypothesis based on what it learned from the previous hypothesis. This is done using a gradient optimization algorithm (though some optimizers are not gradient based). The iterative optimization is truncated when there is no more increase in the objective function for several epochs. The nonlinear challenges in the ANNs is often recognized when plotting the objective function as a function of the epochs. Quite often, it can appear as if one has reached convergence, but then the objective function suddenly drops further. This just illustrates that the objective function is very difficult to optimize.

In Figure 4 the main elements of the backpropagation routine are shown. This is used as an important step in the training of ANNs, where the goal is to find the weights that minimize the objective function. The derivates are then computed based on kernel derivatives that combine to get the desired result.

The objective function surface can look very ugly. An illustration is shown in Figure 5. It is of course extremely difficult to optimize such surfaces on the computer, no matter what technique is used.

The theory of ANNs is not very well developed. This is seen in the ML literature, which very often relies on a trial-and-error procedure and lots of experimentation to arrive at important results. One important theoretical baseline is the Universal Approximation Theorem, which ambitiously states that 'A neural net can do anything'. This result is impressive, since it implies that there is no task that ANNs, in principle, cannot perform, as long as one can somehow arrive at a good value for weights-and-biases vector w. However, it only holds true for infinitely wide ANNs. Real applications have two hard restrictions that limit the capabilities of ANN's: layers must have finite width, and we cannot try out all possible parameter vectors, meaning we have to do some kind of search to find a good hypothesis. Still, the representation and hypothesis space of a large (wide and/or deep) ANN can be huge. Because an ANN relies on numerical optimization to try to find the best solution in this parameter space (i.e. search the argmin of the loss function), there is always a chance that the optimization procedure will find its way to a lo-

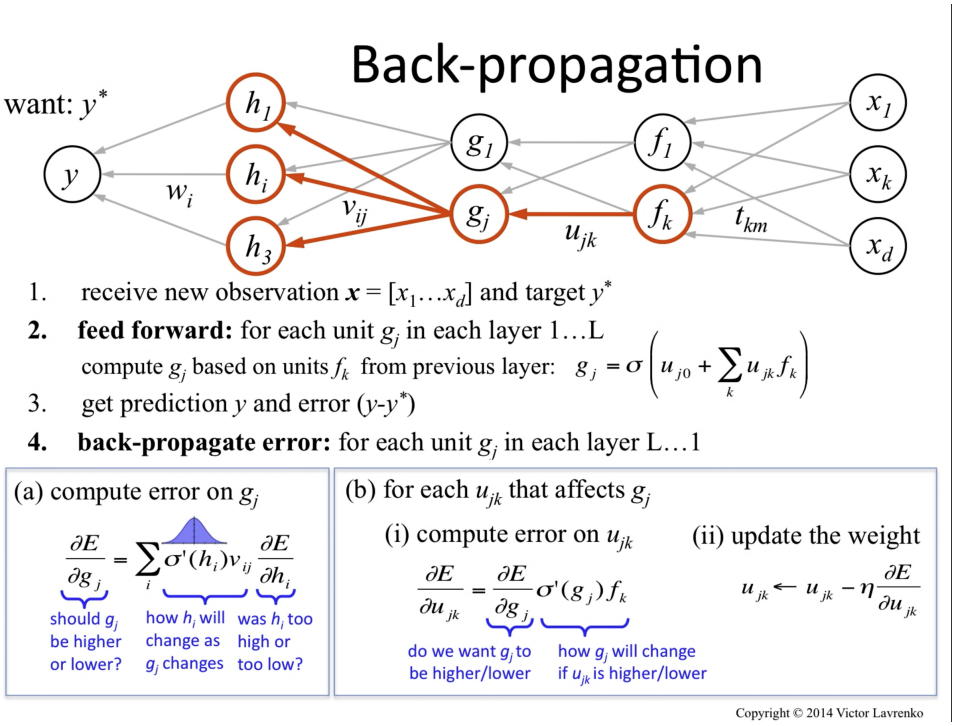


Figure 4: The Backpropagation Algorithm. Here, the input layer is at the right-hand side. A batch x-values has been passed forward, and the loss $L(y * -y)$ has been calculated. The derivative of L with respect to a weight/bias w_j^k will come out as an iterated chain rule, since each neuron activation a_i^k depends on the previous layers' activations a^{k-1}, which in turn depends on a^{k-2}, hence the notion of the loss being propagated "backwards" in the ANN. From youtube.com/watch?v=An5z8lR8asY

cal minimum on the optimization surface which does not satisfy the intent of the practitioner. The large size of the hypothesis space means we can assume that overfitting will be a central tendency in unregularised ANN's. Experience supports this notion: there are lots of add-ons and a large ML literature focused on how to avoid overfitting. However, if regularisation is too strong, it will lead to underfitting. Hence ANN applications often involve some kind of balancing act between too much and too little regularisation. The strong focus on avoiding overfitting has implications for the optimization procedure as well; an ML practitioner ultimately wants not to find the best parameter setting for the training set, but for the test set, leading to practices like early stopping, and more.

There are by today a number of possibly tools to stabilize the optimization procedure, including regularization, batches, batch and weight normalization, weight initialization, etc. [9]. We will focus on presenting the main ideas required for what follows.

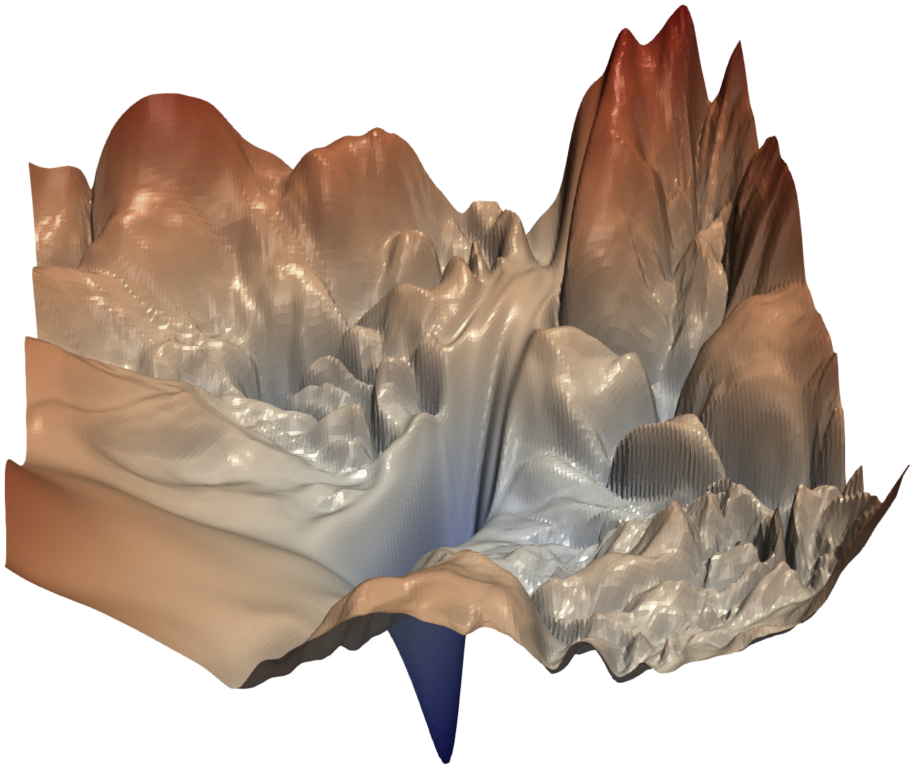


Figure 5: Example optimization surface in a backpropagation setting, plotted with respect to two learnable parameters. Efficiently finding the global minimum of such as surface forms a considerable engineering challenge, particularly when the number of parameters is large.

2.4 Hyperparameter Tuning

We do not give a full detail over what non-learnable parameters the ANN models have here. As we brought up briefly in the previous section, there are a number of switches and dials to turn, to make it work in practice. Even though this has become an art that often required much experience, there are still much room for experimentation to see what settings might work better than others.

This thesis uses the optimizer called ADAM - the Adaptive Moment (or Adaptive Moment Estimation). This algorithm arose from the machine learning community after a decades-long process of development in the field of gradient-based optimization. One can find a starting point for this field in the Gradient Descent algorithm proposed by Cauchy in 1847 [10]. Gradient descent is relatively slow and unstable, so ML researchers developed mini-batch gradient descent, which randomly selects a subset of the parameter over which to optimize in one step (or epoch, in the case of ANN's.) Adding acceleration and momentum parameters to each learnable parameters gives ADAM.

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

Figure 6: The ADAM optimizer algorithm. Using gradient and Hessian to infer the direction and magnitude of the next step in the parameter space, ADAM assigns separate parameters to each of the parameters in a batch. Image from [9].

The following inputs or hyperparameters must often be tweaked in an ANN training exercise:

- Epochs.
- Architecture (Layers, neurons, activation function).
- Learning rate.

3 Bayesian neural networks

3.1 Bayesian View

The Bayesian approach is generally applicable to most statistical inference tasks. By treating the slopes in the linear regression model in equation (3) or the weights in equation (4) as random variables, we turn the optimization problem into a challenge of assessing a probability distribution for the parameters of interest, given the data.

There are at least two reasons for studying Bayesian methods in the context of ANNs. First, the Bayesian viewpoint enables a natural regularizer for the weights in the ANN structure, and this can lead to faster convergence and more robust structures that limit the risk of overfitting to the training data. There are also several other regularizers in the literature, but most of these can be transformed to a Bayesian setting with a more or less natural prior distribution for the weights. Second, the Bayesian viewpoint naturally leads to uncertainty quantification in the sense that there is not just one prediction with fixed network weights but infinitely many predictions when marginalizing over the distribution of the network weights. On this aspect, the standard optimization approaches have come up with a number of ad hoc solutions such as random drop-out of edges or simple addition of the training error in the data space. The Bayesian setting in principle allows for much more nuanced probabilistic statements in this context and has proven highly effective. Bayesian methods for neural networks have been around for a long time [11]. And [12] presented a Bayesian solution of an ANN model for the NIPS competition, which outperformed all competing suggestions. A recent discussion of Bayesian approaches to ANN analysis is provided by [3].

The Bayesian approach assigns a prior distribution to the weights in the ANN. We let $p(\boldsymbol{w})$ denote the prior density function of the weights. A network model as in equation (4) and Figure 1 defines the likelihood model for the response variables $\boldsymbol{y} = (y_1, \ldots, y_n)$. The resulting density function (likelihood) for the response, given the weights, can be represented by $p(\boldsymbol{y}|\boldsymbol{w})$. Bayes' theorem now states that the posterior density function for the weights is defined by

$$p(\boldsymbol{w}|\boldsymbol{y}) = \frac{p(\boldsymbol{w})p(\boldsymbol{y}|\boldsymbol{w})}{p(\boldsymbol{y})} \propto p(\boldsymbol{w})p(\boldsymbol{y}|\boldsymbol{w}). \tag{6}$$

Similarly, Bayes' theorem naturally leads to a marginalization over the trained ANN weights. Assume one aims to predict a new response y_0. The frequentist approach would then plug the predicted weights into the functional model; $f_{\hat{w}}(\boldsymbol{x}_0)$, where the ANN model which is here described by f. In the Bayesian setting the solution is to marginalize over the posterior distribution for the weights:

$$p(y_0|\boldsymbol{y}) = \int p(\boldsymbol{w}|\boldsymbol{y})p(y_0|\boldsymbol{w})d\boldsymbol{w}. \tag{7}$$

In practice, the integral in equation (7) is often difficult to solve. An obvious solution is that of Monte Carlo sampling from the posterior density for the weights, and then predicting y_0 for each weight sample. We then have

$$p(y_0|\boldsymbol{y}) \approx \frac{1}{N}\sum_{j=1}^{M} p(y_0|\boldsymbol{w}_j), \tag{8}$$

which represents a mixture distribution over the predictive distributions for the N sampled weights $\boldsymbol{w}_j \sim p(\boldsymbol{w}|\boldsymbol{y})$, $j = 1, \ldots, N$.

Yet, the sampling from the posterior distribution of the weights is not a straightforward task. One possible solution is provided by Markov chain Monte Carlo (MCMC) sampling, see e.g. [13]. Here, the sampled weights form a Markov chain with limiting distribution equal to the posterior in equation (6). Hence, this provides fantastic opportunities for Bayesian inference in ANNs: the implementation is often relatively easy and the convergence is achieved under very weak regularity condition on the proposal density. In their NIPS work on Bayesian methods for ANN, [12] used MCMC sampling to explore the posterior distribution of the ANN weights.

The challenge with MCMC sampling is related to the often difficult practical exploration of the sample space of the posterior. In large-size applications the MCMC sampler often requires tedious iteration efforts to ensure convergence of the iterative Markov chain simulation. Especially so for difficult posterior surfaces like that of high-dimensional weights that might be hard to separate. For this reason the actual convergence and mixing properties of the Markov chain can be very slow. Hence, the samples in equation (7) become very dependent and could also be biased if convergence is not yet achieved, and it is not always reliable for posterior assessment.

Some researchers feel that Bayesian methods are altogether too slow. Alternative methods for acquiring uncertainty estimates include Evidential Neural Networks

3.2 Variational Bayesian inference in neural networks

In Bayesian inference, variational inference (VI) techniques have gained popularity because they are much faster than MCMC approaches. Unfortunately, it has only rarely been possible to derive the properties of the VI methods and it does not give an exact solution to the posterior, but they have shown useful in practical assessment of weights uncertainty in Bayesian ANNs.

At its core, VI approaches find an approximation $q(\boldsymbol{w})$ to the posterior distribution $p(\boldsymbol{w}|\boldsymbol{y})$. The density $q(\boldsymbol{w})$ is constructed using certain principles, in particular that of minimum Kullback-Leibler divergence between a parametric form of $q(\boldsymbol{w})$ and the actual posterior. This divergence can be phrased as:

$$D_{KL} = \int q_{\boldsymbol{\theta}}(\boldsymbol{w}) \log \frac{q_{\boldsymbol{\theta}}(\boldsymbol{w})}{p(\boldsymbol{w}|\boldsymbol{y})} d\boldsymbol{w}. \tag{9}$$

The parametric form of $q_{\boldsymbol{\theta}}(\boldsymbol{w})$ is most commonly set to be a multivariate Gaussian distribution, and the specification of parameter $\boldsymbol{\theta}$ is done via minimization of the D_{KL} in equation (9). In essence, this approach then fits a Gaussian approximation to the posterior distribution. In practice, the calculation of equation (9) relies on evaluation of this expression up to a proportionality constant, as in equation (6).

The BBB approach [4] provides a highly practical implementation of the VI approach in this setting of ANN. Here, the approximation to D_{KL} is constructed in a

special way. Rather than working with the weights directly, it applies a transformation of variables, where the weights are a function of the parameter and a random sample; $\boldsymbol{w} = \boldsymbol{t}(\boldsymbol{\epsilon}, \boldsymbol{\theta})$. This enables an efficient implementation of the derivatives in the backpropagation in the ANN, see e.g. [4] or [3]. With its tight connection with backpropagation, one can benefit from good implementations such as ADAM in the BBB steps.

The prior model can be of different types, such as a Gaussian with spike-and-slab priors, which can be implemented in the BBB approach that has been studied here.

3.3 Practical elements on implementation

Since the parameter vector $\boldsymbol{w}$ in a deterministic ANN is swapped for a vector of hyperparameters, calculated from Monte Carlo simulations. This means that the stochastic and simulation aspects of the BBB algorithm cannot be separated from the deterministic one during training. The mean-of-simulations $\hat{y}^j$, $j = 1, \ldots, N$, seen in red in the plots below, is calculated after Monte Carlo sampling has been performed, and cannot be calculated otherwise as the algorithm stands. If one wanted to train the BNN to find the mean $\hat{y}$ alone, the only option would be to create a wholly separate, traditional FFNN (feedforward NN) with the same architecture, now with point estimates on parameters.

We add some comments related to the particular Monte Carlo sampling based BBB implementation which this thesis built upon: Statistical experiments often rely on stochastic modelling, i.e. being able to draw samples from a random variable. In order to create replicable experiments, probabilistic programs rely on an RNG (random number generator) to produce a pseudo-random array of number. The numbers in this array are then fetched consecutively, i.e. in the listed order, whenever a random sample is needed. The seed parameter of the RNG determines the exact list that is produced. Hence, using the same seed value in the same RNG will give exactly the same results, allowing completely reproducible scripts in the fields of probabilistic programming. Experimental reproducibility is obviously an important and sought-after feature of in scientific research. However, seeding not only allows for experimenters to reproduce eachother's results - it is highly useful when developing an ANN model, since two different train-and-predict runs of identical scripts can give vastly different results, especially when training is prone to poor model fitting or producing NaNs. With seeding, one can be fairly certain that a change in results is produced by a change in the code, and not by dumb luck. In order to guarantee this, one would make sure that the similar parts of two scripts are receiving the same RNG number string, but this level of rigour is usually not required. The distribution of the "base" RNG can be uniform. From a uniformly distributed stochastic variable, many well known distributions can be simulated, making a uniform RNG very useful in any program that involve simulation. However, expecting the result to be the same requires seeding.

At the beginning of the experimentation phase for this thesis, the Keras objects

in the code would not allow themselves to be seeded, meaning that the only way to know the behaviour of a piece of code was to run it multiple times. A seed value was provided for random.seed(), numpy.random.seed() and tf.compat.v1. set_random_seed(seed_value) as well as for the os.environ object, which is used to set environment variables. The code being non-seedable had a considerable effect on the experimental setup, since it meant that any change in the outcome of fitting an ANN model and using it to predict could have been caused by pure chance.

On the other hand, the simulation and deterministic aspects are separate in the sense that they rely on different tunable parameters in their entirety, meaning that the mean hyperparameter can be tuned in an separate process from the standard deviation hyperparameter.

Keras with Tensorflow is the chosen framework in with the BBB algorithm implementations have been written. This is a high-level language made for ease of use. Alas, it can have bugs, as evidenced by the numerous bug-fixes Keras:bug-fixes.

A note for newcomers to Python ANN development is that Keras and tf.keras are not the same frameworks. This is to say that Tensorflow has an implemented sub-library called "keras" which is separate from the standalone Keras framework, with different syntax and other implementation details. This is in spite of the fact that Keras is "built on top of" Tensorflow, meaning TF is the backend of the Keras API. Hence, combining syntax from Keras and tf.keras may lead to errors. This fact is seemingly nowhere to found in their respective documentations - most users will have to infer it either from blogs, online fora or personal experience. Most online tutorials are simply too rudimentary to go into this level of detail, and perhaps it does not affect too many people. Still, it can be a bit of a headache to get around.

The Adam optimizer is said to make learning rate tuning obsolete, since it can work with different values. The idea that Adam is entirely agnostic with respect to learning rate is empirically false from the results shown here. A better, more precise wording may be: the Adam optimizer is less sensitive to learning rate, giving good results in a larger learning rate neighborhood instead of only being good for a single value.

One might want for a BNN to train all parameters of the priors in the training process, inferring all traits of the prior from the data in the same way that a standard ANN learns totally from the data. This is not the case in Krasser's implementation - only the mean of the prior is learned from data, the standard deviation is set by the static (non-learnable) parameters σ_1, σ_2.

Essentially, a BNN is a neural network that is able to say "I don't know". In a regression setting, a standard ANN returns a point estimate $E(Y|x)$, while an BNN yields a whole distribution for the posterior, $p(y|x)$. This density holds all the information that can be known about the data conditional on the model, and can be manipulated using any tool in classical statistic to find e.g. confidence distributions, mode, and 1D values like quantiles and mean.

3.4 Metrics

To compare results on different parameter setting in the ANNs we use a number of criteria. Each one is explained briefly in what follows. They are all based on comparison of predictions or predictive distributions for a size T hold-out test dataset.

The mean square error (MSE) is used a measure for the prediction strength of the method. The goal is to have a small mismatch between the prediction and the actual data. The MSE equals

$$\text{MSE} = \sum_{k=1}^{T} (y_{\text{test},k} - \hat{y}_{\text{test},k})^2, \tag{10}$$

where the hold-out data are denoted $y_{\text{test},k}$ and the associated prediction is $\hat{y}_{\text{test},k}$. In a standard ANN, considering a frequentist viewpoint, this prediction would equal the functional representation for the model; $\hat{y}_{\text{test},k} = f_{\hat{w}}(\boldsymbol{x}_{\text{test},k})$. In a Bayesian setting that we focus on here, it would be more reasonable to take the average value

$$\bar{f}(\boldsymbol{x}_{\text{test},k}) = \frac{1}{N} \sum_{j=1}^{N} f_{\boldsymbol{w}^j}(\boldsymbol{x}_{\text{test},k}),$$

extracted from posterior predictive samples in equation (8), where the ANN is used for each sample $\boldsymbol{w}^j$, $j = 1, \ldots, N$.

The goodness-of-fit (GOF) is assessed by comparing the predictive distribution and the observed test data. In our context, we rank the position of the data $y_{\text{test},k}$ among the samples

$$\hat{y}^j_{\text{test},k} = f_{\boldsymbol{w}^j}(\boldsymbol{x}_{\text{test},k}).$$

The rank statistic of this sample is then the order, such that

$$r_k = \frac{1}{N} \sum_{j=1}^{N} I(\hat{y}^j_{\text{test},k} < y_{\text{test},k}), \qquad k = 1, \ldots, T. \tag{11}$$

Here, if the prediction is reasonably good, the distribution of $r_1, \ldots, r_T$ should be more or less uniformly distributed.

A Kolmogorov-Smirnov (KS) test statistic is used to compare whether the uniform distribution is justified. This is based on the maximal difference between the data empirical cumulative distribution and a reference cumulative distribution function. In the current setting, first, the empirical cumulative distribution function for the ranks r_k, $k = 1, \ldots, T$, is computed, and secondly this is compared with the straight line between 0 and 1 which defines the cumulative distribution function of the uniform distribution.

4 Procedure/experiments

Having gone over the required theoretical groundwork, we now turn to experiments. Section 4.1 brings up some general concerns about systematic ANN experiments, and about this Keras BBB setup in particular. The next sections detail fitting of the variational posterior mean to noisy sinusoidal data. Lastly, in Section 4.4 we look at how to fit uncertainty in a reasonable way, by selecting a small subset of hyperparameters for fine-tuning. Results are presented and discussed.

4.1 Background

Implementing and successfully training an ANN is a wholly different endeavor from understanding deep learning in the abstract theoretical case. Software development know-how, critical thinking, patience and "common sense" implementation choice are at least as important as having a deep understanding of calculus, optimization or probability theory. Part of the reason for this is that the theory of ANNs is not at all sufficiently well developed that it can reliably tell a priori which hyperparameter settings will yield a good result - not even for noiseless data sets. This is a very different scenario than e.g. the theory of linear regression, which guarantees convergence under certain constraints. For this reason, one usually has to simply try out a few different settings and see which, if any, yield an acceptable model fit. Libraries such as Keras makes creating an ANN as simple as 10 lines of code. The challenge comes in selecting an ANN model from a large set of different models and hyperparameter settings.

This being said, theory has a lot of use when it comes to processing the data before inputting it to the ANN, and how to interpret/evaluate model's predictions. Probability theory will help us when it comes to evaluating the uncertainty estimation performance of the BBB algorithm. Concepts like rank statistic, quantiles, probability distributions and the KS metric will be central in that regard.

The complexity of the ANN model, its reliance on calculus, matrix algebra and optimization, as well as the thousands of paper that have been produced in the ever-developing field of research, can make the newcomer ask: where to even begin? A new practitioner is well advised to confer online blogs, tutorials or courses, which may condense decades of research into accessible, step-wise and concise programming strategies. Using academic research publications as a primary guideline for creating one's own ANN should only be done after some basic implementation practices have been establishes - starting from the bottom and jumping straight into a very recent paper on advanced ANN technology will simply be too steep of a learning curve.

As illustrated by Figure 5 and **??**, ANN training can be messy, and a model's performance and ability to generalise can be difficult to assess. This also holds for the experiments conducted here, and they do not prove anything generally about the performance of the BBB algorithm applied to DL problems. Notably, if a trained model performs its task well 10 times in a row, it may still conceivably fail the

next 10 times. Training a model is often rife with similar kinds of instability - if an ANN model is well fitted after running the script once, i.e. it achieves a high performance metric, running the exact same script over again may yield very poor model fits simply as a result of the stochasticity inherent in the model. If NaN's are returned by the optimizer, no model will be produced at all - this can happen at random when using a stochastic optimiser, although there are steps one can take to reduce that chance. Still, experiments can show how well an ANN performs over many replicate trials. The theory gives no guarantees of performance, as opposed to, again, the theory of linear regression, which is computed analytically, minimises least squares exactly, and is guaranteed to converge to any underlying linear trend asymptotically under strict assumptions. Neural networks make very few assumptions about the domain under investigation, and can pick up on much more complex statistical relationships. The great flexibility of ANN's comes with several downsides: since there is no guarantee on convergence, there may always be an error that the network cannot get rid of. During training, there is often an a priori chance of failure (i.e. NaN or bad model fit).

The performance of an ANN on a certain task will be contingent on the training and hold-out datasets, which usually involve inherent randomness/noise. For most purposes, acquiring a new training set will require wholly or partly retraining the network in order to incorporate the new information gained through data. Even a small change in the dataset used may lead to the trained architecture of the ANN no longer being optimal. For these reasons, it is not clear a priori how wide of a family of problems a trained ANN can viably perform for. Capturing out-of-distribution data points (i.e. extrapolation in the case of regression) is a notoriously hard problem for ANNs. Bayesian approaches may help in this regard. Colloquially speaking, a BNN is a neural network that can say "I don't know", instead of simply assigning a prediction (label) without supplying any warning on how far out of distribution the data point is. A standard ANN does not even give us any information on the spread (e.g. variance) exhibited by the sample distribution of the data. The ability to estimate the spread of the data in some region of the covariate space is a great feature of Bayesian approaches to ANN's, but it must rely on considerable inductive biases. The difficulty in investigating and choosing the right inductive bias for a given problem may be central challenge that determines if BNN's will be surpassed by other DL uncertainty estimation techniques in state-of-the-art applications.

ANN predictions are determined by many factors: the data, the model and inference assistance tools such as the optimizer, batch norm layers or . Hence, it would useful to have some guarantee that the ANN is only used to predict on observations from the same distribution as the data. However, this is often an unfeasible demand - for instance, a self-driving car will always, in theory, be at risk of seeing a situation unlike anything it has ever seen. This is a reason why modelling uncertainty must be developed, along with sensible policies on what the agent will do when it encounters a novel challenge. In the case of self-driving cars, the car should perhaps slow down? But at what rate? What if slowing down for

obstacle A puts the car at risk of crashing into obstacle B? To make such difficult decision making, a realistic uncertainty description is helpful.

Learning how to program and properly use ANNs in Tensorflow and Keras has, so far, been a considerable challenge. This is largely due to the complexity of the ANNs, which consequently means that professional ANN libraries, such as Tensorflow, are fairly complicated for a novice in this field of study. While there are many tutorials online, each tutorial has a different approach, making it challenging to transfer knowledge gained from one setting into another. When a novice first creates their very own ANN, there is a good chance the result will be bad, whether the task is classification, fitting a regression line or something entirely different. It will not be obvious why the result is bad - pouring over forum posts and instructional books on deep learning may be the best way to find a way forward. This may the case with traditional statistical methods as well - what's special about ANN's is the hugely modular nature of the model. One can always add another layer, try a different optimizer or regularizer, try different priors in the case of BNN's, or even have two adverserial ANN's (GAN's) compete against one another doing opposite tasks, creating seemingly novel representation spaces. Sometimes it seems that one's imagination is the only limit in coming up with new models. The large variation in ANN models on the intellectual marketplace means that there exists no overarching inference methodology that will give great results in all or even most cases. Instead, one has a number of "rules of thumb" and context sensitive strategies, along with lots of trial and failure.

The particular combination of linear algebra, calculus, optimization and statistics that comprises ANNs makes for a field that is hard to learn, and even when the student has a full overview of the workings of a deep FNN (feedforward neural network), it will not at all be obvious what implementation choices they should make as a practitioner or researcher.

Architecture (i.e. the number of layers and nodes), activation functions (which may have their own parameters), optimizer (contains learning rate) and loss function must always be chosen. This choice in itself can seem daunting, since one is picking a point in a fairly large hyperparameter space. Here, it is easy to get stuck trying out different value combinations on these must-have hyperparameters. Tuning them is a substantial field in ML literature. If the ANN still does not work, one may have to use additional techniques, like feature manipulation and add-ons onto the network like batchnorm. (Data should always be normalised, as is a common saying in the community (Andrew Ng).) This all makes for a substantial challenge when "going Bayesian" on top of all this. Hence, a lot of the time has been spent reading up on ANNs, Bayesian and otherwise.

Neural networks are complicated algorithms - some of the most computationally demanding programs that perform modern, real-world applications are ANNs, taking several weeks to train. Therefore, it is a good rule of thumb to always start out with simple tasks and networks and progress incrementally to more challenging tasks of a similar sort. This way, one can make sure that the code runs and that everything works out as expected along the way. In order to get a practical

understanding of the BBB algorithm and what it can do, let us use a simple 1D regression case.

On the topic of regression data sample simulation: remember that the underlying process is forgotten after it has created the data set. Therefore, the function $f(x)$ has no impact or salience on the neural network fitting on its own. However, for larger data sets, the resulting model would predict increase in proximity to the original underlying process. Hence, it is useful to include the graph of $f(x)$ in the plot of the BNN prediction, to get a visual sense of how well the prediction fits the dataset. Obvious, for very small datasets (say, $N = 10$), one cannot expect the best fit to the data set to also be a good fit to the underlying trend.

In the BBB algorithm, the mean $E(Y|x, w)$ can be seen as separate from prior distributions and the choice of its parameter values - at least when the prior distribution is symmetric. Further, the uncertainty estimates rely on having a good estimated mean $\hat{E}(Y|x, w) = \hat{\mu}(x)$, since the posterior distribution conditional on an observation, will be centered on $\hat{\mu}(x)$. Therefore, we can start out by adjusting the hyperparameters that are commonly found in a non-Bayesian ANN - layers, nodes, activation functions, learning rate, optimizer, and so forth. Then, we will adjust the BBB hyperparameters, namely the prior parameters σ_1, σ_2, π. In our implementation, the latter three will be static and identical for all neurons throughout the training process. Making them learnable and unique (separate) for each neuron/layer could be an interesting further model expansion that we will not investigate here.

4.2 Univariate nonlinear regression

A simple implementation of the BBB algorithm created by M. Krasser provides a good starting point. The implementation is meant to be an introduction, and is based on Blundell et.al. [4] which implements a simple (one-dimensional) regression, with the following set-up.

A set of N equally spaced points $\{x_i\}$ ranging from values a to b is used as the gridpoints for the dataset:

$$\{x_i\} = \{a = x_1, x_2, ..., x_{N-1}, b = x_N\} \tag{12}$$

For each of these covariate values x_i, the stochastic response y_i is generated using a sinusoidal trend function:

$$y_i = 10(\sin(2\pi x_i)) + \varepsilon_i \quad , \quad i = 1, \ldots, N \tag{13}$$

with $\varepsilon \sim N(0, \sigma_\varepsilon^2)$ and $\sigma_\varepsilon = 1$. This process, when applied to each covariate point x_i, generates response data $\{y_i\}$, which is normally distributed about the curve of the underlying trend. We now have a simulated dataset $\mathcal{D}_{\text{train}} = \{y_i, x_i\}$ on which inference can be based, and we can select the best model with respect to some metric. In a simulated data setting we can generate as much data as we like, so we can create a separate, arbitrarily sized data set of the same structure, called $\mathcal{D}_{\text{valid}} = \{y_j, x_j\}$. The training data set is plotted in Figure [7].

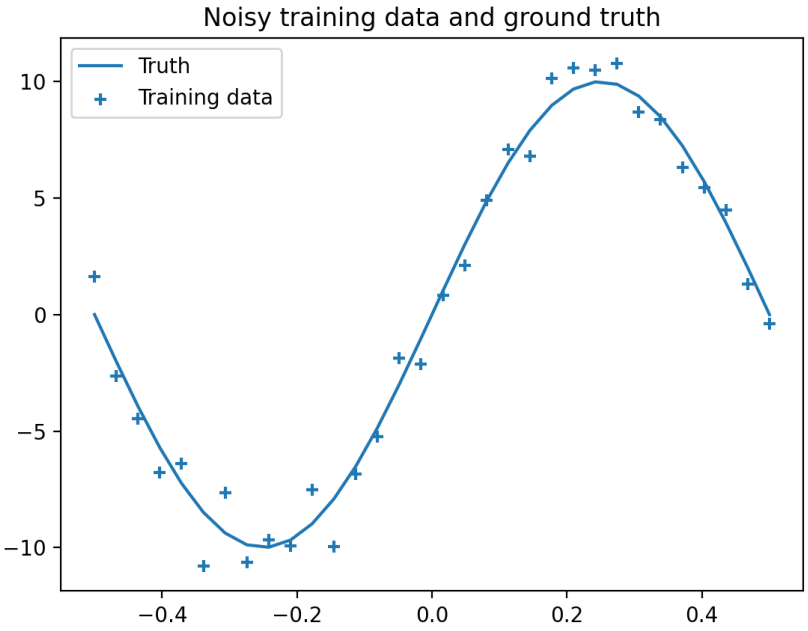


Figure 7: Dataset of 32 points generated from $f(x) = 10\sin(2\pi x) + \varepsilon$

This simulated dataset is well-behaved compared to most datasets seen in the real world. The data generating function $f(x)$ assigns one and only one response value to each point in the linearly spaced grid $\{x_i\}$. This is a far cry from the messy data that tends to result from studies or experiments on real-life phenomena. Uncertainty caused by unseen variables and from inherent aleatoric processes often gives rise to data that is not well-behaved: it may not seem to coalesce with any available theoretical distribution, it may have unpredictable, chaotic outliers, it may be strongly unbalanced, with many observations in one value and few or none in another, or it may have very few observations altogether. It may even have missing values on some variables - estimating these is a field in itself in the machine learning literature. All of these situations can be challenging for data analysts to deal with, which high-lights the benefits of using a simulated dataset starting out: we can begin with a well-behaved simulated setting and add on increasingly more features and complexities as we go along.

Running the BBB script, we are presented with the plot in Figure 8. A variational posterior distribution has been fitted to $\mathcal{D}_{\text{train}}$. The variational mean $\hat{E}(Y|X)$ is seen in red, ostensibly a close fit trend in the training data set. $\hat{E}(Y|X)$ is labelled "Predictive Mean" in the legend. In addition, a lower and upper bound are found about $\hat{E}(Y|X)$, forming an interval given by

$$(\hat{E}(Y|X) - 2\hat{\sigma}_\varepsilon, \hat{E}(Y|X) + 2\hat{\sigma}_\varepsilon)$$

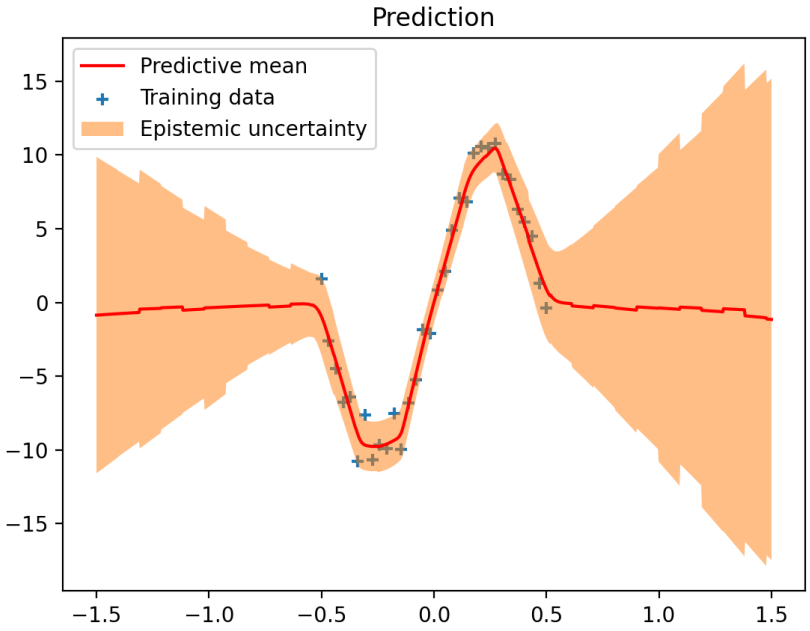


Figure 8: Bayes-by-Backprop 20ReLU-20ReLU model fit. Simulation mean in red, +-2 variational simulation sample standard deviations in yellow. Unedited fork from M. Krasser's github blog.

One may note that this interval is not a credible interval. The term "credible interval" is simply the Bayesian term for an interval formed by posterior quantiles containing some portion of the posterior probability density. The uncertainty interval we are given is referred to as "Epistemic uncertainty" in the legend, meaning uncertainty that could be eliminated with more data. This is somewhat imprecise. A better term would simply be "posterior mean +/- 2* posterior standard deviations", which is neither a credible or confidence interval. Still, it does the job of telling us how spread out the posterior distribution for various covariate values. The bounds are sensible - they closely follow the data, with only a few points falling above or below. In the out-of-distribution regions, given by

$$x \in (-\infty, -.5) \cup (.5, \infty),$$

we find that the uncertainty bounds spread out about linearly away from the training set, although it spreads out at different rates in positive and negative directions (to the left and right along the x-axis). We see the inductive bias of the model at work: the model is completely guessing what will happen outside the regions where observations, i.e. training data point, have been made. Yet it is not a bad guess.

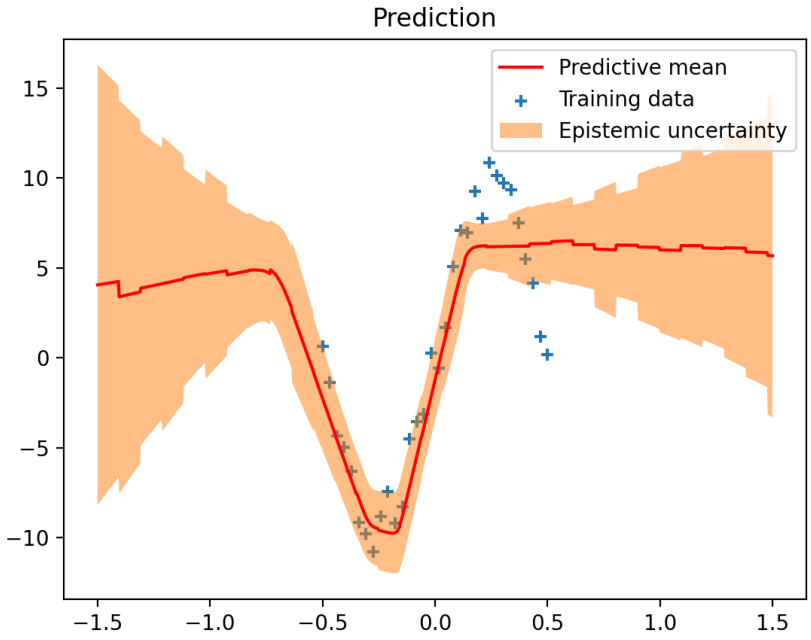


Figure 9: BBB model fit, produced from exactly the same script as in figure 8. Due to stochasticity in the code, the data set has changed, and the model fit is poorer.

It is one thing to estimate the trend $E(Y|X)$ as closely as possible. It is quite a different thing to estimate the error variance σ_ε^2 as well, fulfilling the hopes of what the BBB algorithm delivers on the promises of reliable interpolation/extrapolation and belief-based uncertainty estimates - even in a highly controlled simulated setting. In the language of Bayesian statistics, a fitted model may have a good posterior mean that lies close to $E(Y|X)$, but the $2\hat{\sigma}_\varepsilon$ bounds may be completely out of proportion, and it may not catch on to more complicated patterns such as heteroscedasticity. The implementation used here assumes homoscedasticity, since the data is generated with constant noise variance. Still, the model can pick up on some variation in variance when it forms itself about the data set according to Bayes' Theorem. The uncertainty estimate of the model is largely controlled by separate hyperparameters σ_1, σ_2, although network width also has a lot of impact on uncertainty estimate.

Running the same script again, we get a different result shown in Figure 9. The model fit is now much worse: it has failed to pick up the signal for the rightmost 40% of the data, so instead of tracing the underlying sine wave, it shoots out horizontally. The uncertainty bound also begin growing from this point, the model seemingly oblivious to the data in this region.

Why does the model change from one run to the next? The code is highly

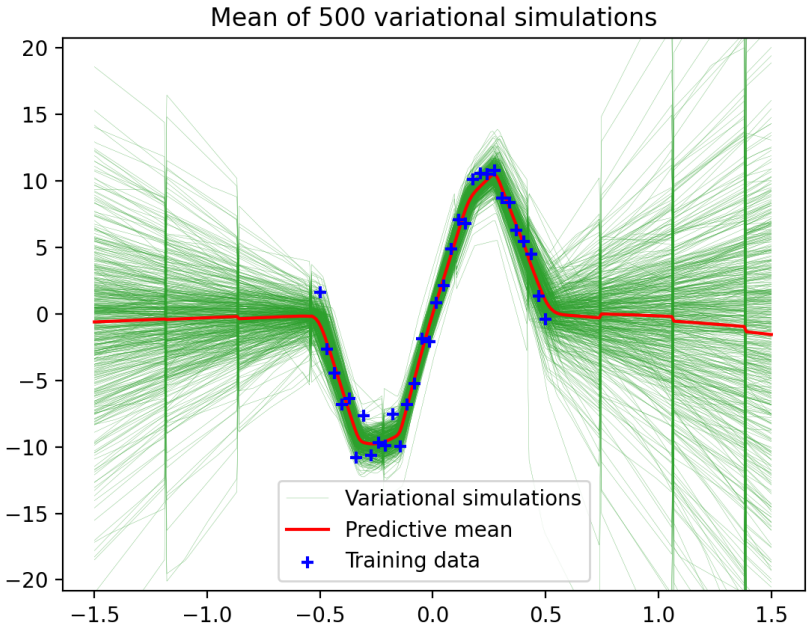


Figure 10: 500 variational simulations making up a sample posterior

probabilistic in nature. It calls a RNG function when generating the data set, all trainable parameters are initialised by a normal distribution, and the optimizer is an advanced form of stochastic gradient descent, which makes stochastic choices during training. The BBB algorithm also randomly samples the weights and biases used for each epoch. This means that seeding the RNG's in the code is necessary in order to ensure that the program does not change results are random. From now on, we will seed the code in order to be able to isolate the effects of changes in the code.

Why does the model fit deteriorate from one run to the next? The simple answer is that the current choice of hyperparameters makes training unstable. We should change it up to find a better permutation of values that makes training more stable, so that the failure or success of a hyperparameter setting to fit well to the data can be relied upon to a higher degree.

The question of whether training stability is important really depends on the application. If we have no constraints on time and compute, we can try out different hyperparameter settings in a roundabout way, and as soon as the model fit is acceptable, simply save the parameter values and the model architecture, and you have a production ready model. However, in on-site training, as seen in e.g. Internet-of-Things applications, data is continuously updated, and inference must be quite fast in order to respond quickly to the change in observations. Here, un-

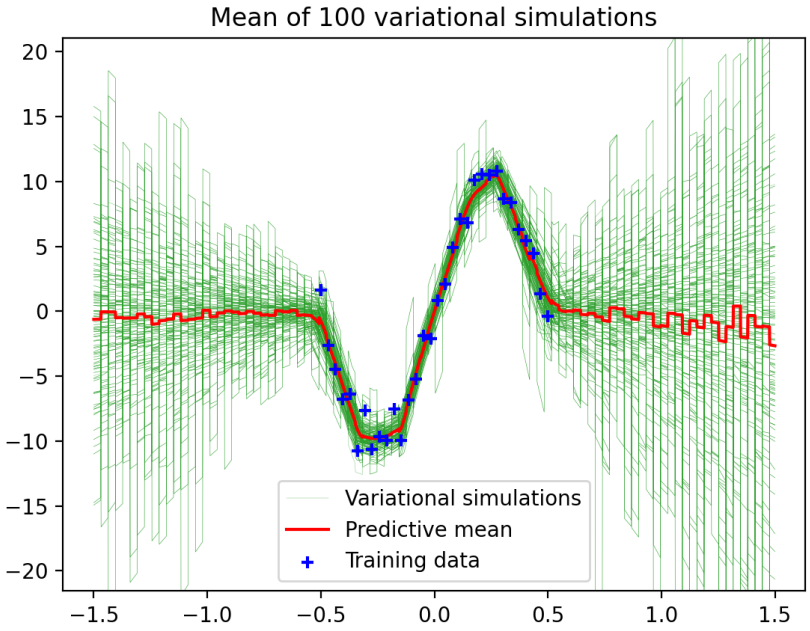


Figure 11: 100 variational simulations giving a new sample posterior. The grid step size 10 times smaller than previously.

stable training can be catastrophic. Hence, creating ANN algorithms that can be relied upon to fit models to the data reasonably well every time has been a goal of importance in the ML literature. One technique is data normalisation, whereby the whole data set is transformed by

$$x_i^{normalised} = (x_i - \mu_{\text{train}})/\sigma_{\text{train}} \tag{14}$$

where $\mu_{\text{train}}, \sigma_{\text{train}}$ are the sample mean and standard deviations of the training set. This technique did not work at all for the sinusoid regression task - results where bad, and fitting the models slowed down, taking more epochs to finish convergence. It is clear that scaling a data set along the x and y directions does not yield an equivalent problem with respect to the model: some data scaling are better than others, and, as with most other factors in the model, one cannot know a prior what scaling to choose - one simply has to try out different ones. In the last stage of experiments, we will scale the data out further in the x-direction, reducing the slope of the sinusoid curve and making model fitting more stable.

The plot in Figure 10 displays 500 variational posterior samples. It is from these samples the uncertainty bounds are calculated, by taking their standard deviation in each point along the x-axis and adding/subtracting 2 of them to the posterior mean, which in turn is just the pointwise mean of the samples along the x-axis.

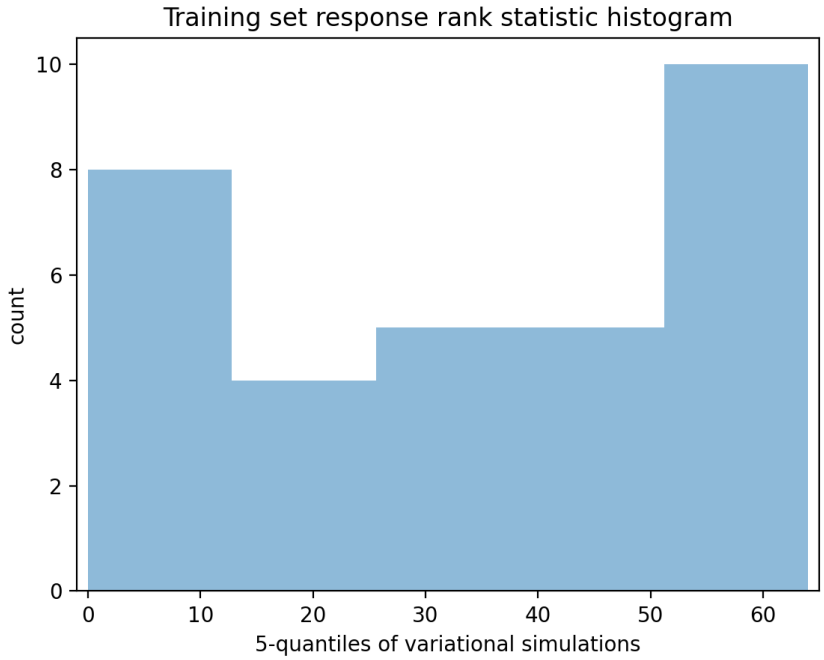


Figure 12: A 5-bin histogram showing the how many data points fall in each quintile (5-quantile) of 64 variational posterior samples. This histogram is somewhat U-shaped, meaning tails are heavy. This is a common problem in statistics: the spread in the data is underestimated, so many data point fall in the upper and lower quintiles. There are too few points in the middle. In order to rectify this issue, we must make the posterior distribution spread out more, so it covers and describes the spread of the data well.

Note that one could calculate any desired statistic from these simulated curves: higher order moments, quantiiles, ect.

The plot in Figure 11 shows the exactly same model fit, but the number V of variational simulations has been reduces by a factor of 5, and the granularity of evaluation has been increased by a factor of 10. These changes do not affect training or model fit - only serves to highlight different parts of the model. The lower V means that the mean will vary more.

The histogram in Figure 12 shows the rank statistic of where the validation data points fall in between 64 variational curves, which are indexed from 1 to 64 at each point on the x-axis. Falling below every one gives a rank statistic of 0 for that data point, while being above all of them gives a rank statistic of 64. With 32 data points, the cumulative count ends up at 32. The horizontal axis shows all variational curves lined up from lowest to highest response value at the covariate point of that observation.

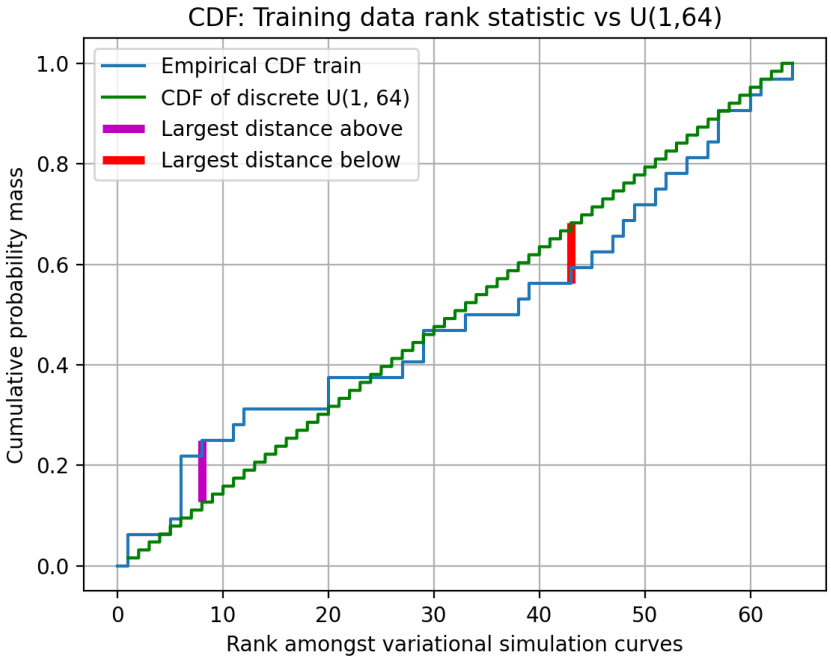


Figure 13: Comparing cumulative distribution functions (CDFs): Empirical training data CDF vs discrete uniform going from 1 to 64 - one step for each variational simulation. Dn+=.126, Dn-=.121 (above, below)

The plot in Figure 13 shows the results in the histogram in Figure 12 in cumulative distribution form. The cumulative sample distribution of the data is compared to the discrete uniform distribution. This uniform is the ideal distribution of the rank statistic of the data in the model perfectly fits the data. The largest distances in cumulative probability mass above and below betweent the two CDF's are shown in magenta and red. The largest of these distances is the K-S statistic for that model with respect to the data.

The plot in Figure 14 is the MSE diagnostic of the 20ReLU-20ReLU model during training. It can be used to monitor how fast the model training converged. However, it is in linear scale, meaning that the changes in the beginning of training had the largest impact on the plot. If we make the y-axis logarithmic, we get the result in Figure 15. Here, the oscillation in MSE near the end of training are much more visible, allowing us to see at what epoch the model truly converged.

Having explored the 2-layer ReLU 20-neuron-per-layer architecture for BBB, the problem still remains that the model sometimes fails to fit. Having somewhat unstable training may not always be a problem in applications - if one has enough compute, one could simply train the model repeatedly until a sufficient fit is achieved. More clever, one could save the best model using a checkpoint call-

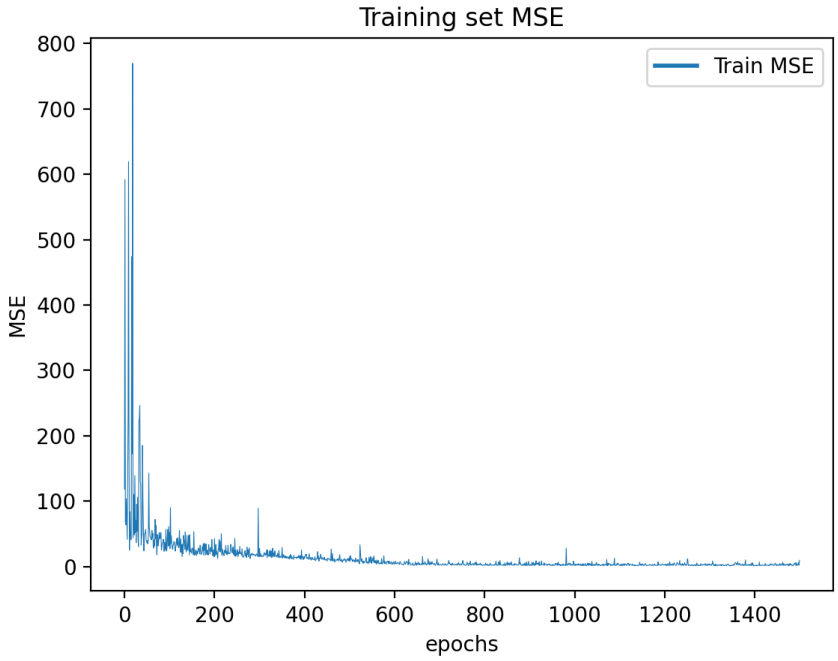


Figure 14: Minimum Squared Error between the 20ReLU-20ReLU model's prediction and the training set, plotted across the 1500 epochs of the training procudure.

back, as in Keras, for saving the best model (i.e. the best parameter setting) for each run, and then start from there in the next run with different hyperparameter settings. We can only speculate as to how far such "common sense" techniques are used in production level AI. Theory can be very helpful in developing new methods for fast, robust ANN training with less hyperparameter tuning. A simple example However, such techniques can only take us so far - many problems still rely on more bruteforce, common sense implements, such as having checkpoints that save the best model fit so far along the way during training.

In the context of this thesis, the main problem with unstable training is that we want to know what hyperparameter and architecture setup gives the best uncertainty estimate, which relies on a good mean fit - the fit of the model should have a low MSE with respect to the training and validation sets. If we increase the weight-and-bias prior variance σ_w, and the posterior mean shifts in the next run, was the change in posterior mean caused by the change in σ_w, or did it happen by chance? We can only know for sure by running the two scripts many times over, where the σ_w value is the only difference between the two. Such a strategy would tease out the impact of this particular hyperparameter given the values of all the other hyperparameters, but in order to know what it does in general, we need more trials. This strategy easily becomes prohibitive with respect to time, as we would

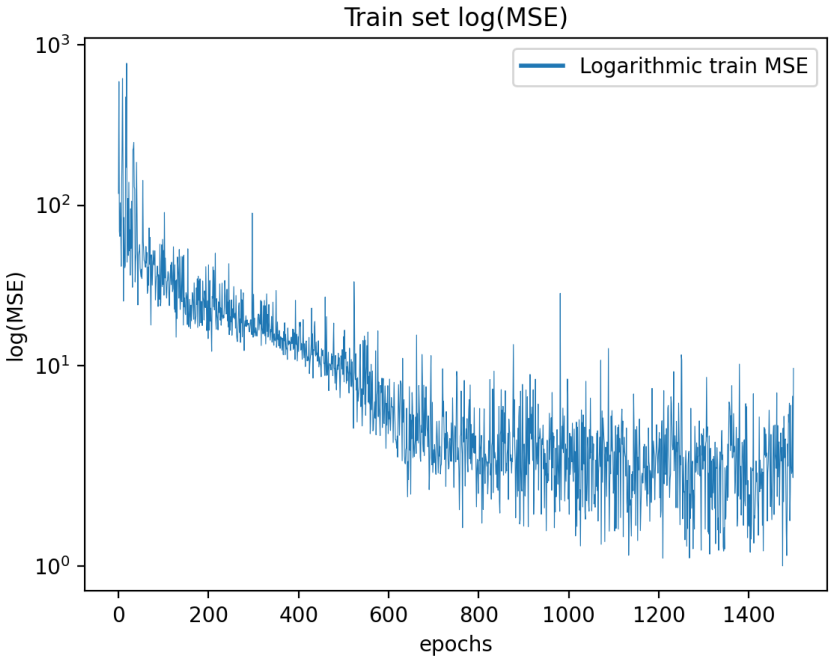


Figure 15: log(MSE) between 20ReLU-20ReLU model and training set, plotted across 1500 epochs. Small changes in the MSE near the end of training are exacerbated by showing logarithmic training MSE, enabling a closer view of whether the MSE trend converged or not.

need several runs for each hyperparameter setting, for many different settings, and each run takes about 5-20 minutes.

We create a new data set, $\mathcal{D}_{\text{valid}}$, used for evaluation purposes. A systematic experiment is performed, where different architectures are tried out. Let us refer to the BBB architecture so far as ”20tanhX2”, meaning 20 neurons, tanh(x) activation function and 2 hidden layers. An architecture called ”5ReLU” will then have 5 neurons a layer, ReLU activation function and only 1 hidden layer, and so forth.

Given the large space of hyperparameter values, there is a challenge in selecting which ones to vary. Changing all of them at once will give little insight in the impact of each hyperparameter.

The scientific process seeks to find out what causal relationships exist in the world, by untangling and isolating them. In an experimental setting, one tries to isolate factor and change only one at a time, to see the effects of each. This highlights the need for seeding code’s RNG, so that one can see the effect of a single change in the code. Even so, can should keep in mind that a full view of what a probabilistic script does can only be found by running it with a constant hyperparameter setting, changing only the seed. This will reveal its stability: if the

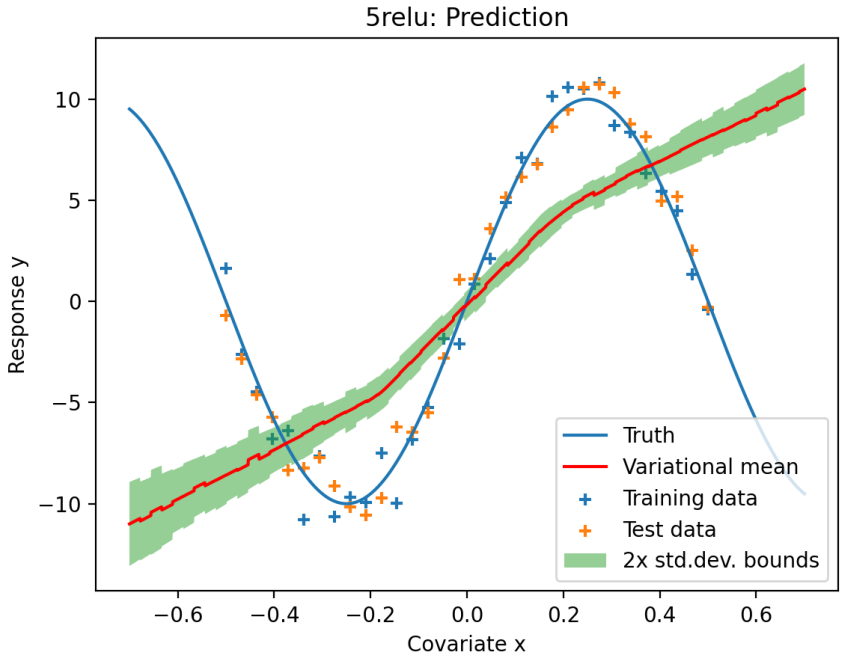


Figure 16: The model fit for the 5ReLU setup. This is the worst fit of all the experiments, with an MSE of 19.179.

model fitness varies greatly across seed values, one can the sure that the training is not stable at this hyperparameter setting.

We will be using our two metrics, MSE and the Kolmogorov-Smirnov statistic, distribution, to evaluate each fitted model. Note that while the MSE is calculated directly from the holdout data set, called validation set in Keras, and can be plotted directly from the Keras history object. The K-S statistic is evaluted using the validation data along with variational simulations, making it very time consuming to evaluate for each epoch. Therefore, it is only produced after training is finished. It would, however, be interesting to see a validation K-S vs epoch plot, as the validation K-S statistic should reach a minimum and start increasing if the model becomes overfitted.

The increase in compute time increases linearly with the granularity of the grid on which the network's prediction is plotted, as well as with the number of variational simulations created. The size of the data also affects compute time linearly, since the implementation evaluates all of the data every epoch. However, since the network in fully connected, adding more neurons and layers will increase the time to compute more than linearly. Adding one neuron to a layer creates one bias, one weight per neuron in the previous layer, and one weight per neuron in the next layer. In addition, the BB algorithm doubles the number of (trainable)

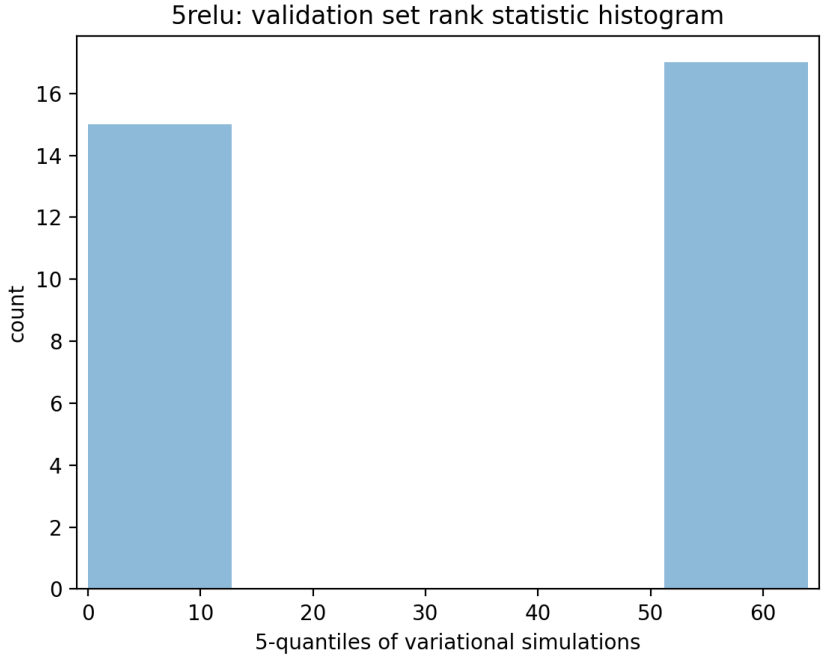


Figure 17: Histogram: The validation response's rank statistic among 5ReLU's variational curves. It is about as U-shaped as a histogram can get.

parameters per weight/bias, since both the centre and spread (mean and variance) must be encoded in parameters. (A regular ANN only expresses the mean, as a point estimate.)

The implementation by M. Krasser uses the BBB algorithm to find uncertainty estimates around the regression line in a standard univariate regression setting. A dataset is made from the deterministic $f(x) = 10\sin(2\pi x)$ on $x \in (-.5, .5)$ with added iid noise $\varepsilon \sim N(0, \sigma^2)$. The underlying process f_{true} is now "forgotten", i.e. it cannot be accessed by the regression algorithm, and it cannot be used to evaluate metrics of model fitness. While f_{true} is what we really want to estimate, we settle for the next best thing: adapting a regression curve that minimizes the RSS (residual sum of squares) to the dataset. For medium sized datasets, polynomial regression can be fitted analytically under strict assumptions, but for n observations and $k = p - 1$ covariates the asymptotic runtime is $O(p^2(n + p))$, which can be restrictive. The BBB algorithm instead draws a sample S of simulated curves s from the estimated posterior $p(y, w|x)$, and uses S as its representation of the distribution from which D was drawn. For each $x \in X$, BBB now takes the mean and variance of $s \in S$ as the regression line and the variance of the stochastic process $f(x) + \varepsilon$, respectively.

The code produces a plot of a regression curve estimating the underlying trend

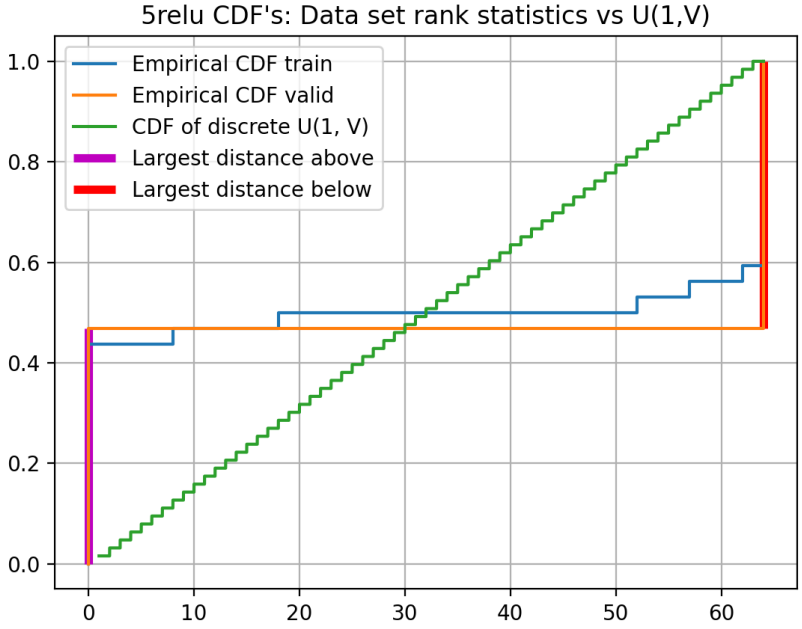


Figure 18: 5ReLU: CDF comparison. Dn+, Dn- evaluated only for the validation data. See tables for values.

along with predictive mean and uncertainty bounds. The predictive mean is seen as the red curve seen in figure 6 - this is the mean of 500 simulation curves, each being a realization of the fitted BNN model with randomly sampled parameter (weights and biases) values. The simulation curves are also used in producing the uncertainty bounds. The bounds correspond to 2 standard deviations of the data set along the y-axis. Since the errors are Gaussian and iid, i.e. $\varepsilon \sim N(0, \sigma^2)$, these bounds are approximate to 95% confidence intervals to the regression line. This is to say that given that the regression is an accurate fit to the underlying process, and that the uncertainty bounds are also accurate, the bounds will contain about 95% of all data points in the data set. Since the errors are Gaussian, we can create CI's not only for 95% of the data, but for arbitrary data set portions centred at the sample mean. Since the posterior from which simulations S are drawn is itself normal, their empirical distribution should approach the data set distribution as the sizes of both data set and simulation set go to infinity - assuming the posterior is a perfect fit. This means that the data points (the elements of the data set) should appear among the simulation curves in a uniformly distributed fashion. More specifically, the rank statistic of the data points among the simulation curves along the y-axis should be uniformly distributed - the data points should fall in the lower 5% of the simulation curves 5% of the time, it should fall between the lower 95 and

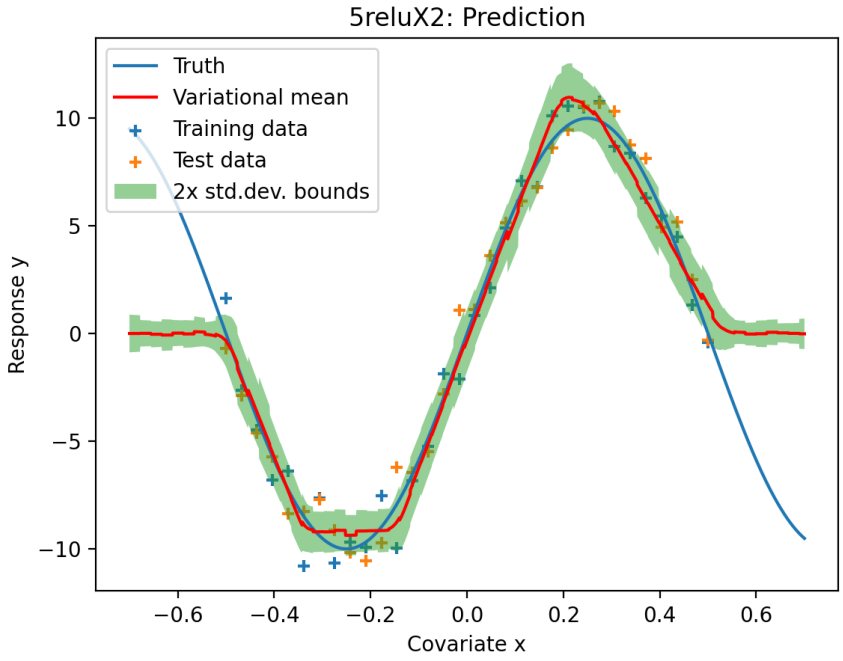


Figure 19: 5ReLUX2 model fit - the most well fitted model to the training data.

90 percent 5% of the time, and so forth. The degree of uniform goodness-of-fit (GOF) of this rank statistic is therefore a good metric for the accuracy of the uncertainty bounds produced by the algorithm. Plotting a histogram with 20 bins, one for each 5-percentile, gives a decent visual tool to see where the uncertainty bound has strengths and weaknesses - e.g. large tails is interpreted to mean that the bounds are too small, and we need to increase the according parameters innate in the DenseVariational hidden layer class.

Some experimentation was undertaken to find better architectures and hyperparameter settings for the sinusoid regression. Sometimes, training produces NaN's as predicted response values, no matter the covariate value. This issue could always be resolved by decreasing the learning rate of the Adam optimizer to some small number ($\alpha = 0.001$ always gives real-valued response values, i.e. not NaN or Inf, for all models used), though it may cause the model to not learn well. This implies that the minimization topology is challenging for the optimizer - it likely gets stuck in a local minimum, after which it has trouble exploring a wider region of the parameter space.

As illustrated in another blog , Laplace distributions may serve as a better prior than the Gaussian mixture used in this BBB model. This again underscores the multivariate nature of the hyperparameter space of the model - in a certain sense. The choice of prior distribution not a hyperparameter per se, but has importance

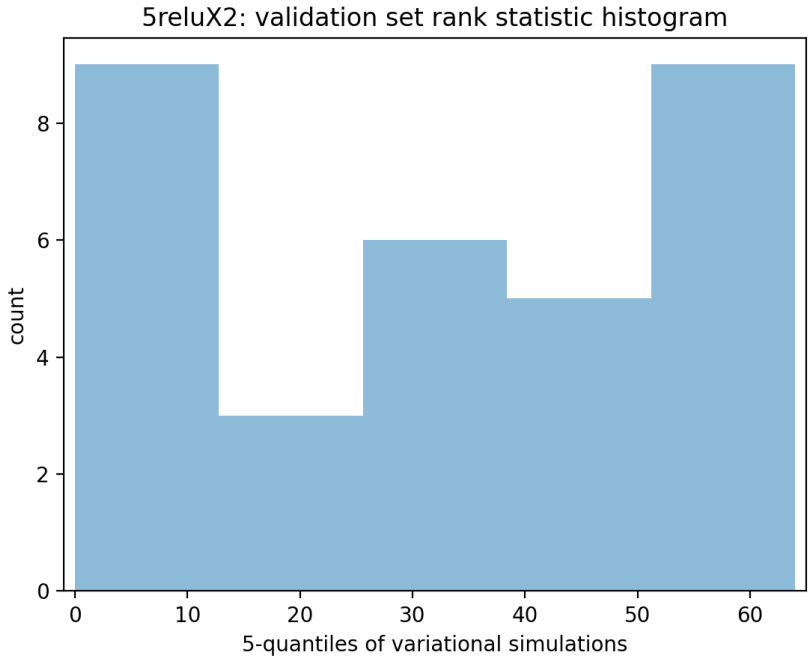


Figure 20: 5ReLUX2 histogram: rank stastistic of data response among variational curves.

for inference. The space of possible prior models one could choose is large, and the number of trainable parameters involved in different priors is variable. The use of Gaussian or Laplace gives L2 and L1 regularization, respectively, but that does not prohibit the use of other distributions. A hyperparameter space search would necessarily be limited in its coverage. Ideally, one could try to update the priors in a non-parametric fashion, as seen in MCMC methods, though such methods have their own difficulties. This would remove the need for directly training the prior parameters. But it requires an extremely rich model with a fully hierarchical prior model.

How stable is this algorithm? We want to be able -optimizer may get stuck in local minimum of parameter space. Once in a local minimum, for the optimizer to get out of it and traverse the argument space can be unlikely. Adam decreases its learning rate steadily, which can make it focus too hard on minima it comes across. - the methods .compile() and .fit() of Keras.models.Model() clearly do not refresh and start over when they are called multiple times in a for-loop. This is demonstrated by Figure **??**.

Table 1: one could include KS metric as a loss metric in Keras, to try to include uncertainty tuning more closely to the training of the model. This would require writing a custom K-S metric class, then create an object for keras.model.compile to

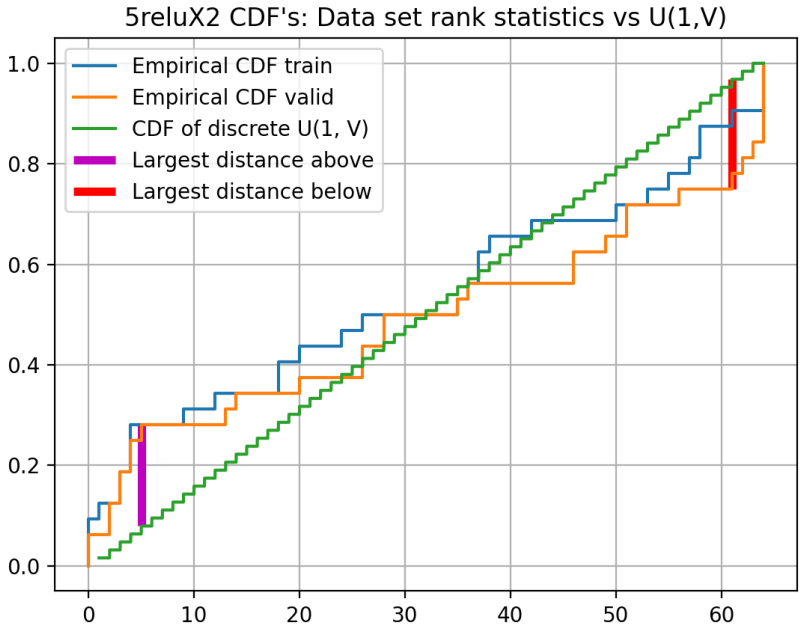


Figure 21: The 5ReLUX2 model's CDF's: comparison of cumulative distributions for data and variational curves.

accept as a loss argument. It is not at all clear a priori that such an uncertainty training technique would work well, since the K-S metric cannot distinguish between the model's estimated uncertainty bounds being to narrow or too broad (i.e. uncertainty underestimation and overestimation, respectively). The K-S metric only give a measure of the "distance" from a uniform distribution to the empirical rank statistic distribution, but not the "direction" of said divergence.

Table 1: one could here run one script many times over, to find the sample distribution of MSE's (while changing the RNG seed). This should give us a better idea of how stable the architecture is, since it eliminates the possibility that the results in this table were derived completely by chance. Doing so would take some measure of time, and we will instead focus on taking our lessons from the trials thus far and use them to try to fit a better uncertainty estimate in a setting with more data.

Results in table Table 1 are noisy. The 20tanh model sometimes fit the data well, but in this case it failed. This instability could be removed by decreasing the error in the data set, or get more data points in it. This table only shows a snapshot of the results for one seed value - a more thorough way of doing it would be to run each script repeatedly, and analyse the set of all results for each case.

(Here, saying that 20tanh the model "failed" this task simply means that its

Table 1: MSE experimental results for sinusoid data

			BBB		Deterministic ANN	
ϕ	Layers	Nodes	MSE, train	MSE, valid	MSE, train	MSE, valid
Tanh	1	5	5.910	4.647	.753	1.038
Tanh	2	5	7.429	6.016	9.476	7.776
Tanh	1	20	4.055	3.058	.712	1.022
Tanh	2	20	6.960	5.860	.401	1.641
ReLU	1	5	19.179	15.945	19.325	16.136
ReLU	2	5	.854	1.276	.986	1.109
ReLU	1	20	2.215	1.436	2.291	2.055
ReLU	2	20	.888	1.135	.380	1.391

Table 2: Kolmogorov-Smirnov statistic for experimental results for sinusoid data

			BBB	
ϕ	Layers	Nodes	KS, train	KS, valid
Tanh	1	5	.296	.313
Tanh	2	5	.327	.313
Tanh	1	20	.250	.281
Tanh	2	20	.313	.296
ReLU	1	5	.438	.531
ReLU	2	5	.218	.218
ReLU	1	20	.234	.296
ReLU	2	20	.105	.150

prediction was substantially different from the ”underlying truth” - in this case, it was strongly under-fitted. Of course, the data set we gathered could conceivably been generated from a trend curve identical to that of the badly fitted 20tanh model. In other words, there is no objective answer to what constitutes a bad fit - it depends on what sort of prediction we are looking for. The question of whether the model would keep producing the same under-fitted prediction is a different matter - the 20tanh model)

Table 2 shows results of the KS statistic for these examples. With 2 layers, the ReLU results are her pretty promising in this Bayesian model.

...

As we have seen, different 1D regression problems require different neural network architectures to find the best solutions - at least for shallow networks (not deep). One could desire to find a single architecture than could solve a large family of problems, so that one does not have to hyperparameter tune (i.e. search for a good setup of numbers of nodes and layers) for every new problem one comes

across. This has the offset/disadvantage of more challenging training - deeper networks yield more difficulties with training - so it may be more pragmatic to create a new architecture for every new problem. This is likely true for multivariate regression and classification tasks as well.

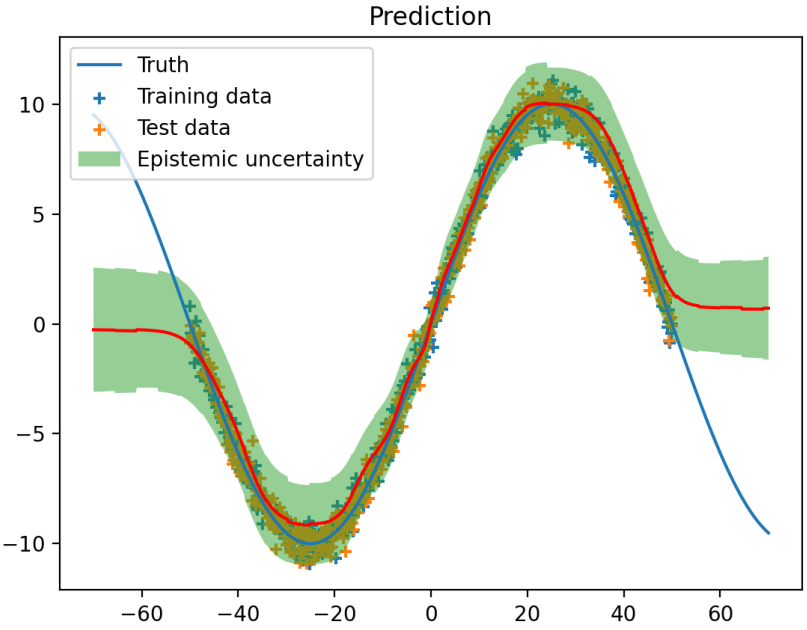


Figure 22: A 2-layer model with form (200tanh,200ReLU) fitted to a training set of 500 datapoints. The first layer smoothly follows the ground truth, while the second layer increases the representation space of the uncertainty bounds, without disturbing the posterior mean.

4.3 Uncertainty estimation

Tuning the hyperparameters that have the biggest impact on uncertainty may create a bias in the posterior mean - fitting the variational posterior is a ”single package” in this sense, as fitting its centre and spread (specifically mean and variance) cannot be entirely decoupled. This phenomenon has a clear analogy to the bias-variance trade-off in ”regular” ANN’s, i.e. ANN’s where weights and biases w and ANN predicted response on the test labels are both point estimates, incidentally conditional on the training set. The analogy is found in the fact that tuning hyperparameters that increase uncertainty estimation may affect the posterior mean, making it biased. Note that ”bias” in this context simply denotes that the estimated mean is different from the trend.

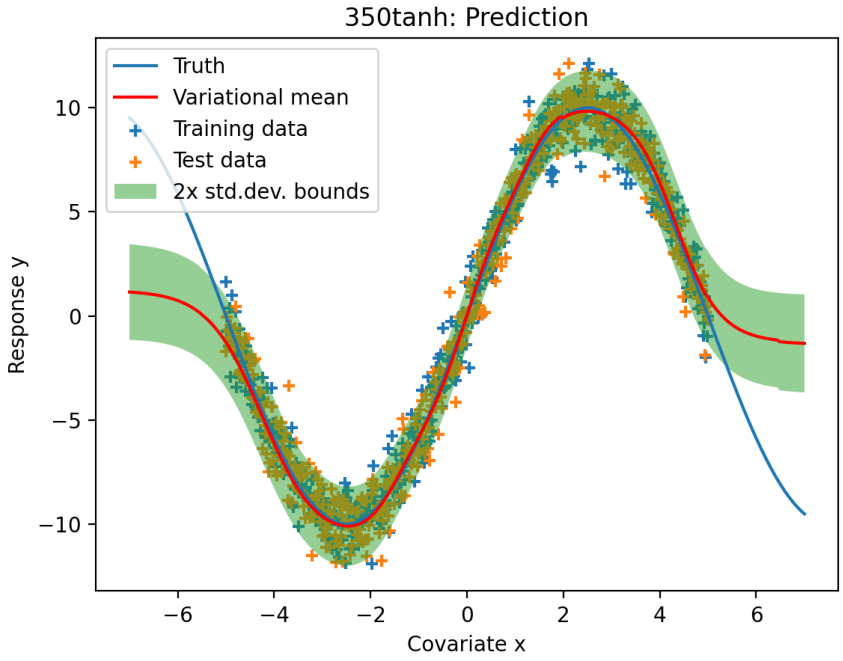


Figure 23: The best uncertainty fit achieved: the 350tanh BBB architecture. After many attempts tuning other hyperparameters, tuning the network breadth turned out to be the most reliable way of tuning uncertainty, i.e. posterior spread, without biasing the posterior mean. A surprising, yet satisfying result.

When the process of getting a good variational posterior variance creates a bias in the posterior mean, a naive tactic may come to mind: partition the domain of the covariates into separate sections, and train different ANN's on the training data that falls into each section. Then, simply combine the predictions into a piece-wise function that covers the whole domain. However, such a strategy would be incredibly cumbersome in many applied settings, because of the enormous dimensionality of their covariate domains. A better strategy for creating more general purpose AI is creating ANN's that automatically adapt to arbitrarily complex and high-dimensional functions, with little to no human data preparation needed. This is what one should seek to achieve when developing ANN methods, and it makes applying them much more practical once their proper use is better understood.

5 Closing remarks

Insight in properties of the BBB method has been gained in this work, via sensitivity studies to the various settings of hyperparameters in the network models. One may come into the ANN field assuming that a well-chosen architecture

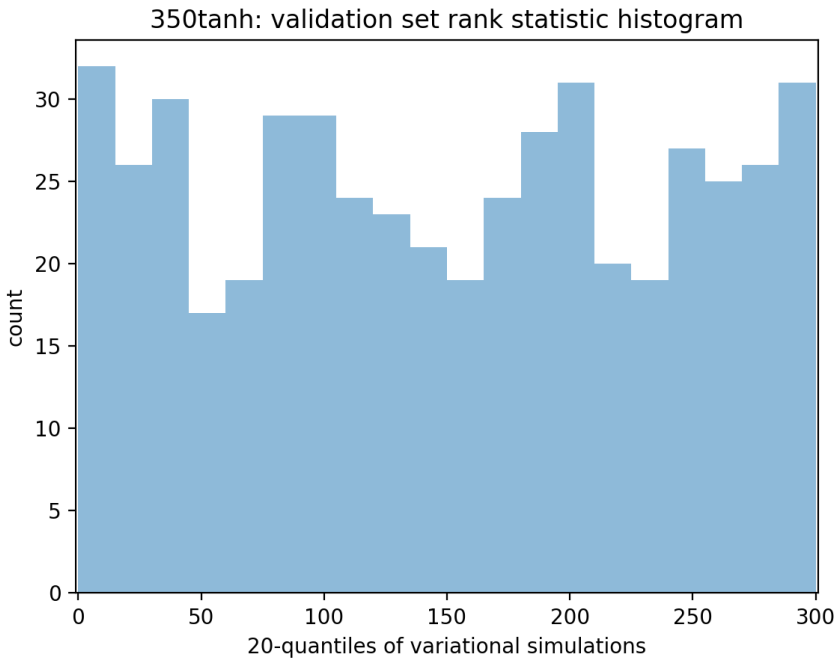


Figure 24: Histogram show data rank statistic for 350tanh BBB architecture.

(layer-and-neuron arrangement) will learn equally well on all appropriately sized data sets drawn from the same distribution. Surprisingly, the BBB algorithm (or at the very least, this implementation of it) is highly sensitive to the amount of data used in training. Substantially increasing the size of the data set requires retraining the network, returning and retraining the data set - either the ANN must be made wider, or the priors must be modified to allow for the posterior to have greater spread, balancing out the increased certainty induced by including more data. Activation function have some impact on representation space and inductive bias - tanh functions easily fit to sinusoidal data, but it is not as good as ReLU at picking up on signals when there is a lot of noise. Compared with a frequentist approach, the BBB results are usually less prone to overfitting when the number of layers is more than one in our example.

This work looks only at a simple example of a sinusoid model. Much work on other more complex cases is obviously interesting, but it is likely difficult to come up with clear-cut advice. A possible solution to these cases, which is currently being tried by some researchers, is to employ a search for optimal designs of hyperparameters, for instance using Bayesian optimization methods. Still, in high dimensions and with very difficult posterior density surfaces, it is not straightforward how to approach this suggestion. Like most other deep learning methods, generalization is difficult.

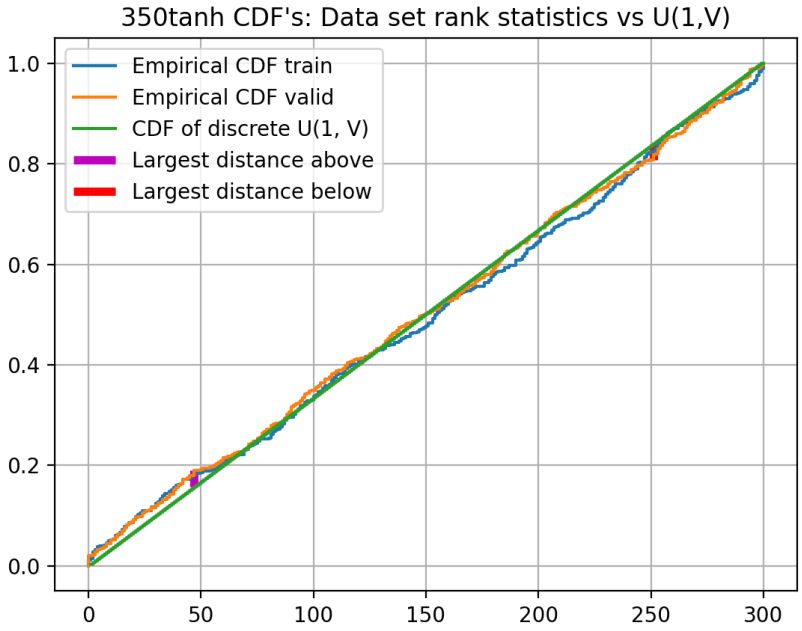


Figure 25: CDF comparison, detailing the Dn+ and Dn-. The max() of these is the validation set K-S statistic for the 350tanh model - it comes out to be .0328, an order of magnitude lowewr than any K-S statistic in the small data scenario. One should keep in mind that the one-way K-S statistic is sensitive to the size of the data set being compared to an ideal distribution.

Bibliography

[1] Anil K Jain, Jianchang Mao, and K Moidin Mohiuddin. Artificial neural networks: A tutorial. *Computer*, 29(3):31–44, 1996.

[2] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[3] Laurent Valentin Jospin, Wray Buntine, Farid Boussaid, Hamid Laga, and Mohammed Bennamoun. Hands-on bayesian neural networks–a tutorial for deep learning users. *arXiv preprint arXiv:2007.06823*, 2020.

[4] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks, 2015.

[5] Keming Yu, Zudi Lu, and Julian Stander. Quantile regression: applications

and current research areas. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 52(3):331–350, 2003.

[6] Pedro L. Fernández-Cabán, Forrest J. Masters, and Brian M. Phillips. Predicting roof pressures on a low-rise structure from freestream turbulence using artificial neural networks. *Frontiers in Built Environment*, 4:68, 2018.

[7] J. Cano et al. Accelereting deep neural networks on low power heterogeneous architectures. *1th International Workshop on Programmability and Architectures for Heterogeneous Multicores*, 2004.

[8] Scott Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *arXiv preprint arXiv:1705.07874*, 2017.

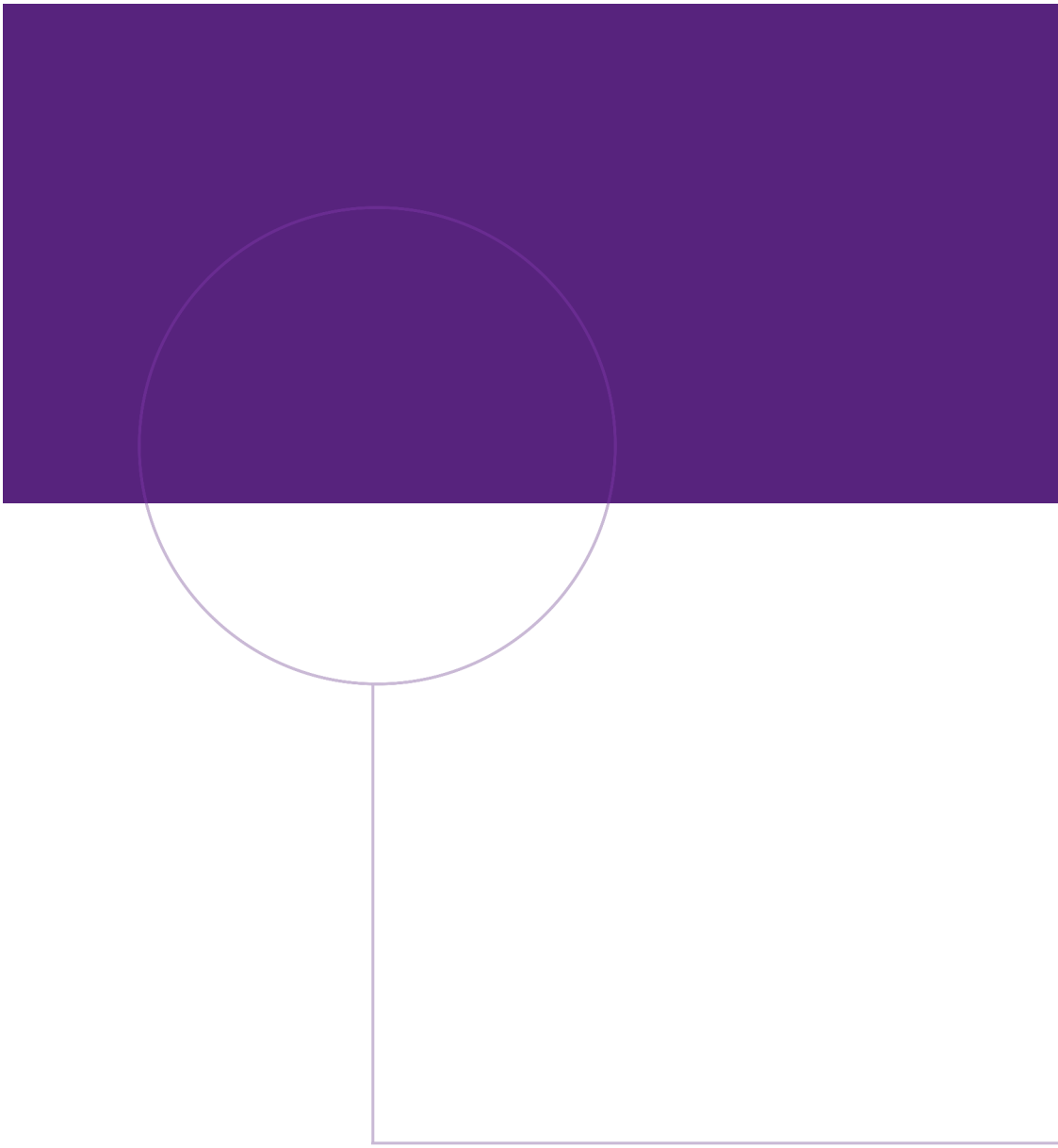
[9] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep learning*, volume 1. MIT press Massachusetts, USA:, 2017.

[10] Radford M Neal and Jianguo Zhang. Méthode générale pour la résolution de systèmes d'équations simul- tanées. In *Compte rendu des séances de l'académie des sciences*, pages 536–538. 1847.

[11] D.M. Titterington et al. Bayesian methods for neural networks and related models. *Statistical Science*, 19(1):128–139, 2004.

[12] Radford M. Neal and Jianguo Zhang. High dimensional classification with bayesian neural networks and dirichlet diffusion trees. In *Feature Extraction*, pages 265–296. Springer, 2006.

[13] Dani Gamerman and Hedibert F Lopes. *Markov chain Monte Carlo: stochastic simulation for Bayesian inference*. CRC Press, 2006.



NTNU

Norwegian University of
Science and Technology