TKP4580 - Chemical Process Technology, Specialization project

Bayesian neural networks: A comparison of Monte Carlo variational inference and Monte Carlo dropout

Hege Landbø

Submission date:	18.12.2020
Supervisors:	Johannes Jaschke, IKP
	Halvor Aarnes Krog, IKP

NTNU

Department of Chemical Engineering

Norwegian University of Science and Technology

Abstract

This project aims to compare two of the most common methods used for approximating a Bayesian deep learning model, namely Monte Carlo variational inference (MCVI) and Monte Carlo dropout (MCDO). The purpose of comparing the two methods is to obtain knowledge on which approximation gives the most robust and accurate model that also produces the expected results according to the theoretical background. Bayesian learning techniques, which incorporates model uncertainty into the results, have become increasingly popular in industrial context and have been proven as an efficient tool for data-driven applications. This is mainly due to its ability to indicate just how certain or uncertain the model is about the outcome that was predicted. However, several techniques are needed to approximate and simplify the mathematical model of a Bayesian neural network due to its complexity and intractable analytical solutions. Many have proposed different methods during the last two decades, showing varying results, but this project will focus on two of the seemingly most popular and commonly used methods. To examine the properties of the two model structures, and the differences between the two, both models were implemented and trained on the same three datasets. The hyperparameter tuning of the models were investigated, and the average training times, model precision and visual prediction results for the two models were compared in each case. Results showed somewhat similar prediction results and accuracy, however, the MCDO model proved to have shorter average training time for all three cases, as well as slightly lower mean squared error losses. It was also more user-friendly in terms of intuitive model structure and parameter tuning.

Preface

Declaration of Compliance

I, Hege Landbø, hereby declare that this is an independent work according to the exam regulations of the Norwegian University of Science and Technology (NTNU).
Signature:

Hege Landbø

Place and Date: Trondheim - Gløshaugen, December 2020

Table of Contents

List of Tables

List of Figures

Abbreviations

AI	Artificial intelligence
BNN	Bayesian neural network
EI	Expected improvement
ELBO	Evidence lower bound
HP	Horsepower
KL	Kullback-Liebler
LSTM	Long short-term memory
MAP	Maximum a posteriori
MCDO	Monte Carlo dropout
MCVI	Monte Carlo variational inference
ML	Machine learning
MLE	Maximum likelihood estimate
MPG	Miles per gallon
MSE	Mean squared error
NaN	Not a Number
NN	Neural network
PCA	Principal component analysis
Relu	Rectified linear unit

RNN	Recurrent neural network
SRS	Simple random sampling

CHAPTER 1

Introduction

Machine learning (ML) is one of the most promising developements within artificial intelligence (AI) over the past decade, and has become an increasingly popular tool within several types of industries, due to the increasing amounts of data being collected. For instance, machine learning models are used within the manufacturing industry for optimization, control and troubleshooting, in medicine for medical diagnosis and within finance for credit applications and fraud detection, just to mention a few.

Machine learning is basically to program computers to optimize a performance criterion, an optimization objective, using example data or past experience. This can be done to either make predictions about the future or to gain knowledge about the data. Neural networks (NN) are a class of models within machine learning. They are computational models that integrate the principles from the different information levels in the human brain, and are efficient tools for sovling complex problems. However, an important factor that is missing from the neural network architectures, is the measure of uncertainty in the predictions, which may result in overconfident decision-making. This is why Bayesian neural networks (BNN) have gathered increased attention, because it offers uncertainty estimates of its parameters as probability distributions, providing uncertainty measurements of the prediction.

Bayesian neural networks have the ability to combine the properties of neural networks with Bayesian uncertainty modelling. However, the difficulties with choosing meaningful prior distributions and the intracability of inferring the posterior distribution, limits the effectiveness of these networks. Approximations for the model posterior is then necessary, and one popular approach is called variational inference. The posterior is then approximated by a variational distribution of a known functional form, in which the parameters are optimized so that it comes as close to the true posterior as possible. Blundell et al.

(2015) [4] provided a variational inference approach that resembles backpropagation in regular neural networks, and this method is called Bayes by backprop. It is, however, stated that variational inference doubles the amount of parameters that needs to be trained, without significant improvement of the results. Gal and Ghahramani (2016) [10] proposed in their paper an approach to variational inference, involving dropout during both training and inference, providing both the properties of uncertainty estimates as well as model robustness. This approach is called Monte Carlo dropout, and it is supposedly able to avoid the difficulties regarding priors and the posterior approximation, and still getting similar results.

The aim of this project is to compare the two mentioned methods for approximating a Bayesian neural network, namely Bayes by backprop, which in this project will mainly be referred to as Monte Carlo vaiational inference (MCVI), and Monte Carlo dropout (MCDO). The purpose of this is to find the model that shows best potential in regards to prediction results, robustness and practical implementation. The best approach will then be the basis for further work.

This project thesis will be structured in the following way:

1. **Chapter 2** will present some basic theory on machine learning, specifically neural networks, and then a theoretical background on Bayesian neural networks and the two proposed methods for approximation.

2. In **chapter 3**, a brief review of some general theory regarding the building of a neural network, and evaluation of a model, will be presented.

3. In **chapter 4**, the implemented models using the two approaches will be presented, together with the results from testing both on three different datasets, in three separate case-studies. For each case, the models will be discussed and evaluated based on the results.

4. Lastly, in **chapter 5**, a final evaluation and conclusion of the results will be presented, together with suggestions for future work.

CHAPTER 2

Background and theory

The following chapter will present a theoretical background for this project. The first two sections will address some general concepts of machine learning and explain the basic mechanisms of a feedforward neural network. In section 2.3, background theory on probabilistic learning and Bayesian neural networks will be presented. In addition, a more detailed explanation of two different approaches for implementation of a BNN, Monte Carlo variational inference and Monte Carlo dropout, will be provided.

2.1 The Basics of Machine Learning and Neural Networks

Machine learning is a branch within artificial intelligence which is the study of algorithms that improve themselves based on experience. Briefly explained, ML-algorithms use statistical approaches to discover patterns in data and apply that pattern to make educated guesses [13]. The ML model is constituted by a set of assumptions that makes inferences about unobserved data possible [11]. There are two types of learning that are most commonly used, namely supervised and unsupervised learning. Supervised learning is applied when the dataset the algorithm is being trained on, includes labeled data. This basically means that the intended or desired output is known, and the algorithm is learning the relationships and parameters based on the minimization of a cost function of the desired and predicted output. Unsupervised learning is applied when the training data is not labeled, and when the objective is to find some pattern, structure or clusters in the data, not knowing what the output should look like. An example of unsupervised learning is the mapping of genes into groups of genes with similar properties [21]. There are also two main types of problems one can encounter in when working with supervised machine learning, namely regression and classification problems. A problem is a classification problem if the objective is to predict a discrete output, a class label, for a training example, while its a regression problem if the output predicted is a continuous quantity [5]. This project, including the cases presented in chapter 4, will focus on regression with supervised learning.

2.2 Feed forward neural network

Figure 2.1 presents a typical neural network architecture. It is composed of layers of nodes, one input layer, n intermediate layers called hidden layers, and one output layer. Determining the number of hidden layers and the number of nodes in each layer is dependent on the problem and the type of model, and the numbers will vary. The figure does not show the node containing the bias term which should be added to each hidden layer and output layer.

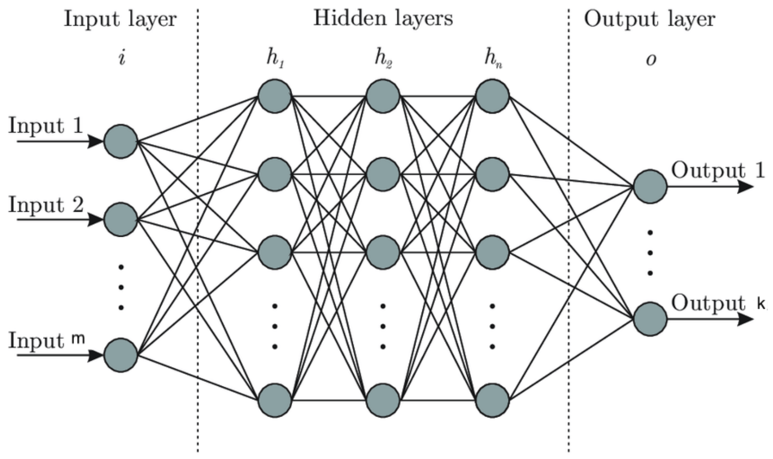


Figure 2.1: Simplified NN with an input layer, n hidden layers and an output layer. The bias unit is not shown in this figure.

The input layer receives external input variables from a dataset, and the number of nodes in this layer is determined by the number of different features in each training example. One training example i is composed of a set of features and their corresponding label, $\{\mathbf{x}_i, \mathrm{y}_i\}$. The nodes propagate the information in a "feed-forward" matter, through each layer where the output from a node in one layer constitutes an input for the next layer. A network layer can be expressed as a vector, $\mathbf{a}^i$, where $i \epsilon [0, n]$, n representing the number of hidden layers. The network can generally be represented as

$$\begin{aligned} \mathbf{a}^0 &= \mathbf{x}_i \\ \mathbf{a}^i &= \sigma(\mathbf{W}^i \mathbf{a}^{i-1} + \mathbf{b}^i) \\ \hat{y}_i &= \mathbf{a}^n \end{aligned} \tag{2.1}$$

Here, $\mathbf{W}^i$ is the weight matrix applied to incoming connections of layer i, that amplifies or dampens the signals of the transferred data. $\sigma(\cdot)$ represents the activation function, which is a bounded function such as a sigmoid function, that introduces non-linearity to the model [2]. Figures of two of the most commonly used activation functions can be found in Appendix D.

Ultimately, the information reaches the output layer and an optimization problem, $min_w J(\mathbf{w})$, can be solved. An example of a commonly used cost function for the optimization of regression problems is the mean squared error, $MSE(\hat{y}_i,y_i)$ [7]. The computed cost is the accumulated loss of the whole model, and this is propagated back into the network by computing the partial derivative with respect to the weight associated with every node in all the layers. This technique is called backpropagation, and it obtains the gradient of the cost function, $\nabla J(\mathbf{w})$, for the optimization, enabling tuning of the weights for best possible prediction performance [22].

The transferring of the entire dataset through the network with the corresponding cost computation, backprogagation and parameter tuning, make up one iteration or one *epoch* of the network training algorithm.

2.3 Probabilistic modelling: Bayesian neural networks

The basic principles of a probabilistic modelling approach is to replace unobserved quantities where uncertainty will occur, with a probability distribution [11]. BNN's are neural networks where the applied weights, normally estimated by point estimates, are the uncertain unobserved quantity that are represented by probability distributions, from which estimates can be drawn. It is then the parameters of these distributions that will be learned instead of the weights directly, and in this way uncertainty in the weight estimation, and thus in the model itself, is taken into account.

Uncertainties

There exists different types of uncertainties, and two main types of uncertainty in Bayesian/probabilistic modelling is aleatoric and epistemic uncertainty [11]. Aleatoric uncertainty refers to the noise in the observed data. This uncertainty is covered by placing a probability distribution upon the predicted value, and is an uncertainty that can not be reduced by e.g. adding more data. Epistemic uncertainty refers to the uncertainty of the model itself and is covered by the posterior distribution. This is modelled by placing a prior distribution over the model weights and trying to capture how much they vary with the given data. This is a type of uncertainty that can be reduced by adding more data. An example showing both aleatoric and epistemic uncertainty in the predictions of a Bayesian neural network, and a more thorough explanation of these, can be found in Appendix A.

Prior, likelihood and posterior

There are three important components that defines the probabilistic model, the likelihood, prior and posterior distributions. The likelihood is a conditional distribution of the data as a function of the model parameters and can be defined as $l(\mathbf{w}) = p(\mathbf{D}|\mathbf{w})$. It indicates the likelihood of observing the data, $\mathbf{D}$, given the parameters, $\mathbf{w}$. The prior, $p(\mathbf{w})$, is a distribution placed upon the parameters and gives a prior estimation of the parameters before observing any data. The posterior is a distribution defined by Bayes theorem, and is proportional to the product of the likelihood and the prior,

$$p(\mathbf{w}|\mathbf{D}) = \frac{p(\mathbf{D}|\mathbf{w})p(\mathbf{w})}{p(\mathbf{D})} \propto l(\mathbf{w})p(\mathbf{w}) \tag{2.2}$$

$p(\mathbf{D})$ is called the normalizing term, and will be explained in section 2.3.1. Following the properties of the prior and the likelihood, the posterior distribution posits all information that can be learned about the parameters, $\mathbf{w}$, from the given data. It gives the most likely parameters, given the observed data [19].

Estimating parameters

In general, when a neural network is introduced to an observed data input, $\mathbf{x}$, the objective is to make a prediction of the output, y, based on that data. To give an as accurate prediction as possible, one needs to have good estimates of the model weights, i.e. the model parameters, based on the training of the network on a labeled dataset.

One common way of estimating the parameters, $\mathbf{w}$, is to compute the maximum likelihood estimate (MLE). In this case, one seeks to find the parameter that maximizes the objective,

$$\mathbf{w}^* = \underset{\theta}{\operatorname{argmax}}\, p(\mathbf{D}|\mathbf{w}), \tag{2.3}$$

By applying the proportionality property of the posterior, given above, one can incorporate prior knowledge about the model parameters which then serves as a form of regularization. By finding the parameter that maximizes this posterior,

$$\mathbf{w}^* = \underset{\theta}{\operatorname{argmax}}\, p(\mathbf{D}|\mathbf{w})p(\mathbf{w}), \tag{2.4}$$

one gets the maximum a posteriori (MAP) estimate. In this case, the prior is assumed to be uniform, $p(\mathbf{w}) \propto 1$ [19]. However, these give only a point estimate that neglects the fact that there are other possible parameters that are nearly likely [15].

By instead using the full probability distribution of the posterior as an estimate, and by evaluating the shape of the distribution, one can acquire the confidence of the parameter estimates. The parameter estimate is then the average over all possible parameters of the distribution, which is known as marginalisation. This is referred to as Bayesian estimation, and is the basis of Bayesian inference in neural networks [15].

2.3.1 Bayesian inference

Bayesian inference makes it possible to make predictions that also take into account the uncertainties of all the weights of the network with respect to the posterior distribution, and one is able to more accurately model the epistemic uncertainty of the network [19].

The posterior distribution is, as mentioned, computed by Bayes' theorem, and in the case of a posterior distribution of the model parameters w given the dataset **D**, Bayes theorem can be written as

$$p(\mathbf{w}|\mathbf{D}) = \frac{p(\mathbf{D}|\mathbf{w})p(\mathbf{w})}{\int_w p(\mathbf{D}|\mathbf{w'})p(\mathbf{w'})d\mathbf{w'}} \tag{2.5}$$

Both the likelihood and the prior is fairly easy to compute, but the denominator component, $p(\mathbf{D}) = \int_w p(\mathbf{D} \mid \mathbf{w'})p(\mathbf{w'})d\mathbf{w'}$, can make the analytic computation of the posterior intractable for neural networks. $p(\mathbf{D})$ is known as the normalizing term and shows whether the data **D** is generated from the model. This computation requires an integration over all possible model parameters, and is called marginalization [20].

A posterior distribution is, in addition to estimating the weights of the network, used to make predictions of the output, y$_{pred}$, given some unseen input data, $\mathbf{D}_{new} = \{\mathbf{x}\}$. This can be done by computing the posterior predictive distribution with the approximated posteriors of the parameters as the prior, given by

$$p(y_{pred}|\mathbf{x}, \mathbf{D}_{new}) = \int p(y_{pred}|\mathbf{x}, \mathbf{w}, \mathbf{D})p(\mathbf{w}|\mathbf{D})dw \tag{2.6}$$

The distribution is the density of the likelihood of the prediction multiplied with the posterior distribution of the weight parameters **w**, and is equivalent to averaging predictions weighted by the posterior probability of the parameters. This makes it possible to quantify the aleatoric uncertainty by extracting the standard deviation of the output distribution.[17] On a more practical note, this means that the predicted output is retrieved by sampling from the output distribution. By extracting the model prediction N number of times and getting N sampled predictions, one can retrieve the mean prediction together with the epistemic uncertainty by taking the mean and standard deviation of the N prediction vectors, $\mathbf{y}_{pred}$. A good demonstration of this output is shown in Figure A.1 in Appendix A.

Due to the difficulties of analytically expressing the posterior, an alternative method for characterizing an approximate posterior is needed, and one such method is called variational inference, presented in the following section [19].

2.3.2 Variational inference

To deal with the problem of an intractable posterior, a popular approximation method called variational inference can be applied. The posterior is then instead approximated by a variational distribution, q($\mathbf{w}|\theta$) of a known functional form. In theory, any distribution could be chosen for this purpose, however, a family of distributions that is often chosen as the functional form for the approximation is the Gaussian family. This was also proven by Graves (2011) [12], in his paper on variational inference, to be the best choice for both posterior and prior distribution. θ = (μ,σ) is then the variational parameter which is learned during training of the network, so as to retrieve a member of the Gaussian family as close to the true posterior as possible [17]. Consequently, the problem becomes an optimization problem, where the objective is to minimize some divergence between the approximate and the true posterior [19]. Usually the Kullback-Liebler (KL) divergence as a function of the variational parameter, is used as the objective funtion, shown in Equation 2.7.

$$\theta^* = argmin_\theta D_{KL}[q(\mathbf{w}|\theta)||p(\mathbf{w}|\mathbf{D})] \tag{2.7}$$

θ^* is here the estimated variational parameter. Since the KL-divergence function originally requires computation of the posterior, one can derive the evidence lower bound (ELBO) as an alternative optimization objective. The ELBO for this case is defined as,

$$ELBO = -D_{KL}[q(\mathbf{w}|\theta)||p(\mathbf{w})] + \mathbb{E}[log(p(\mathbf{D}|\mathbf{w}))] \tag{2.8}$$

and is derived from the KL-divergence in the following way, [16]

$$\begin{aligned} D_{KL}[q(\mathbf{w}|\theta)||p(\mathbf{w}|\mathbf{D})] &= \mathbb{E}_q[log\frac{q(\mathbf{w}|\theta)}{p(\mathbf{w}|\mathbf{D})p(\mathbf{w})}p(\mathbf{D})] \\ &= \mathbb{E}_q[log(q(\mathbf{w}|\theta)) + log(p(\mathbf{D}) - log(p(\mathbf{D}|\mathbf{w}) - p(\mathbf{w})] \\ &= \mathbb{E}_q[log(q(\mathbf{w}|\theta)) - log(p(\mathbf{D}|\mathbf{w}) - log(p(\mathbf{w}))] + log(p(\mathbf{D}) \\ &= \mathbb{E}_q[log\frac{q(\mathbf{w}|\theta)}{p(\mathbf{w})} - log(p(\mathbf{D}|\mathbf{w})] + log(p(\mathbf{D}) \\ &= D_{KL}[q(\mathbf{w}|\theta)||p(\mathbf{w})] - \mathbb{E}[log(p(\mathbf{D}|\mathbf{w}))] + log(p(\mathbf{D})) \end{aligned} \tag{2.9}$$

The first term is the KL-divergence between the variational distribution and the true prior, which is known as the complexity cost, and works as a form of penalty term in the optimization. The middle term is the negative expected log-likelihood, called the likelihood cost, and the last term is called the marginal cost. Since the marginal cost is independent of θ and therefore constant when varying q, the resulting cost, also known as the variational free energy, is defined as [4]

$$J(\theta) = D_{KL}[q(\mathbf{w}|\theta)||p(\mathbf{w})] - \mathbb{E}[log(p(\mathbf{D}|\mathbf{w}))] \tag{2.10}$$

where $J(\theta) = -ELBO$. Therefore, minimizing the KL-divergence is in fact equivalent to maximizing the ELBO, and the resulting optimization objective is shown in Equation 2.11 [19].

$$\theta^* = argmax_\theta\{\mathbb{E}_q[log(p(\mathbf{D}|\mathbf{w})] - D_{KL}[q(\mathbf{w}|\theta)||p(\mathbf{w})]\} \tag{2.11}$$

However, analytical computation of the ELBO's gradients is not possible and introduces additional difficulty regarding optimization [19]. Blundell et al. [4] proposed a method using an unbiased estimator for the gradients, often referred to as the Monte Carlo estimator, resulting in a backpropagation-like algorithm for variational inference called Bayes by Backprop. This method is introduced below.

Monte Carlo variational inference

A monte carlo estimatior is useful for computing expectations of distributions. It draws samples from this distribution to compute a sample average of the respective function. With some altering of the ELBO, it can be found that one needs to compute an expectation w.r.t. the variational approximation $q(\mathbf{w}|\theta)$. Monte Carlo estimation thus involves sampling N samples, $\theta \sim q_\theta$, and computing the average so that the ELBO and its gradient can be estimated [19]. The resulting approximated cost can be shown as,

$$J(\theta) = \sum_{i=1}^{n} log(q(w^i|\theta)) - log(p(w^i)) - log(p(\mathbf{D}|w^i)) \tag{2.12}$$

where w^i is the ith Monte Carlo sample drawn from the approximate variational distribution, $q(\mathbf{w}|\theta)$ [4].

One MC estimator referred to as a path-wise derivative estimator or reparameterization gradient estimator, involves a re-parameterization trick which allows gradient computation [19]. The mentioned Bayes by backprop method by Blundell et al uses a generalization of this re-parameterization trick to learn the distributions of the weights. In cases where q_θ is a normal distribution, a sample, ϵ, is drawn from a parameter-free distribution, $N(0, \mathbf{I})$, and is then transformed with a deterministic function, t(μ,σ,ϵ) = μ+$\sigma \odot \epsilon$ [17]. This transformation makes normal backpropagation through the network, with calculations of gradients for parameter updates, possible [4]. In more general terms, this method is also known as Monte Carlo variational inference (MCVI).

Prior distribution

Regarding the priors of the network, is should be mentioned that there are several options for choosing both the applied distributions and the parameters of them. As mentioned previously in this section, Graves suggests that distributions from the Gaussian family for both the prior and the posterior is the better choice [12]. In his paper, the prior parameters μ and σ^2 (mean and variance) are also learned during training. Blundell et al. argues in their paper that using a fixed scaled mixture prior of two Gaussian distributions, yields best performance for Bayes by backprop, and will therefore be used in the implemented MCVI models presented in chapter 4 [4]. This means that the parameters of the two distributions are *hyperparameters*, meaning they are determined and fixed prior to training the model.

2.3.3 Monte Carlo dropout

Gal and Ghahramani [10] proposed a more practical approach to obtaining uncertainty of a model that is equivalent to variational inference with MC sampling, explained in the previous section. This approach is called Monte Carlo Dropout (MCDO), and includes performing dropout on every layer in the network both during training and testing, with some probability of units being dropped out. The variational distribution resembles in this case a Bernoulli distribution by being a mixture of two Gaussian processes with small variances and one of the means fixed at zero. A mathematical derivation of this, and proof of exactly how and why this is equivalent to MCVI, is beyond the scope of this project. However, this can be found in Gal's phd thesis (2016) [8] and the mentioned paper of Gal and Ghahramani. By approximating the minimization of the KL-divergence between the variational and the posterior distribution, one arrives at the following minimization objective for regression problems,

$$J(\mathbf{M}_i, m_i) = -\frac{1}{N}\sum_{n=1}^{N} log(p(D_n|\hat{w}_n)) + \sum_{i=1}^{L}(\frac{1-p}{2N}||\mathbf{M}_i||^2 + \frac{1}{2N}||m_i||^2) \tag{2.13}$$

where $\mathbf{M}_i$ and m_i are the variational parameters of q_w, p is the dropout probability, N is the number of data points and $\hat{w}_n$ is the weight sampled from the variational distribution, $\hat{w}_n \sim q_w$, which is now of the approximate form [9]

$$\begin{aligned} &w = \{\mathbf{W}_i\}_{i=1}^{L} \\ &\mathbf{W}_i = \mathbf{M}_i \cdot diag([z_{i,j}]_{j=1}^{K_i}) \\ &b_i = m_i \\ &z_{i,j} \sim Bernoulli(p_i), \text{ for i = 1,..,}L\text{ , j=1,..,}K_{i-1} \end{aligned} \tag{2.14}$$

L is here the number of layers, K_i the number of units in layer i and p_i the dropout probability of layer i. $\mathbf{M}_i$ and $\mathbf{m}_i$ are the variational parameters for the weight matrices, w_i, and the bias vector, b_i, respectively. This makes q_w a distribution over matrices which columns are set to zero [9].

The practical implementation of MCDO is done by dropping random units during training and at each test iteration, for N iterations, in which one is left with N empirical samples from an approximate posterior of the predictions. From this, one can compute the estimate of the predictive mean and predictive standard deviation from the N sample vectors, in the same way as for Bayesian inference, to collect the model prediction and epistemic uncertainty. During training, the objective is to minimize the cost function, to estimate the weight parameters. The first term of the cost function given above is the negative log-likelihood, and it can be shown that maximizing the likelihood (MLE), as described in section 2.3, gives the same optimal parameters as when minimizing the mean squared error (MSE). The optimization objective for the practical implementation can thus be derived as

$$J(\mathbf{M}_i, \mathbf{m}_i) = \frac{1}{N}\sum_{n=1}^{N} ||y^i - \hat{y}^i||^2 + \lambda\sum_{i=1}^{L}(||\mathbf{M}_i||^2 + ||m_i||^2) \tag{2.15}$$

with a generalized weight decay, λ, for the $l2$ regularization [9]. Using $l2$ regularization in point estimate neural networks is equal to setting a Gaussian prior with zero mean and a fixed variance, from a Bayesian point of view [12]. One could say that the regularization term works as a soft constraint, just as the prior does for the posterior. These are similar arguments to those used in the paper by Gal and Ghahramani to demonstrate the equivalence between MCVI and MCDO, and their respective loss functions [17].

CHAPTER 3

Building and evaluating a neural network

The following chapter will address some general considerations that needs to be taken into account when building and evaluating a neural network. Firstly, the hyperparameters of a neural network, and specifically of a MCVI and MCDO network, will be addressed. Then, some methods for pre-processing of data necessary before training a machine learning model will be briefly explained. Lastly, some important aspects regarding the evaluation of a model will be presented in terms of performance evaluation and some steps to avoid poor performance.

3.1 Tuning of hyperparameters

Hyperparameters are the parameters that are determined prior to training the model, and their values determine the structure of the network. Examples of such parameters are the number of hidden units in the network, the learning rate fed to the optimizer and the dropout probability of a dropout NN. Tuning these parameters correctly is crucial to getting good results. The tuning could be done manually, however, this can be tedious work and give non-optimal solutions. Methods exist for automatic tuning by optimization techniques such as the random search algorithm or the grid-search algorithm. However, these methods have shown to be both computationally costly and time-consuming, and later research have shed light on Bayesian optimization as an approach that outperforms these methods as well as manual tuning by experts. An elaboration and further explanation of Bayesian optimization can be found in Appendix C.

Another hyperparameter that is important for network performance, is the choice of activation function for the different layers. All activation functions approximate a different Gaussian process covariance function that corresponds to different uncertainty estimates. One should therefore think of what the expected result is before choosing the activation.

For instance, if one expects the prediction uncertainty to increase the further outside of a training data range it predicts (as is the case in the implementation case-studies in chapter 4), then a rectified linear unit (Relu) function should be used. This function is shown in Figure D.1a in Appendix D. When comapring it with the Sigmoid function shown in Figure D.1b, one can see that the Relu function does not saturate as the Sigmoid does, and this will lead to an increasing uncertainty [10].

Table 3.1 shows the different hyperparameters that are necessary to tune for an MCVI and a MCDO network.

Method	Hyperparameter	Description
MCVI	Prior distribution	For example a Normal distribution, $N(\mu, \sigma)$. Could also choose mixed priors of two or more distributions. This parameter could also be trained.
	Prior distribution parameters	In the case of a Normal distribution, the parameters are μ and σ
	Posterior distribution	For example a Normal distribution, $N(\mu, \sigma)$, where μ and σ are trained.
MCDO	Dropout probability, p	The probability of a unit not being dropped, (1-p) of being dropped.
	Weight decay, λ	Weight decay of $l2$-regularization.
General	Learning rate, lr	Determining size of optimization steps.
	Units, K	The number of hidden units in each hidden layer.
	Layers, L	The number of hidden layers in the network.
	Activation function, a	Activation function applied to each unit in the network, for example Relu or Sigmoid.

Table 3.1: This table shows the different hyperparameters that needs tuning for both Monte Carlo variational inference, and Monte Carlo dropout. The general hyperparameters are common for both methods.

3.2 Data pre-processing

Data pre-processing is a data mining technique that can have significant impact on the performance of a machine learning model, as it transforms raw data into an understandable format. It produces a dataset that is better fitted for model training, and hopefully results in better generalization performance of the model, depending on the processing steps chosen. Common pre-processing steps include data-cleaning and data-transformation. There also exists several methods for both cleaning and transformation of data, and below are some of the most popular methods listed [14][1].

Data Cleaning

- *Removing outliers:* This method involves removing data points that lies outside of some pre-determined bound that incorporates a percentage part of all the same type data. Outliers have a great impact on model prediction, and is therefore an important step.

- *Missing data treatment:* These methods involves treating incomplete datasets, by for example listwise deletion, where all data corresponding to the missing point is removed.

Data Transformation

- *Scaling the data:* The data is scaled so that all features are within the same order of magnitude, and no features will wrongfully influence the the model more than others. Standardization is a very common scaling method, and is shown in Equation 3.1.

- *Smoothing the data:* Smoothing is done when the raw data is very noisy, and some techniques often used include binning or regression, which will not be elaborated further in this report.

The mentioned standardization method for scaling of data can be shown as,

$$x_{scaled} = \frac{x_{train} - \mu_{train}}{\sigma_{train}} \tag{3.1}$$

where x_{scaled} are the training and test-features, both scaled in terms of the training features [1]. The importance of data preprocessing and the choice of processing methods will vary depending on the type of data, the type of machine learning problem and the network that is being trained.

3.3 Model evaluation

It is important to review the performance of a model before using it for prediction tasks. There are many procedures and methods for discovering different types of problems with the model and developing solutions to fix them. However, this section will focus on the very common problems of over- and underfitting, how to detect it and what measures that can improve the model performance.

3.3.1 Over- and underfitting

Over- and underfitting are some of the most common pitfalls when building and training a machine learning model, and occurs when the model hypothesis is too generalized or too specific to the training data. Luckily, there are ways of detecting such behaviour in the model that are also quite intuitive, such as cross-validation which is explained below.

- When *over-fitting* occurs, it means that the output hypothesis has *high variance* and the model fits very well to the training data, while not generalizing well to unseen data, like the validation or test set.

- When *Under-fitting* occurs, it means that the output hypothesis has a *high bias* and the model is underestimating the model parameters and is generalizing too much. This can happen when the model is not capturing the complexity of the data, or is the wrong type for that particular dataset.

The ideal is to find an optimal trade-off between these two phenomena and this is referred to as the bias and variance dilemma [23]. Depending on if the model suffers from high variance or high bias, there are different measures that can be considered to improve the outcome. An example of an overfitted model and how it was improved can be found in Appendix B.

3.3.2 Hold-out cross validation and data splitting

A common procedure when training a machine learning model on some training data, is to split the data into a training, validation and test set. This way, one can train the model on the sampled training data and at the same time, periodically validate the model according to some loss function applied to the validation data. This is called *hold-out cross validation* [23]. The same loss function is applied to the training data, and by tracking both the training loss and the validation loss, one is able to see whether the model is overfitting, underfitting or performing in an expected matter overall. This works as a good indicator of how the model is performing, and what might be wrong or lacking in the model. The remaining test data is then used to test the model performance by predicting a result based on the test features, and evaluating that result according to some error metric comparing the test labels to the predicted values.

Figure 3.1 show a common split of training-, validation and test-datasets, where typically 20% of the available data is first sampled and held out for later testing. The validation set is then sampled from the resulting training set, and typically consists of 20% of the training data. Statistical sampling techniques can be used for splitting, and one common procedure is to use simple random sampling (SRS). The different datasets are then sampled randomly with a uniform distribution, and each sample has equal probability of being selected. One problem that can occur with this method, often when the dataset itself is not uniformly distributed, is that the sampled set does not cover the original dataset in a sufficient way. This can in some cases lead to overfitting to the training data [23].

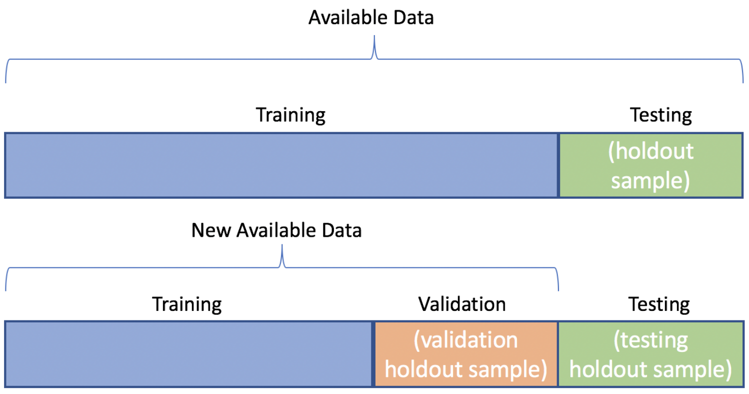


Figure 3.1: Figure shows typical data-splitting procedure. First, some fraction of the available data is sampled and held out for later testing of the model. The remaining dataset is sampled to create the validation set which is also held out and used to validate the model during training. Typical fractions of the resulting training-, validation- and training-datasets are 0.6, 0.2 and 0.2 respectively [3].

3.3.3 Evaluation

Figure 3.2 shows three examples of plotting the validation and training loss for each training epoch. In case B the validation loss is higher than the training loss and increasing, and this is usually a sign that the model is overfitting to the training data and is not generalizing well to unseen data. Several factors can lead to overfitting, but a typical cause is a too high model complexity, i.e. too many hidden layers and units, following a large number of trainable parameters. Methods for reducing overfitting may include adding regularization terms to the cost function, adding dropout layers to the network or increasing the training size, among others [18]. In case A, one can see that the validation and training loss has stagnated and is generally high. This is a typical sign of underfitting, and that the model is not able to learn from the data. In this case, one might have a network that does not have enough parameters to properly fit to the given training data, and a solution may be to increase the model complexity. Case C shows a trend that is preferable in most cases, where both training and validation loss is decreasing in a similar matter throughout the training period.

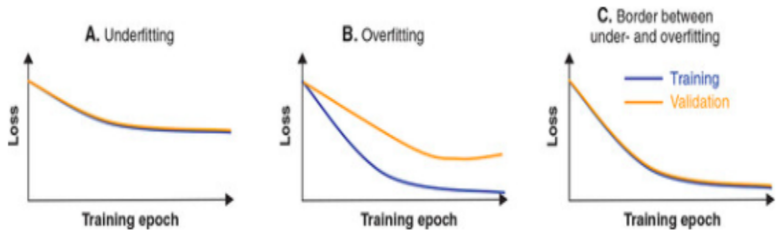


Figure 3.2: The figure shows three simplified plots of validation and training loss during training. Case **A** shows underfitting, where the training and validation loss seizes to decrease after a short period, remaining at a high value. Case **B** shows overfitting, where the validation loss starts to increase, while training loss decreases further. Case **C** shows a "just right" fitted model, where training and validation loss decreases somewhat simultaneously throughout the training period [25].

Another problem that might occur when trying to fit a network to some training data, is that validation error is much lower than the training error. This is counter-intuitive as ML models should not be better at predicting the unknown than what it has learned. On the other hand, this could also make sense if the data splitting process has given a validation set that is generally easier to predict from, than it was to learn from the training set. In any case, this is an undesirable result, and re-evaluation of the splitting method as well as adding more data could be solutions to this problem.

CHAPTER 4

Model testing and evaluation

This chapter will present three case-studies in three separate sections. In each case, an MCVI model and an MCDO model will train on different datasets, to examine the properties of, and comparing the models. The datasets and the following pre-processing steps that were used prior to training the models, will be presented in the beginning of each section. The models will then be described, followed by a presentation of the plots of the test prediction results and the training and validation losses for both models. At the end of each section, the results for the respective case will be evaluated in light of implementation, average training time, prediction accuracy and the plotted uncertainties of the models. The model implementations in Python can be found in Appendix F.

For implementation of the models in Python, the machine learning libraries TensorFlow and Keras were used. Table 4.1 show the versions of libraries in Python used for the implementation, and the version of Python used was Python 3.7. The aleatoric uncertainty in all three cases was neglected in the plotting of the results, and only the epistemic uncertainty is shown. In addition, only the MSE-plots will be shown in all cases for comparison purposes. However, the training and validation loss in terms of the ELBO function for the MCVI models can be shown in Appendix E, in addition to other left out figures.

Library	Version
TensorFlow	2.3.0
TesnorFlow probability	0.11.2
Keras	2.3.1
Keras tuner	1.0.2
Scikit learn	0.23.2
Scipy	1.4.1
Pandas	1.1.2

Table 4.1: Versions of Python libraries used for implementation.

The CPU specifications for the computer used for implementation of the models: Intel(R) Core(TM) i7-6600U CPU 2.60GHz 2.80GHz.

4.1 Case 1: Sinusoidal function

This case-study was based upon an example implemented by Krasser (2019) [16], and the dataset consists of an input vector **x** with m number of equally spaced data points in the range (-0.5,0.5), and an output vector, **y**, with the associated y-labels. The input and output data are related by the following sinusoidal function,

$$f(\mathbf{x}) = 10\sin(2\pi\mathbf{x}) \tag{4.1}$$

Gaussian noise was added to create some randomness in the data such that **y** = $f(\mathbf{x})$ + ϵ, where $\epsilon \sim N(0,\sigma^2)$ with constant σ=1.5 for all input points. The training dataset can be presented as,

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdots \\ x_m \end{bmatrix}, \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \cdots \\ y_m \end{bmatrix}$$

where a training example is represented as $\mathrm{D}_i = \{\mathrm{x}_i, \mathrm{y}_i\}$. In this example, there is only one input feature to each output.

The training data was split into a training and validation set by extracting 20% of the data for validation. The test set was constructed by 1000 equally spaced x-values in the range (-1.0,1.0). This is not normal procedure, but was done for the purpose of demonstrating the increasing uncertainty when using the model for extrapolation. For this reason, the MSE between the test labels and the predictions will not be used to evaluate this model since these values would be higher than what is expected when following normal procedure. The training, validation and test set was scaled according to Equation 3.1 in

Section 3.2, with the *StandardScaler*-function in the *Sklearn* library, and no other data-preprocessing steps were necessary to follow.

The model mean and standard deviation shown in the results below are based on 500 predictions on the test data, and all data was scaled back to original value before plotting.

4.1.1 MCVI model

This model is based on Krasser's (2019) [16] implementation of an MC variational inference BNN. The variational distribution, q_θ, was chosen to be of the Gaussian family of distributions. The prior distribution was fixed and of a mixed form of two Gaussian distributions, with π determining the ratio, and σ_1 and σ_2 determining the variances between the two. The network was composed of 2 dense variational layers with 20 hidden units in each layer, and the Relu activation function was applied to the units in both layers. The hyperparameters are given in table 4.2 below, and were determined as suggested by Krasser (2019) in his implementation. The number of training epochs, trainable parameters and training data points are given in table 4.3.

	Layers, L	**Units**, K	**Activation**, a	**lr**	σ_1	σ_2	π
MCVI	2	20 20	Relu Relu	8e-3	1.5	0.1	0.5

Table 4.2: Hyperparameters of MCVI model. σ, μ and π are specific for the variational inference model.

Epochs	**Trainable parameters**	**Training data points**
3000	962	200

Table 4.3: The number of training epochs, trainable parameters in the network and number of training data points used to train MCVI model for case 1.

4.1.2 MCDO model

The MC dropout model was built with the TensorFlow-Keras implemented *Dense* and *Dropout* layers. It consisted of 2 layers with 20 hidden units in each layer, and the Relu activation function was used in this model as well. The hyperparameters are given in table 4.4 below, and were determined by manual tuning due to non-optimal results using Bayesian optimization. The optimization algorithm gave, however, a good starting point for manual tuning which was exploited. The number of training epochs, trainable parameters and training data points are given in table 4.5.

	Layers	**Units**	**Activation**	**LR**	p	λ
MCDO	2	20	Relu	1e-3	0.05	1e-4
		20	Relu			

Table 4.4: Hyperparameters of MCDO model. p and λ are specific for the MC dropout model.

Epochs	**Trainable parameters**	**Training data points**
3000	962	200

Table 4.5: The number of training epochs, trainable parameters in the network and number of training data points used to train MCDO model for case 1.

4.1.3 Results

Figures 4.1 and 4.2 show the MCVI model's prediction of the output when made on the test set, and the corresponding losses computed during training, respectively.

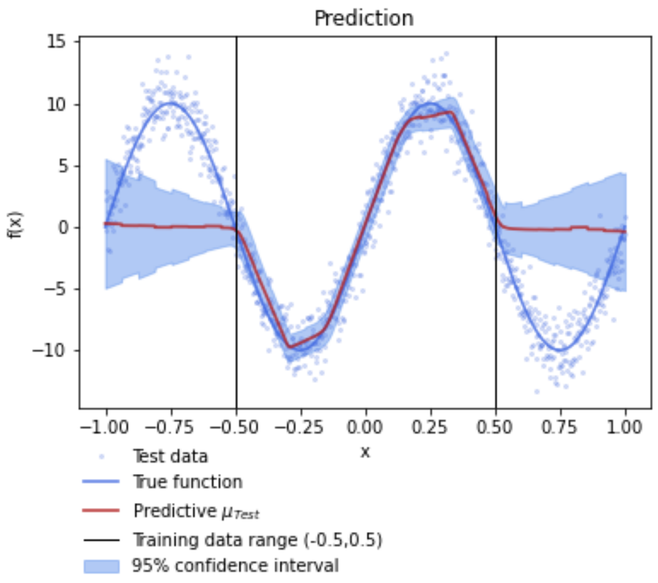


Figure 4.1: Prediction based on the test data in case 1 for the Monte Carlo variational inference model, with epistemic uncertainty.

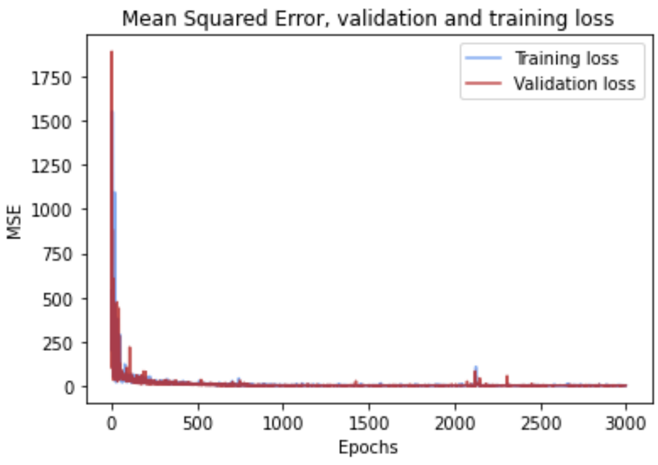


Figure 4.2: Mean squared error between model output and y-label for the Monte Carlo variational inference model in case 1, computed for validation and training data at each training epoch.

Figures 4.3 and 4.4 show the MCDO model's predicted output of the test data and the corresponding losses computed during training, respectively.

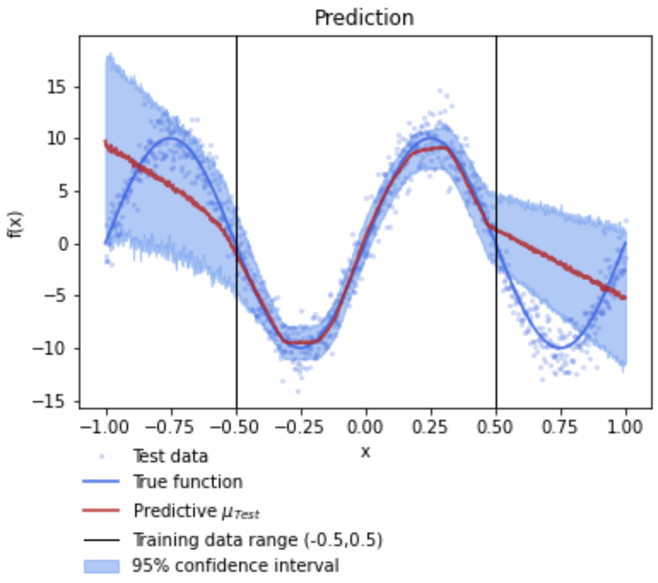


Figure 4.3: Prediction based on the test data in case 1 for the Monte Carlo dropout model, with epistemic uncertainty.

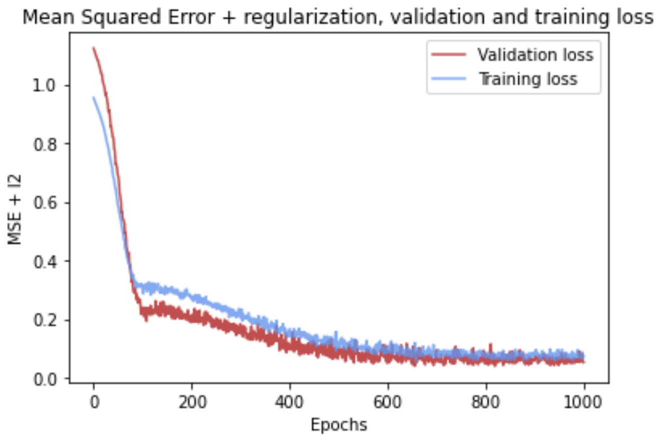


Figure 4.4: Mean squared error between model output and y-label with additional $l2$ regularization, for the Monte Carlo dropout model in case 1, computed for validation and training data at each training epoch.

Model	MSE training	MSE validation	Avg. training time [s]
MCVI	2.9283	2.8216	48.14
MCDO	0.0674	0.1044	25.78

Table 4.6: Mean squared error for training and validation set and average training time, for both models in case 1.

4.1.4 Discussion

The result of the prediction done with the MCVI model, shown in Figure 4.1, showed the expected trend in regards to uncertainty. The model seems to be quite certain about its prediction within the range of the training data, while the epistemic uncertainty increases when trying to predict outside the range. This is in accordance with the fact that machine learning in general is not good for extrapolation, and should not predict with certainty in those cases. The model mean, shown in red, also seem to give a good prediction of the true function within the training range. The fact that the uncertainty seems to increase linearly outside the training data range, can be connected to the type of activation function used for this model, as explained in Section 3.1. The Relu activation function shown in Figure D.1a in Appendix D, which increases linearly with increasing input, is used in both models in this case. Figure E.1 show an example where the MCVI model with the same hyperparameter tuning, was trained with Sigmoid activation in both layers instead, and it is clear that the uncertainty has reached some kind of saturation in which it does not increase.

When evaluating the training and validation loss, shown in Figure 4.2, it is difficult to interpret whether the trend is decreasing with the training epochs due to the scale of the plot. However, Figure E.3 in Appendix E.1 show the same plot, only here with the limits of the y-axis significantly decreased. In this figure, one can see that the loss is relatively noisy, which is common for Bayes by backprop when only one MC sample is taken each iteration for evaluation of the ELBO, presented in Section 2.3.2 [17]. However, one can see a clear downwards trend until around 1500 epochs. One could then have considered to stop the training at 1500, but when trying this the prediction result became significantly worsened. This might be due to the noisiness in the error, and thus that one should increase the number of epochs to reach a state with more stable gradient computation. The plot for the ELBO-loss, which the parameter optimization was based on, is shown in Figures E.2a and E.2b in Appendix E.1. Here, the validation loss is lower than the training loss for all epochs. As explained in Section 3.3.3, this is counter-intuitive, as the model should be better at predicting the data it is being trained on than unseen data. Due to the fact that computing the ELBO cost function involves stochastic sampling from distributions, as explained in Section 2.3.2, and that it is not as intuitive as the MSE, it is difficult to interpret what might be the reason for this result. It could be a sign that the splitting of the data, done with simple random sampling, result in a validation set that does not cover the training range in a sufficient way. It could also be that the model should have been trained for longer even though the loss seemed to have converged. However, neither increasing the epochs nor changing the split ratio or random seed for extracting datapoints, seemed to make a difference.

The prediction with the MCDO model, shown in Figure 4.3, showed similar results to the MCVI model in terms of uncertainty and prediction mean. However, the uncertainty within the training range is slightly larger, which could mean that this model would perform better with more data provided for training, as epistemic uncertainty can be reduced by adding more data. It could also be a sign of sub-optimal hyperparameter tuning, as the tuning in this case was done by trial and error, while the tuning of the MCVI model in this case followed the suggestions of Krasser (2019) [16], which are assumed to give optimal results. However, as the test data is noisy, it could be that the MCDO model gives the more reasonable result as the confidence interval covers more of the test data points than the MCVI model, and therefore gives a more realistic prediction of the test data.

Figure 4.4 show the MSE + regularization loss for the MCDO model. The loss-curves are far less noisy than for the MCVI model, which was expected as the practical implementation of this model does not include stochastic sampling. It shows a clear downwards trend, and even though the validation loss is lower than the training loss in the beginning of the curve, the deviation is only 0.037 at the end of the training epochs, which in this case could be regarded as a sufficient result. In this case the training data was constructed by equally spaced datapoints, and this could have affected the randomness of the selected validation set, and further the models ability to predict good and even better results from the validation data.

Table 4.6 show the MSE after the last training epoch for validation and training loss, as well as the average training time, for each model. The MCVI model used 3000 epochs for training, before reaching a sufficiently low MSE. This was three times the number that the MCDO model needed, which makes sense as learning the posterior with variational inference is in general much slower than regular backpropagation [17]. The loss-curves for this model also converged at a higher mean squared error than the MCDO model. It was expected that the runtime of the MCVI model would be even slower compared to the MCDO model, which only trained for 1000 epochs, due to the higher number of trainable parameters in this model. However, when testing the time it took for the MCDO model to train for 3000 epochs, it was only 4 seconds faster. This might be due to the simplicity of the dataset, and that a larger time difference would occur with more complex data and model structure.

4.2 Case 2: Cubic function

In this case-study the dataset was contructed, and can be represented, in the same way as for case 1, but with a different function, $f(\mathbf{x})$. The $\mathbf{x}$-vector consisted in this case of m number of equally spaced points in the range (1.5,5). The dataset was constructed with the following cubic function,

$$f(\mathbf{x}) = 0.1\mathbf{x}^3 - 0.7\mathbf{x}^2 + \mathbf{x} \tag{4.2}$$

with Gaussian noise such that $\mathbf{y} = f(\mathbf{x}) + \epsilon$, where $\epsilon \sim N(0,\sigma^2)$ with constant $\sigma = 0.1$.

The training data was split into a training and validation set by extracting 20% of the data for validation. The test data was also in this case constructed separate from the training data, for the same reasons as for case 1, by 1000 equally spaced x-values in the range (0,6.5). The training, validation and test data was scaled according to Equation 3.1 in Section 3.2, with the *StandardScaler*-function in the *Sklearn* library, and no other data-preprocessing steps were necessary to follow.

The model mean and standard deviation shown in the results below are based on 500 predictions on the test data, and all data was scaled back to original value before plotting.

4.2.1 MCVI model

The MCVI model in this case was built with the *DenseVariational* layer in TensorFlow-Keras, with mixed non-trainable priors. A Gaussian distribution was chosen as the posterior. This layer is based on the same logic as the dense variational layers in Krasser's model from case 1. The model consisted of 1 hidden layer with 10 units, and the number of training epochs, trainable parameters and training data points are given in table 4.8. The Hyperparameters are given in table 4.7, and were determined by manual tuning due to non-optimal results using Bayesian optimization. The optimization algorithm gave, however, a good starting point for manual tuning which was exploited.

	Layers	**Units**	**Activation**	**LR**	σ_1	σ_2	π
MCVI	2	10	Relu	8e-3	1	1e-7	0.5
		10	Relu				

Table 4.7: Hyperparameters of MCVI model for case 2. σ, μ and π are specific for the variational inference model.

Epochs	**Trainable parameters**	**Training data points**
3500	84	700

Table 4.8: The number of training epochs, trainable parameters in the network and number of training data points used to train MCVI model for case 2.

4.2.2 MCDO model

The MCDO model was built with the same TensorFlow-Keras layers as in case 1, and the model had 2 hidden layers with 20 hidden units in each layer. The Relu activation function was used in both layers. The number of trainable parameters, training epochs and training data points are given in table 4.10. The hyperparameters are given in table 4.9, and were determined by manual tuning due to non-optimal results using Bayesian optimization, as described in Appendix C. The optimization algorithm gave, however, a good starting point for manual tuning which was exploited. The resulting number of trainable parameters was 481.

	Layers	**Units**	**Activation**	**LR**	p	λ
MCDO	2	20 20	Relu Relu	1e-3	0.05	5e-5

Table 4.9: Hyperparameters of MCDO model for case 2. p and λ are specific for the MC dropout model.

Epochs	**Trainable parameters**	**Training data points**
500	481	700

Table 4.10: The number of training epochs, trainable parameters in the network and number of training data points used to train MCDO model for case 2.

4.2.3 Results

Figures 4.5 and 4.6 show the prediction made on the test data and the loss computed during training for the MCVI model, respectively.

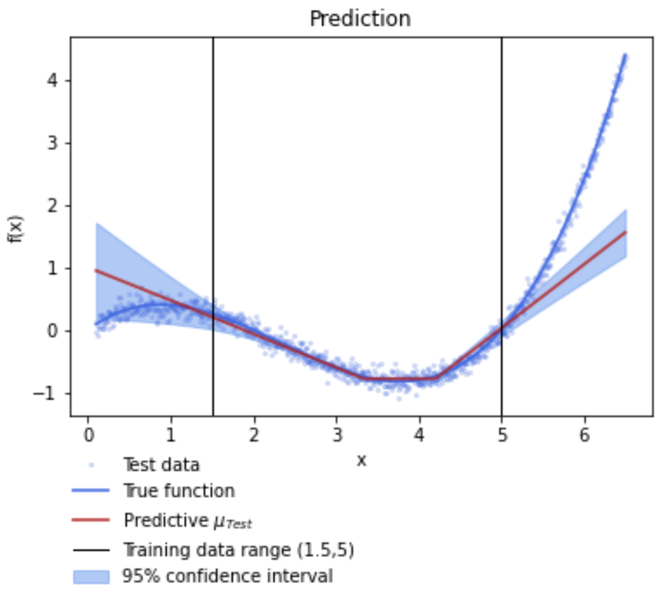


Figure 4.5: Prediction based on test data in case 2 for the Monte Carlo variational inference model, with epistemic uncertainty.

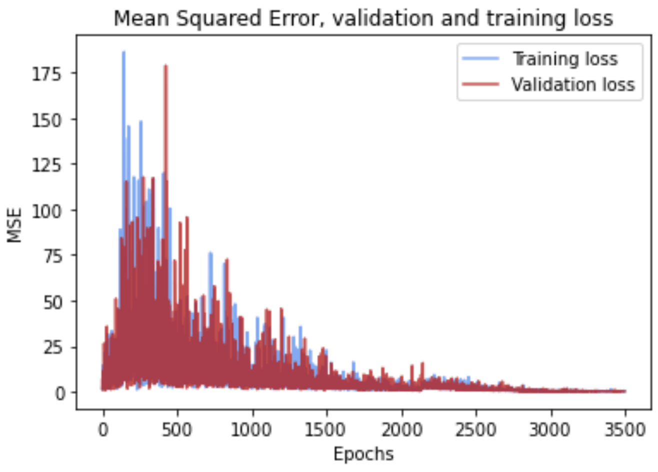


Figure 4.6: Mean squared error between model output and y-label for the Monte Carlo variational inference model in case 2, computed for validation and training data at each training epoch.

Figures 4.7 and 4.8 show the prediction made on the test data and the loss computed during training for the MCDO model, respectively.

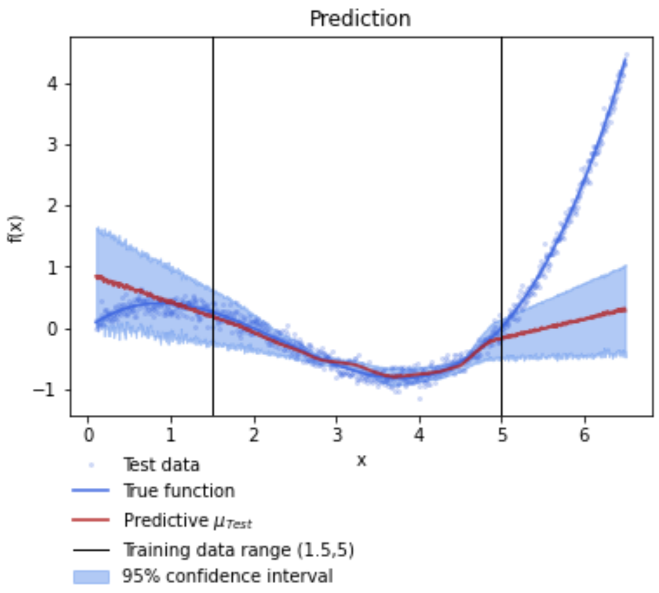


Figure 4.7: Prediction based on test data in case 2 for the Monte Carlo dropout model, with epistemic uncertainty.

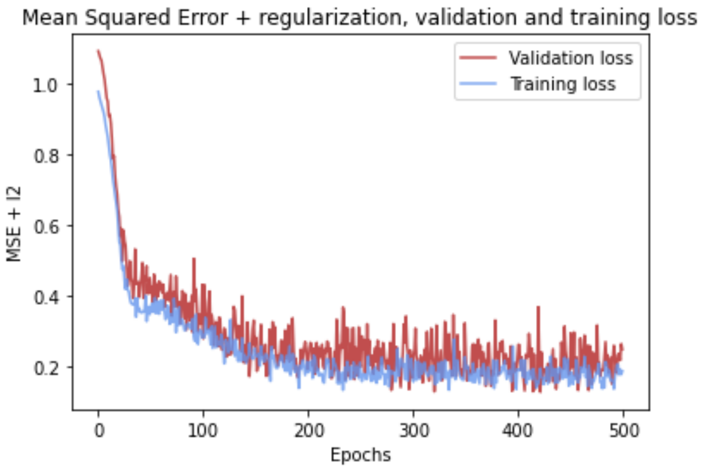


Figure 4.8: Mean squared error between model output and y-label with additional $l2$ regularization, for the Monte Carlo dropout model in case 2, computed for validation and training data at each training epoch.

Model	MSE training	MSE validation	Avg. training time [s]
MCVI	0.2470	0.2072	55.24
MCDO	0.1858	0.2475	13.58

Table 4.11: Mean squared error for training and validation set and average training time for both models in case 2.

4.2.4 Discussion

Figure 4.5 shows the same expected results as for case 1, with a linearly increasing uncertainty outside the training data range. However, it seems slightly overconfident within the training data range, where the standard deviation is very small and the model mean is not identical to the true function over the entire range. This might be due to the fact that the model structure is relatively simple with only one hidden layer, and therefore not introducing much uncertainty. The hyperparameters regarding the prior distribution of the MCVI model were tuned based on the article by Blundell et al. [4], in which the example was a classification problem. Still, these parameter values seemed to give good results for this model as well, despite this being a regression problem.

The training and validation loss curves, shown in Figure 4.6, show a relatively gradual downwards trend, even though the decrease seems to be mainly due to the decreasing level of noise. Figures E.4a and E.4b in Appendix E.2 show the ELBO-loss of the training and validation data, and here it is more clear that the actual loss is decreasing with the number of epochs.

The MCDO model needed only 500 epochs to converge to minimum MSE values for the training and validation loss. Figure 4.7 show the prediction on the test data, and it shows a similar trend as for the MCVI model, only with larger confidence interval both within the training range, and a higher increase on the outside. This model had two hidden layers with 20 units in each layer, which coincides with the theory that higher complexity in the model introduces more uncertainty in the predictions.

Figure 4.8 show the loss curves computed during training, and the trend is clearly decreasing also in this case. However, training loss is now slightly lower than validation loss for all epochs, but both curves are decreasing simultaneously, which is the preferred result as described in Section 3.3.3. This shows that the model is training and improving performance on the training data, and managing to generalize to the unseen validation data.

Table 4.11 show the MSE of the training and validation data at the end of the training epochs, as well as the average training time, for each model. The MCDO model was significantly faster, as was expected due being trained for only 500 epochs, while the MCVI model was trained for 3500 epochs. One might have expected the runtime of the MCVI model to be quite fast, as it only had 84 trainable parameters. However, as mentioned for case 1, the training time might not be affected by the number of parameters in a very large sense when the dataset is as simple as this one, with only one training feature. Overall, it can be seen that the MSE training loss for the MCDO model is lower than for the MCVI model, and that both the prediction plot and the MSE-loss plot showed a more expected and better result.

4.3 Case 3: Auto MPG dataset

This case-study will examine how the two approximative models handles larger datasets with several features, and compare the results as done in the previous cases. The models will not try to predict out of range data in this case-study.

This dataset was collected from the UCI Machine learning repository [6], and consists of data that describes the city-cycle fuel consumption of a number of old cars, in miles per gallon (MPG). The original dataset includes 5 continuous attributes and 3 discrete. The discrete attributes were removed from the dataset, to avoid any difficulties regarding mixing categorical and continuous features. The dataset consists of 398 instances, which gives an **x**-matrix of dimensions 398x4 and **y**-vector of dimensions 398x1, and the data can be presented as,

$$\mathbf{x} = \begin{bmatrix} x_1^1 x_2^1 x_3^1 ... x_n^1 \\ x_1^2 x_2^2 x_3^2 ... x_n^2 \\ x_1^3 x_2^3 x_3^3 ... x_n^3 \\ ... \\ x_1^m x_2^m x_3^m ... x_n^m \end{bmatrix} ,\mathbf{y} = \begin{bmatrix} y^1 \\ y^2 \\ y^3 \\ ... \\ y^m \end{bmatrix}$$

where m is the number of instances and n is the number of features in each training example, which is here 398 and 4 respectively. After pre-processing the data with scaling and removing outliers and missing values, the number of training examples was reduced from 398 to 314.

The data was scaled with Equation 3.1 in Section 3.2, and the missing values were removed using the *dropna()* function in *Pandas*. The outliers were removed with the *z-score* function in *stats* from the *Scipy*-library. The training data was split into a training and test set by extracting 20% of the data for testing. During training, 20% of the training data was extracted for validation.

The correlation between the different features can be shown in Figure E.5, and based on which features showed the most visible trend with the MPG data, the features weight and horsepower (HP) were used to show whether the prediction of MPG followed the expected trend. Further feature analysis have not been performed in this project, and the results should therefore only be reviewed in light of the visual trends in the plots. The model mean and standard deviation shown in the results are based on 500 predictions on the test data, and all data was scaled back to original value before plotting.

4.3.1 MCVI model

The Monte Carlo variational inference model was built with the *DenseVariational* layer in TensorFlow-Keras, with mixed non-trainable priors. A Gaussian distribution was chosen as the posterior. The model had 1 hidden layer with 10 units, and the number of epochs, trainable parameters and training data points can be found in table 4.13. The hyperparameters were tuned manually due to non-optimal solutions with Bayesian optimization. Sigmoid activation was used in the hidden layer of this model due to exploding values during loss-calculation when using the Relu activation function.

	Layers	**Units**	**Activation**	**LR**	σ_1	σ_2	π
MCVI	1	10	Sigmoid	1e-3	1	1e-7	0.5

Table 4.12: Hyperparameters of MCVI model for case 3.

Epochs	**Trainable parameters**	**Training data points**
5000	144	314

Table 4.13: The number of training epochs, trainable parameters in the network and number of training data points used to train MCVI model for case 3.

4.3.2 MCDO model

The MC dropout model was built with the TensorFlow-Keras implemented *Dense* and *Dropout* layers. Bayesian optimization was used to find the optimal hyperparameter tuning, which resulted in a model with 2 hidden layers, the first with 100 units and the second with 20 units. Four of the hyperparameters were included in the search space of the Bayesian optimization, namely dropout-rate, number of hidden layers, number of units in each hidden layer and the weight decay for the $l2$-regularizer. The remaining parameters were manually tuned, and Relu activation was used in both layers. The hyperparameters are given in table 4.14, and the number of epochs, trainable parameters and training data points can be found in table 4.15.

	Layers	Units	Activation	LR	Dropout	λ
MC	2	100	Relu	1e-3	0.1	1e-4
		20	Relu			

Table 4.14: Hyperparameters for MCDO model. p and λ are specific for the MC dropout model.

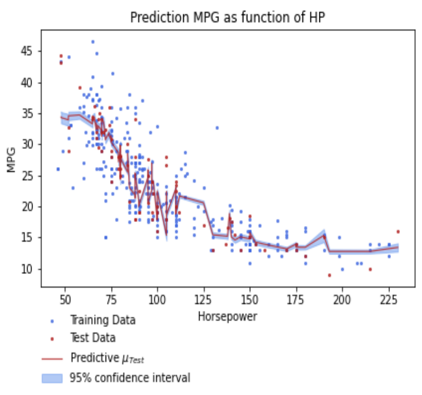
Epochs	Trainable parameters	Training data points
50	2541	314

Table 4.15: The number of training epochs, trainable parameters in the network and number of training data points used to train MCDO model for case 3.

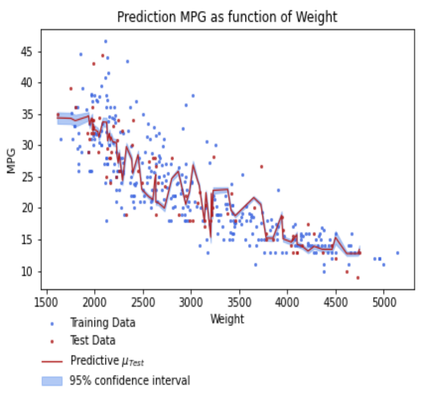
The Bayesian optimization algorithm was implemented with the *Keras tuner* application in the *Keras*-library, and the implementation can be found in Appendix F.

4.3.3 Results

Figures 4.9a and 4.9b show the predictions of MPG as a function of horsepower and weight respectively, and Figure 4.10 show a 3D-plot of the prediction of how MPG vary with both features.



(a) MPG as a function of horsepower.



(b) MPG as a function of weight.

Figure 4.9: Predictions based on test data in case 3 for the Monte Carlo variational inference model. MPG in 2D-plot, as function of (a) horsepower and (b) weight.

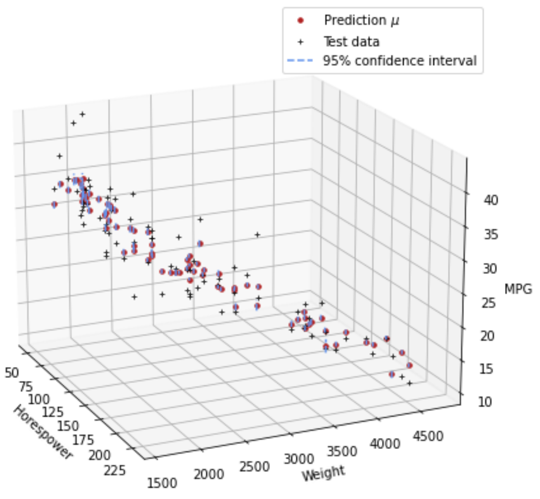


Figure 4.10: Prediction based on test data in case 3 for the Monte Carlo variational inference model. MPG in 3D-plot as function of horsepower and weight.

Figure 4.11 show the correlation between the predicted values of miles per gallon and the true value.

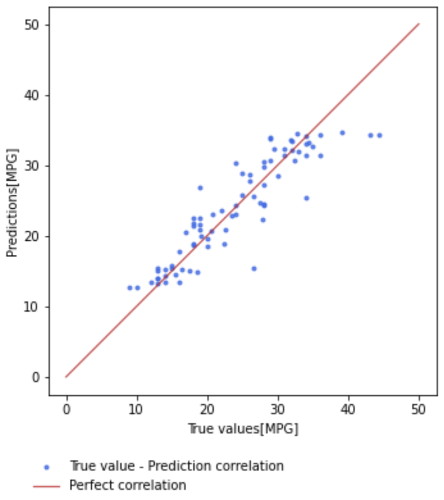


Figure 4.11: Correlation between model output and true y-label values for Monte Carlo variational inference model.

Figure 4.12 show the MSE-loss computed during training of the model.

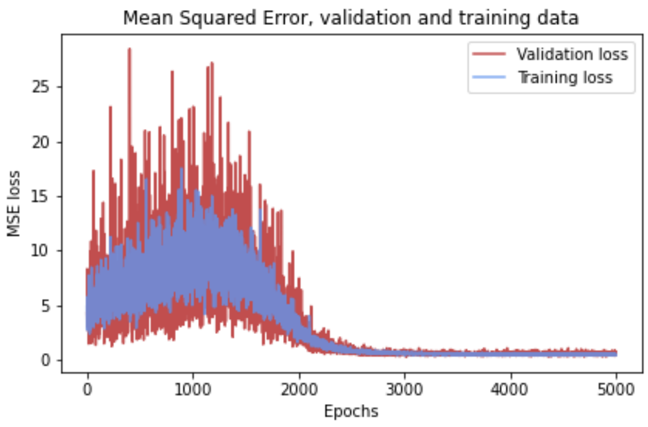
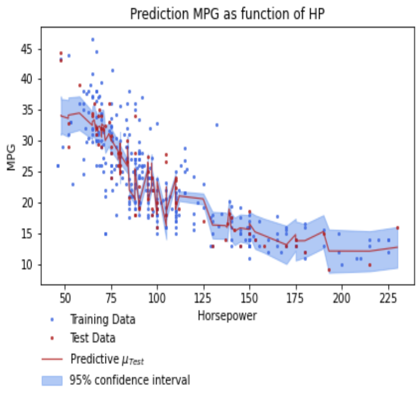
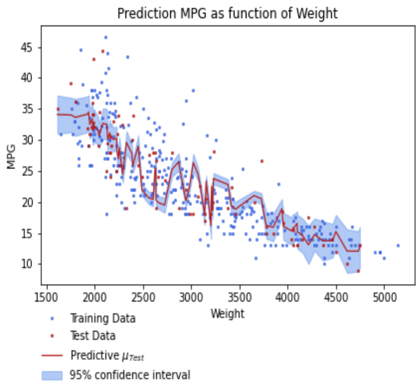


Figure 4.12: Mean squared error between model output and y-label, for the Monte Carlo variational inference model, computed for validation and training data at each training epoch.

Figures 4.13a and 4.13b show the predictions of miles per gallon as a function of horsepower and weight respectively, and Figure 4.14 show a 3D-plot of the prediction of how MPG vary with both features.



(a) MPG as a function of horsepower.



(b) MPG as a function of weight.

Figure 4.13: Predictions base on test data in case 3 for the Monte Carlo dropout model. MPG in 2D-plot, as function of (a) horsepower and (b) weight.

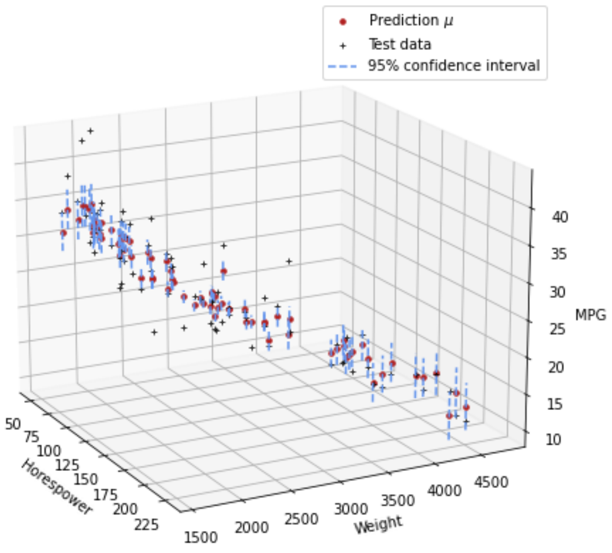


Figure 4.14: Prediction based on test data in case 3 for the Monte Carlo dropout model. MPG in 3D-plot as function of horsepower and weight.

Figure 4.15 show the correlation between the predicted values of miles per gallon and the true value.

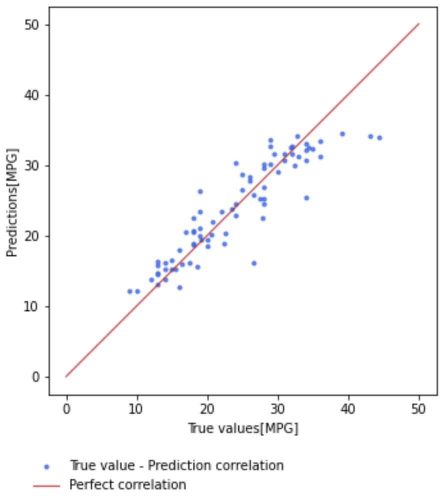


Figure 4.15: Correlation between model output and true y-label values for Monte Carlo dropout model.

Figure 4.16 show the MSE-loss computed during training of the model.

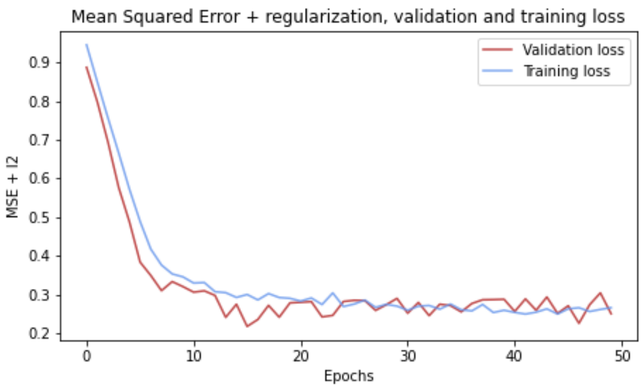


Figure 4.16: Mean squared error between model output and y-label with additional $l2$-regularization, for the Monte Carlo dropout model, computed for validation and training data at each training epoch.

Table 4.16 show the MSE at the end of the training epochs, the MSE between the true test labels and the prediction output, and the average training time. The MSE test values are computed with non-scaled values after prediction is completed.

Model	MSE training	MSE validation	MSE test [mpg^2]	Avg. training time [s]
MCVI	0.4874	0.5858	11.243	114.01
MCDO	0.2669	0.2514	10.939	1.66

Table 4.16: Mean squared error for training and validation set and average training time for MCDO model in case 3.

4.3.4 Discussion

Figures 4.9a and 4.9b, showing the 2D-plot of the prediction of MPG as function of horsepower and weight, respectively, show an indication that the prediction is following the expected trend of the correlation between the output and input feature. The Sigmoid activation function was used in this model, which is probably the reason for the very low uncertainty in the model. The Relu function was not suitable for this model, due to exploding loss values during the first epochs. As this model has more features than the datasets in case 1 and 2, the added loss for each parameter during training got significantly higher. With the relu function being a linearly increasing function, one can assume that this resulted in high loss values that could not be handled by the algorithm, and thus giving out *Not a Number (NaN)*-values for the rest of the iterations. In both of of the 2D-plots, it might seem like the lack of smoothness in the curves could be a sign of overfitting, however, both HP and weight are just one out of four features affecting the prediction, and the plots does therefore not give an indicateion of whether or not the model is overfitting. Figure 4.10 show the 3D-plot of the correlation between the MPG-values and both the horsepower and weight features. This figure gives a better visual of the models prediction of the test data, and the prediction seems to be very similar to the actual values of the test data. In table 4.16, the MSE between the predicted output and the test labels was 11.243 mpg^2, which seems to be a reasonably good result.

Figure 4.11 show the correlation between the true values of the test data and the predicted output as the blue dots. They seem to be gathered around the ”perfect correlation” line, which is a good indication that the model was able to predict the test data quite accurately.

The MSE-loss is shown in Figure 4.12, and show a clear improvement with the number of epochs. It could look like the loss is at a minimum at around 3000 epochs, and that there was no need for running for another 2000. However, Figures E.6a and E.6b in Appendix E.3 show the computed ELBO-loss, in which it is more visible that the loss is in fact decreasing slowly until around 5000 epochs. The curves showing both the computed MSE and the ELBO are quite noisy, as was expected. This could be seen in both previous case-studies for the variational inference models as well, and is due to the stochasticity of the model itself.

Both Figure 4.13a and 4.13b, showing the prediction for the MCDO model, are quite similar to those of the MCVI model. However, the uncertainty is larger overall, and especially for the outer points of both prediction curves, where there is less training data. This could both be due to the choice of activation function and the increased complexity. This model gives a more reasonable result as the 95% confidence interval is able to include 48.7% of the true test data values, while the MCVI models uncertainty only covers 12.82% of the true values and thus being overconfident about its prediction. Figure 4.14 show the 3D-plot of the correlation between horsepower, weight and MPG, in which one can see that the prediction follows the test data in a good way. This can also be seen in figure 4.15, where the correlation between the predicted outcome and the true test labels lie along the ”perfect correlation” line. The MSE between the true test value and the predicted outcome was 10.939 mpg^2, which was slightly lower than for the MCVI model.

The MSE loss curves for the MCDO model, shown in figure 4.16 show a rather steep descend during the first 5-7 epochs, followed by a more gradual decrease in both validation and training loss for the rest of the training period. The validation loss is lower in the fist 30 epochs, which is an undesired result, but seems to keep a stable level as the training loss decreases to a generally lower value at the end. Due to some noise in the validation loss curve, the final MSE-value is actually lower for the validation loss, as shown in table 4.16.

In this case-study, the Bayesian optimization algorithm was able to give slightly better hyperparameter tuning than the manually tuned Monte Carlo dropout model. This proved that there is some potential in implementing the optimization method, however, it needs to be improved for generalized use on different models and datasets. The average training time for MCDO, shown in table 4.16, was very low, which was expected due to the few number of epochs that the model was trained for. Overall, it seems that the MCDO model gave the best results in terms of lower mean squared error, lower number of necessary epochs for training, and a relatively fast algorithm.

CHAPTER 5

Final evaluation and future work

The results in Section 4.1 and 4.2 showed overall expected results according to theory of Bayesian neural networks, and the models were able to predict well inside the training data range with a reasonable increase in uncertainty on the outside.

The manual tuning of the hyperparameters regarding the priors of the MSVI-model proved to be very difficult, and tuning based on intuitive knowledge was practically impossible. Bayesian model performance is in general sensitive to the choice of prior and it is difficult to choose suitable and meaningful priors in MCVI, even with prior knowledge about the data [27]. The hyperparameters of the MCDO-model was in general easier to tune, as each parameter had an intuitive effect on the model performance. For example, when the model was overfitting is was a natural choice to try increasing the weight decay, λ, or the dropout probability, p, as explained in Appendix B. This model also showed to be more robust, and not as sensitive to small changes in the model structure or hyperparameters as the MCVI model.

The average training time did not seem to be affected by the number of parameters in the three cases, but a rather significant increase in runtime could be seen with increasing epochs. Due to the larger number of epochs needed for the MCVI models, those models were naturally a lot slower than the MCDO-models. This is an important factor to consider in further work, as average runtime becomes more important with large and complex networks, and one may want to choose the faster model.

The Bayesian optimization algorithm only provided good results for the MCDO-model in case 3, and there are several possible reasons for the poor performance. The search space may have been too small, or in the wrong range. It could also be that the search algorithm did not get enough trials to reach the optimal solution, or that it simply needed more features to make the right decisions.

Based on the evaluation of the three case-studies conducted in this project, using Monte Carlo dropout as a method for approximating a Bayesian neural networks seems to be the better choice. This based on the results from each case presented in tables 4.6, 4.11 and 4.16, where MCDO showed both lower MSE-values and overall faster network training (mainly due to lower necessary number of epochs). Therefore, in further studies, this method will be used as a basis for new models.

Further work should focus on using this model as a starting point, and explore possibilities for improving it for specific types of data. Improvements may include the implementation of Long short-term memory recurrent neural networks (LSTM-RNN) for working with time-dependent datasets. This for making the model able to predict on realistic data, and be used as a tool in a research context. An improvement of the Bayesian optimization algorithm would also be necessary, to avoid time consuming hyperparameter tuning. When working with high dimensional datasets, feature analysis, e.g. Principal Component Analysis (PCA), should be performed for better understanding of the data and sorting out important features to simplify the data structure.

Bibliography

[1] A.B. Kotsiantis, D. K. and Pintelas, P. [2006]. Data preprocessing for supervised learning, *International journal of computer science* **1**(1).
URL: *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.8413rep=rep1type=pdf*

[2] Anthony T.C. Goh, Fred H. Kulhawy, C. C. [2005]. Bayesian neural network analysis of undrained side resistance of drilled shafts, *Journal of geotechnical and geoenvironmental engineering* **131**.

[3] Bland, G. [n.d.]. Train/test split and cross validation – a python tutorial.
URL: *https://algotrading101.com/learn/train-test-split/*

[4] Blundell, C., Cornebise, J., Kavukcuoglu, K. and Wierstra, D. [2015]. Weight uncertainty in neural networks.

[5] Brownlee, J. [2017]. Difference between classification and regression.
URL: *https://machinelearningmastery.com/classification-versus-regression-in-machine-learning/*

[6] Dua, D. and Graff, C. [2017]. UCI machine learning repository.
URL: *http://archive.ics.uci.edu/ml*

[7] Feindt, M. and Kerzel, U. [2006]. The neurobayes neural network package, *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **559**(1): 190 – 194. Proceedings of the X International Workshop on Advanced Computing and Analysis Techniques in Physics Research.
URL: *http://www.sciencedirect.com/science/article/pii/S0168900205022679*

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

[9] Gal, Y. and Ghahramani, Z. [2015]. On modern deep learning and variational inference.

[10] Gal, Y. and Ghahramani, Z. [2016]. Dropout as a bayesian approximation: Representing model uncertainty in deep learning.

[11] Ghahramani, Z. [2015]. Probabilistic machine learning and artificial intelligence, *Nature* **521**: 452–459.

[12] Graves, A. [2011]. Practical variational inference for neural networks, *in* J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira and K. Q. Weinberger (eds), *Advances in Neural Information Processing Systems*, Vol. 24, Curran Associates, Inc., pp. 2348–2356.
URL: *https://proceedings.neurips.cc/paper/2011/file/7eb3c8be3d411e8ebfab08eba5f49632-Paper.pdf*

[13] Hao, K. [2018]. What is machine learning, *MIT Technology Review* .

[14] Huang, J., Li, Y.-F. and Xie, M. [2015]. An empirical analysis of data preprocessing for machine learning-based software cost estimation, *Information and Software Technology* **67**: 108 – 127.
URL: *http://www.sciencedirect.com/science/article/pii/S0950584915001275*

[15] Kenji Doya, Shin Ishii, A. P. and Rao, R. P. N. [2007]. *Bayesian Brain - Probabilistic approaches to neural coding*, The MIT press.

[16] Krasser, M. [2019]. Variational inference in bayesian neural networks.
URL: *http://krasserm.github.io/2019/03/14/bayesian-neural-networks/*

[17] Laurent Valentin Jospin, Wray Buntine, F. B. H. L. and Bennamoun, M. [2020]. Hands-on bayesian neural networks - a tutorial for deep learning users, *ACM Comput. Surv.* **1**(1).

[18] Michelucci, U. [2018]. *Applied Deep Learning - A case-based approach to understanding deep neural networks.*, Apress Media LLC.

[19] Miller, A. C. [2018]. *Advances in Monte Carlo variational inference and applied probabilistic modelling*, PhD thesis, Harvard University.

[20] Murphy, K. P. [2012]. *Machine Learning - A Probabilistic Perspective*, The MIT Press.

[21] Ng, A. [2020]. Machine learning, https://www.coursera.org/learn/machine-learning/home/welcome.

[22] Nielsen, M. A. [2015]. *Neural networks and deep learning*, Determination press.

[23] Reitermanová, Z. [2010]. Data splitting.
URL: *https://www.mff.cuni.cz/veda/konference/wds/proc/pdf10/WDS10*$_105_i1_Reitermanova.pdf$

[24] Shahriari, B., Swersky, K., Wang, Z., Adams, R. P. and de Freitas, N. [2016]. Taking the human out of the loop: A review of bayesian optimization, *Proceedings of the IEEE* **104**(1): 148–175.

[25] Shanqing Cal, S. B. and Nielsen, E. [2019]. *Deep learning with JavaScript*, Manning publications.

[26] Snoek, J., Larochelle, H. and Adams, R. P. [2012]. Practical bayesian optimization of machine learning algorithms.

[27] Wu, A., Nowozin, S., Meeds, E., Turner, R. E., Hernández-Lobato, J. M. and Gaunt, A. L. [2018]. Fixing variational bayes: Deterministic variational inference for bayesian neural networks, *CoRR* **abs/1810.03958**.
URL: *http://arxiv.org/abs/1810.03958*

APPENDIX A

Visual example of uncertainty

As this is meant to be an example to visualize the uncertainties of an MCVI model, the dataset and tuning of the model will not be elaborated. As explained in section 2.3 of this report, there are two main types of uncertainties to consider Bayesian modelling, aleatoric and epistemic uncertainties. The output layer of a Bayesian network gives a distribution, from which a mean and variance can be retrieved. Inference is done with this model 10 times resulting in 10 mean predictions with corresponding standard deviation. The result is shown in Figure A.1 below.

The figure show the training data from which the model was trained on as the blue dots, and the red lines represents the prediction mean, μ, from each prediction. The blue lines represents the standard deviation, σ, for each μ. From this illustration one can clearly see the uncertainty in the model as the variations of the different prediction results, and this is the epistemic uncertainty. By adding more data, the variation of these red lines could be reduced. The aleatoric uncertainty, which respresents the noise in the data, is shown by the different blue lines. This is inherent noise, which can not be reduced by adding more data.

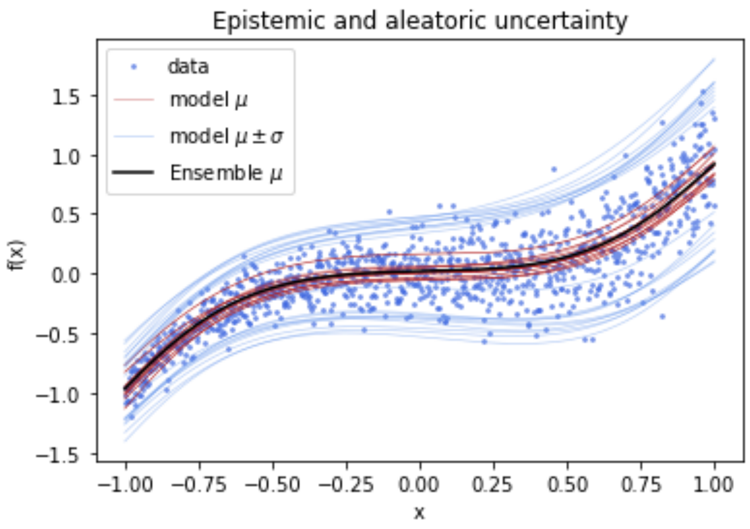


Figure A.1: The figure show the model mean and standard deviation of 10 predictions made on the training data. The red lines are the model mean, the blue lines are the standard deviations and the black line is the ensemble mean of the 10 prediction means.

APPENDIX B

Case 2: Example of overfitting

When manually trying to fit the MC dropout model to the data, several combinations of hyperparameters were tried out. One of the many achieved results of this case is shown in Figures B.1 and B.2, and is presented to give an example of an overfitted model. The different causes that can lead to overfitting and how it can be spotted was introduced in chapter 3, and the plots below show clearly some of the typical signs of overfitting. This model had 2 hidden layers with 100 hidden units in each layer. The dropout rate was set to 0.05, and the model was trained for 30000 epochs.

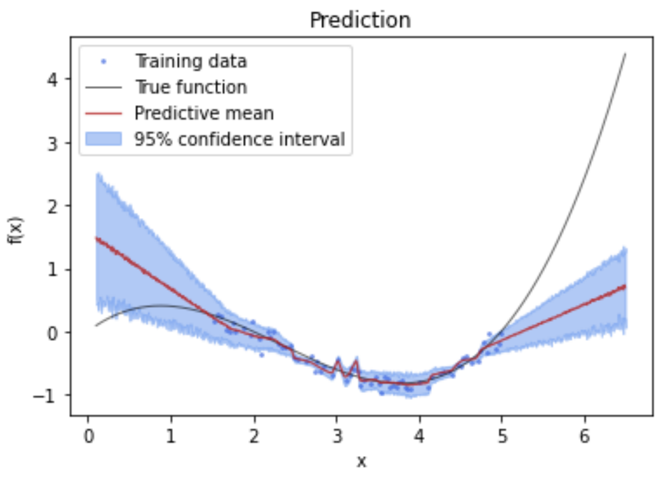


Figure B.1: The figure shows a prediction made with the MCDO model, with the epistemic uncertainty. The red line is showing the predictive mean and can be seen tracing specific data points instead of the general trend, which is an indication of overfitting.

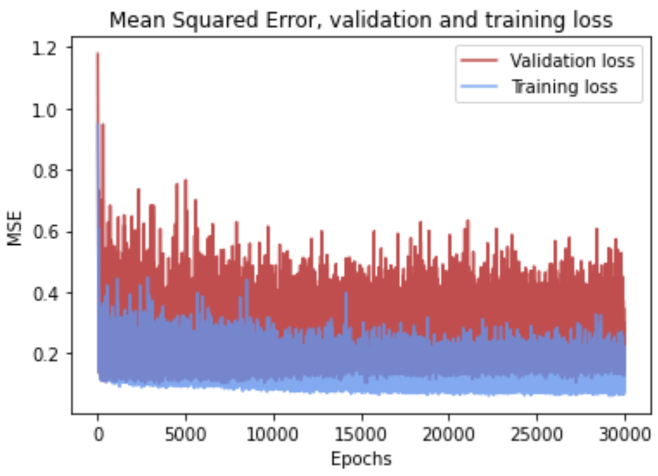


Figure B.2: The plot shows the training and validation loss computed during training of the model. After around 5000 epochs, the validation loss is starting to increase while the training loss is continously decreasing. This is a sign that the model is overfitting to the training data.

One can see from Figure B.2 that after around 5000 epochs, the validation loss is slightly starting to increase while the training loss is decreasing further. This indicates that the model is fitting better to the training data, and getting worse at generalizing to other data. When looking at the prediction plot, Figure B.1, one can in fact see that the predictive mean is trying to follow the different training data points instead of following the overall trend of the data. As mentioned in chapter 3, there are several possible factors that can lead to this result, but a somewhat obvious way of improving the performance is to reduce the number of hidden units. The number of trainable parameters is very large (10401) with 100 units in two sequential layers. This increases the complexity of the model, and with a simple dataset like this, it over-complicates the fitting making the model very specific to the training data. Other ways of improving the generalizability of the model is to increase the regularization factor lambda, or to increase the dropout factor. This will induce a larger regularizing effect on the network weights, making it less prone to overfitting. Also, this is an indication that far less epochs is necessary for training, and one should stop training when validation loss starts to increase, as described in section 3.3.3.

In the result presented from this case in section 4.2.3, in Figure 4.7 and 4.8, the model was improved by reducing the number of hidden units to only 20 hidden units per layer, and additionally increasing lambda from 1e-5 to 5e-5, needing only 500 epochs for training.

APPENDIX C

Bayesian optimization for hyperparameter tuning

Bayesian optimization has in cases of tuning hyperparameters of a deep neural networks been proven as the superior choice of approach due to the way the next combination of hyperparameters is chosen during optimization [26]. The method is designed to choose the next location to evaluate a cost function based on the results of previously tested hyperparameter combinations.

One is interested in finding the minimum of the objective function J(θ), which is constructed as a probabilistic model and represents the performance of the network. A prior distribution of functions is selected, usually a Gaussian process over the hyperparameter values, from which a prior belief about the objective function can be drawn. This Gaussian process is known as the surrogate model for the optimization, and is used to approximate the objective function. An aquisition function is used to select the optimal location to sample the objective function in the next step, and is based on the posterior of the previously sampled objective functions. There are two ways of choosing the next optimal point, namely by exploitation or exploration, and the aquisition function bases its choice on a trade off between these two. Exploitation means to evaluate the cost function where the surrogate model predicts high objective values, and where selecting this location is likely to increase performance. Exploration means to evaluate where the uncertainty of the surrogate model prediction is high and information is lacking. Both high objective values and high prediction uncertainty corresponds to high aquisition values, and thus the objective is to maximize the aquisition function. A graphical example of such a process is shown in Figure C.1, where it is clearly shown how the decision on where to evaluate the objective function next is determined by the maximization of the aquisition function, shown in the bottom of each graph [24].

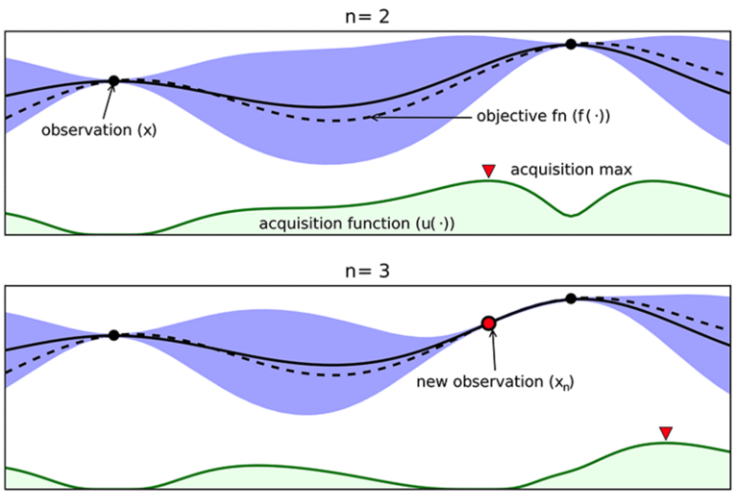


Figure C.1: Illustration of Bayesian optimization. The objective function is shown as the black stippled line, while the true objective function is shown as the black whole line. The green shaded function in the bottom of each graph represents the aquisition function, in which the objective function will be evaluated where this function has high values.

The next location x$_n$ to sample the objective function $f(x_n)$, is defined as

$$x_n = argmax_x a(x|x_{n-1}, y_{n-1}) \tag{C.1}$$

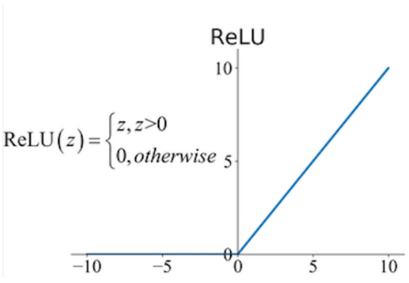
where x is the set of hyperparameters, a is the aquisition function, and (x$_{n-1}$,y$_{n-1}$) denotes the $n-1$ samples drawn from the objective function so far, where y$_{n-1}$ = f(x$_{n-1}$) + ϵ_{n-1} is the noisy sample of the objective function. The the new sample is then used to update the Gaussian process prior. A popular choice for the aquisition function is the expected improvement(EI), which can be defined as

$$EI(x) = E[max(f(x) - f(x^+), 0)] \tag{C.2}$$

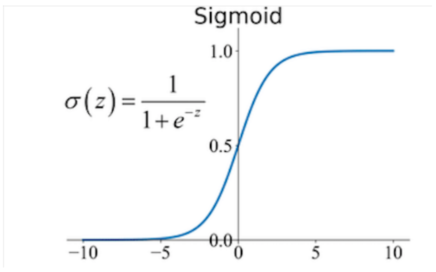
Where f(x$^+$) denoted as the value of the best sample so far, and x$^+$ is the location of that sample. This represents the potential improvement over the space of hyperparameters. Even though this procedure requires more computation than for example random search method, the implementation of previous information at each step results in fewer steps towards the optimum [26].

APPENDIX D

Activation functions



(a) Relu activation function



(b) Sigmoid activation function

Figure D.1: Two of the most common activation functions applied to the units in the network hidden layers.

APPENDIX E

Additional results

E.1 Case 1

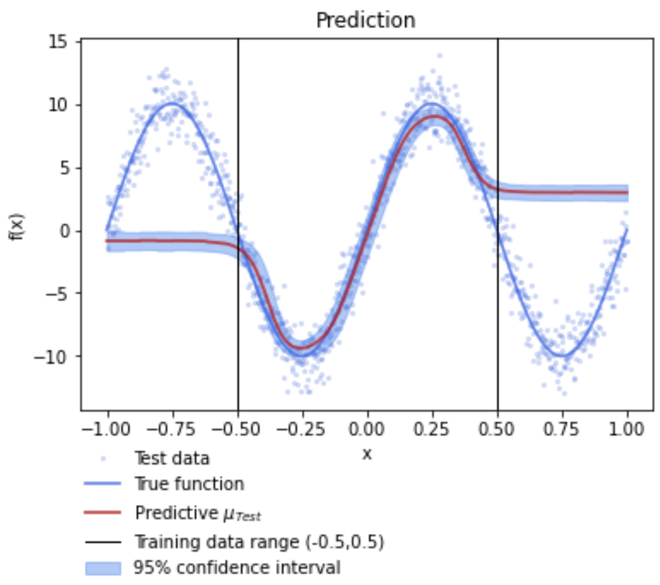
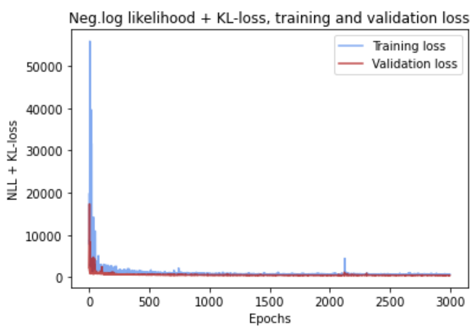
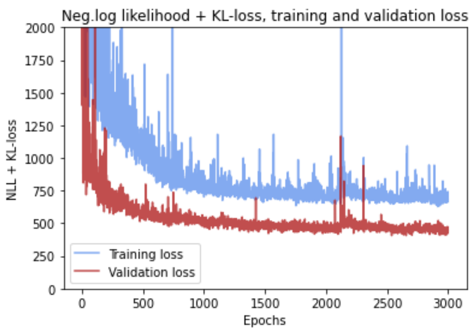


Figure E.1: Prediction on test data for the MCVI model in case 1: Sinusoidal function. Sigmoid activation function is here used in both hidden layers, resulting in a smaller and nearly constant confidence interval over the whole range.



(a) Training and validation ELBO-loss.



(b) Training and validation ELBO-loss, enlarged plot.

Figure E.2: Training and validation loss curves for the MCVI model in case 1: Sinusoidal function. The curves show the computed loss from the maximization of the ELBO; Neg.log likelihood + $D_{KL}[q_\theta|p(w)]$, used for optimizing the parameters, as shown in section 2.3.2.

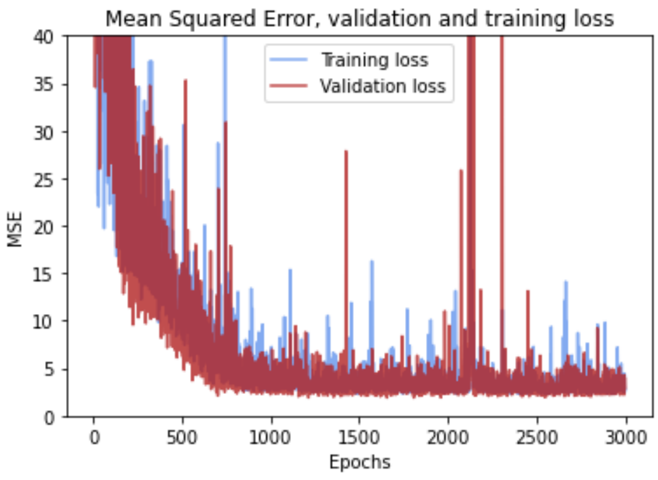
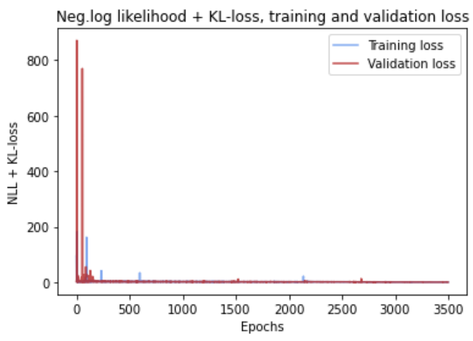
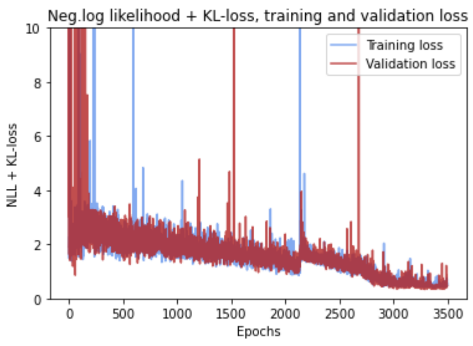


Figure E.3: Training and validation loss curves for the MCVI model in case 1: Sinusoidal function. The curves show the MSE-loss between the output y and the true label value for the training and validation data. Enlarged plot, corresponding to Figure 4.2.

E.2 Case 2



(a) Training and validation ELBO-loss.



(b) Training and validation ELBO-loss, enlarged plot.

Figure E.4: Training and validation loss curves for the MCVI model in case 2: Polynomial function. The curves show the computed loss from the maximization of the ELBO; Neg.log likelihood + $D_{KL}[q_\theta|p(w)]$, used for optimizing the parameters, as shown in section 2.3.2.

E.3 Case 3

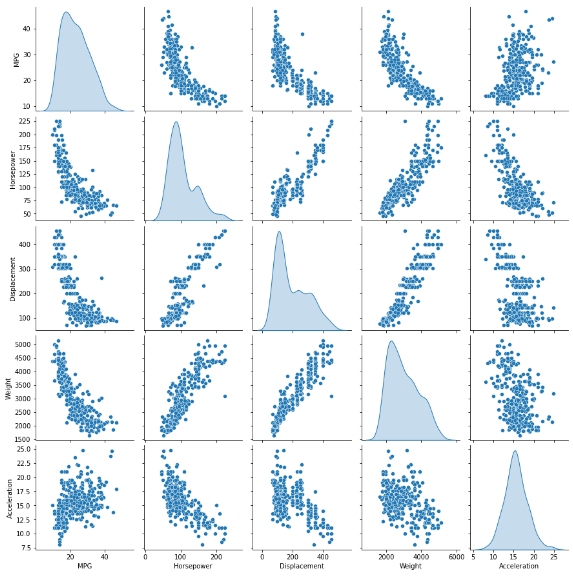
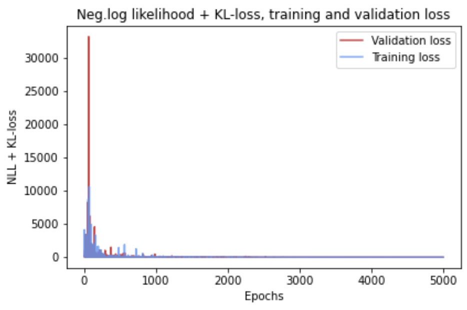
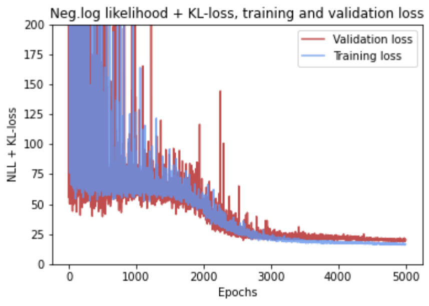


Figure E.5: Correlation between all features in the dataset in case 3: Auto MPG. The plot was constructed with the *pairplot* function from the *Pandas*-library.



(a) Training and validation ELBO-loss.



(b) Training and validation ELBO-loss, enlarged plot.

Figure E.6: Training and validation loss curves for the MCVI model in case 3: Auto MPG. The curves show the computed loss from the maximization of the ELBO; Neg.log likelihood + $D_{KL}[q_\theta|p(w)]$, used for optimizing the parameters, as shown in section 2.3.2.

APPENDIX F

Implementation of models in Python

```
"""
______________________________________________________________
IMPORTED LIBRARIES:
______________________________________________________________
"""

import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions
tfpl = tfp.layers
tfkl = tf.keras.layers

from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense , Input
from tensorflow.keras.losses import MeanSquaredError
from tensorflow.keras import  regularizers

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

import numpy as np
import matplotlib.pyplot as plt

import tqdm
```

```
"""
______________________________________________________________________
SINUS-FUNCTION:

    f(x) = 10sin(2pix) + epsilon

The training data was scaled with StandardScaler()-function,
(x-mu)/sigma, before being split into a training and validation
set. Validation set was 20% of original training set.

epsilon ~ N(0,sigma^2)
______________________________________________________________________
"""
def f(x, sigma):
    epsilon = np.random.randn(*x.shape) * sigma
    return 10 * np.sin(2 * np.pi * (x)) + epsilon

train_size = 200
noise = 1.5

x = np.linspace(-0.5, 0.5, train_size).reshape(-1, 1)
y = f(x, sigma=noise)


x_train_scaler = StandardScaler() #scales (x-mu)/sigma
x_train_scaler.fit(x)
x_train_scaled = x_train_scaler.transform(x)
y_train_scaler = StandardScaler()
y_train_scaler.fit(y)
y_train_scaled = y_train_scaler.transform(y)

x_train, x_val, y_train, y_val = train_test_split(x_train_scaled,
                                                  y_train_scaled,
                                                  test_size = 0.2,
                                                  random_state = 42)
x_test = np.linspace(-1.,1.,1000)[:,np.newaxis]
y_true = f(x_test,0)
x_test = x_train_scaler.transform(x_test)
```

```
"""
________________________________________________________________
POLYNOMIAL FUNCTION:

    f(x) = 0.1x^3 - 0.7x^2 + x + epsilon

The training data was scaled with StandardScaler()-function,
(x-mu)/sigma, before being split into a training and validation
set. Validation set was 20% of original training set.

epsilon ~ N(0,sigma^2)
________________________________________________________________
"""
def f(x, sigma):
    epsilon = np.random.randn(*x.shape) * sigma
    return 0.1*x**3 - 0.7*x**2 + 1*x + epsilon

train_size = 700
noise = 0.1

x = np.linspace(1.5, 5, train_size).reshape(-1, 1)
y = f(x, sigma=noise)

x_train_scaler = StandardScaler()
x_train_scaler.fit(x)
x_train_scaled = x_train_scaler.transform(x)
y_train_scaler = StandardScaler()
y_train_scaler.fit(y)
y_train_scaled = y_train_scaler.transform(y)

x_train, x_val, y_train, y_val = train_test_split(x_train_scaled,
                                                  y_train_scaled,
                                                  test_size = 0.2,
                                                  random_state = 42)

x_test = np.linspace(0.1,6.5,1000)[:,np.newaxis]
y_true = f(x_test,0)
x_test = x_train_scaler.transform(x_test)
y_test = f(x_test,noise)
```

```
"""
MONTE CARLO VARIATIONAL INFERENCE MODEL:

Hyperparameters for mixed, non-trainable priors with two Gaussian
distributions:
    scale = sigma1,
    scale = sigma2,
    mix = Pi
Other hyperparameters:
    batch_size
    activation
    units
    learning_rate
    epochs
"""

def prior_mix(kernel_size, bias_size, dtype = None):
    """
    The belief of which model parameters are likely, before any
    data is seen. Mixed, non-trainable prior of two Gaussian
    distributions <==> independent normal distr. placed upon
    each weight and bias, all with equal variance.

    The function defines the prior distribution for a given
    Dense Variational layer.

    Parameters
    ----------
    kernel_size :
        Number of parameters in the dense layer weight matrix
    bias_size :
        Number of parameters in the dense layer bias vector
    dtype : TYPE, optional
        Default = None.

    Returns
    -------
    Callable lambda function, which takes input tensor T and
    returns independent normal distr with mean = 0 and variance = 1.
    This is the prior over the dense layer parameters.

    """
    n = kernel_size + bias_size
    mix = Pi
    p = np.array([[mix, 1-mix]]*n)

    bimix_gauss = tfd.Mixture(cat=tfd.Categorical(probs=p),
                              components=[
                                  tfd.Normal(loc=tf.zeros(
                                                  n,
                                                  dtype = dtype),
                                             scale=sigma1),
                                  tfd.Normal(loc=tf.zeros(
                                                  n,
                                                  dtype = dtype),
                                             scale=sigma2])
    prior_model = lambda t: tfd.Independent(
                                bimix_gauss,
                                reinterpreted_batch_ndims = 1
                                )

    return prior_model



def posterior_distLam(kernel_size, bias_size, dtype = None):
    """
    Variational distribution of a known functional form, which
    parameters will be optimized to get as close to the true
    posterior as possible. Placed upon the model parameters.
    Gives the most likely parameters, given the data.

    Parameters
    ----------
    kernel_size :
        Number of parameters in the dense layer weight matrix
    bias_size :
        Number of parameters in the dense layer bias vector
    dtype : TYPE, optional
        Default = None.

    Returns
    -------
    Callable which takes an input and produces a tfd.Distribution
    instance.

    """
    n = kernel_size + bias_size
    return Sequential([
        tfpl.VariableLayer(2*n, dtype = dtype),
        tfpl.IndependentNormal(n)
        ])


batch_size = x_train.shape[0]
num_batches = x_train.shape[0]/ batch_size
activation = activation

"""
Build model:
------------
    Input layer
    1 hidden layer, 10 hidden units
    1 layer providing mu and sigma for the output layer distribution
    Output layer (distribution)
"""
model = Sequential([
    tfpl.DenseVariational(input_shape = (x_train.shape[1],),
        units = units,
        make_prior_fn=prior_mix,
        make_posterior_fn = posterior_distLam,
        kl_weight = 1/x_train.shape[0],
        activation = activation),
    tfpl.DenseVariational(
        units = tfpl.IndependentNormal.params_size(1),
        make_prior_fn=prior_mix,
        make_posterior_fn = posterior_distLam,
        kl_weight = 1/x_train.shape[0]),
    tfpl.IndependentNormal(1)
    ])

"""
Training the model:
-------------------
"""
def nll(y_true, y_pred):
    """
    Negative log likelihood, for computing the ELBO loss function.

    Parameters
    ----------
    y_true :
        True values, y-labels.
    y_pred :
        Prediction output distribution.

    Returns
    -------
    Negative log likelihood of the y-labels given the predicted
    distribution.

    """
    return -y_pred.log_prob(y_true)

model.compile(loss = nll,
              optimizer = tf.optimizers.Adam(
                              learning_rate = learning_rate),
              metrics = "mse")
history = model.fit(x_train, y_train,
                    batch_size = batch_size,
                    validation_data = (x_val,y_val),
                    epochs = epochs, verbose = 0)
```

```
"""
______________________________________________________________
MONTE CARLO DROPOUT MODEL:

Hyperparameters for dropout model:
    dropout = dropout probability
    lam = lambda (weight decay)
Other hyperparameters:
    batch_size
    activation
    units
    learning_rate
    epochs
______________________________________________________________
"""
tau = 1
dropout = dropout
lam = lam
N = x_train.shape[0]
batch_size = N
num_batches = N/batch_size

"""
Build model:
------------
    Input layer
    L hidden layers, K hidden units  (L=2, K=20 in this case)
    Dropout layer after every hidden layer
    Output layer
"""
x_in = Input(shape=(1,))
L1 = Dense(units = units,
                activation = activation,
                kernel_regularizer = regularizers.l2(lam),
                kernel_initializer = "random_normal")(x_in)
K1 = tfkl.Dropout(dropout)(L1, training=True)
L2 = Dense(units = units,
                activation = activation,
                kernel_regularizer = regularizers.l2(lam),
                kernel_initializer = "random_normal")(K1)
K2 = tfkl.Dropout(dropout)(L2, training=True)
x_out = Dense(units = y_train.shape[1])(K2)

model = Model(x_in,x_out)
```

```
"""
Training the model:
-------------------
"""
model.compile(loss = MeanSquaredError(),
              optimizer = tf.optimizers.Adam(
                  learning_rate = learning_rate)
              )

history = model.fit(x_train, y_train,
                    batch_size = batch_size,
                    validation_data=(x_val,y_val),
                    epochs = epochs, verbose = 0)
```

```
"""
______________________________________________________________
BAYESIAN OPTIMIZATION FOR MONTE CARLO DROPOUT MODEL:
______________________________________________________________
"""

def BuildAndCompile(hp):
    """
    Callable that takes in hyperparameters and produces a
    model with those hyperparameter values.

    For Int and Float hyperparams, minimum and maximum values
    for search space must be determined, optional: step size.

    Parameters
    ----------
    hp :
        Hyperparameters

    Returns
    -------
    model :
        An instance of the Model class.

    """
    dropout = hp.Float("dropout",
                       min_value = minimum_dropout,
                       max_value = maximum_dropout)
    lam = hp.Float("lamda",
                   min_value = minimum_lam,
                   max_value = maximum_lam)


    x_in = Input(shape=(train_features.shape[1]))

    x = Dense(units = hp.Int("HidLay_0",
                             min_value = minimum_units,
                             max_value = maximum_units,
                             step = step_units),
              activation = "relu",
              kernel_regularizer = regularizers.l2(lam),
              kernel_initializer = "random_normal")(x_in)
    x = Dropout(dropout)(x, training=True)
    for i in range(hp.Int("num_layers",
                          minimum_layers,
                         maximum_layers)):
        x = Dense(units = hp.Int("HidLay_"+str(i+1),
                                min_value = minimum_units,
                                max_value = maximum_units,
                                step = step_units),
                  activation = "relu",
                  kernel_regularizer = regularizers.l2(lam),
                  kernel_initializer = "random_normal")(x)
        x = Dropout(dropout)(x, training=True)

    x_out = Dense(units =1)(x)

    model = Model(x_in,x_out)
    model.compile(loss = MeanSquaredError(),
              optimizer = tf.optimizers.Adam(
                  learning_rate = learning rate)
              )

    return model



"""
Search for best hyperparameters:
--------------------------------
    BayesianOptimization class: creates tuner objects
        Parameters:
        -----------
        hypermodel:
            Model instance, model
        objective:
            Model metric to minimize or maximize
        num_initial_points:
            The number of randomly generated samples as initial
            training data for Bayesian optimization
        max_trials:
            Total number of triels to test at most
        beta:
            Balancing factor of exploration and exploitation

    Search method: Performs a search for best hyperparameter
    configurations
        Parameters:
        -----------
            Takes in same parameters as the fit-method for
            training.
"""

tuner = BayesianOptimization(hypermodel = BuildAndCompile,
                              objective = Objective("model_metric",
                                                    direction = "min"),
                              num_initial_points = num_initial_points,
                              max_trials = max_trials,
                              beta = beta,
                              project_name="Project_name")

tuner.search(train_features, train_labels,
              epochs = epochs,
              validation_split = 0.2,
              shuffle = True,
              verbose = 0)

best_hps = tuner.get_best_hyperparameters()[0]

"""
___________________________________________________________
INFERENCE:
    500 predictions on test data.
    Resulting in mean and standard deviation vectors of all
    predictions.
___________________________________________________________
"""
import tqdm

y_hats = []
for i in tqdm.tqdm(range(500)):
    y_hat = model.predict(x_test)
    y_hats.append(y_hat)

y_pred = np.mean(y_hats, axis = 0)
y_std = np.std(y_hats, axis = 0)
```