Uncertainty Quantification with Bayesian Neural Networks

Master's thesis submitted to

Prof. Dr. Nadja Klein

Humboldt-Universität zu Berlin

School of Business and Economics

Chair of Applied Statistics

Prof. Dr. Stefan Lessmann

Humboldt-Universität zu Berlin

School of Business and Economics

Chair of Information Systems

by

Per Joachims

(598082)

in partial fulfilment of the requirements

for the degree of

Master of Science Statistics

Berlin, June 30 2021

Acknowledgement

"If I have seen further, it is by standing upon the shoulders of giants."

– Isaac Newton in a letter to Robert Hooke, 1675

It is truly a blessing that it is feasible for me to conduct research in such a fascinating area. I would like to thank all researchers that provide open access to their work and all developers of open-source software, as without them this project would not have been possible.

I would also like to thank my friends and family for their support. Last but not least, I would like to thank my supervisor Nadja Klein for vital advice and discussions, and for giving me the opportunity to dive deep into the Bayesian neural network rabbit hole.

Per Joachims, Berlin, June 2021

Abstract

Neural networks (NNs) are used to a great extent in various fields and are becoming increasingly relevant to decisions in our everyday life. However, NNs are typically not aware of the uncertainty that accompanies their predictions, inhibiting their use in safety-critical applications. Bayesian NNs (BNNs) allow quantifying uncertainty by modeling weights as distributions over possible values. It is usually infeasible to compute BNNs analytically. Various adaptations that lower the computational cost of BNNs and non-Bayesian approaches have been proposed to model uncertainty in neural networks. As of now, there is no clear winner among those methods.

In this thesis, we benchmark the popular models Monte Carlo dropout (MC dropout), deep ensemble (PNN-E), and probabilistic backpropagation (PBP) as well as the last-layer inference methods neural linear model (NLM) and marginally calibrated deep distributional regression (DNNC), an adaption to NLM that ensures marginal calibration. We assess their – marginal and probabilistic – calibration and obtain robust predictive performance results by evaluating the models' ability to predict uncertainty for in-between and out-of-range inputs. Further, we provide empirical evidence that DNNC may be used as a post-calibration method on a NN fit with standardized inputs and outputs. This enables a wider usability of the general approach.

Contents

Nomenclature

Acronyms / Abbreviations

BNN	Bayesian neural network
CDF	Cumulative distribution function
CRPS	Continuous ranked probability score
DGP	Data generating process
ELBO	Evidence lower bound
GP	Gaussian process
HMC	Hamiltonian Monte Carlo
KDE	Kernel density estimation
KL	Kullback-Leibler
LogS	Logarithmic score
MAE	Mean absolute error
MC	Monte Carlo
MCMC	Markov chain Monte Carlo
MDL	Minimum description length
MPIW	Mean prediction interval width
MSE	Mean square error
NLL	Negative log-likelihood
NN	Neural network
NUTS	No-U-Turn sampler
PBP	Probabilistic backpropagation
PDF	Probability density function
PICP	Prediction interval coverage probability
PNN	Probabilistic neural network
ReLU	Rectified linear unit
RMSE	Root mean square error
SGD	Stochastic gradient descent
VI	Variational inference
e.g.	For example
i.e.	That is
i.i.d.	Independent and identically distributed
w.r.t.	With respect to

Roman Symbols

$\boldsymbol{A}$	Matrix
$\boldsymbol{a}$	Vector
a	Scalar
$\mathcal{D}$	Dataset

n	Number of samples in $\mathcal{D}$	$\widehat{y_i}$	Model output for input $\mathbf{x}_i$, scalar
d	Number of input features in $\mathcal{D}$	L	Number of layers in neural network
$\mathbf{x}_i$	Input for model	V_l	Number of network units in layer l
y_i	Target for model, scalar	$\boldsymbol{z}^{(l)}$	Output of layer l

List of Figures

List of Tables

1 Introduction

"Data! Data! Data!", he [Sherlock Holmes] cried impatiently. "I cannot make bricks without clay."

– *The Adventure of the Copper Beeches* by Sir Arthur Conan Doyle

With the massive increase in computing power over the last decades, there has been a rise in the application of machine learning methods. Most notably, neural networks (NNs) – essentially the concatenation of simple tensor[1] operations – can run on specialized hardware (GPUs and TPUs) and show excellent predictive power in many applications. Thus, NNs are used in various fields such as computer vision (Krizhevsky et al., 2012) or natural language processing (Mikolov et al., 2013). However, a NN typically outputs a scalar value in a regression context or unreliable probabilities for the output classes in a classification setting (Gal and Ghahramani, 2016).

The Need for Uncertainty Quantification

Unawareness about the certainty of machine learning methods can lead to problems for their use in safety-critical applications. E.g., imagine we use a neural network to predict whether a patient has diabetic retinopathy (diabetes damaging the retina) based on pictures of the retina (Pratt et al., 2016). A prediction that fails to identify an existing illness (false negative) might seriously put the patient's eyesight at risk. As a second example, a wrong prediction of a steering angle in autonomous driving may result in casualties or fatalities. In both examples, awareness of the model about the certainty of its predictions might prevent disasters. If the model is *certain*, we may trust the model's prediction, and if it is not, we pass the problem on to the doctor or driver.

Sources of Uncertainty

Uncertainty can be categorized into *aleatoric* and *epistemic* uncertainty (Hora, 1996; Kiureghian and Ditlevsen, 2009). Aleatoric or statistical uncertainty arises from noise in the data. Consider a single coin flip with a probability of heads $p_H = 0.5$, where we cannot – under normal circumstances

[1]Loosely speaking, a tensor can be thought of as the generalization of a matrix.

– foresee the outcome of the experiment even if we know the true value p_H. On the other hand, epistemic or model uncertainty stems from a lack of knowledge of the world and reflects a model's uncertainty about its parameters. Epistemic uncertainty can be reduced with data and expresses our ignorance (Kendall and Gal, 2017).

Non-Bayesian NN Approaches

There exist a variety of approaches to model uncertainty in neural networks. A probabilistic neural network (PNN; Nix and Weigend, 1994) learns input-dependent noise in the data and thus captures aleatoric uncertainty. An ensemble of NNs relies on the random initialization of the weights to find various local minima in weight space, thus modeling epistemic uncertainty. We can combine both and form a deep ensemble (PNN-E; Lakshminarayanan et al., 2016) that captures both types of uncertainty.

Bayesian NN Approaches

With Bayesian approaches, we model the parameters as distributions over possible values to encode our beliefs about the true values. We can model the weights of a NN not as scalars but rather as probability distributions. Such a network is referred to as Bayesian neural network (BNN; Denker and Lecun, 1991; MacKay, 1992; Neal, 1995) and is at the intersection of neural networks and Bayesian inference. However, the uncertainty modeling with BNNs comes at a price. It is usually computationally infeasible to perform exact Bayesian inference as this involves solving integrals in the dimension of the number of weights. We can resort to variational inference (VI) to approximate the true posterior of the BNN (Hinton and Camp, 1993) with, e.g., Bayes by Backprop (BBB; Graves, 2011; Blundell et al., 2015) or probabilistic backpropagation (PBP; Hernández-Lobato and Adams, 2015). Alternatively, we might use Markov chain Monte Carlo (MCMC) to sample from the posterior (Neal, 1992; Welling and Teh, 2011; Chen et al., 2014; Zhang et al., 2019). Monte Carlo dropout (MC dropout; Gal and Ghahramani, 2016; Kendall and Gal, 2017) generates samples from the approximate posterior by applying dropout – randomly setting the output of some layer neurons to zero – during training *and* testing. To lower the computational cost of BNNs, we can perform Bayesian inference in the last layer only, e.g., with a heteroscedastic variant of the neural linear model (NLM; Moberg et al., 2019). A promising adaption to NLM is the marginally calibrated deep distributional regression method (DNNC; Smith and Klein, 2021; Klein et al., 2021) that guarantees marginal calibration.

Aim and Methodology

Modeling uncertainty with neural networks is a relatively new field as it requires vast amounts of computational resources. At present, there is no clear best method. It is necessary to extensively benchmark them to derive insights about their characteristics and the quality of their predictive uncertainty. This is what we aim for in this thesis. Specifically, our benchmark experiments are designed to answer the following questions:

1. How well do non-Bayesian and Bayesian NNs estimate uncertainty in a regression setting?
2. In which scenarios is capturing epistemic uncertainty important, in which aleatoric?
3. Are the last-layer inference methods NLM and DNNC competitive?

We proceed in the following way. First, we run the methods on toy data to accentuate the basic properties of the methods and compare them to a *ground truth* Gaussian process (GP). Second, we assess both the calibration and the predictive performance on UCI regression datasets (Dua and Graff, 2017), standard real-world regression datasets used in the literature. We consider their – marginal and probabilistic – calibration on the full data to gauge the consistency of the forecasts and the observations (Gneiting et al., 2007). We obtain robust predictive performance results and assess the models' ability to interpolate and extrapolate uncertainty via gap splits (Foong et al., 2019b) and tail splits, respectively. We evaluate the models on a broad range of performance metrics such as logarithmic score (LogS; Good, 1952) and continuous ranked probability score (CRPS; Matheson and Winkler, 1976; Unger, 1985), both of which are proper scoring rules (Gneiting et al., 2007). We further contribute to the literature by providing empirical evidence that DNNC may not require the transformation of the targets with the empirical distribution of the response to fit the underlying NN, and instead, a zero-mean and unit-variance standardization suffices. This could enable easier and wider applicability of the general approach.

Due to limited computational power and time constraints, we must limit our benchmarking in three ways. First, we focus solely on the regression case and do not evaluate methods on (multi-class) classification. Second, we do not compare all existing methods/variants. We select PNN and PNN-E as easy-to-implement non-Bayesian benchmarks and further compare the promising methods MC dropout, PBP, NLM, and DNNC. Third, we follow the typical experimental setup for UCI regression tasks from the literature (Hernández-Lobato and Adams, 2015) and thus do not evaluate the influence of the number of layers on the quality of the methods' uncertainty estimates. This factor might strongly influence the performance of the last-layer inference methods NLM and DNNC.

Related Works

The study of uncertainty-aware NN methods is an active field of research. Model performance is typically assessed with random splits on UCI regression tasks (Hernández-Lobato and Adams, 2015; Gal and Ghahramani, 2016; Lakshminarayanan et al., 2016; Moberg et al., 2019). Foong et al. (2019b) introduced gap splits for the UCI datasets and studied the (in)ability of variational methods and MC dropout to predict in-between uncertainty. Ober and Rasmussen (2019) found that the neural linear model performs well for random and gap splits if (costly) hyperparameter tuning is performed.[2]

Kendall and Gal (2017) proposed a variant of MC dropout that captures aleatoric and epistemic uncertainty, and benchmarked it to standard MC dropout on depth regression problems. Capturing epistemic uncertainty is essential for small datasets or safety-critical applications. In contrast, modeling aleatoric uncertainty is important for large datasets where epistemic uncertainty is explained away. Further predictive performance for dataset shifts was studied by Ovadia et al. (2019). They found that existing methods struggle with dataset shifts. Ovadia et al. (2019) argue that the importance of modeling epistemic uncertainty increases with the dataset shift. Generally, ensemble methods performed best in their experiments.

Outline

The remainder of this thesis is outlined as follows. Chapter 2 begins by introducing uncertainty. Then, we review the basics of Bayesian statistics and neural networks to lay the foundation for all methods considered in this thesis. Chapter 3 gives an extensive overview of methods that enable us to combine neural networks with uncertainty estimates. Some of these methods are benchmarked in Chapter 4 on toy regression data and real-world UCI regression datasets, using various metrics. We conclude in Chapter 5 and outline areas for future research.

[2]The version used by Ober and Rasmussen (2019) differs from our implementation. We used the heteroscedastic version outlined in Moberg et al. (2019).

2 Background

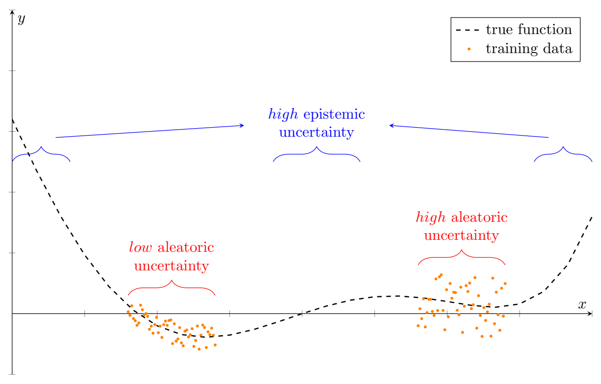
As this thesis aims to compare a variety of models on their ability to predict uncertainty, we start this chapter by introducing uncertainty and explaining the classification into *aleatoric* and *epistemic* uncertainty. The benchmarked models are at the intersection of Bayesian statistics and neural networks (NNs). In this chapter, we thus outline both areas to enable an unfamiliar reader to comprehend the methods presented in Chapter 3.

2.1 Uncertainty

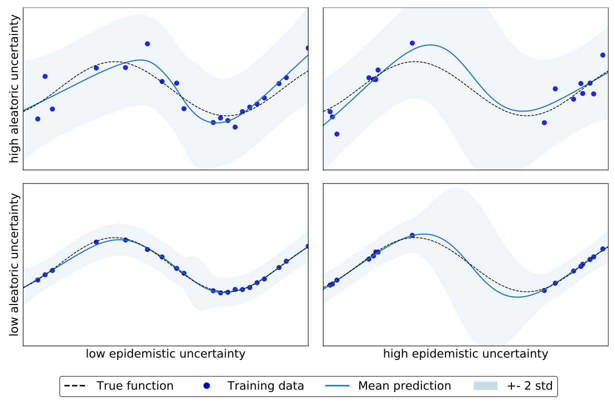
Let us denote some data with $\mathcal{D} := (\boldsymbol{X}, \boldsymbol{y}) = \{(\boldsymbol{x_i}, y_i)\}_{i=1}^{n}$, where $\boldsymbol{x_i} \in \mathbb{R}^d$ are feature vectors and $y_i \in \mathbb{R}$ scalar target variables. Assume we have trained some model on this data to learn a relationship $y_i = f(\boldsymbol{x}_i)$ where the targets are corrupted by noise ϵ_i. Given a new input $\boldsymbol{x}_*$, we are interested in a predictive density $\hat{p}(y_*|\boldsymbol{x}_*)$ for the output rather than a point prediction $\hat{y}_*$. The predictive density encodes a model's uncertainty about its predictions for y_*. This uncertainty is the sum of *aleatoric* and *epistemic* uncertainty (Hora, 1996; Kiureghian and Ditlevsen, 2009).

Aleatoric uncertainty is an inherent feature of the data generating process (DGP). This type of uncertainty reflects noise in the observations, e.g., measurement errors in the data or sensor noise. It is independent of the amount of data we collect. For example, we cannot predict the realization of the measurement error of a thermometer. The noise in the observations can be either *homoscedastic* or *heteroscedastic*. Homoscedastic noise is constant for all inputs, in the sense that the variance of the noise-generating random process does not change, $\epsilon_i \sim N(0, \sigma)\ \forall i$. Heteroscedastic noise implies that the variance of the noise differs for inputs, $\epsilon_i \sim N(0, \sigma_i)$.

Epistemic uncertainty or *model uncertainty* refers to uncertainty in the model parameters and is a *property* of the model (Kendall and Gal, 2017). To draw an analogy to a certain detective: if there are few data points in a fictitious murder case, multiple explanations (or model parameters) may fit the data obtained so far. Observing then that the murderer is left-handed and smokes a particular type of tobacco shrinks the set of possible explanations significantly. That is, the more data points are available, the lower the uncertainty in the model parameters. We can *explain away* epistemic uncertainty with data (Kendall and Gal, 2017).



(a)



(b)

Figure 2.1: Which predictive uncertainty do we expect, and where does it come from? (a) Regression problem with regions of altering aleatoric uncertainty in the data and areas without data. (b) Regressing sin(x) with a Gaussian process for different scenarios. In the lower-left, there is low noise in the data. In the lower-right, we do not observe data for a subset of the input range. The upper graphs depict the scenarios as below but with increased noise in the data.

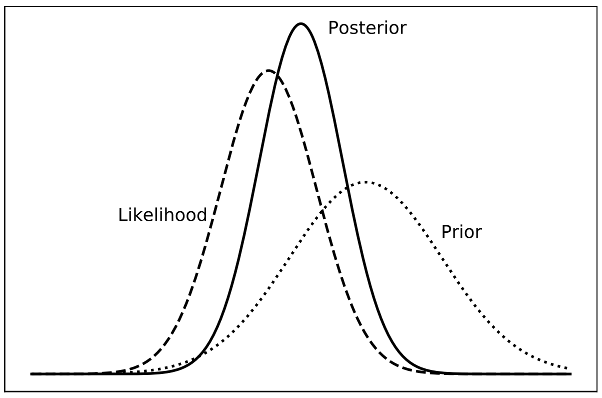
In Figure 2.1a we illustrate both types of uncertainty. For some (regression) problem, there might exist regions where we have collected data with varying aleatoric uncertainty due to different measuring processes. Further, we might have areas without data. Here we expect (high) predictive epistemic uncertainty.

We show the predictive uncertainty of a model that behaves in accordance with our beliefs in Figure 2.1b. We depict four scenarios with all possible combinations of low/high aleatoric/epistemic uncertainty. If there is no gap in the training data and we have low noise in the data (lower-left), the model captures the true function and has relatively low predictive uncertainty for the whole data range. In the lower-right, we show a scenario where we have an input region with no data. In the region without data, the predictive uncertainty is increased to account for epistemic uncertainty. The upper graphs depict the scenarios as below but with increased noise in the data. We see that the total uncertainty increases and that the models' ability to capture the true function decreases.

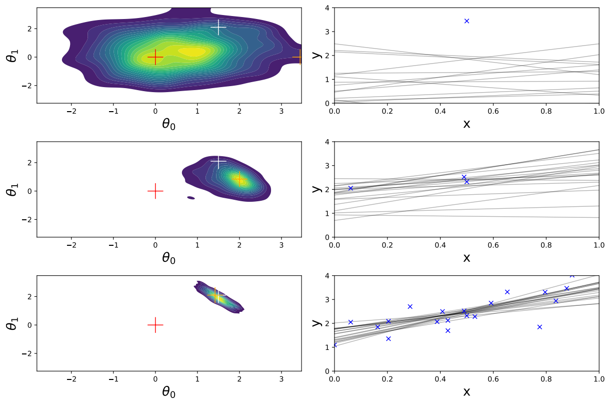
It can be helpful to decompose the predictive uncertainty of a model into its aleatoric and epistemic parts (Depeweg et al., 2017). In active learning, for example, we can ask for data inputs where the epistemic uncertainty is high to improve the quality of our model's prediction. Similarly, in autonomous driving, we want to derive scenarios where our model needs more training samples, e.g., images of a road on a rainy day at sunset.

2.2 Bayesian Statistics

This section introduces the basic concepts of Bayesian statistics with a simple Bayesian linear regression. We will see that it is often impossible to derive analytical solutions to quantities we



(a) Bayesian belief update



(b) Sequential Bayesian inference

Figure 2.2: Bayesian learning. (a) We update our beliefs about the distribution of a parameter θ after seeing some data. (b) Sequential Bayesian inference for a linear regression model $p(y|x) = N(y|\theta_0 + \theta_1 x, \sigma^2)$. Left column: Posterior given n data points. The true parameters that generate the data are marked as white cross. We plot the prior mode as red cross and the maximum likelihood estimation of the parameters as orange cross. The posterior is given as contour plot. Right column: samples from the posterior predictive. Blue Xes mark the training data. Row 1: after $n = 1$ data point. Row 2: after $n = 3$ data points. Row 3: after $n = 20$ data points. (b) based on Murphy (2012).

are interested in. In these cases, typically Markov chain Monte Carlo (MCMC) and variational inference (VI) are used to derive approximate solutions.

2.2.1 Bayes by Example: Bayesian Linear Regression

Denote the data with $\mathcal{D} := (\boldsymbol{X}, \boldsymbol{y}) = \{(\boldsymbol{x_i}, y_i)\}_{i=1}^n$, where $\boldsymbol{x_i} \in \mathbb{R}^d$ are feature vectors and $y_i \in \mathbb{R}$ scalar target variables. We use the classical linear regression setup:

$$y_i = \boldsymbol{\theta} \boldsymbol{x}_i^T + \epsilon_i, \quad \epsilon_i \sim \mathcal{N}(0, \sigma^2) \text{ i.i.d.} \tag{2.1}$$

In other words, we assume that the observed value of the response variable is corrupted with observation noise ϵ_i that is normally distributed. This corresponds to the following likelihood – the probability of observing the data as a function of the (unknown) model parameters:

$$p\left(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}\right) = \mathcal{N}\left(\mathbf{y}|\mathbf{X}\boldsymbol{\theta}, \sigma^2 \mathbf{I}\right) , \tag{2.2}$$

where for now we assume σ^2 is known. In the frequentist method of *maximum likelihood*, we estimate the unknown parameters by values $\hat{\boldsymbol{\theta}}$ that maximize this likelihood. For example, with ordinary least squares, we find estimates for the parameters with $\hat{\boldsymbol{\theta}} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{y}$.

In the Bayesian approach, we incorporate knowledge about the unknown parameters, i.e., we put a *prior* over the parameters with $p(\boldsymbol{\theta})$. The prior encodes our beliefs about the distribution of parameters before we observe any data. When we observe data, we *update* our beliefs. This process is shown in Figure 2.2a where our *posterior* beliefs are influenced by our prior beliefs and the data we see. In Figure 2.2b we display the idea of sequential Bayesian inference. Given just one data point in the first row, the prior highly influences our posterior beliefs. The more data we observe, the more the posterior is influenced by the likelihood. In the limit of infinite data, the posterior mean coincides with maximum-likelihood estimation.

We use Bayes' theorem to compute the posterior distribution for the parameters:

$$\underbrace{p(\boldsymbol{\theta}|\boldsymbol{X},\boldsymbol{y})}_{\text{posterior}} = \frac{\overbrace{p\left(\mathbf{y}|\mathbf{X},\boldsymbol{\theta}\right)}^{\text{likelihood}}\overbrace{p(\boldsymbol{\theta})}^{\text{prior}}}{\underbrace{\int p\left(\mathbf{y}|\mathbf{X},\boldsymbol{\theta}\right)p(\boldsymbol{\theta})d\boldsymbol{\theta}}_{\text{evidence}}} = \frac{p\left(\mathbf{y}|\mathbf{X},\boldsymbol{\theta}\right)p(\boldsymbol{\theta})}{p(\boldsymbol{y}|\boldsymbol{X})}\,, \tag{2.3}$$

where $p(\boldsymbol{y}|\boldsymbol{X}) = \int p\left(\mathbf{y}|\mathbf{X},\boldsymbol{\theta}\right)p(\boldsymbol{\theta})d\boldsymbol{\theta}$ is called *evidence* or *marginal likelihood* as $\boldsymbol{\theta}$ has been marginalized out. Note that we do not view the parameters $\boldsymbol{\theta}$ as point estimates as in the frequentist approach but as a distribution over possible values. The uncertainty in the parameter values propagates to the prediction of y^* given an input $\boldsymbol{x}^*$. We can compute the *posterior predictive* as:

$$p(y^*|\boldsymbol{X},\boldsymbol{y},\boldsymbol{x}^*) = \int p(y^*|\boldsymbol{x}^*,\boldsymbol{\theta})p(\boldsymbol{\theta}|\boldsymbol{X},\boldsymbol{y})d\boldsymbol{\theta}\,. \tag{2.4}$$

The integrals in equations (2.3) and (2.4) are usually difficult to compute analytically. However, if we choose a *conjugate* prior, we can derive analytical solutions. For example, we use a Gaussian prior for the parameters $p(\boldsymbol{\theta}) = \mathcal{N}\left(\boldsymbol{\theta}|\boldsymbol{\theta}_0,\mathbf{V}_0\right)$ with mean θ_0 and covariance $\mathbf{V}_0$. Let the inputs be centered and set the outputs to $\boldsymbol{y} = \boldsymbol{y} - \bar{y}\mathbb{1}_n$, where $\bar{y} = \frac{1}{n}\sum_{i=1}^n y_i$ denotes the empirical mean. We derive the posterior as (Murphy, 2012):

$$p\left(\boldsymbol{\theta}|\mathbf{X},\mathbf{y},\sigma^2\right) \propto \mathcal{N}\left(\mathbf{y}|\mathbf{X}\boldsymbol{\theta},\sigma^2\mathbf{I}\right)\mathcal{N}\left(\boldsymbol{\theta}|\boldsymbol{\theta}_0,\mathbf{V}_0\right) = \mathcal{N}\left(\boldsymbol{\theta}|\boldsymbol{\theta}_n,\mathbf{V}_n\right)\,. \tag{2.5}$$

The posterior predictive distribution of the response y_* for a test input $\boldsymbol{x}_*$ is:

$$p\left(y_*|\mathbf{x}_*,\mathcal{D},\sigma^2\right) = \mathcal{N}\left(y_*|\boldsymbol{\theta}_n^T\mathbf{x}_*,\sigma^2 + {\mathbf{x}_*}^T\mathbf{V}_n\mathbf{x}_*\right)\,. \tag{2.6}$$

If σ^2 is unknown, we could choose a conjugate prior of the form $p(\boldsymbol{\theta},\sigma^2) = p(\sigma^2)p(\boldsymbol{\theta}|\sigma^2)$ where $p(\sigma^2)$ follows an inverse-gamma distribution. We refer to Ober and Rasmussen (2019) for the analytically derived posterior and posterior predictive. If we chose a non-conjugate prior, it would likely be infeasible to perform marginalization, and we need to perform approximate inference with Markov chain Monte Carlo or variational inference.

2.2.2 Approximate Bayesian Inference

For notational simplicity, let $\boldsymbol{X}$ denote some data and θ an unknown hidden variable (which may has several components). We are interested in the posterior from equation (2.3) where it is infeasible to compute the denominator analytically. Thus, we only know the posterior up to a normalizing constant. A popular method to avoid the computation of the integrals is to *sample* from the posterior with Markov chain Monte Carlo and use the samples to approximate the multi-dimensional integral numerically. MCMC, however, can be susceptible to the curse of dimensionality or computationally too expensive, and we may use variational inference to approximate the true posterior with a parametrized distribution $q(\theta; \lambda)$.

Markov Chain Monte Carlo

A class of techniques known as Markov chain Monte Carlo methods can be used to sample from probability distributions. The idea is to generate a *Markov chain* of (correlated) samples $\left\{\theta^{(i)}\right\}, i = 1, 2, ..., J$ that converges to the true distribution and evaluate the samples to compute an integral over that variable (Neal, 1993). The *Monte Carlo* (MC) part characterizes the fact that we use a state of the Markov chain as an approximately random sample from the target distribution. In particular, we use can the sample mean $\bar{\theta} = \frac{1}{J}\sum_{j=1}^{J}\theta^{(j)}$ as an estimate for the posterior mean.

There are several algorithms to perform MCMC with different pros and cons. We briefly introduce a selection of widely used MCMC algorithms. We start with Metropolis-Hastings (Metropolis et al., 1953; Hastings, 1970), the first MCMC algorithm. Gibbs sampling (Geman and Geman, 1984; Gelfand and Smith, 1990) can perform better if the dimension of the parameter space is high. Finally, the basics of Hamiltonian Monte Carlo (HMC; Duane et al., 1987; Neal, 1992) and HMC's extension, the No-U-Turn sampler (NUTS; Hoffman and Gelman, 2014) are presented.

Metropolis-Hastings MCMC algorithms date back to Metropolis et al. (1953) who simulated random draws from a Boltzmann distribution of particles to compute an expectation $\mathbb{E}[g(\boldsymbol{x})] = \int g(\boldsymbol{x})p(\boldsymbol{x})dx$ where $p(\boldsymbol{x})$ was only known up to a normalizing constant. More precisely, the algorithm used does not require simulating from the target distribution directly but only from the *unnormalized* target distribution, a property that makes the algorithm particularly useful (Martin et al., 2020). The Metropolis-Hastings algorithm (Hastings, 1970) generalizes Metropolis et al. (1953) with non-symmetrical proposal densities. The procedure to obtain the next sample in the Markov chain is outlined in Algorithm 1.

The proposal density $q(\theta|\theta^{(t)})$ is used to generate a candidate θ^* given the current state. We accept the candidate with a certain probability that depends on the function values $f(\cdot)$ (and the

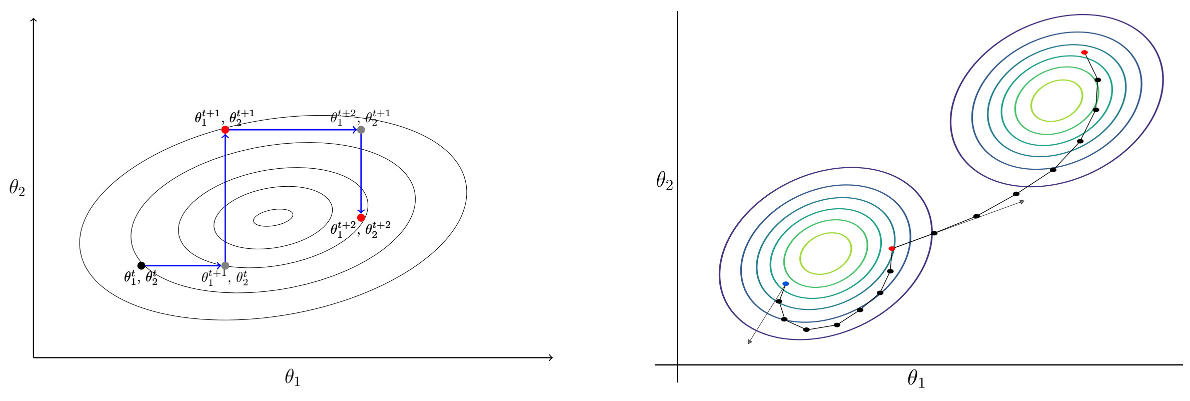
Algorithm 1: Metropolis–Hastings

```
  input: Arbitrary starting point $\theta^{(t)}$
         Unnormalized target density $f(\theta)$
         Proposal density $q(\theta|\theta^{(t)})$
1 Generate proposal $\theta^* \sim q(\theta|\theta^{(t)})$
2 Calculate acceptance probability $\alpha = \min\left(1, \frac{f(\theta^*)}{f(\theta^{(t)})}\frac{q(\theta^{(t)}|\theta^*)}{q(\theta^*|\theta^{(t)})}\right)$
3 Sample $u \sim \text{unif}(0,1)$
4 if $u < \alpha$ then
5     $\theta^{(t+1)} \leftarrow \theta^*$   // accept proposal
6 else
7     $\theta^{(t+1)} \leftarrow \theta^{(t)}$   // reject proposal
8 end
```

non-symmetric proposal densities). The algorithm examines the parameter space in a *random walk*. Intuitively, the algorithm stays longer in regions with a high probability density and visits regions with a low density less frequently. We usually discard a large proportion of the samples by using only every n'th sample since the algorithm produces successive samples that are correlated. The initial distribution depends on the starting point, and it is thus recommended to use an *initial phase* (Neal, 1995) or *burn-in period* (Gelman et al., 2013) and discard those samples.

Gibbs Sampling If the dimension of the parameter space is high, it can be challenging to find a suitable proposal density and Metropolis-Hastings might suffer from *slow mixing*. Gibbs sampling (Geman and Geman, 1984; Gelfand and Smith, 1990) instead updates the components of a parameter vector $\boldsymbol{\theta}^t = (\theta_1^t, ..., \theta_p^t)^T$ sequentially. The algorithm is shown in Figure 2.3a and requires generating samples from the full conditional distributions $p(\theta_i|\theta_1^{t+1}, ..., \theta_{i-1}^{t+1}, \theta_{i+1}^t, ..., \theta_p^t)$. However, strongly correlated components can lead to slow convergence (Cowles and Carlin, 1996).

HMC Both Metropolis-Hastings and Gibbs sampling practically explore the parameter space in a random walk and may take a long time to converge. Hamiltonian Monte Carlo (HMC) is a MCMC method that uses gradient information to avoid this random walk behavior. HMC or *hybrid Monte Carlo* was originally developed in lattice field theory for numerical simulation (Duane et al., 1987) and introduced to statistics by Neal (1992). We use the analogy presented by Neal (2012) and visualize Hamiltonian dynamics as a puck that slides over some surface without friction. This system can be described by states that are tuples of *position* and *momentum* of the puck, both represented as vectors. The idea of using HMC for MCMC is to push the puck in a random direction, pick it up after a while and use this position as the next state in the Markov chain. Thus, HMC interchanges the problem of sampling from the posterior with simulating Hamiltonian dynamics (Neal, 2012; Hoffman and Gelman, 2014).


θ_2
$\theta_1^{t+1}, \theta_2^{t+1}$
$\theta_1^{t+2}, \theta_2^{t+1}$
$\theta_1^{t+2}, \theta_2^{t+2}$
$\theta_1^{t}, \theta_2^{t}$
$\theta_1^{t+1}, \theta_2^{t}$
θ_1
θ_2
θ_1

(a) Gibbs sampling (b) Hamiltonian Monte Carlo

Figure 2.3: Markov chain Monte Carlo. (a) Two iterations of the Gibbs sampler which generates the next sample (red) in the Markov chain by sampling sequentially from the full conditional. (b) Sampling a mixture of two bivariate Gaussians with HMC. We show two trajectories with $L = 8$ leapfrog steps. New states for the Markov chain are marked as red dots, momentum samples as gray arrows, and leapfrog steps as black dots. Figure (b) is loosely based on Neal (1995).

To simulate Hamilton dynamics in a probabilistic setup, we introduce an auxiliary momentum vector r_k for each parameter. Thus, we expand the p-dimensional parameter space to a $2p$-dimensional *phase space* (Neal, 2012; Betancourt, 2017):

$$\theta_k \to (\theta_k, r_k)\,. \tag{2.7}$$

We focus here on the core algorithm and refer to Neal (1995) for an in-depth introduction to Hamilton dynamics.

We have to simulate Hamilton dynamics in discrete time. First, we sample a momentum vector from the multivariate normal. Then, we perform L – reversible and volume-preserving – *leapfrog* steps with step size ϵ to generate a proposal position. We accept or reject this proposal, similarly to the Metropolis algorithm outlined in Algorithm 1. In Figure 2.3b, we illustrate sampling a mixture of two 2D Gaussians with HMC. The full HMC procedure is summarized in Algorithm 2.

Due to the non-random behavior induced by following the first-order gradient, HMC converges faster than other MCMC algorithms. We can generate an *independent* sample from the target distribution in about $O(d^{5/4})$, compared to $O(d^2)$ for random-walk Metropolis (Hoffman and Gelman, 2014). However, the leapfrog updates in HMC require the gradient of the log-posterior, which can be impossible to obtain analytically. We can approximate the gradient numerically, which might result in computational overhead.

Further, HMC requires extensive hand-tuning of step size ϵ and number of steps L. These are crucial parameters to generate reliable samples from the true posterior and must be selected with

Algorithm 2: Hamiltonian Monte Carlo

```
input: Number of samples $M$
       Arbitrary starting point $\theta^{(0)}$
       (Unnormalized) Log joint density $\mathcal{L}$
       Number of steps $L$, step size $\epsilon$
for $m$=1 to $M$ do
    Sample momentum vector $r^{(0)} \sim N(0, I)$
    $\tilde{\theta} \leftarrow \theta^{(m-1)}, \tilde{r} \leftarrow r^{(0)}$  // init leapfrog
    for $i$=1 to $L$ do
        $\tilde{\theta}, \tilde{r} \leftarrow$ Leapfrog($\tilde{\theta}, \tilde{r}, \epsilon$)  // perform $L$ leapfrog steps of size $\epsilon$
    end
    Calculate acceptance probability $\alpha = \min\left(1, \frac{\exp\left(\mathcal{L}(\tilde{\theta}) - \frac{1}{2}\tilde{r}\cdot\tilde{r}\right)}{\exp\left(\mathcal{L}(\theta^{(m-1)}) - \frac{1}{2}r^{(0)}\cdot r^{(0)}\right)}\right)$
    Sample $u \sim \text{unif}(0, 1)$
    if $u < \alpha$ then
        $\theta^{(m)} \leftarrow \tilde{\theta}$  // accept state as new sample
    else
        $\theta^{(m)} \leftarrow \theta^{(m-1)}$  // reject state
    end
end

function Leapfrog($\theta, r, \epsilon$):
    $\tilde{r} \leftarrow r + \frac{\epsilon}{2}\nabla_\theta \mathcal{L}(\theta)$
    $\tilde{\theta} \leftarrow \theta + \epsilon \cdot \tilde{r}$
    $\tilde{r} \leftarrow \tilde{r} + \frac{\epsilon}{2}\nabla_\theta \mathcal{L}(\tilde{\theta})$
    return $\tilde{\theta}, \tilde{r}$;
return
```

great care, a task that typically requires expert knowledge and expensive training runs (Hoffman and Gelman, 2014).

NUTS The *No-U-Turn sampler* (NUTS; Hoffman and Gelman, 2014) is an extension to HMC that eliminates the need to fine-tune the number of steps L and step size ϵ. As the name reveals, the core idea is to stop when the trajectory – the path of the positions in the Hamilton system – starts to turn back. NUTS proves to be an efficient method to explore the parameter space and requires no hand-tuning of parameters (Hoffman and Gelman, 2014). This makes NUTS the default sampler in most Bayesian inference tools such as Stan (Carpenter et al., 2017) or PyMC3 (Salvatier et al., 2016).

Variational Inference

The goal of VI is to approximate the intractable true posterior $p(\theta|\boldsymbol{X})$ by a simpler density $q(\theta; \lambda)$ that is parametrized by variational parameters λ (Bishop, 2006). Variational inference exchanges the marginalization problem of Bayesian inference with optimization. Figure 2.4 depicts the idea

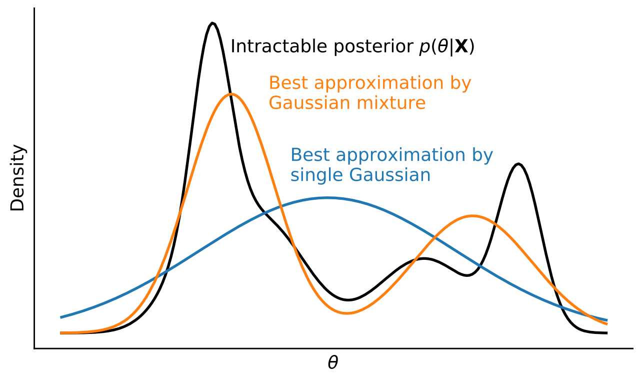


Figure 2.4: Approximating the posterior with variational inference. We want to find a simpler, parametrized density $q(\theta;\lambda)$ that minimizes the KL divergence to the posterior $p(\theta|\boldsymbol{X})$. We show in blue the best approximation using a single Gaussian, in orange the best approximation using a mixture of two Gaussians.

of variational inference. We see that the success of VI is highly dependent on the choice of the variational family $\mathcal{Q}$, i.e., in this example we cannot approximate the posterior well with a Gaussian distribution and lose a lot of information.

With VI, we want to find an approximating density or *variational posterior* $q(\theta;\lambda)$ that minimizes the *Kullback-Leibler* (KL) divergence (Kullback, 1959; Kullback and Leibler, 1951) – a non-symmetric measure of the discrepancy of two probability distributions (Waterhouse et al., 1996) – to the posterior $p(\theta|\boldsymbol{X})$. The KL divergence is defined as:

$$D_{\mathrm{KL}}[q(\theta;\lambda)||p(\theta|\boldsymbol{X})] := E_{q(\theta;\lambda)}\left[\log\frac{q(\theta;\lambda)}{p(\theta|\boldsymbol{X})}\right] = E_{q(\theta;\lambda)}\left[\log q(\theta;\lambda) - \log p(\theta|\boldsymbol{X})\right] . \tag{2.8}$$

ELBO We cannot minimize the KL divergence directly as we would need to compute $p(\theta|\boldsymbol{X})$. We can, however, rearrange the formula and express the KL divergence in terms of the evidence lower bound (ELBO) plus an unknown constant that does not depend on $q(\theta;\lambda)$ (Bishop, 2006):

$$\begin{aligned} D_{\mathrm{KL}}[q(\theta;\lambda)||p(\theta|\boldsymbol{X})] &= E_{q(\theta;\lambda)}[\log q(\theta;\lambda)] - E_{q(\theta;\lambda)}[\log p(\theta|\boldsymbol{X})] \\ &= E_{q(\theta;\lambda)}[\log q(\theta;\lambda)] - E_{q(\theta;\lambda)}[\log p(\theta,\boldsymbol{X})] + \log p(\boldsymbol{X}) \\ &= -E_{q(\theta;\lambda)}[\log p(\theta,\boldsymbol{X}) - \log q(\theta;\lambda)] + \log p(\boldsymbol{X}) \\ &= -\mathrm{ELBO}[q(\theta;\lambda)] + \log p(\boldsymbol{X}) . \end{aligned} \tag{2.9}$$

The ELBO is defined as $E_{q(\theta;\lambda)}[\log p(\theta,\boldsymbol{X}) - \log q(\theta;\lambda)]$. We minimize the KL divergence by maximizing the ELBO. The optimization problem of VI becomes:

$$\lambda^* = \arg\max_{\lambda} \mathrm{ELBO}[q(\theta;\lambda)] . \tag{2.10}$$

Essentially, by maximizing the ELBO or the *variational free energy*, we place probability mass where the true posterior has a high density with $E_{q(\theta;\lambda)}[\log p(\theta, \boldsymbol{X})]$ and simultaneously penalize entropy – a measure of complexity – in q.

It is usually difficult to gauge whether variational inference has *worked* well, and the results are often validated by comparing them to those obtained with MCMC. This poses a challenge to VI since we could solely run MCMC in this case. Developing diagnostics to assess whether the approximation with VI is appropriate is an active field of research (Yao et al., 2018).

2.3 Neural Networks

This section illustrates the fundamental concepts of neural networks (NNs). We first introduce the building blocks and outline the process of training a NN. Next, the optimization problem is depicted in more detail. Finally, we introduce a few regularization methods.

2.3.1 Building Blocks and General Procedure

A neural network can be described as a function $f(\boldsymbol{x})$ that maps input values $\boldsymbol{x}_i$ to output values y_i (here scalars). We can think of this function as the composition of many simpler functions (Goodfellow et al., 2016):

$$f(\boldsymbol{x}) = \left(f^{(1)} \circ f^{(2)} \circ ... \circ f^{(L)}\right)(\boldsymbol{x}) \tag{2.11}$$

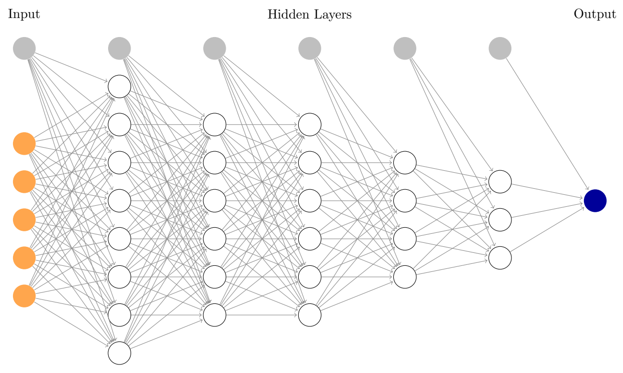
where L denotes the number of layers in the network. An example of a NN with five hidden layers is depicted in Figure 2.5a. Each layer in the NN transforms its input data by an affine transformation followed by a non-linear transformation:

$$f^{(l)}(\boldsymbol{z}^{(l-1)}) = \varphi^{(l)}(\boldsymbol{A}^{(l)}\boldsymbol{z}^{(l-1)} + \boldsymbol{b}^{(l)})\,, \tag{2.12}$$

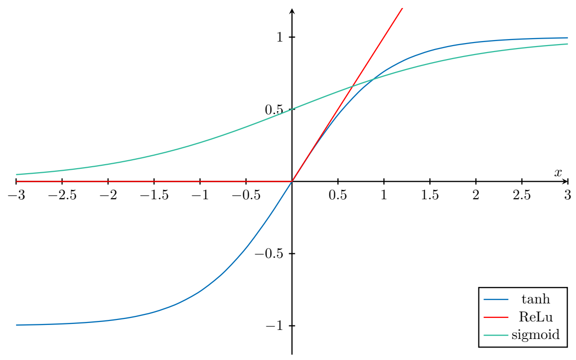
where $\boldsymbol{A}^{(l)}$ and $\boldsymbol{b}^{(l)}$ are the weights and biases that connect layers $l-1$ and l, $\boldsymbol{z}^{(l)}$ denotes the output of layer l, and φ denotes a non-linear transformation or *activation function.* We show some typical activation functions in Figure 2.5b. The most commonly used activation function in hidden layers is the rectified linear unit (ReLU; Nair and Hinton, 2010):

$$\varphi_{\text{ReLU}}(a) = max(0, a)\,. \tag{2.13}$$

The activation function for the output layer depends on the problem at hand. That is, for regression problems, where the target is unbounded, we use the linear – also called identity – activation function $\varphi_{\text{Linear}}(a) = a$.



(a)



(b)

Figure 2.5: Components of a neural network. (a) Schematic example of a deep NN with $L = 6$ layers and single output neuron. Input neurons are marked in orange, output neurons in blue. Hidden neurons are depicted in white, biases in gray. (b) Typical activation functions that neurons apply after an affine transformation of their inputs. The linear activation function $\varphi_{\text{Linear}}(a) = a$, common for the output layer in regression, left out for visibility.

Let $\mathcal{W} = \{\boldsymbol{W}^{(l)}\}_{l=1}^{L}$ denote the trainable parameters or *weights* that characterize the NN. $\boldsymbol{W}^{(l)}$ is a $V_l \times (V_{l-1} + 1)$ weight matrix, where V_l denotes the number of network units in layer l and the $+1$ accounts for the biases. All parameters are typically initialized randomly. The training of the NN consists of the repetition of the following five steps (Chollet, 2017):

1. Select a (mini) *batch* with size m of training samples $\boldsymbol{x}_{1\ldots m}$ and target values $y_{1\ldots m}$.
2. Make a *forward pass* to generate the network's outputs $\widehat{y}_{1\ldots m} = f_{\mathcal{W}}(\boldsymbol{x}_{1\ldots m})$.
3. Calculate the *loss* $\mathcal{L}(y_{1\ldots m}, \widehat{y}_{1\ldots m})$ on the batch, where $\mathcal{L}(\cdot)$ denotes some loss function.
4. Compute the gradients of the loss w.r.t. the parameters via *backpropagation*.
5. Update the network weights using the gradients to (moderately) reduce the loss on the batch.

2.3.2 Optimization

We can view the training of a neural network as the following optimization problem:

$$\mathcal{W}^* = \arg\min_{\mathcal{W}} \left[\mathcal{L}\left(\boldsymbol{y}, f_{\mathcal{W}}\left(\boldsymbol{X}\right)\right)\right] , \tag{2.14}$$

where we want to find the set of weights $\mathcal{W}^*$ that has the smallest loss on our training data $\mathcal{D} = (\mathbf{X}, \mathbf{y})$. The loss function measures the discrepancy between predictions $\widehat{\boldsymbol{y}} = f_{\mathcal{W}}(\boldsymbol{X})$ and targets $\boldsymbol{y}$. Depending on the problem, different loss functions are used.

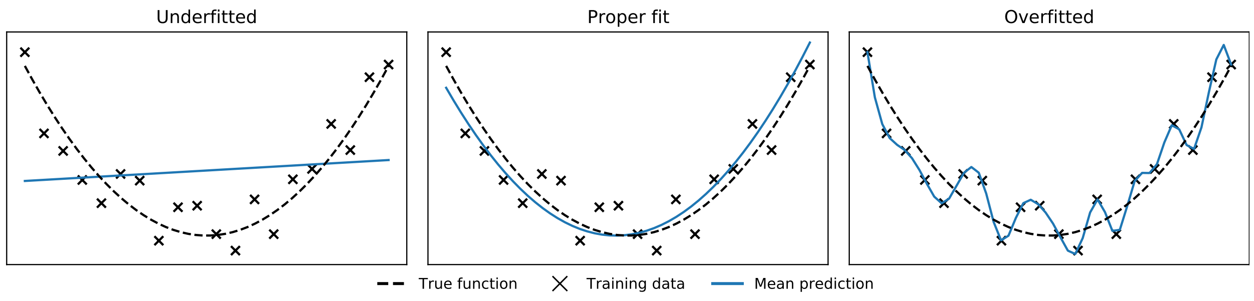


Figure 2.6: Over-fitting and under-fitting: A key challenge in machine learning is to find a model architecture that is both complex enough to identify the signal in the data and not too complex as to over-fit on noise. Loosely based on Goodfellow et al. (2016).

For regression problems, the most commonly used loss functions are mean square error (MSE) and mean absolute error (MAE):

$$\mathcal{L}_{\text{MSE}}(y_{1\ldots m}, \widehat{y}_{1\ldots m}) = \frac{1}{m}\sum_{i}^{m}(y_i - \hat{y}_i)^2 \,, \tag{2.15}$$

$$\mathcal{L}_{\text{MAE}}(y_{1\ldots m}, \widehat{y}_{1\ldots m}) = \frac{1}{m}\sum_{i}^{m}|y_i - \hat{y}_i| \,, \tag{2.16}$$

where we compute the loss for a batch of size m in this notation.

The weight update occurs in two steps. First, we compute the gradients of the loss with regard to the weights. This is usually done via backpropagation (Rumelhart et al., 1986) that obtains these gradients for all weights using the chain rule iterating backward from the last layer. Second, we use the computed gradient information and move the weights by a step factor in the direction opposite to the gradient: $\mathcal{W}_{j+1} = \mathcal{W}_j - \eta(\frac{\delta\mathcal{L}}{\delta\mathcal{W}_j})$, where $\eta > 0$ is the size of the step or *learning rate*.

There exist plenty of so-called *first-order* optimization techniques, techniques that require the gradient. Stochastic gradient descent (SGD) uses mini-batches to compute the gradient of the loss with respect to the parameters and uses a constant learning rate η (Goodfellow et al., 2016). Interesting variants of SGD are SGD with momentum (Sutskever et al., 2013) as well as RMSprop and Adagrad (Duchi et al., 2011) that maintain a learning rate for *each* parameter. Throughout this thesis, we train our neural networks with the Adam optimizer from Kingma and Ba (2015). Adam is recommended as the default optimizer since it achieves good results quickly (Ruder, 2016). Essentially, the Adam optimizer computes an adaptive learning rate for each network weight based on the gradients' first two moments. Adam thus combines RMSprop with momentum and is suitable for fast network training with noisy or sparse gradients (Kingma and Ba, 2015).

2.3.3 Regularization

A key challenge in machine learning, in general, is to distinguish signal from noise. The more we increase the complexity of the model of choice, the better it adapts to the training data with enough training time. That is, it learns both signal *and* noise from the training data. An example of this *over-fitting* to the data is depicted in Figure 2.6. Here we can see that the model fits the training data perfectly using a high degree polynomial (right). An over-fitted model likely does not generalize well to unseen data.

The technique of preventing a model from over-fitting is called *regularization*. Given a fixed amount of data and NN architecture, we can choose from different regularization techniques. In this section we describe *weight regularization*, *dropout* (Srivastava et al., 2014) and *batch normalization* (Ioffe and Szegedy, 2015).

The concept of weight regularization is used for a variety of machine learning models, from linear regression to neural networks. It works by adding a penalty term to the optimization problem (Hastie et al., 2009):

$$\arg\min_{\mathcal{W}} \left[\mathcal{L}\left(\boldsymbol{y}_i, f_{\mathcal{W}}\left(\boldsymbol{x}_i\right)\right) + \lambda J(f_{\mathcal{W}})\right] , \tag{2.17}$$

where $J(\cdot)$ is a penalty function. The most common penalties for neural networks are L1 regularization and L2 regularization. L1 regularization adds a penalty proportional to the absolute values of the weights for all layers. L2 regularization or *weight decay* penalizes proportional to the squared values of the parameters.

Dropout is a popular and effective strategy to perform regularization in neural networks. It is applied layer-wise and randomly sets some output values of the layer to zero during the NN training. Intuitively, dropout masks the output of a layer $\boldsymbol{z}^{(l)}$ with a tensor $\boldsymbol{d}^{(l)}$ of the same size filled with ones and zeros. Every element in $\boldsymbol{d}^{(l)}$ is drawn from a Bernoulli distribution with probability $p = 1 - d$ where d denotes the *dropout rate*. The output of $\boldsymbol{z}^{(l)} \circ \boldsymbol{d}^{(l)}$ then is multiplied by the reciprocal of the dropout rate $1/d$ (Chollet, 2017). This process is shown in Figure 2.7. At test time, the outputs of the layers are usually not affected by dropout.

Training a neural network can be challenging as the distribution of the inputs for each layer may change during training after each mini-batch. This is also referred to as *internal covariate shift* (Ioffe and Szegedy, 2015). Batch normalization aims to *standardize* the inputs for the layers for each mini-batch. We can formalize it as the transformation $BN_{\gamma^{(l)},\beta^{(l)}} : \boldsymbol{z}^{(l)}_{1...m} \to \gamma^{(l)}\hat{\boldsymbol{z}}^{(l)}_{1...m} + \beta^{(l)}$, where $\hat{\boldsymbol{z}}^{(l)}_{1...m} = (\sigma^2_B + \boldsymbol{\epsilon})^{1/2}(\boldsymbol{z}^{(l)}_{1...m} - \mu_B)$, $\boldsymbol{\epsilon}$ is some noise, and μ_B and σ^2_B mark the first two moments of the inputs for a mini-batch B. The parameters γ and β are used for scaling and shifting, respectively, and have to be learned. Batch normalization is an effective technique for regularization that further

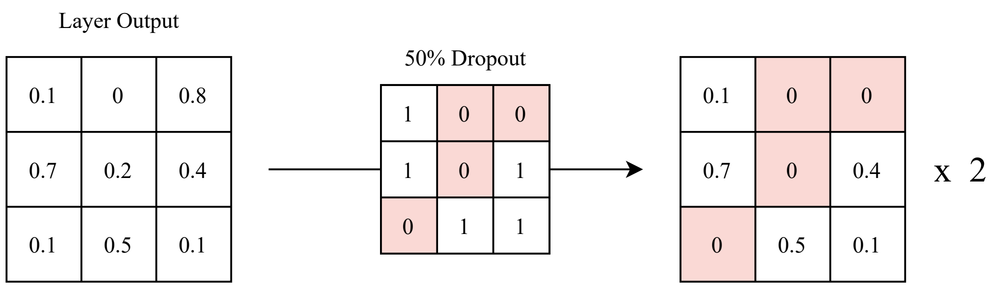


Figure 2.7: Dropout in neural networks. Dropout applied to a layer masks the output and scales it by the inverse of the dropout rate. Dropout is usually only applied during the training process for regularization. Visualization based on Chollet (2017).

accelerates the training process (Ioffe, 2017). The transformation is differentiable and thus does not interfere with back-propagation. For the details about the use in the inference stage and more, see Ioffe and Szegedy (2015).

3 Methods for Uncertainty Quantification in Neural Networks

In this chapter, we outline NN methods that allow for uncertainty quantification in regression problems. First, we describe a probabilistic NN (PNN; Nix and Weigend, 1994) and a deep ensemble (PNN-E; Lakshminarayanan et al., 2016), both of which are non-Bayesian approaches. Second, we introduce the concept of a Bayesian neural network (BNN; Denker and Lecun, 1991; MacKay, 1992; Neal, 1995), which is infeasible to calculate analytically. We illustrate approximate inference in BNNs with MCMC and VI with methods such as Bayes by Backprop (BBB; Graves, 2011; Blundell et al., 2015) and probabilistic backpropagation (PBP; Hernández-Lobato and Adams, 2015). Third, we present the basics of Gaussian processes (GPs; Rasmussen and Williams, 2006) that can be related to BNNs. A Gaussian process with ReLU-kernel (GP-ReLU; Lee et al., 2017) is be used as a *gold standard* benchmark in our toy problems in Chapter 4. Monte Carlo dropout (MC dropout; Gal and Ghahramani, 2016; Kendall and Gal, 2017) approximates a Gaussian process and is easily implemented in standard neural network software. Fourth, we outline two methods that perform Bayesian inference in the last layer of a NN: 1) a variant of the neural linear model (NLM; Moberg et al., 2019) and 2) the marginally calibrated deep distributional regression method (DNNC) of Klein et al. (2021). Finally, we qualitatively summarize the methods that we benchmark in this thesis.

3.1 Probabilistic NN and Deep Ensemble

3.1.1 Probabilistic NN

As described in Section 2.3.2, a NN is typically optimized – by minimizing the MSE or MAE – to output the predictive mean or predictive median, respectively. We can combine a neural network with a mixture of distributions in a mixture density network (MDN; Bishop, 1994; Bishop, 2006). More concretely, for an input $\boldsymbol{x}_i$, the output layer of the NN learns all parameters (weights and parameters for all mixture components) of the specified mixture of distributions. A probabilistic

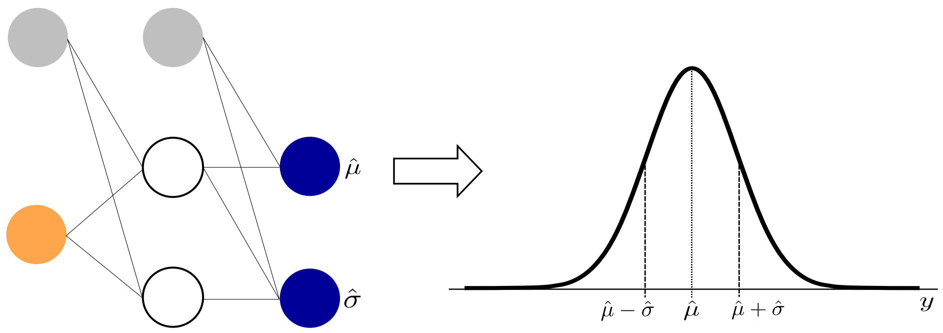


Figure 3.1: Probabilistic neural network: learning input-dependent noise in the data. Input neurons are marked in orange, output neurons in blue. Hidden neurons are depicted in white, biases in gray. Given an input $\boldsymbol{x}_i$, a PNN predicts mean $\hat{\mu}_i$ and aleatoric variance $\hat{\sigma}_i$. Note that there is no uncertainty in the model parameters (scalar weights), and thus a PNN cannot capture epistemic uncertainty.

NN can be described as a MDN with a single Gaussian and assumes that the outputs are corrupted with Gaussian noise. The PNN learns both the mean and aleatoric variance as functions of the input. See Figure 3.1 for more details. A PNN cannot capture epistemic uncertainty as this reflects uncertainty in the model's parameters, which are scalars for a PNN.

For our experiments, we follow Lakshminarayanan et al. (2016) and Kendall and Gal (2017) and choose an architecture where mean and variance output share the same weights for all but the last layer, as illustrated in Figure 3.1. However, one could use more sophisticated architectures where mean and variance outputs do not even share a connection, and/or a model learns means and variances individually for each element in a target vector $\boldsymbol{y}_i$.

A PNN is trained by using an appropriate loss function. Assuming normally distributed errors for the observed value of the scalar response variable, the weights are optimized by minimizing the negative log-likelihood (NLL) (Nix and Weigend, 1994):

$$\mathcal{L}_{\text{NLL}}(y_{1\ldots m}, \widehat{y}_{1\ldots m}) = \frac{1}{m}\sum_{i}^{m} \frac{\|y_i - \hat{y}_i\|^2}{2\hat{\sigma}_i^2} + \frac{1}{2}\log \hat{\sigma}_i^2 \,. \tag{3.1}$$

For numerical stability, the network is optimized to predict the log variance $s_i := \log \hat{\sigma}_i^2$ (Kendall and Gal, 2017). The loss function used in training becomes:

$$\mathcal{L}_{\text{PNN}}(y_{1\ldots m}, \widehat{y}_{1\ldots m}) = \frac{1}{m}\sum_{i}^{m} \frac{\|y_i - \hat{y}_i\|^2}{2\exp(s_i)} + \frac{1}{2}s_i \,. \tag{3.2}$$

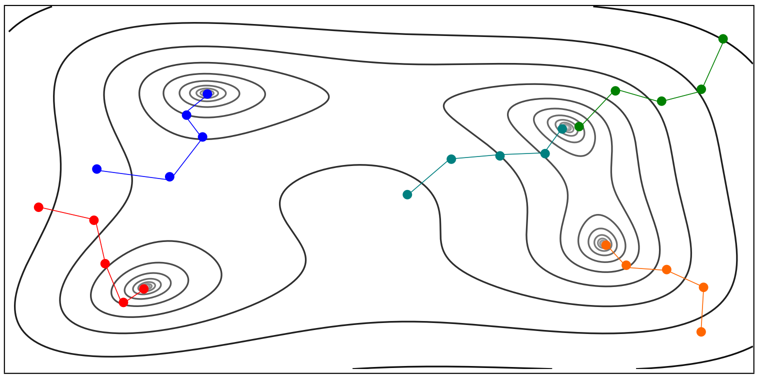


Figure 3.2: Deep ensemble. Random initialization may lead to various local minima in weight space, capturing epistemic uncertainty. We show an example training scenario with five ensemble members via contour plot.

3.1.2 Deep Ensemble

A PNN, however, is only able to capture the aleatoric noise inherent in the data. Lakshminarayanan et al. (2016) propose a *deep ensemble* that further allows modeling epistemic uncertainty. Due to the random initialization of the weights, the ensemble members usually find different local minima in weight space. This idea is depicted in Figure 3.2. Note that we train the individual ensemble members without adversarial training.

We can combine M independent neural networks trained to maximize the NLL into an ensemble E. This is done by uniformly weighting the PNNs' predictions to obtain an ensemble prediction of:

$$p_E(y|\mathbf{x}) = \frac{1}{M}\sum_{m=1}^{M} p_{\mathcal{W}_m}(y|\mathbf{x}) = \frac{1}{M}\sum_{m=1}^{M} \mathcal{N}\left(\mu_{\mathcal{W}_m}(\mathbf{x}), \sigma^2_{\mathcal{W}_m}(\mathbf{x})\right) , \tag{3.3}$$

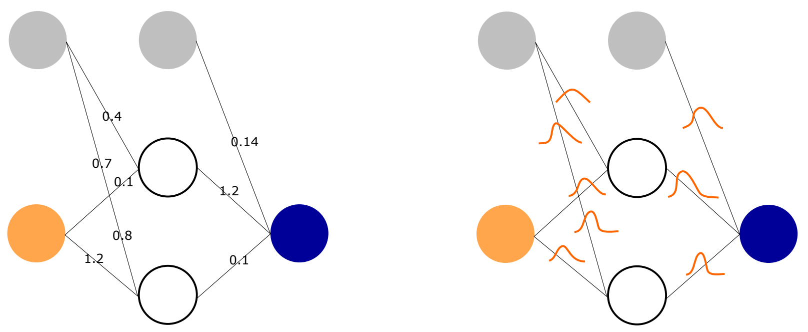
where $\{\mathcal{W}_m\}_{m=1}^{M}$ denotes the trainable parameters of all ensemble members. The ensemble's prediction of a target y_* given an input $\boldsymbol{x}_*$ is a mixture of Gaussian distributions since each PNN predictive distribution is Gaussian. We follow Lakshminarayanan et al. (2016) and further approximate the prediction of the ensemble by a Gaussian distribution:

$$p_E(y_*|\boldsymbol{x}_*) \sim \mathcal{N}\left(\mu_E(\boldsymbol{x}_*), \sigma^2_E(\boldsymbol{x}_*)\right) . \tag{3.4}$$

The mean and variance are given by:

$$\mu_E(\boldsymbol{x}_*) = \frac{1}{M}\sum_{m=1}^{M} \mu_{\mathcal{W}_m}(\boldsymbol{x}_*) \quad \text{and} \tag{3.5}$$

$$\sigma^2_E(\boldsymbol{x}_*) = \frac{1}{M}\sum_{m=1}^{M} \left(\sigma^2_{\mathcal{W}_m}(\boldsymbol{x}_*) + \mu^2_{\mathcal{W}_m}(\boldsymbol{x}_*)\right) - \mu^2_E(\boldsymbol{x}_*) \,. \tag{3.6}$$



(a) NN (b) BNN

Figure 3.3: Contrasting NNs and BNNs. In a Bayesian neural network (right) we model the weights not as scalars but as probability distributions over possible values. Input neurons are marked in orange, output neurons in blue. Hidden neurons are depicted in white, biases in gray. Based on Blundell et al. (2015).

In our experiments, we refer to this method as *PNN-E*. Training an ensemble is computationally expensive for large and complex models. Further, depending on the model architecture, the evaluation of all ensemble members might take a long time, significantly impacting the usability of this method for real-time applications.

3.2 Bayesian Neural Networks

The idea of Bayesian neural networks is to model the weights of a NN not as scalars but rather as distributions over possible values, as depicted in Figure 3.3b. Hence, we can harness the predictive power of neural networks jointly with the uncertainty modeling of Bayesian inference (Sun et al., 2019).

A BNN is defined by specifying a – e.g., Gaussian – likelihood, placing a prior distribution over the weights, and updating the probability distributions of the weights when we see data. For example, we could use Gaussian priors for all weights matrices $p(\boldsymbol{W}^{(l)}) = N(\mathbf{0}, \boldsymbol{I})$ (Gal, 2016). It is relatively easy to formulate a BNN but hard to perform posterior inference as we have to calculate integrals in the dimension of the number of weights.[1]

3.2.1 MCMC

Neal (1992) proposed to use HMC (see Section 2.2.2) to sample from the posterior in Bayesian neural networks. Concretely, Neal (1995) suggested the alteration of HMC updates for the network weights and Gibbs sampling updates for the hyperparameters. The derivatives of the loss w.r.t.

[1]Selecting a meaningful prior for the weights that corresponds to our knowledge is normally very difficult (Neal, 1995).

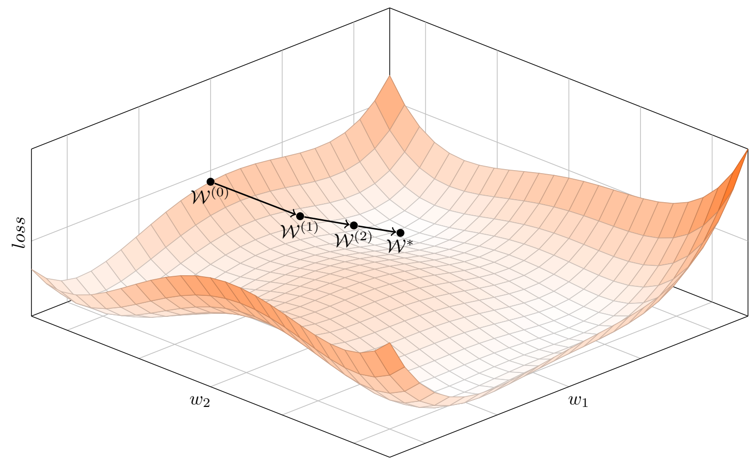


Figure 3.4: Neural network training as optimization in a simplified weight space $\mathcal{W} = (w_1, w_2)$.

the network weights necessary to perform leapfrog updates in HMC are obtained by standard backpropagation. HMC does not make assumptions about the form of the posterior distribution and is seen as *gold standard* for BNNs. However, we need to evaluate the derivatives of the loss for every leapfrog step on the full training data. This puts a substantial computational burden on the use of HMC to perform Bayesian learning for neural networks, especially for large datasets.

In recent years, a lot of work has been done to overcome the computational challenges of HMC in Bayesian NNs. The focus has been mainly on MCMC methods based on stochastic gradients, where only a subset of the full data is used to calculate the gradient. Stochastic gradient Langevin dynamics (Welling and Teh, 2011) produces samples from the posterior distribution of weights by introducing additive noise in the stochastic gradient estimator. Imagine if we add Gaussian noise to the gradients in the optimization problem in Figure 3.4, we do not convergence to a single optimum but generate samples by the successive moving in weight space.[2]

Stochastic gradient Hamiltonian Monte Carlo (Chen et al., 2014) builds on Welling and Teh (2011) and adds momentum to the Langevin dynamics to counteract the noise in the gradient estimation. Zhang et al. (2019) introduced cyclical stochastic gradient MCMC that uses a cyclical step size. The idea is that we break up the training process into partitions: in each partition, we start with a large step size, e.g., 0.1, to discover new modes of the posterior in an *exploration stage*. We decrease the step size gradually to zero to sample at these modes in a *sampling stage* and repeat the process. For the exact step size schedule see Zhang et al. (2019). This approach seems promising and efficient in both locating modes and sufficiently describing them.

However, imagine we have sampled a set of posterior weights $\mathcal{W}^{(1)}, ..., \mathcal{W}^{(J)}$, we need to evaluate an ensemble of J neural networks to obtain the posterior predictive. We likely cannot make full

[2]We can think of the NN training from Section 2.3.2 as the successive moving in weight space starting at a random point (random initialization) to minimize some loss.

use of the immense scaling capacities – achieved by specialized hardware and software – of neural networks and evaluate all ensemble members parallelly due to GPU memory limitations. Further, given a large network, we would need to store the full set of posterior weights, which can be very memory-intense. The approach of sampling all model parameters and averaging the outputs is appealing but largely unpractical. Korattikara et al. (2015) try to overcome these problems by *distilling* the knowledge into a single neural network. Concretely, the Monte Carlo ensemble acts as a teacher NN and its predictions are used as targets for a student NN that approximates the distribution of the teacher with a Gaussian distribution (Korattikara et al., 2015). However, this approach leads to Gaussian predictive densities that may not be useful if we are interested in the true – not necessarily Gaussian – predictive distributions.

3.2.2 Variational Inference

Hinton and Camp (1993) first introduced variational inference for BNNs and use a *mean-field* approximation to the true posterior with a factorized density $q_{\boldsymbol{\theta}}(\mathcal{W})$ that minimizes the KL divergence to the true posterior (Gal, 2016):

$$q_{\boldsymbol{\theta}}(\mathcal{W}) = \prod_{l=1}^{L} q_{\boldsymbol{\theta}}(\boldsymbol{W}^{(l)}) = \prod_{l=1}^{L} \prod_{i=1}^{V_l} \prod_{j=1}^{V_{l-1}+1} q_{\mu_{ij,l},\sigma_{ij,l}}(w_{ij,l}) = \prod_{i,j,l} N(w_{ij,l} | \mu_{ij,l}, \sigma^2_{ij,l}) \,, \tag{3.7}$$

where $w_{ij,l}$ is the weight connecting the i'th neuron in layer l with the j'th neuron/bias in layer $l-1$. This variational distribution doubles the number of parameters in a network and does not model any correlations between the weights. To illustrate this concept, we might view the training of BNNs as starting with a multivariate Gaussian distribution over the weights – specified by mean vector and variance matrix – and move in this larger parameter space (mean and variance per weight) to minimize a loss. However, the computation of integrals over the variational posteriors is still a difficult task. Hinton and Camp (1993) only present analytical results for a network with a single hidden layer (Graves, 2011; Gal, 2016).

Graves (2011) builds on Hinton and Camp (1993) and performs variational inference for NNs by optimizing a minimum description length (MDL; Rissanen, 1994) loss function. Graves (2011) computes the intractable (log-likelihood) loss with by Monte Carlo approximation using samples drawn independently from the factorized Gaussian approximation to maximize a lower bound on the likelihood (Hernández-Lobato and Adams, 2015). The work of Graves (2011) was the first scalable variational method for models to go beyond a single hidden layer. However, the approach did not perform well in practice (Hernández-Lobato and Adams, 2015). Blundell et al., 2015 introduced *Bayes by Backprop*, a method that in turn builds upon Graves (2011) but obtains unbiased gradients. These unbiased gradients are then used to optimize the parameters via stochastic gradient descent. In every training iteration, the mean and the variance of all weights are updated using a reparametrization trick (Kingma and Welling, 2014). Notably, this method further allows for the use of non-Gaussian priors (Blundell et al., 2015).

Probabilistic Backpropagation

Similarly to other variational methods for Bayesian neural networks, probabilistic backpropagation approximates the posterior with a product of univariate Gaussians. The training of PBP is similar to the standard training process of neural networks outlined in Section 2.3.2. First, we make a forward pass of the probabilities and calculate the loss associated with the prediction, using the marginal likelihood as the loss function. Second, we use gradient information to update the parameters of the Gaussians. PBP differs from BBB in the updating step of means and variances for the weights.

We outline the approach in more detail and again denote the data with $\mathcal{D} := (\boldsymbol{X}, \boldsymbol{y}) = \{(\boldsymbol{x_i}, y_i)\}_{i=1}^n$. Assuming that the errors for the observed value of the scalar response variable are normally distributed, the likelihood given weights $\mathcal{W}$ and noise precision $\gamma^{-1} = \sigma$ is:

$$p(\mathbf{y}|\mathcal{W}, \mathbf{X}, \gamma) = \prod_{i=1}^{n} \mathcal{N}\left(y_i | f_{\mathcal{W}}(\boldsymbol{x}_i), \gamma^{-1}\right) . \tag{3.8}$$

Hernández-Lobato and Adams (2015) further specify a Gaussian prior for each weight in $\mathcal{W}$:

$$p(\mathcal{W}|\lambda) = \prod_{l=1}^{L} \prod_{i=1}^{V_l} \prod_{j=1}^{V_{l-1}+1} \mathcal{N}\left(w_{ij,l} | 0, \lambda^{-1}\right) , \tag{3.9}$$

where $w_{ij,l}$ is the weight connecting the i'th neuron in layer l with the j'th neuron/bias in layer $l-1$. λ is again a precision parameter. Weakly informative gamma priors are chosen for the precision parameters, $p(\gamma) = \text{Gamma}(\gamma|\alpha_0^\gamma, \beta_0^\gamma), p(\lambda) = \text{Gamma}(\lambda|\alpha_0^\lambda, \beta_0^\lambda)$ with $\alpha_0^\gamma = 6$, $\beta_0^\gamma = 6$ and $\alpha_0^\lambda = 6$, $\beta_0^\lambda = 6$.

This completes the probabilistic model and the posterior distribution for $\mathcal{W}$, γ and λ is calculated using Bayes' rule:

$$p(\mathcal{W}, \gamma, \lambda \mid \mathcal{D}) = \frac{p(\mathbf{y}|\mathcal{W}, \mathbf{X}, \gamma) p(\mathcal{W}|\lambda) p(\lambda) p(\gamma)}{p(\mathbf{y}|\mathbf{X})} . \tag{3.10}$$

The Gaussian posterior predictive distribution for y_* given an input $\boldsymbol{x}_*$ then is determined by:

$$p\left(y_* | \mathbf{x}_*, \mathcal{D}\right) = \int p\left(y_* | \mathbf{x}_*, \mathcal{W}, \gamma\right) p(\mathcal{W}, \gamma, \lambda | \mathcal{D}) d\gamma d\lambda d\mathcal{W} , \tag{3.11}$$

where $p\left(y_*|\mathbf{x}_*, \mathcal{W}, \gamma\right) = N(y_*|f(\boldsymbol{x}_*), \gamma)$. However, in most cases – due to the high dimensionality – equations (3.10) and (3.11) are intractable. We have to resort to approximate Bayesian inference.

Probabilistic backpropagation approximates the exact posterior in equation (3.10) with a factored distribution.

$$q(\mathcal{W}, \gamma, \lambda) = \left[\prod_{l=1}^{L}\prod_{i=1}^{V_l}\prod_{j=1}^{V_{l-1}+1} \mathcal{N}\left(w_{ij,l}|m_{ij,l}, v_{ij,l}\right)\right] \mathrm{Gam}(\gamma|\alpha^\gamma, \beta^\gamma)\mathrm{Gam}(\lambda|\alpha^\lambda, \beta^\lambda)\,. \tag{3.12}$$

Concretely, PBP approximates the posterior with a collection of one-dimensional Gaussians that match each weight's marginal mean and variance. Denote w a single weight and $q(w) = \mathcal{N}(w|m, v)$ our current beliefs regarding w. When we see new data, we update our beliefs by Bayes' rule (Hernández-Lobato and Adams, 2015):

$$s(w) = Z^{-1} f(w)\mathcal{N}(w|m, v)\,, \tag{3.13}$$

where $f(x)$ is some likelihood function and Z is the normalization constant.

The – likely more complex – posterior beliefs $s(w)$ are approximated with a one-dimensional Gaussian. The new beliefs $\tilde{q}(w)$ minimize the KL divergence to $s(w)$:

$$\tilde{q}(w) = \mathcal{N}(w|\tilde{m}, \tilde{v}) = \operatorname*{arg\,min}_{m,v} D_{\mathrm{KL}}[s(w)||\mathcal{N}(w|m, v)]\,. \tag{3.14}$$

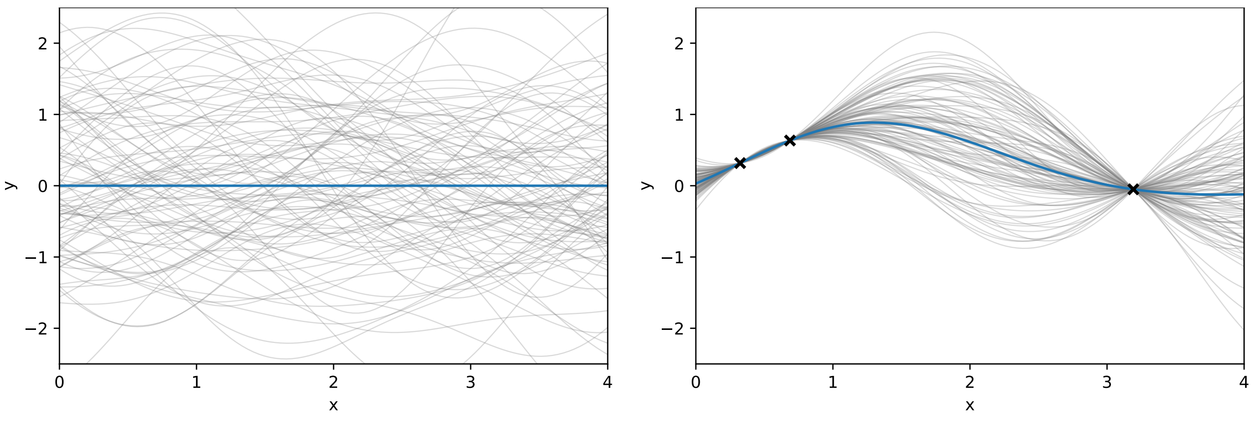
For details on the updating equations for $\tilde{m}$ and $\tilde{v}$ see Hernández-Lobato and Adams (2015) and Minka (2001).

3.3 Gaussian Process

Neal (1995) has shown that a single-layer BNN with a Gaussian prior for the weights approximates a Gaussian process when the width of the network tends to infinity. This result has also been shown to hold for deep BNNs (Lee et al., 2017). We utilize this relationship between BNNs and GPs in Section 4.1, where we benchmark some models presented in this section. That is, we use the prediction of a Gaussian process as a reference benchmark for simple toy problems.

A GP can be described as a distribution over functions (Rasmussen and Williams, 2006). Essentially, a GP is a Bayesian approach that places a prior over the distribution of functions and is formalized by (Murphy, 2012):

$$f(\boldsymbol{x}) \sim GP(m(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x}'))\,, \tag{3.15}$$



(a) Samples from the prior (b) Samples from the posterior

Figure 3.5: Gaussian process as a distribution over functions. Grey lines indicate samples from the prior in (a) or the posterior in (b), the blue line shows the mean prediction. Training samples in (b) are marked as black crosses.

where $m(\boldsymbol{x})$ is a mean function and $k(\boldsymbol{x}, \boldsymbol{x}')$ a covariance function or *kernel*:

$$m(\boldsymbol{x}) = \mathbb{E}[f(\boldsymbol{x})]\,, \tag{3.16}$$

$$k(\boldsymbol{x}, \boldsymbol{x}') = \mathbb{E}[(f(\boldsymbol{x}) - m(\boldsymbol{x}))(f(\boldsymbol{x}') - m(\boldsymbol{x}'))^T]\,. \tag{3.17}$$

The kernel here largely encodes our prior knowledge. Typical kernels include the squared exponential kernel, the exponential kernel, or the Matérn kernel.

For a finite number of points $\boldsymbol{X}$, which is of size $n \times d$, a GP defines a joint Gaussian distribution of functions (Murphy, 2012):

$$p(\boldsymbol{f}|\boldsymbol{X}) = \mathcal{N}(\boldsymbol{f}|\boldsymbol{\mu}, \boldsymbol{K}) \tag{3.18}$$

with mean $\boldsymbol{\mu} = (m(\boldsymbol{x_1}), ..., m(\boldsymbol{x_n}))$ and positive definite covariance matrix $\boldsymbol{K}$ with elements $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$.

When we observe data, this prior distribution is transformed to a posterior distribution (Murphy, 2012). We illustrate this in Figure 3.5. Consider a test set $\boldsymbol{X}_*$ of shape $n_* \times d$. We want to predict the function outputs $\boldsymbol{f}_*$. The function outputs of the training data $\boldsymbol{y}$ – which are corrupted with some Gaussian noise ϵ with variance σ_y^2 – and $\boldsymbol{f}_*$ are distributed jointly:

$$\begin{pmatrix} \boldsymbol{y} \\ \boldsymbol{f}_* \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{0} \end{pmatrix}, \begin{pmatrix} \boldsymbol{K}_y & \boldsymbol{K}_* \\ \boldsymbol{K}_*^T & \boldsymbol{K}_{**} \end{pmatrix}\right), \tag{3.19}$$

where $\boldsymbol{K}_y = \boldsymbol{K} + \sigma_y^2 I_n$ is $n \times n$, $\boldsymbol{K}_* = k(\boldsymbol{X}, \boldsymbol{X}_*)$ is $n \times n_*$, $\boldsymbol{K}_{**} = k(\boldsymbol{X}_*, \boldsymbol{X}_*)$ is $n_* \times n_*$ For simplicity, we assume zero means. If we do not know the noise variance σ_y^2, we can place a hyperprior $p(\sigma_y^2)$ and sample via MCMC.

The posterior predictive is obtained as (Murphy, 2012):

$$p(\boldsymbol{f}_*|\boldsymbol{X}_*, \boldsymbol{X}, \boldsymbol{y}) \sim \mathcal{N}(\boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*)\,, \tag{3.20}$$

where

$$\boldsymbol{\mu}_* = \boldsymbol{K}_*^T \boldsymbol{K}_y^{-1} \boldsymbol{y}\,, \tag{3.21}$$

$$\boldsymbol{\Sigma}_* = \boldsymbol{K}_{**} - \boldsymbol{K}_*^T \boldsymbol{K}_y^{-1} \boldsymbol{K}_*\,. \tag{3.22}$$

To compute the posterior predictive, we have to invert the $n \times n$ matrix $\boldsymbol{K}_y$, which is computationally infeasible for large datasets. For a more in-depth introduction to Gaussian processes, see Rasmussen and Williams (2006).

Relationship to BNNs Rasmussen and Williams (2006) derive an analytical kernel for a GP that corresponds to a single layer NN with bounded activation functions in the hidden layer such as tanh or sigmoid. This result follows from the central limit theorem (Rasmussen and Williams, 2006). Recently, Lee et al. (2017) have developed a method that recursively computes a covariance function so that a GP with zero mean function is equal to an infinitely wide BNN that uses an i.i.d. Gaussian prior for each layer. Concretely, for a layer with V_l inputs we choose i.i.d. Gaussian priors for the weights $N(0, \sigma_w^2/V_l)$ with a constant σ_w^2 and a prior $N(0, \sigma_b^2)$ for the bias. Notably, with the recursive equations from Lee et al. (2017), we can generate a GP that corresponds to a neural network with ReLU activation functions.

3.4 Monte Carlo Dropout

As presented in Section 2.3.3, dropout is a regularization technique that prevents a neural network from overfitting. Applied to the outputs of a layer, dropout randomly samples binary variables and masks the output of the layer. That is, it sets the output of some neurons of the layer to zero. Typically, dropout is applied only during the training process. However, when activated at test time, it yields a method for uncertainty quantification in neural networks referred to as *Monte Carlo dropout* (Gal and Ghahramani, 2016). MC dropout does not make any assumptions about the model architecture and can be used with all variants of dropout (Gal and Ghahramani, 2016).

Mathematically, MC dropout minimizes the KL divergence to the posterior of a deep Gaussian process. The approximate distribution is a mixture of two Gaussians with small variances and one mean set to zero (Kendall and Gal, 2017). MC dropout generates samples from the approximate posterior by applying dropout during test time (Kendall and Gal, 2017). This is easily implemented in standard neural network software such as TensorFlow (Abadi et al., 2015) and PyTorch (Paszke et al., 2019). Once trained, we use weights $\mathcal{W}$ that characterize the neural network for all draws.

We use the approach outlined by Kendall and Gal (2017) to combine both heteroscedastic aleatoric uncertainty and epistemic uncertainty in one model. That is, we apply dropout during train *and* test time before every weight connection in a NN that is trained to predict mean and log-variance (PNN, see Section 3.1.1) in order to minimize the negative log-likelihood defined in equation (3.2). For inference, we compute T forward passes through the network to obtain a set $\{\hat{y}_t, \hat{\sigma}_t^2\}_{t=1}^T$ of T sampled outputs given an input $\boldsymbol{x}_*$. The predictive density of an output y_* is then a mixture of T Gaussians with equal weights.

For computational ease, we approximate this predictive density with a univariate Gaussian distribution $p(y_*|\boldsymbol{x}_*) \sim N(\mu_*, \sigma_*^2)$, where mean μ_* and variance σ_*^2 are given by (Kendall and Gal, 2017):

$$\mu_* = \frac{1}{T}\sum_{t=1}^{T} \hat{y}_t \quad \text{and} \tag{3.23}$$

$$\sigma_*^2 = \frac{1}{T}\sum_{t=1}^{T} \hat{y}_t^2 - \left(\frac{1}{T}\sum_{t=1}^{T} \hat{y}_t\right)^2 + \frac{1}{T}\sum_{t=1}^{T} \hat{\sigma}_t^2 \,. \tag{3.24}$$

Here $\frac{1}{T}\sum_{t=1}^{T} \hat{y}_t^2 - (\frac{1}{T}\sum_{t=1}^{T} \hat{y}_t)^2$ captures the model uncertainty and $\frac{1}{T}\sum_{t=1}^{T} \hat{\sigma}_t^2$ the aleatoric uncertainty in the data. One could also use the mixture of T Gaussians directly or approximate this by a mixture of $K \ll T$ Gaussians. However, this increases the computational cost.

Contrary to the MCMC approach of performing predictive inference by ensembling T NNs with weights $\mathcal{W}^{(1)}, ..., \mathcal{W}^{(T)}$, we can parallelize the computation of the posterior predictive. Computing T forward passes can be relatively cheap if dropout is only applied in the last layers of a NN. However, it is computationally expensive if we sample the whole architecture, e.g., generating 100 Monte Carlo samples from the approximate posterior can lead to a worst-case 100-fold increase in inference time (Kendall and Gal, 2017). This seriously restricts the applicability of MC dropout for real-time applications that involve complex and large neural networks. Recently, Brach et al. (2020) proposed a *single shot* adaption to MC dropout that approximates the first two moments of MC dropout in each layer. Their results indicate that this approach might facilitate the use of MC dropout in real-time applications (Brach et al., 2020).

3.5 Last-Layer Bayesian Inference

To overcome the computational burden of BNNs, we can perform Bayesian inference in the final layer of a NN only. In a regression context, this is done by using the output of the last hidden layer as basis functions for Bayesian linear regression (Snoek et al., 2015; Riquelme et al., 2018; Ober and Rasmussen, 2019). We outline a heteroscedastic variant of the standard neural linear model and an adaption that ensures marginal calibration.

3.5.1 Neural Linear Model

Suppose we have trained a neural network that minimizes the NLL as seen in Section 3.1.1. The neural network outputs a Gaussian distribution – a tuple of predictive mean and variance $(\hat{y}_*, \hat{\sigma}_*^2)$ – for an input $\boldsymbol{x}_*$. Further, if we have trained the model with a linear activation function for the output layer, it is possible to use the last hidden layer output as basis functions for a linear regression:

$$y_i = f_{\widetilde{\mathcal{W}}}(\boldsymbol{x}_i) + \epsilon_i = \phi_{\widetilde{\mathcal{W}}}(\boldsymbol{x}_i)^T \boldsymbol{\beta} + \epsilon_i \,, \tag{3.25}$$

where $\widetilde{\mathcal{W}} = \left\{\boldsymbol{W}^{(l)}\right\}_{l=1}^{L-1}$ denotes the weights of all layers of the neural network but the last, and $\phi_{\widetilde{\mathcal{W}}}(\cdot)$ denotes the vector of basis functions obtained from the last hidden layer.

We can obtain a vector of basis functions for every input $\boldsymbol{x}_i$ and perform Bayesian linear regression as outlined in Section 2.2. Let $\boldsymbol{Z}_{\widetilde{\mathcal{W}}}(\boldsymbol{X}) = [\phi_{\widetilde{\mathcal{W}}}(\boldsymbol{x}_1), ..., \phi_{\widetilde{\mathcal{W}}}(\boldsymbol{x}_n)]^T$ denote the $n \times q$ data matrix obtained by feed-forwarding all inputs through a neural network and retrieving the outputs of the last hidden layer with q neurons. We follow Moberg et al. (2019) and use the predicted variances from the NN as covariance matrix for the likelihood:

$$p\left(\mathbf{y}|\mathbf{Z}, \boldsymbol{\beta}\right) = \mathcal{N}\left(\mathbf{y}|\mathbf{Z}\boldsymbol{\beta}, \boldsymbol{\Sigma}\right) \,, \tag{3.26}$$

where

$$\boldsymbol{\Sigma} = \begin{bmatrix} \hat{\sigma}_1^2 & & \\ & \ddots & \\ & & \hat{\sigma}_n^2 \end{bmatrix} . \tag{3.27}$$

Following Moberg et al. (2019), we use a Gaussian prior $\mathcal{N}\left(\boldsymbol{\beta}|\boldsymbol{\beta}_0, \mathbf{V}_0\right)$ with mean $\beta_0 = \mathbf{0}$ and covariance $\mathbf{V}_0 = \tau^2 \boldsymbol{I}$. As we are using a conjugate prior, we can obtain the posterior in closed form:

$$p\left(\boldsymbol{\beta}|\mathbf{Z}, \mathbf{y}, \boldsymbol{\Sigma}\right) \propto \mathcal{N}\left(\mathbf{y}|\mathbf{Z}\boldsymbol{\beta}, \boldsymbol{\Sigma}\right) \mathcal{N}\left(\boldsymbol{\beta}|\boldsymbol{\beta}_0, \mathbf{V}_0\right) = \mathcal{N}\left(\boldsymbol{\beta}|\boldsymbol{\beta}_n, \mathbf{V}_n\right) \,, \tag{3.28}$$

where mean and variance are obtained by applying Bayes' rule for linear Gaussian systems (Murphy, 2012):

$$\mathbf{V}_n = \left(\mathbf{V}_0^{-1} + \mathbf{Z}^T\boldsymbol{\Sigma}^{-1}\mathbf{Z}\right)^{-1} \quad \text{and} \tag{3.29}$$

$$\boldsymbol{\beta}_n = \mathbf{V}_n\mathbf{V}_0^{-1}\boldsymbol{\beta}_0 + \mathbf{V}_n\mathbf{Z}^T\boldsymbol{\Sigma}^{-1}\mathbf{y}\,. \tag{3.30}$$

The posterior predictive of the response y_* for a test input $\boldsymbol{x}_*$ is given by:

$$p\left(y_*|\mathbf{x}_*, \mathcal{D}, \boldsymbol{\Sigma}\right) = \mathcal{N}\left(y_*|\boldsymbol{\beta}_n^T\phi_{\widetilde{\mathcal{W}}}(\boldsymbol{x}_*), \hat{\sigma}_*^2 + \phi_{\widetilde{\mathcal{W}}}(\boldsymbol{x}_*)^T\mathbf{V}_n\phi_{\widetilde{\mathcal{W}}}(\boldsymbol{x}_*)\right)\,. \tag{3.31}$$

Note that the predictive variance is the sum of the NN's predicted aleatoric uncertainty $\hat{\sigma}_*^2$ and a term that depends on the variance in the model parameters, V_N (Murphy, 2012).

This approach is an easily implemented and computationally cheap method that builds on a probabilistic neural network and that can capture both aleatoric and epistemic uncertainty. In our experiments in Chapter 4 we refer to this method as *NLM*. As with PNNs, we can combine multiple NLMs into an ensemble. We refer to the ensemble model as *NLM-E*. NLM-E suffers from the same drawbacks as a deep ensemble (Section 3.1.2). I.e., training and prediction are costly, inhibiting the use for many applications.

3.5.2 Marginally Calibrated Deep Distributional Regression

Klein et al. (2021) built upon Klein and Smith (2019) and Smith and Klein (2021) and extended the marginally calibrated distributional regression method to neural networks. Similarly to the neural linear model presented above, the method uses the last hidden layer outputs of a NN as basis functions for regression. From the feature vector outputs of the last hidden layer, an implicit copula process is obtained. Combined with a non-parametric estimation of the marginal distribution of the response, the method ensures marginal calibration. The approach yields a scalable regression method with predictive distributions that are flexible functions of the input and can be readily computed (Klein et al., 2021).

Loosely speaking, a copula is a function that *couples* marginal probability distributions of random variables together to a joint probability distribution. It allows for the separate modeling of the marginal distributions and the dependence structure of random variables (Nelsen, 2006). Consider a random vector $(Y_1, Y_2, ..., Y_n)$ of continuous responses with cumulative distribution functions $F_i(y) = \Pr[Y_i \leq y]$. We apply a probability integral transform to each random variable:

$$(U_1, ..., U_n) = (F_1(Y_1), ..., F_n(Y_n))\,, \tag{3.32}$$

where the marginals of the random vector $(U_1, ..., U_n)$ are uniformly distributed on $[0,1]$. A copula $C : [0,1]^n \rightarrow [0,1]$ is a joint cumulative distribution function on the unit cube (Nelsen, 2006):

$$C(u_1, u_2, ..., u_n) = \Pr[U_1 \leq u_1, ..., U_n \leq u_n]\,. \tag{3.33}$$

Denote the realizations as $\boldsymbol{y} = (y_1, ..., y_n)^T$ and $\boldsymbol{X} = (\boldsymbol{x}_1, ..., \boldsymbol{x}_n)^T$ the corresponding features. The joint density of the distribution $\boldsymbol{y}|\boldsymbol{X}$ can be expressed by (Sklar, 1959):

$$p(\boldsymbol{y}|\boldsymbol{X}) = c^{\dagger}\left(F\left(y_1|\boldsymbol{x}_1\right), \ldots, F\left(y_n|\boldsymbol{x}_n\right)|\boldsymbol{X}\right) \prod_{i=1}^{n} p\left(y_i|\boldsymbol{x}_i\right)\,, \tag{3.34}$$

where $c^{\dagger}(\boldsymbol{u}|\boldsymbol{X})$ is a n-dimensional copula density and $F(y_i|\boldsymbol{x}_i)$ the distribution function of $Y_i|\boldsymbol{x}_i$. The key idea is to replace the unknown $c^{\dagger}$ with an implicit copula of a NN regression $c_{NN}(\boldsymbol{u}|\boldsymbol{X}, \boldsymbol{\theta})$ and calibrate the distributions of the response to its invariant margin (Klein et al., 2021; Smith and Klein, 2021):

$$p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}) = c_{\mathrm{NN}}\left(F_Y\left(y_1|\boldsymbol{x}_1\right), \ldots, F_Y\left(y_n|\boldsymbol{x}_n\right)|\boldsymbol{X}, \boldsymbol{\theta}\right) \prod_{i=1}^{n} p_Y\left(y_i\right)\,, \tag{3.35}$$

where F_Y is estimated non-parametrically. We present two methods to estimate F_Y in Section 4.2.2.

We transform the response values by their estimated inverse marginal cumulative distribution function (CDF) $\tilde{\boldsymbol{z}} = \Phi^{-1}(\hat{F}_Y(\boldsymbol{y}))$ – depicted in Figure 3.6 – and train a neural network with the transformed responses as targets and a linear activation function for the output. As with the neural linear model, we use the last hidden layer output as basis functions for a linear regression:

$$\tilde{z}_i = \phi_{\widetilde{\mathcal{W}}}(\boldsymbol{x}_i)^T \boldsymbol{\beta} + \epsilon_i, \epsilon_i \sim N(0, \sigma^2) \text{ i.i.d.}\,, \tag{3.36}$$

or in matrix notation:

$$\tilde{\boldsymbol{z}} = \boldsymbol{B}_{\widetilde{\mathcal{W}}}(\boldsymbol{X})\boldsymbol{\beta} + \boldsymbol{\epsilon}, \boldsymbol{\epsilon} \sim N(0, \sigma^2 I)\,, \tag{3.37}$$

where $\tilde{\boldsymbol{z}} = (\tilde{z_1}, \tilde{z_2}, ..., \tilde{z_n})^T$ is a vector of n pseudo-responses, $\widetilde{\mathcal{W}} = \left\{\boldsymbol{W}^{(l)}\right\}_{l=1}^{L-1}$ denotes the weights of all layers of the neural network but the last, $\phi_{\widetilde{\mathcal{W}}}(\cdot)$ is the vector of basis functions from the last hidden layer, and $\boldsymbol{B}_{\widetilde{\mathcal{W}}}(\boldsymbol{X}) = [\phi_{\widetilde{\mathcal{W}}}(\boldsymbol{x}_1), \phi_{\widetilde{\mathcal{W}}}(\boldsymbol{x}_2), ..., \phi_{\widetilde{\mathcal{W}}}(\boldsymbol{x}_n)]^T$ is a $n \times q$ data matrix. The regression in equation (3.37) is modeled without an intercept β_0 as an intercept term is not identified in the copula (Klein et al., 2021).

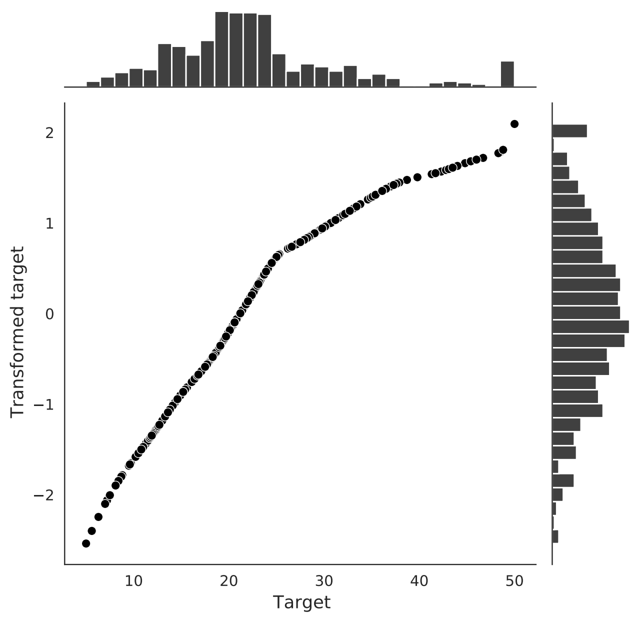


Figure 3.6: Transforming the response by its inverse marginal CDF. The x-axis shows the target variables $\boldsymbol{y}$, the y-axis the transformed targets $\tilde{\boldsymbol{z}} = \Phi^{-1}(\hat{F}_Y(\boldsymbol{y}))$. Data: Boston Housing.

We use a Gaussian prior for the basis coefficient vector:

$$\boldsymbol{\beta}|\boldsymbol{\theta}, \sigma^2 \sim N(0, \sigma^2 P(\boldsymbol{\theta})^{-1}), \tag{3.38}$$

where $P(\boldsymbol{\theta})$ denotes a sparse precision matrix that depends on regularization parameters $\boldsymbol{\theta}$. We obtain a distribution of the pseudo-vector $\tilde{\boldsymbol{z}}$ (Klein et al., 2021):

$$\tilde{\boldsymbol{z}}|\boldsymbol{X}, \sigma^2, \boldsymbol{\theta} \sim N\left(0, \sigma^2(I + \boldsymbol{B}_{\widetilde{\mathcal{W}}}(\boldsymbol{X})P(\boldsymbol{\theta})^{-1}\boldsymbol{B}_{\widetilde{\mathcal{W}}}(\boldsymbol{X})^T)\right), \tag{3.39}$$

where $\boldsymbol{\beta}$ is integrated out. Let $\boldsymbol{z} = (z_1, z_2, ..., z_n) = \sigma^{-1}S(\boldsymbol{X}, \boldsymbol{\theta})\tilde{\boldsymbol{z}}$ denote the standardized vector of pseudo-responses, where $S(\boldsymbol{X}, \boldsymbol{\theta})$ is a $n \times n$ diagonal scaling matrix with diagonal elements $s_i = \left(1 + \phi_{\widetilde{\mathcal{W}}}(\boldsymbol{x}_i)^T P(\boldsymbol{\theta})^{-1}\phi_{\widetilde{\mathcal{W}}}(\boldsymbol{x}_i)\right)^{-1/2}$.

The density of the *implicit* Gaussian copula (Klein et al., 2021) is:

$$c_{NN}(\boldsymbol{u}|\boldsymbol{X}, \boldsymbol{\theta}) = \frac{\phi_n(\boldsymbol{v}; \boldsymbol{0}, R(\boldsymbol{X}, \boldsymbol{\theta}))}{\prod_{i=1}^n \phi_1(v_i)}, \tag{3.40}$$

where

$$R(\boldsymbol{X}, \boldsymbol{\theta}) = S(\boldsymbol{X}, \boldsymbol{\theta})\left(I + \boldsymbol{B}_{\widetilde{\mathcal{W}}}(\boldsymbol{X})P(\boldsymbol{\theta})^{-1}\boldsymbol{B}_{\widetilde{\mathcal{W}}}(\boldsymbol{X})^T\right)S(\boldsymbol{X}, \boldsymbol{\theta}), \tag{3.41}$$

$v_i = \Phi_1^{-1}(u_i)$, $\boldsymbol{v} = (v_1, ..., v_n)^T$, ϕ_1, and ϕ_n denote the densities of $N(0,1)$ and $N_n(\boldsymbol{0}, R)$, respectively. Note that the copula density does not depend on σ, and we thus set $\sigma = 1$. The Gaussian

```
Algorithm 3: Marginally calibrated deep distributional regression
1 Estimate the marginal CDF of the responses $F_Y$ non-parametrically.
2 Fit a NN with targets transformed by the marginal CDF $\tilde{\boldsymbol{z}} = \Phi^{-1}(\hat{F}_Y(\boldsymbol{y}))$ and with linear
    activation function for the output layer. Obtain the data matrix $\boldsymbol{B}_{\widetilde{\mathcal{W}}}(\boldsymbol{X})$ from the last
    hidden layer by feed-forwarding all training inputs $\boldsymbol{X}$.
3 Use MCMC to compute the augmented posterior distribution $p(\boldsymbol{\beta}, \boldsymbol{\theta}|\boldsymbol{y})$.
```

copula density c_{NN} is linked to $c^{\dagger}$ by (Klein et al., 2021):

$$c^{\dagger}(\boldsymbol{u}|\boldsymbol{X}) = \int c_{NN}(\boldsymbol{u}|\boldsymbol{X}, \boldsymbol{\theta}) p(\boldsymbol{\theta}) d\boldsymbol{\theta}\,, \tag{3.42}$$

where the obtained copula $c^{\dagger}$ is not a Gaussian copula (Klein and Smith, 2019).

We explore two forms for the prior in equation (3.38). First, we use a *ridge prior* that shrinks the coefficients *globally*:

$$\beta_j|\tau^2 \sim N(0, \tau^2)\,, \tag{3.43}$$

so that $\boldsymbol{\theta} = \left\{\tau^2\right\}$. We choose the scale-dependent prior of Klein and Kneib (2016) as prior for τ^2:

$$p\left(\tau^2\right) \propto \left(\tau^2/r\right)^{-1/2} \exp\left(-\left(\tau^2/r\right)^{1/2}\right)\,, \tag{3.44}$$

with scale parameter r set to 2.5.

Second, we can likely identify a richer dependency structure by shrinking the coefficients *globally-locally*. That is, we use a *horseshoe prior* for the coefficients in the following form:

$$\beta_j|\lambda_j \sim N(0, \lambda_j^2)\,, \tag{3.45}$$

$$\lambda_j|\tau \sim \text{Half-Cauchy}(0, \tau)\,, \tag{3.46}$$

$$\tau \sim \text{Half-Cauchy}(0, 1)\,, \tag{3.47}$$

and thus have the copula parameters $\boldsymbol{\theta} = \left\{(\lambda_1, \lambda_2, ..., \lambda_j)^T, \tau^2\right\}$.

To compute the copula density in equation (3.40), we need to invert the $n \times n$ matrix $R(\boldsymbol{X}, \boldsymbol{\theta})$, which is computationally expensive and infeasible for large n. Klein and Smith (2019) propose a MCMC method to generate J draws $\left\{(\boldsymbol{\beta}^{(1)}, \boldsymbol{\theta}^{(1)}), ..., (\boldsymbol{\beta}^{(J)}, \boldsymbol{\theta}^{(J)})\right\}$ from the augmented posterior distribution $p(\boldsymbol{\beta}, \boldsymbol{\theta}|\boldsymbol{y})$. The posterior predictive $p(y_*|\boldsymbol{x_*})$ of a response Y_* given an input $\boldsymbol{x}_*$ is given by:

$$p\left(y_*|\boldsymbol{x}_*, \boldsymbol{X}, \boldsymbol{y}\right) = \int p\left(y_*|\boldsymbol{x}_*, \boldsymbol{\beta}, \boldsymbol{\theta}\right) p(\boldsymbol{\beta}, \boldsymbol{\theta}|\boldsymbol{X}, \boldsymbol{y}) \mathrm{d}(\boldsymbol{\beta}, \boldsymbol{\theta})\,. \tag{3.48}$$

In practice, we can estimate equation (3.48) based on the MCMC samples by (Klein and Smith, 2019; Klein et al., 2021):

$$\hat{p}_*\left(y_*|\boldsymbol{x}_*\right) = \frac{\hat{p}_Y\left(y_*\right)}{\phi_1\left(\Phi_1^{-1}\left(\hat{F}_Y\left(y_*\right)\right)\right)} \frac{1}{\hat{s_*}}\phi_1\left(\frac{\Phi_1^{-1}\left(\hat{F}_Y\left(y_*\right)\right) - \hat{s}_*\hat{f}_{\widetilde{\mathcal{W}}}\left(\boldsymbol{x}_*\right)}{\hat{s_*}}\right), \tag{3.49}$$

where $\hat{s_*} = \frac{1}{J}\sum_{j=1}^{J} s_*^{(j)}$, $s_*^{(j)} = (1 + \phi_{\widetilde{\mathcal{W}}}(\boldsymbol{x}_*)P(\boldsymbol{\theta}^{(j)})^{-1}\phi_{\widetilde{\mathcal{W}}}(\boldsymbol{x}_*)^{-1/2})$, $\hat{f}_{\widetilde{\mathcal{W}}}(x_*) = \phi_{\widetilde{\mathcal{W}}}(x_*)^T\hat{\boldsymbol{\beta}}$ and $\hat{\boldsymbol{\beta}} = \frac{1}{J}\sum_{j=1}^{J}\boldsymbol{\beta}^{(j)}$. The posterior predictive density $\hat{p}_*\left(y_*|\boldsymbol{x}_*\right)$ is the marginal density $\hat{p}_Y$ transformed by the non-linear functions $\hat{f}_{\mathcal{W}}(\boldsymbol{x}_*)$ and $\hat{s_*}$. This leads to predictive densities that are flexible functions of the input $\boldsymbol{x}_*$. We refer to Klein and Smith (2019) for the details on the MCMC sampling scheme and Klein et al. (2021) for the derivation of the posterior predictive.

The full approach is summarized in Algorithm 3. We refer to the marginally calibrated deep distributional learning method as *DNNC-R* for the ridge prior and as *DNNC-HS* for the horseshoe prior.

3.6 Summary of Implemented Methods

We select the following methods for our benchmarking experiments in the next chapter:

- A **PNN** is quickly computed and a useful comparison method since it only captures aleatoric uncertainty and thus allows us to study the importance of epistemic uncertainty quantification.
- The deep ensemble **(PNN-E)** is a strong benchmark (Lakshminarayanan et al., 2016). However, as we train multiple NNs, there is a large computational overhead during training and at test time due to GPU memory constraints.
- **MC dropout** is a popular method in the literature. A NN with dropout takes significantly longer to converge (Gal and Ghahramani, 2016) and at test time, we sample e.g., 1,000 times from the approximate posterior, which is very costly for large networks.
- **PBP** produces fast variational results and captures both aleatoric and epistemic uncertainty.
- **NLM (or an ensemble of NLMs)** performs inference in the last layer. It is a simple and computationally feasible method at the cost of disregarding uncertainty in preceding layers. Naturally, NLM-E has the same limitations as PNN-E.
- **DNNC** is related to the NLM but ensures marginal calibration and its predictive density is a flexible function of the data.

Table 3.1 summarizes the models qualitatively w.r.t. the computational cost and the predictive distributions. Note that the predictive density of ensemble methods (MC dropout, PNN-E and NLM-E) is a mixture of Gaussians approximated by a Gaussian (Kendall and Gal, 2017; Lakshminarayanan et al., 2016; Moberg et al., 2019). DNNC is thus the only method that does not output Gaussian predictive distributions.

Abbreviation	Method (Section)	Computational Cost		Predictive Density
		Training	Prediction	
PNN	Probabilistic neural network (3.1.1)	Low	Low	Gaussian
PNN-E	Deep ensemble, ensemble of PNNs (3.1.2)	High	Moderate to high	Gaussian ≈ mixture of M (e.g., 5) Gaussians
MC dropout	Monte Carlo dropout (3.4)	High	High	Gaussian ≈ mixture of T (e.g., 1,000) Gaussians
PBP	Probabilistic backpropagation (3.2.2)	Low to moderate	Low	Gaussian
NLM	Neural linear model (3.5.1)	Moderate	Low	Gaussian
NLM-E	Ensemble of NLMs (3.5.1)	High	Moderate to high	Gaussian ≈ mixture of M (e.g., 5) Gaussians
DNNC	Marginally calibrated deep distributional regression (3.5.2)	Moderate	Low	Flexible functions

Table 3.1: Summary of implemented models: computational cost and predictive distributions.

4 Experiments and Results

This chapter first studies the basic characteristics of the implemented models on synthetic toy regression problems. Subsequently, we benchmark the models on UCI regression datasets (Dua and Graff, 2017) to assess their calibration and predictive performance. Finally, we compare DNNC-R to a slightly modified version trained with standardized inputs and outputs.

4.1 Toy Data

As in Foong et al. (2019a), we utilize the relationship between BNNs and GPs (Section 3.3) and compare the predictive mean and variance of some methods described in Chapter 3 with those of a *ground truth* Gaussian process with ReLU-kernel. Input and output variables are standardized to have zero-mean and unit variance.[1] At test time, we rescale the outcome by mean and standard deviation of the training set. All methods share the same basic architecture and have one hidden layer with 50 neurons. We train the (underlying) neural network for 40 epochs with the Adam optimizer and a learning rate of 0.01. See Appendix A.1 for the selection of hyperparameters for all methods.

4.1.1 1D Input Space

To evaluate the methods' predictive performance on a toy data set, we follow the experimental setup of Hernández-Lobato and Adams (2015). We set up a homoscedastic regression problem with scalar inputs x_i and scalar responses y_i:

$$y_i = x_i^3 + \epsilon_i, \text{ where } \epsilon_i \sim \mathcal{N}(0, 3^2) \text{ i.i.d.} \tag{4.1}$$

We generate the training data by sampling 20 scalar inputs x_i uniformly at random from the interval $[-4, 4]$ and computing corresponding targets corrupted with homoscedastic noise. We then assess the predictive performance of the methods on a test interval $[-6, 6]$.

[1]For DNNC, we transform the targets by their empirical marginal distribution $\hat{F_Y}$. See Section 3.5.2 for details.

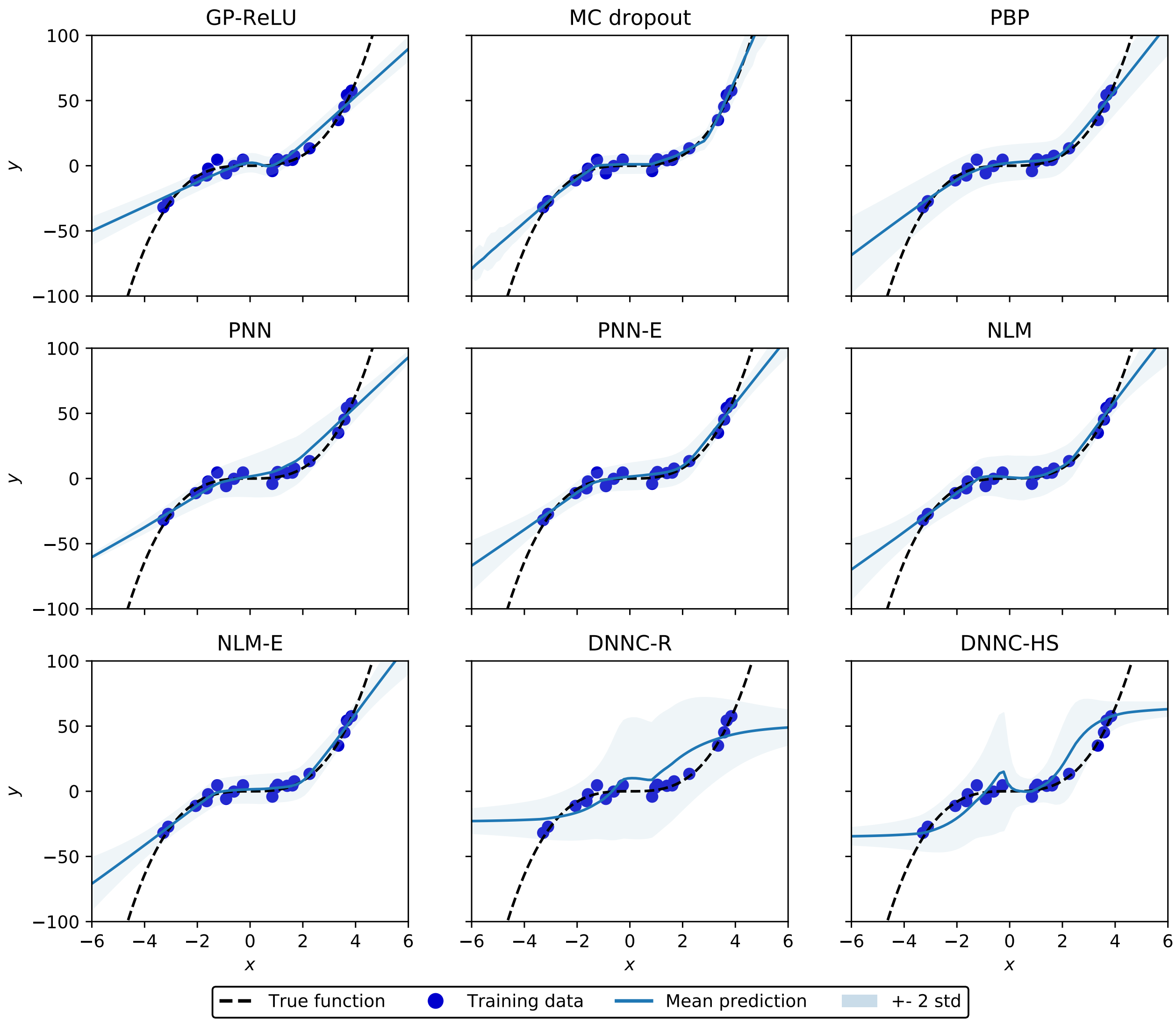


Figure 4.1: Results for the Toy1D dataset. Blue dots mark the training data with homoscedastic noise and the dashed black line indicates the true function. The predictive mean is shown as a turquoise line. The transparent blue area depicts the credible intervals of ± 2 standard deviations from the mean.

The results are depicted in Figure 4.1. Ground truth GP-ReLU (top left, $\sigma_w = 5$, $\sigma_b = 1$) is able to capture the true relationship between inputs and outputs for most of the training data range and assumes a linear relationship where it has not seen data. The predictive uncertainty is reasonably low in the training range (driven by the noise in the data) and increases the further we predict outside the training range. This fits our intuition as predictive (epistemic) uncertainty should increase in areas where the model has not seen data. MC dropout (top-center) comes relatively close to GP-ReLU and correctly predicts increasing epistemic uncertainty outside the training data. PBP (top-right) correctly captures the true function and has a higher aleatoric uncertainty for the full data range.

Some methods are based on a probabilistic neural network that minimizes the NLL. We show the predictive results of a PNN in the center-left. The PNN captures the true function for a large fraction of the data range and predicts varying aleatoric variance. For regions outside the training data, we find low predictive uncertainty. The deep ensemble (PNN-E; center) and the neural linear model (NLM; center-right) improve upon the PNN as they model epistemic uncertainty. Significantly, the models predict higher uncertainty for regions where they have not seen data. The predictions of an ensemble of NLMs (NLM-E; bottom-left) are similar to those of a single NLM.

DNNC-R and DNNC-HS incorporate the empirical marginal distribution $\hat{F_Y}$ and are rather unsuitable for our toy experiments. Essentially, we restrict an outcome outside our training range to be distributed as the outcomes we have trained on. This is reflected in their predictions in the bottom-middle and bottom-right of Figure 4.1. For inputs well in the training range, $x_i \in [-1.5, 1.5]$, DNNC-R and DNNC-HS output a predictive variance that resembles the variance of the empirical marginal target density $\sigma(\frac{\mathrm{d}F_y}{\mathrm{d}y})^2$. The further we predict near the boundaries of the training range, the lower the predictive uncertainty. Outside the training range, the methods predict low uncertainty and mean values similar to the boundaries of the training range.

4.1.2 2D Input Space

In a second step, we model the quality of the predictive uncertainty of all methods for a regression problem with 2-dimensional features $\boldsymbol{x}_i = (x_{i,1}, x_{i,2})$. We roughly follow Foong et al. (2019a) and set up the regression problem as:

$$y_i = 0.8x_{1,i} + 0.8x_{2,i} + \epsilon_i, \text{ where } \epsilon_i \sim \mathcal{N}(0, 0.1^2) \text{ i.i.d.} \tag{4.2}$$

where the output y_i is corrupted with relatively low noise with standard deviation $\sigma_\epsilon = 0.1$. The (clustered) inputs $\boldsymbol{X}$ are sampled from a multivariate Gaussian:

$$\boldsymbol{x_i} \sim \mathcal{N}(\boldsymbol{\mu}, \begin{pmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_x^2 \end{pmatrix}), \tag{4.3}$$

with $\sigma_x^2 = 0.02$ and two cluster centers $\boldsymbol{\mu_1} = (-1, -1)$, $\boldsymbol{\mu_2} = (1, 1)$. We generate 50 samples for each cluster.

Figure 4.2 shows the predictive uncertainty $\sigma[f(\boldsymbol{x})]$ for all models on a test grid $[-2, 2] \times [-2, 2]$, where the noisy observations are depicted as white crosses. A brighter color corresponds to higher predictive uncertainty. The ground truth GP-ReLU ($\sigma_w = 2$ and $\sigma_b = 1$; top-left) predicts low uncertainty at and close to the training clusters, driven by the noise in the data. The further a

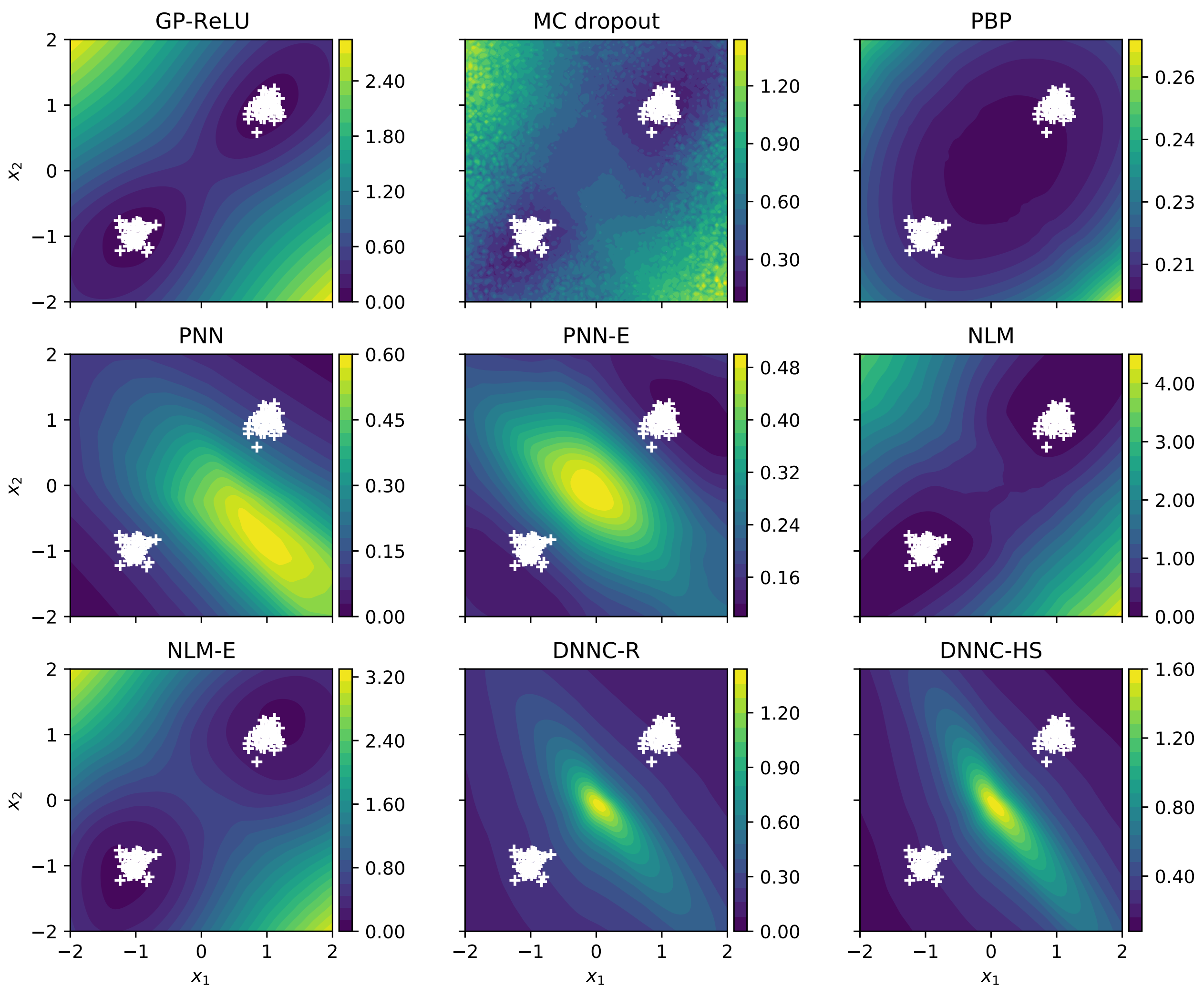


Figure 4.2: Predictive uncertainty $\sigma[f(\boldsymbol{x})]$ of all methods for toy regression problem with 2D input space. Noisy observations are depicted as white crosses. Note that the color scaling differs for all methods.

test input is located away from the clusters, the higher the predicted (epistemic) uncertainty. MC dropout (top-center) is again relatively close to the ground truth GP-ReLU. PBP fails to capture the *in-between* uncertainty between the clusters, similarly to the variational inference method studied in Foong et al. (2019a).

The simple PNN (center-left) predicts higher uncertainty in the center of the test data and decreasing uncertainty as we move away from the center. As in the Toy1D experiment, the PNN predicts low uncertainty *outside* of the training range $[-1, 1] \times [-1, 1]$. The deep ensemble (PNN-E; center) can capture some epistemic uncertainty between the data clusters but struggles for most of the test grid. The neural linear model (NLM) and an ensemble of NLMs (NLM-E) show predictive uncertainties comparable to the ground truth.

The marginally calibrated methods (DNNC-R and DNNC-HS) suffer from the same problems as in Toy1D. Incorporating the empirical marginal distribution – estimated on the training data –

is not meaningful in this context. Especially, we find high predictive uncertainty in feature space orthogonal to the line connecting the training data clusters.

4.2 UCI Regression Datasets

In this section, we compare the performance of the algorithms on the real-world regression datasets based on their calibration and predictive performance. These UCI datasets (Dua and Graff, 2017) are typically used in the literature to gauge the predictive accuracy of regression algorithms. We evaluate the – probabilistic and marginal – calibration and the predictive performance of the models. The calibration is assessed in-sample with full data. For the predictive performance, we obtain robust results by splitting the data in three ways: 1) random splits, 2) gap splits (Foong et al., 2019b) to gauge the models' ability to predict *in-between* uncertainty, and 3) tail splits to evaluate the models on *out-of-range* inputs.

The outline of this section is as follows. First, we introduce the datasets and the process of splitting the data into train and test sets. Second, the calibration criteria and performance metrics are defined. Third, we summarize the algorithmic settings. Finally, all results are presented and discussed.

Abbreviation	Full Name	n	d
boston	Boston Housing	506	13
concrete	Concrete Compression Strength	1,030	8
energy	Energy Efficiency	768	8
kin8nm	Kin8nm	8,192	8
powerplant	Combined Cycle Power Plant	9,568	4
wine	Wine Quality Red	1,599	11
yacht	Yacht Hydrodynamics	308	6

Table 4.1: Summary of the UCI datasets with n samples and d-dimensional features.

4.2.1 Description of Data and Train-Test Splits

The datasets are described in Table 4.1. Due to limited resources, we focus on relatively small datasets with less than 10,000 samples. The marginally calibrated deep distributional regression method from Section 3.5.2 requires continuous target variables. However, the *wine* dataset regresses an integer from three to eight. We decided to include it nevertheless and see some interesting results in Section 4.2.4. A histogram of the response values and the kernel density estimation for all UCI datasets are depicted in Figure 4.3.

To ensure robust results, we benchmark the predictive performance of the methods on three different train/test-split schemes. We use the splitting process of Hernández-Lobato and Adams

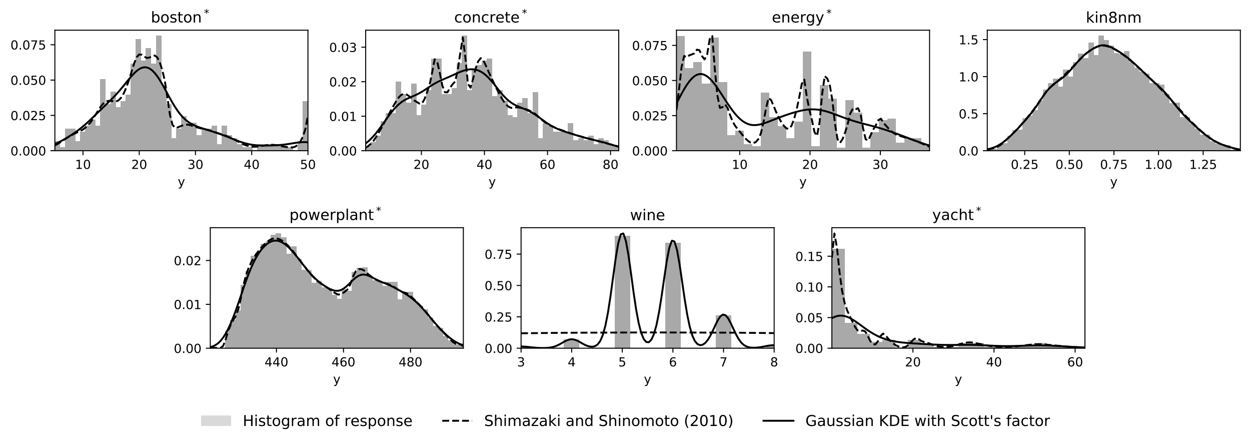


Figure 4.3: Kernel density estimation for the UCI datasets. We choose the best KDE method by visual inspection. We use the kernel density estimation of Shimazaki and Shinomoto (2010) for datasets marked with a star, a Gaussian KDE with Scott's factor (Scott, 1992) for the others.

(2015), i.e., we split the data *randomly* into training and test set with a test ratio of 0.1. This process is repeated 20 times.

Further, we evaluate the performance using the *gap splits* proposed by Foong et al. (2019b). Denote a given dataset with $\mathcal{D} = (\boldsymbol{X}, \boldsymbol{y})$, where $\boldsymbol{X}$ is a $n \times d$ matrix. For every continuous feature d_{cont} of $\boldsymbol{X}$, we split the data into 2/3 training set and 1/3 test set. We do this by sorting the data in increasing order of that feature and selecting the middle third as test set, the rest as training set. Thus, we obtain d_{cont} gap splits for a dataset with d_{cont} continuous features.

As we have observed in Section 4.1, some methods predict low uncertainty for regions *outside* the training range. Thus, we introduce another way of splitting the data into train and test sets to evaluate the models' prediction given out-of-range inputs. The process to create these *tail splits* is similar to the process that generates the gap splits, and we obtain d_{cont} tail splits for a dataset with d_{cont} continuous features. After sorting, we obtain the test set as the bottom and top 1/6s and use the rest as training data. However, for out-of-distribution inputs, the predicted variance output of a PNN might not be well-behaved (Chua et al., 2018). We, therefore, bound the variance output by its minimum and maximum predicted variance for the training data. This is feasible as for data outside the training range, the aleatoric uncertainty typically does not increase (Kendall and Gal, 2017).

4.2.2 Evaluation Criteria

We benchmark the models on a multitude of metrics. These can be roughly split into two categories: calibration criteria and predictive performance metrics.

Calibration Criteria

Given an input $\boldsymbol{x}_i$, all methods used in this thesis forecast a predictive density $\hat{p}_i(\cdot|\boldsymbol{x}_i)$, or by integration a predictive distribution $\hat{F}_i(\cdot|\boldsymbol{x}_i)$. We assess the *calibration* of the algorithms by considering *probability calibration* and *marginal calibration*. This helps us to measure the consistency of forecasts and observations (Gneiting et al., 2007). For the evaluation of calibration, we train the methods on the full data.

A method is probabilistically calibrated if the proportion of observations $\tilde{p}_j$ that fall below the p_j-quantile is approximately equal to this quantile: $\tilde{p}_j \approx p_j \; \forall p_j \in \{0 \leq p_1 \leq ... \leq p_k \leq 1\}$. We compute a proportion $\tilde{p}_j$ as:

$$\tilde{p}_j = \frac{1}{n} \# \left\{ y_i | \hat{F}_i(y_i|\boldsymbol{x}_i) < p_j, i = 1, ..., n \right\} . \tag{4.4}$$

We assess the probability calibration in Section 4.2.2 by plotting the deviation of the proportion to the quantile $\tilde{p}_j - p_j$ versus p_j. A perfectly calibrated method is thus a horizontal line.

A method is marginally calibrated if the average predictive density equals the marginal density observed from the data. We compute the average predictive density $\hat{p}_{\text{avg}}$ as:

$$\hat{p}_{\text{avg}}(y) = \frac{1}{n} \sum_{i=1}^{n} \hat{p}_i(y|\boldsymbol{x}_i) . \tag{4.5}$$

The empirical marginal density is unknown and we have to estimate it. We compare two methods for non-parametrically kernel density estimation (KDE) for each dataset. The first is a Gaussian kernel density estimation with Scott's rule (Scott, 1992) as bandwidth method. As an alternative, we use the non-parametric method of Shimazaki and Shinomoto (2010). We chose the best KDE for each dataset by visual inspection. The histogram of the response and the kernel density estimation for all UCI datasets are depicted in Figure 4.3. In our experiments, we use the method of Shimazaki and Shinomoto (2010) for the datasets *boston*, *concrete*, *powerplant*, *energy* and *yacht*, and a Gaussian KDE for the rest.

Predictive Performance Metrics

We gauge the quality of the model predictions mainly with two strictly proper scoring rules. A scoring rule $\mathcal{S}$ can be thought of as a function $\mathcal{S} \colon (\hat{p}_i, y_i) \to \mathbb{R}$ that maps a predictive density $\hat{p}_i$ – or by integration $\hat{F}_i$ – and true event y_i to a numerical value. A scoring rule is *proper* if its expected score is maximized when the forecaster forecasts the true distribution and *strict* if the maximum is unique (Gneiting and Raftery, 2007).

We calculate the logarithmic score (LogS; Good, 1952) as:

$$\mathrm{LogS}(\hat{p}_i, y_i) = \log \hat{p}_i(y_i|\boldsymbol{x}_i)\,, \tag{4.6}$$

and the continuous ranked probability score (CRPS; Matheson and Winkler, 1976; Unger, 1985) as:

$$\begin{aligned}\mathrm{CRPS}(\hat{F}_i, y_i) &= \int_{-\infty}^{\infty} (\hat{F}_i(y) - \mathbf{1}\{y \geq y_i\})^2 \,\mathrm{d}y & (4.7)\\ &= \int_{-\infty}^{y_i} \hat{F}_i(y)^2 \,\mathrm{d}y + \int_{y_i}^{\infty} (\hat{F}_i(y) - 1)^2 \,\mathrm{d}y\,. & (4.8)\end{aligned}$$

For events where the predictive density has a low value at the true target ($\hat{p}_i(y_i|\mathbf{x}_i)$ is small), the logarithmic score gives a high penalty. Thus, the LogS is sensitive to outliers, i.e., $\hat{p}_i(y_i|\mathbf{x}_i) = 0$ leads to a LogS of minus infinity. Further, to compute the logarithmic score, we only evaluate the predictive density at a single point. In contrast, the CRPS takes the full predictive distribution into account and has a lower sensitivity w.r.t. extreme cases (Gneiting and Raftery, 2007). Given n observations, the average of LogS and CRPS are obtained as:

$$\begin{aligned}\overline{\mathrm{LogS}} &= \frac{1}{n}\sum_{i=1}^{n} \mathrm{LogS}(\hat{p}_i, y_i) \quad \text{and} & (4.9)\\ \overline{\mathrm{CRPS}} &= \frac{1}{n}\sum_{i=1}^{n} \mathrm{CRPS}(\hat{F}_i, y_i)\,. & (4.10)\end{aligned}$$

We supplement the scoring rules with the root mean square error (RMSE):

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}\,, \tag{4.11}$$

and two interval metrics, the 95% prediction interval coverage probability (PICP) and the average width of the 95% interval (MPIW). In some applications – e.g., safety-critical environments – it can be important to assure that the true value lies in the prediction interval: $y_i \in [q_i(\alpha), q_i(1-\alpha)]$ where q_i denotes the quantile function for some predictive density $\hat{p}_i(\cdot|\boldsymbol{x}_i)$. We use $\alpha = 0.025$ to compute the 95% PICP:

$$\mathrm{PICP} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{q_i(0.025) \leq y_i \leq q_i(0.975)}\,. \tag{4.12}$$

However, the wider the prediction interval, the higher the PICP. For example, a Gaussian predictive density with an infinite variance leads to a PICP of 1. To account for this, we further calculate the average width of the 95% interval (MPIW):

$$\text{MPIW} = \frac{1}{n}\sum_{i=1}^{n}\left(q_i(0.975) - q_i(0.025)\right). \tag{4.13}$$

The MPIW helps us to evaluate the *sharpness* of the predictions (Gneiting et al., 2007). Generally, a good model should have a high coverage probability and output narrow 95% intervals.

4.2.3 Algorithms and Implementation

For the UCI experiments, we follow the experimental setup of Hernández-Lobato and Adams (2015). That is, for all methods, we use an (underlying) NN with a single hidden layer and 50 hidden units. We use a ReLU activation function for the hidden layer and a linear activation function for the output layer. The network is trained for 40 epochs with the Adam optimizer. As a network with dropout takes longer to converge, we train MC dropout with 10x the epochs as in Gal and Ghahramani (2016). For algorithms that require hyperparameter-tuning, we use 20% of the training data to obtain the best values. After this, we fit the NN using the full training data.

Abbreviation	Method (Section)	Settings
MC dropout	Monte Carlo dropout (3.4)	We obtain the optimal dropout rate by a grid search in $[0.005, 0.01, 0.05, 0.1]$ with a validation size of 20% optimized for NLL/LogS. We train the NN for 400 epochs and use 1,000 forward passes for inference.
PBP	Probabilistic backpropagation (3.2.2)	See Section 3.2.2 for details on the priors for the precision parameters.
PNN	Probabilistic neural network (3.1.1)	
PNN-E	Deep ensemble, ensemble of PNNs (3.1.2)	We use five models for our ensemble and approximate the predictive density with a Gaussian distribution. See Section 3.1.2 for details.
NLM	Neural linear model (3.5.1)	We set a prior for beta: $\beta_j\|\tau^2 \sim N(0, \tau^2)$ with τ^2 obtained by grid search in $[0.005, 0.01, 0.05, ..., 500, 1000]$ optimized for NLL/LogS.
NLM-E	Ensemble of NLMs	See NLM and PNN-E.
DNNC-R	Marginally calibrated deep distributional regression with ridge prior (3.5.2)	We set a prior for beta: $\beta_j\|\tau^2 \sim N(0, \tau^2)$, where $p\left(\tau^2\right)$ is scale-dependent hyper-prior. We run MCMC with 4,000 iterations and discard the first 1,000 samples (burn-in).

Table 4.2: Algorithmic settings for the UCI experiments.

Table 4.2 summarizes the methods that are benchmarked on the UCI datasets and their settings. For DNNC-R, we sample 4,000 iterations with a burn-in of 1,000 iterations (contrary to 15,000 in Klein et al. (2021)) to reduce the computational burden. We observe high acceptance rates and find that the reduced sample size does not significantly influence the method's performance on all datasets considered in this thesis. Contrary to the experiments on the toy data, we do not benchmark the methods GP-ReLU and DNNC-HS. We exclude GP-ReLU due to the bad scaling for large data and the homoscedastic regression restriction in our implementation. DNNC-HS is left out as it requires expensive sampling via CPU and we ran our experiments on Google Colaboratory, which offers relatively weak CPU performance.

4.2.4 Results and Discussion

Marginal Calibration

We assess the marginal calibration of the models via visual inspection in Figure 4.4. The figure shows a histogram of the response values, the empirical marginal density (see Figure 4.3 for the selection of the KDE method) as a dashed black line, and the average predictive density $\hat{p}_{\text{avg}}(y) := \frac{1}{n}\sum_{i=1}^{n}\hat{p}_i(y|\boldsymbol{x}_i)$ of all methods per dataset. For *yacht*, we show the log-transformed values for better visibility. A method is marginally calibrated if the average predictive density matches the empirical marginal density.

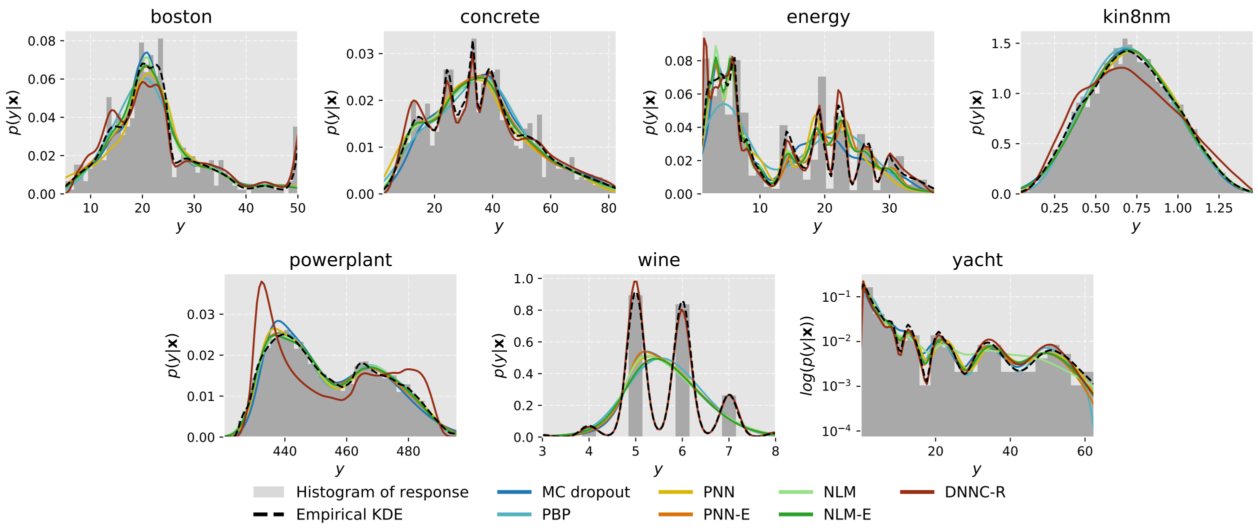


Figure 4.4: Evaluation of marginal calibration for UCI datasets. All methods are trained on the full data. A method is marginally calibrated if the average predictive density $\hat{p}_{\text{avg}}(y) := \frac{1}{n}\sum_{i=1}^{n}\hat{p}_i(y|\boldsymbol{x}_i)$ matches the empirical marginal density which is depicted as dashed black line. Note that we only observe draws from the true distribution and estimate the marginal density. We log-transform the y-axis for *yacht* for better visibility.

We can make a few observations regarding the marginal calibration of the methods. First, we find that DNNC-R is best marginally calibrated for most of the datasets. However, this method struggles for the large datasets *kin8nm* and especially *powerplant*. We tried various KDEs and transformations of the response but could not resolve the issues. A possible solution is the heteroscedastic implicit copula from Smith and Klein (2021). Second, DNNC-R is the only method where the average predictive density matches the right tail behavior of the empirical marginal density for *boston*. Further, for *wine*, DNNC-R is the only marginally calibrated method. The other models are not well calibrated due to their Gaussian predictive distributions and the integer nature of the response variable. Third, the other methods are well calibrated for *kin8nm* and *powerplant*. Fourth, MC dropout and PBP perform badly for *energy*. For *yacht*, NLM struggles for most of the response range and PBP at the right tail.

Probability Calibration

In Figure 4.5 we show the probability calibration of the methods by plotting the deviation $\tilde{p}_j - p_j$ versus p_j. Here $\tilde{p}_j$ (see Section 4.2.2) denotes the proportion of observations that fall below the p_j-quantile. For a perfectly probabilistically calibrated method, this proportion is approximately equal to that quantile, $\tilde{p}_j \approx p_j$. As we plot the deviation to the quantile, a perfectly probabilistically calibrated method corresponds to a horizontal line (depicted as a dashed black line). Values above the horizontal line indicate over-confident models, values below under-confident models. We use

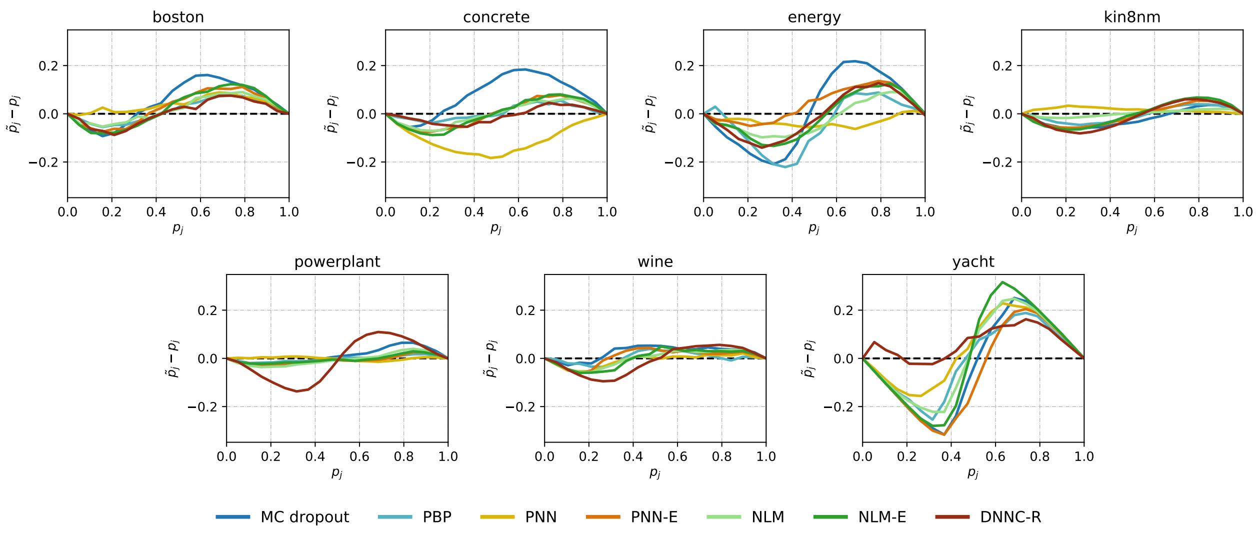


Figure 4.5: Evaluation of probability calibration for UCI datasets. All methods are trained on the full data. We show the expected confidence level (p_j-quantiles) on the x-axis and the observed minus the expected confidence level ($\tilde{p}_j - p_j$) on the y-axis. $\tilde{p}_j$ is the proportion of observations that fall below the p_j-quantile. A dashed black line indicates a perfectly probability calibrated model.

the same scaling of the y-axis for all datasets to easily identify datasets where (some) methods are not well calibrated.

In general, we find there is no clear winner across all datasets. NLM, the ensemble models (PNN-E, NLM-E), and DNNC-R are best calibrated. All methods are probabilistically calibrated for the datasets *kin8nm*, *powerplant* (except DNNC-R), and *wine*. The bad probabilistic calibration of DNNC-R is congruent with the results of the marginal calibration and may result from a bad kernel density estimation of the response. MC dropout is not well calibrated for *boston*, *concrete*, and *energy*. All methods but DNNC-R are poorly probabilistically calibrated for the *yacht* dataset.

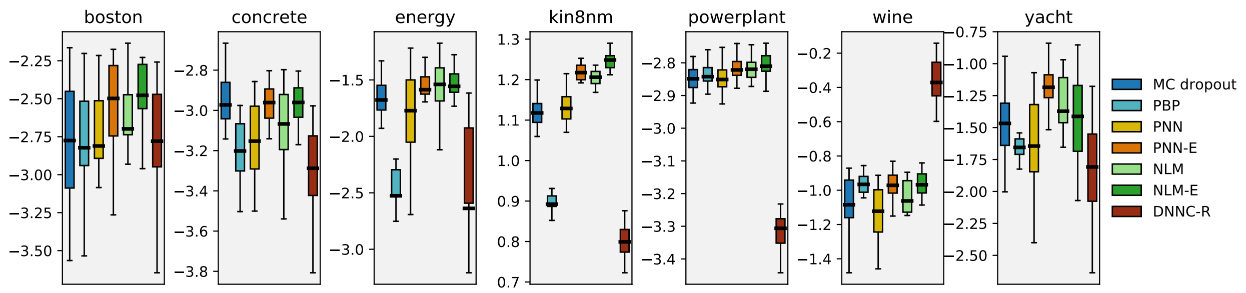
Predictive Performance: Random Splits

For all UCI datasets, we compute the $\overline{\text{LogS}}$, $\overline{\text{CRPS}}$, RMSE, PICP, and MPIW for 20 random train/test splits with a 10% test ratio. Table 4.3 depicts the mean values for all metrics with standard errors in parentheses. For each dataset (row), the best method (column) for a metric is marked in bold. For $\overline{\text{LogS}}$ and PICP, higher values correspond to better performance. For $\overline{\text{CRPS}}$ and RMSE, lower values represent accurate predictions. MPIW is given as additional information about the predictive variance of the methods. Figure 4.6 depicts the predictive $\overline{\text{LogS}}$, $\overline{\text{CRPS}}$, and RMSE values for all splits graphically via box-plots. The in-sample results can be found in Appendix B.1.

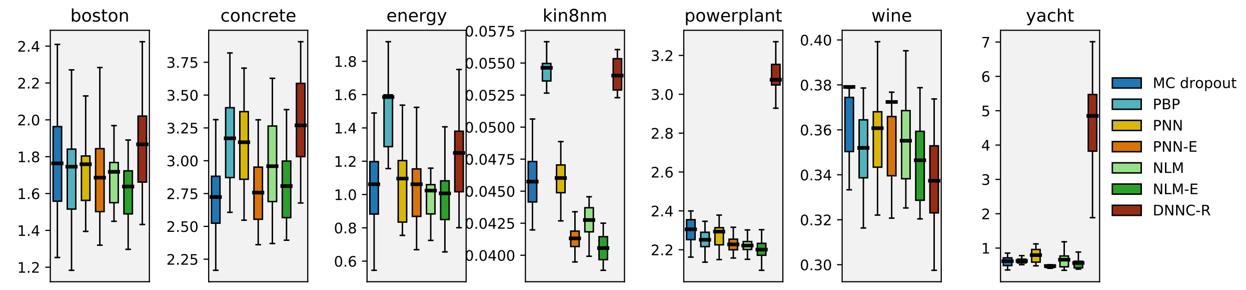
We make five observations regarding the predictive performance of the methods. First, the ensemble methods (PNN-E and NLM-E) perform best across most UCI datasets, while MC dropout and NLM are competitive. Second, a simple probabilistic neural network (PNN) is comparable for the large datasets *kin8nm* and *powerplant*. A large part of the predictive uncertainty is driven by aleatoric uncertainty (noise in the data) as epistemic uncertainty has been explained away. Third, a deep ensemble (PNN-E; an ensemble of PNNs) performs significantly better than a PNN for datasets with fewer samples. This makes sense as the ensemble is able to capture epistemic uncertainty. Similarly, the neural linear model (NLM) improves upon a PNN by capturing epistemic uncertainty. An ensemble of NLMs (NLM-E) slightly improves upon a single NLM and shows the best predictive performance w.r.t. RMSE, $\overline{\text{CRPS}}$, $\overline{\text{LogS}}$, and PICP. Fourth, PBP performs about on par with a PNN but struggles with *energy* and *kin8nm*. Fifth, DNNC-R performs significantly worse for the large datasets *kin8nm* and *powerplant*, in line with the observations about the marginal calibration. For *yacht*, we observe high RMSE and $\overline{\text{CRPS}}$ for DNNC-R as the method likely puts a lot of probability mass near the left tail. However, for *wine*, DNNC-R has by far the highest $\overline{\text{LogS}}$ since the empirical marginal distribution, interwoven into the approach, leads to predictive densities with high probability mass at the integer target values.

		MC dropout	PBP	PNN	PNN-E	NLM	NLM-E	DNNC-R
boston	$\overline{\text{LogS}}$	-2.78 (0.12)	-2.82 (0.11)	-2.81 (0.11)	-2.50 (0.06)	-2.70 (0.10)	**-2.48 (0.06)**	-2.78 (0.09)
	$\overline{\text{CRPS}}$	1.76 (0.07)	1.75 (0.07)	1.76 (0.07)	1.69 (0.06)	1.72 (0.06)	**1.64 (0.06)**	1.87 (0.08)
	RMSE	3.93 (0.24)	**3.54 (0.20)**	3.94 (0.21)	3.72 (0.19)	3.88 (0.20)	3.72 (0.20)	3.64 (0.18)
	PICP	0.89 (0.02)	0.90 (0.01)	0.86 (0.01)	0.91 (0.01)	0.88 (0.01)	**0.92 (0.01)**	0.88 (0.01)
	MPIW	10.96 (1.04)	10.02 (0.07)	8.69 (0.37)	10.97 (0.76)	9.47 (0.54)	10.51 (0.45)	12.25 (0.25)
concrete	$\overline{\text{LogS}}$	-2.97 (0.04)	-3.20 (0.04)	-3.15 (0.05)	**-2.96 (0.02)**	-3.07 (0.05)	**-2.96 (0.03)**	-3.29 (0.05)
	$\overline{\text{CRPS}}$	**2.72 (0.07)**	3.17 (0.08)	3.14 (0.07)	2.76 (0.06)	2.96 (0.08)	2.81 (0.06)	3.27 (0.08)
	RMSE	5.37 (0.18)	5.83 (0.16)	6.10 (0.15)	**5.35 (0.15)**	5.80 (0.20)	5.50 (0.16)	5.95 (0.16)
	PICP	0.94 (0.00)	0.92 (0.01)	0.91 (0.01)	0.95 (0.00)	0.94 (0.01)	**0.97 (0.00)**	0.90 (0.01)
	MPIW	18.88 (0.38)	20.67 (0.06)	19.08 (0.51)	19.83 (0.31)	19.74 (0.42)	21.21 (0.21)	21.23 (0.21)
energy	$\overline{\text{LogS}}$	-1.68 (0.05)	-2.53 (0.09)	-1.77 (0.08)	-1.59 (0.04)	**-1.54 (0.05)**	-1.56 (0.05)	-2.64 (0.33)
	$\overline{\text{CRPS}}$	1.06 (0.08)	1.59 (0.10)	1.10 (0.07)	1.06 (0.07)	1.02 (0.06)	**1.01 (0.06)**	1.25 (0.08)
	RMSE	2.90 (0.34)	3.61 (0.32)	2.97 (0.33)	3.02 (0.33)	3.03 (0.27)	**2.88 (0.30)**	3.20 (0.33)
	PICP	**0.99 (0.00)**	0.91 (0.02)	0.94 (0.01)	**0.99 (0.00)**	0.97 (0.01)	**0.99 (0.00)**	0.97 (0.00)
	MPIW	9.56 (0.71)	10.56 (0.09)	7.21 (0.47)	7.77 (0.35)	7.31 (0.43)	8.22 (0.31)	9.59 (0.27)
kin8nm	$\overline{\text{LogS}}$	1.12 (0.01)	0.89 (0.01)	1.13 (0.01)	1.22 (0.00)	1.21 (0.01)	**1.25 (0.00)**	0.80 (0.01)
	$\overline{\text{CRPS}}$	0.05 (0.00)	0.05 (0.00)	0.05 (0.00)	**0.04 (0.00)**	**0.04 (0.00)**	**0.04 (0.00)**	0.05 (0.00)
	RMSE	**0.08 (0.00)**	0.10 (0.00)	0.09 (0.00)	**0.08 (0.00)**	**0.08 (0.00)**	**0.08 (0.00)**	0.09 (0.00)
	PICP	**0.98 (0.00)**	0.95 (0.00)	0.94 (0.00)	**0.98 (0.00)**	0.96 (0.00)	**0.98 (0.00)**	0.95 (0.00)
	MPIW	0.37 (0.01)	0.41 (0.00)	0.31 (0.00)	0.35 (0.00)	0.32 (0.00)	0.34 (0.00)	0.45 (0.00)
powerplant	$\overline{\text{LogS}}$	-2.85 (0.01)	-2.84 (0.01)	-2.85 (0.01)	-2.82 (0.01)	-2.82 (0.01)	**-2.81 (0.01)**	-3.31 (0.02)
	$\overline{\text{CRPS}}$	2.31 (0.02)	2.25 (0.01)	2.29 (0.02)	2.23 (0.01)	2.22 (0.01)	**2.20 (0.01)**	3.07 (0.04)
	RMSE	4.25 (0.05)	4.15 (0.04)	4.22 (0.05)	4.13 (0.04)	4.11 (0.04)	**4.08 (0.04)**	4.67 (0.05)
	PICP	**0.97 (0.00)**	0.96 (0.00)	0.96 (0.00)	**0.97 (0.00)**	0.96 (0.00)	**0.97 (0.00)**	0.96 (0.00)
	MPIW	17.65 (0.19)	16.06 (0.02)	16.42 (0.24)	16.62 (0.07)	16.49 (0.18)	16.34 (0.10)	30.09 (0.44)
wine	$\overline{\text{LogS}}$	-1.08 (0.04)	-0.97 (0.01)	-1.12 (0.04)	-0.97 (0.02)	-1.06 (0.04)	-0.97 (0.02)	**-0.37 (0.04)**
	$\overline{\text{CRPS}}$	0.38 (0.02)	0.35 (0.00)	0.36 (0.00)	0.37 (0.02)	0.36 (0.00)	0.35 (0.00)	**0.34 (0.00)**
	RMSE	0.67 (0.01)	**0.63 (0.01)**	0.66 (0.01)	**0.63 (0.01)**	**0.63 (0.01)**	**0.63 (0.01)**	0.64 (0.01)
	PICP	0.92 (0.01)	**0.93 (0.01)**	0.90 (0.01)	**0.93 (0.00)**	0.91 (0.01)	**0.93 (0.00)**	0.92 (0.01)
	MPIW	2.74 (0.28)	2.34 (0.00)	2.27 (0.04)	2.84 (0.37)	2.40 (0.11)	2.43 (0.03)	2.29 (0.02)
yacht	$\overline{\text{LogS}}$	-1.47 (0.07)	-1.66 (0.02)	-1.64 (0.10)	**-1.18 (0.04)**	-1.37 (0.09)	-1.41 (0.07)	-1.81 (0.11)
	$\overline{\text{CRPS}}$	0.62 (0.03)	0.62 (0.02)	0.80 (0.06)	**0.48 (0.02)**	0.66 (0.07)	0.57 (0.04)	4.85 (0.39)
	RMSE	1.05 (0.07)	1.07 (0.06)	1.65 (0.14)	**0.93 (0.08)**	1.35 (0.14)	0.98 (0.12)	5.56 (0.44)
	PICP	0.99 (0.01)	0.98 (0.01)	0.93 (0.02)	**1.00 (0.00)**	0.99 (0.00)	**1.00 (0.00)**	0.88 (0.01)
	MPIW	7.63 (0.45)	6.80 (0.05)	6.38 (0.72)	6.05 (0.29)	7.40 (1.01)	7.85 (0.63)	26.63 (1.11)

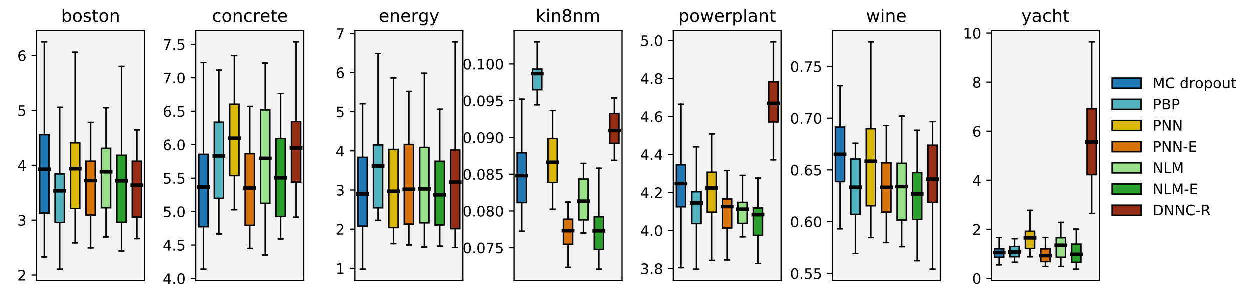
Table 4.3: Predictive results for UCI datasets with **random splits**. We depict the mean $\overline{\text{LogS}}$, $\overline{\text{CRPS}}$, RMSE, PICP, and MPIW for all methods and datasets with standard errors in parentheses. The method with the best mean score per dataset is depicted in bold. For RMSE and $\overline{\text{CRPS}}$ lower values correspond to better performance, for $\overline{\text{LogS}}$ and PICP high values represent accurate predictions.



(a) Average logarithmic score ($\overline{\text{LogS}}$)



(b) Average continuous ranked probability score ($\overline{\text{CRPS}}$)



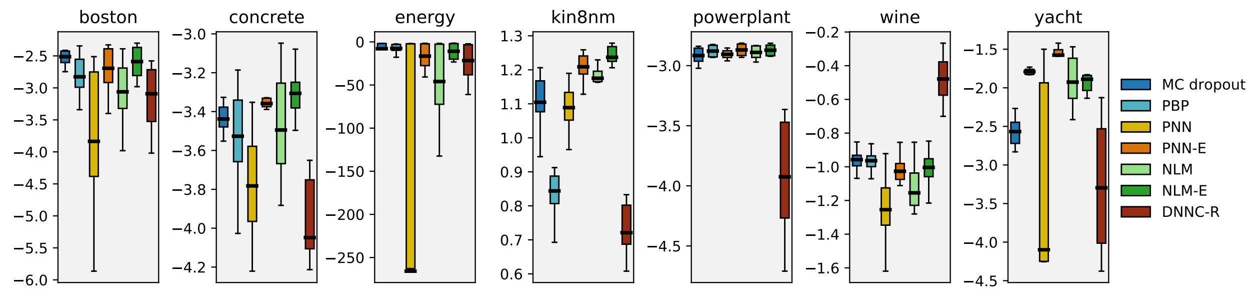
(c) Root mean square error (RMSE)

Figure 4.6: Predictive box-plots for all models for UCI datasets with **random splits**. (a) $\overline{\text{LogS}}$: the higher the value the better performance of the model. (b) $\overline{\text{CRPS}}$: a small value corresponds to a good performance. (c) RMSE: the smaller the better. Note that the vertical black lines in each box-plot correspond to the mean.

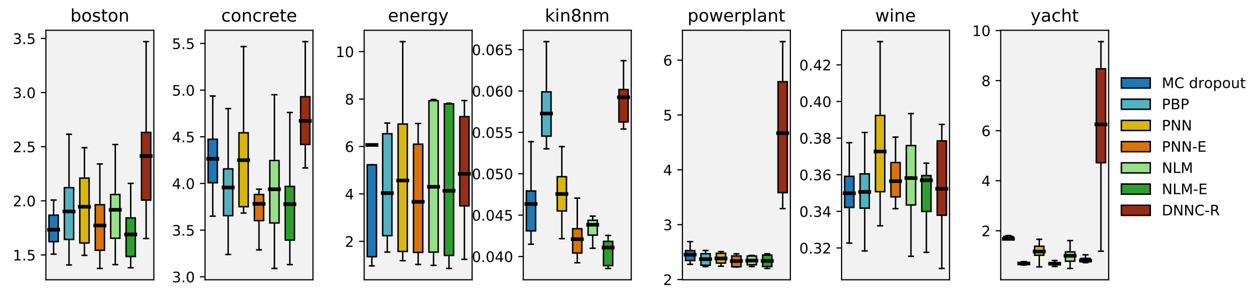
Predictive Performance: Gap and Tail Splits

We further benchmark the models' performance on the UCI data with gap and tail splits as outlined in Section 4.2.1. The gap and tail splits are generated based on the continuous features of the datasets. We provide the predictive box-plots for $\overline{\text{LogS}}$, $\overline{\text{CRPS}}$, and RMSE in Figures 4.7 and 4.8. The predictive mean values and standard errors for all metrics are presented in Tables 4.4 and 4.5. For in-sample results see Appendix B.1. We train all methods with 2/3 of the data as training data for these split types, compared to 9/10 for the random splits. Therefore, we expect to observe worse performance for all models across all datasets.

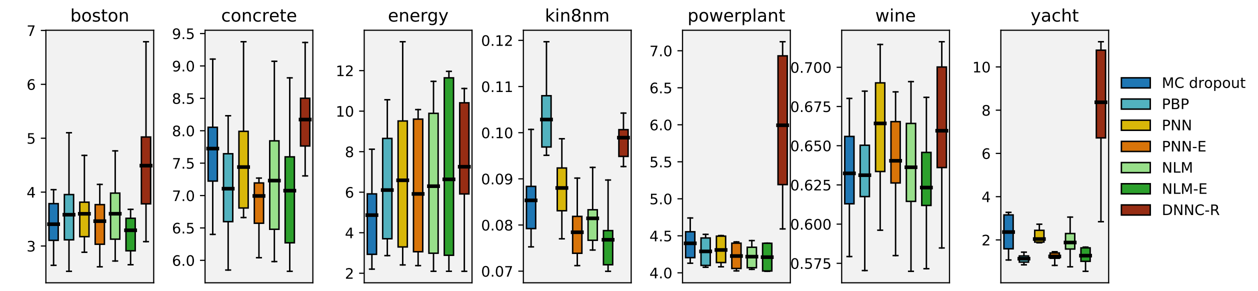
With the gap splits, we measure – similarly to the toy experiment in Section 4.1.2 – whether the methods have high predictive uncertainty in regions between clusters of observations (Foong



(a) Average logarithmic score ($\overline{\text{LogS}}$)



(b) Average continuous ranked probability score ($\overline{\text{CRPS}}$)



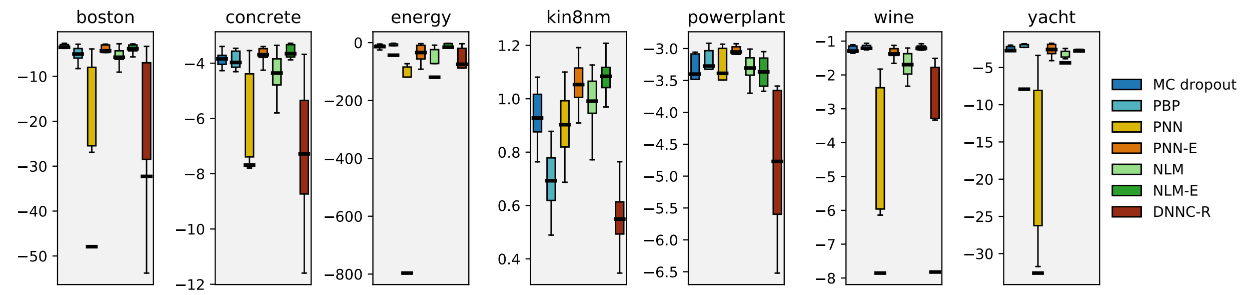
(c) Root mean square error (RMSE)

Figure 4.7: Predictive box-plots for all models for UCI datasets with **gap splits**. (a) $\overline{\text{LogS}}$: the higher the value the better performance of the model. (b) $\overline{\text{CRPS}}$: a small value corresponds to a good performance. (c) RMSE: the smaller the better. Note that the vertical black lines in each box-plot correspond to the mean.

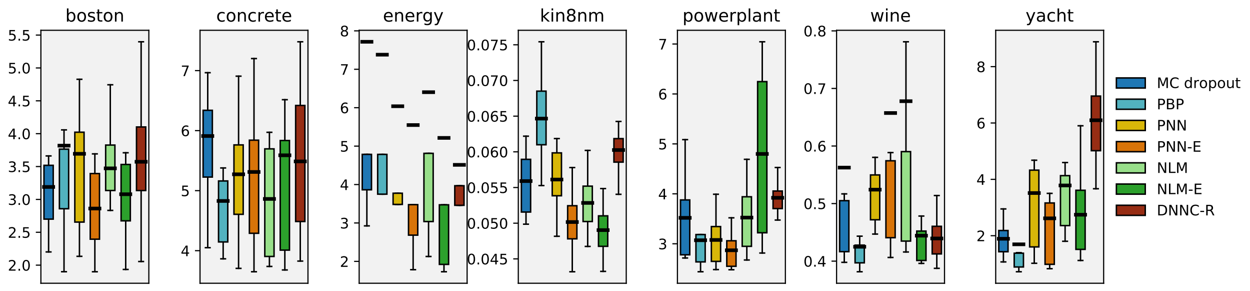
et al., 2019b). In general, the ensemble methods and MC dropout perform best for the gap splits. Most methods perform almost as well as with random splits for the large datasets *kin8nm* and *powerplant*. For some datasets (*boston* and *energy*), NLM-E performs significantly better than NLM, whereas the gains for random splits are only minor. For *energy* in general, the predictive performance of most methods (notably also PNN-E) declines massively. Interestingly, we find, that PBP, where the posterior is a factorized Gaussian distribution, is among the models that retain a relatively good predictive $\overline{\text{LogS}}$, contrary to the mean-field Gaussian approximation benchmarked in Foong et al. (2019b). The performance of DNNC-R is negatively affected as the quality of the kernel density estimation likely declines. This is evident in the performance drop for the datasets *concrete* and *yacht*. For *kin8nm* and especially *wine*, the method performs about as good as with

		MC dropout	PBP	PNN	PNN-E	NLM	NLM-E	DNNC-R
boston	$\overline{\text{LogS}}$	**-2.51 (0.03)**	-2.83 (0.13)	-3.84 (0.34)	-2.69 (0.11)	-3.06 (0.18)	-2.59 (0.07)	-3.09 (0.14)
	$\overline{\text{CRPS}}$	1.73 (0.05)	1.90 (0.11)	1.95 (0.10)	1.77 (0.09)	1.92 (0.12)	**1.69 (0.07)**	2.41 (0.19)
	RMSE	3.41 (0.13)	3.58 (0.21)	3.60 (0.18)	3.46 (0.19)	3.60 (0.20)	**3.29 (0.15)**	4.49 (0.33)
	PICP	**0.93 (0.01)**	0.89 (0.02)	0.77 (0.02)	0.89 (0.02)	0.83 (0.03)	0.90 (0.01)	0.86 (0.02)
	MPIW	12.21 (0.55)	10.55 (0.19)	8.42 (0.39)	10.30 (0.45)	9.99 (0.77)	10.43 (0.40)	13.85 (0.65)
concrete	$\overline{\text{LogS}}$	-3.44 (0.03)	-3.53 (0.10)	-3.78 (0.11)	-3.36 (0.06)	-3.49 (0.10)	**-3.31 (0.05)**	-4.05 (0.16)
	$\overline{\text{CRPS}}$	4.26 (0.14)	3.96 (0.18)	4.25 (0.24)	**3.78 (0.21)**	3.94 (0.21)	**3.78 (0.19)**	4.67 (0.15)
	RMSE	7.72 (0.28)	7.11 (0.30)	7.44 (0.35)	**6.99 (0.37)**	7.23 (0.36)	7.08 (0.36)	8.17 (0.23)
	PICP	**0.99 (0.00)**	0.86 (0.02)	0.80 (0.02)	0.87 (0.02)	0.85 (0.02)	0.90 (0.01)	0.82 (0.02)
	MPIW	36.96 (0.90)	20.56 (0.43)	19.69 (0.95)	20.79 (1.00)	20.73 (1.13)	21.86 (1.10)	22.86 (0.44)
energy	$\overline{\text{LogS}}$	-7.91 (3.47)	**-7.55 (2.87)**	-265.80 (200.43)	-16.50 (7.64)	-45.96 (25.13)	-10.88 (4.56)	-21.65 (11.96)
	$\overline{\text{CRPS}}$	6.06 (3.53)	4.03 (1.14)	4.57 (1.79)	**3.67 (1.21)**	4.30 (1.53)	4.13 (1.53)	4.85 (1.23)
	RMSE	**4.87 (1.06)**	6.11 (1.49)	6.59 (2.11)	5.92 (1.64)	6.30 (1.87)	6.63 (2.15)	7.26 (1.64)
	PICP	**0.75 (0.10)**	0.60 (0.12)	0.52 (0.18)	0.71 (0.10)	0.60 (0.13)	0.72 (0.11)	0.57 (0.18)
	MPIW	64.50 (54.91)	9.41 (1.03)	5.81 (1.83)	8.10 (1.15)	9.54 (2.69)	8.47 (0.85)	10.23 (2.06)
kin8nm	$\overline{\text{LogS}}$	1.10 (0.03)	0.84 (0.03)	1.09 (0.03)	1.21 (0.02)	1.18 (0.02)	**1.24 (0.02)**	0.72 (0.05)
	$\overline{\text{CRPS}}$	0.05 (0.00)	0.06 (0.00)	0.05 (0.00)	**0.04 (0.00)**	**0.04 (0.00)**	**0.04 (0.00)**	0.06 (0.00)
	RMSE	0.09 (0.00)	0.10 (0.00)	0.09 (0.00)	**0.08 (0.00)**	**0.08 (0.00)**	**0.08 (0.00)**	0.10 (0.00)
	PICP	0.97 (0.01)	0.96 (0.01)	0.94 (0.01)	0.97 (0.00)	0.96 (0.00)	**0.98 (0.00)**	0.96 (0.00)
	MPIW	0.35 (0.01)	0.44 (0.00)	0.32 (0.01)	0.34 (0.00)	0.33 (0.01)	0.33 (0.00)	0.49 (0.01)
powerplant	$\overline{\text{LogS}}$	-2.92 (0.04)	-2.88 (0.03)	-2.90 (0.02)	**-2.87 (0.03)**	-2.89 (0.03)	**-2.87 (0.03)**	-3.93 (0.31)
	$\overline{\text{CRPS}}$	2.45 (0.09)	2.37 (0.07)	2.38 (0.06)	**2.34 (0.06)**	**2.34 (0.05)**	**2.34 (0.07)**	4.67 (0.72)
	RMSE	4.40 (0.14)	4.29 (0.12)	4.31 (0.11)	4.23 (0.10)	4.22 (0.09)	**4.21 (0.11)**	5.99 (0.60)
	PICP	0.96 (0.01)	0.95 (0.01)	0.93 (0.01)	0.95 (0.00)	0.94 (0.00)	0.95 (0.01)	**0.98 (0.01)**
	MPIW	18.02 (0.52)	16.27 (0.23)	15.54 (0.85)	16.17 (0.31)	16.06 (0.24)	16.44 (0.64)	42.65 (4.10)
wine	$\overline{\text{LogS}}$	-0.96 (0.02)	-0.96 (0.02)	-1.25 (0.06)	-1.03 (0.03)	-1.15 (0.07)	-1.00 (0.03)	**-0.48 (0.04)**
	$\overline{\text{CRPS}}$	**0.35 (0.01)**	**0.35 (0.01)**	0.37 (0.01)	0.36 (0.01)	0.36 (0.01)	0.36 (0.01)	**0.35 (0.01)**
	RMSE	0.63 (0.01)	0.63 (0.01)	0.66 (0.02)	0.64 (0.01)	0.64 (0.01)	**0.62 (0.01)**	0.66 (0.01)
	PICP	0.92 (0.00)	**0.93 (0.01)**	0.87 (0.01)	0.90 (0.00)	0.88 (0.01)	0.91 (0.00)	0.92 (0.01)
	MPIW	2.35 (0.03)	2.33 (0.02)	2.15 (0.08)	2.31 (0.02)	2.11 (0.04)	2.42 (0.14)	2.32 (0.04)
yacht	$\overline{\text{LogS}}$	-2.57 (0.09)	-1.79 (0.02)	-4.10 (1.37)	**-1.58 (0.06)**	-1.93 (0.15)	-1.89 (0.09)	-3.30 (0.38)
	$\overline{\text{CRPS}}$	1.68 (0.09)	0.69 (0.02)	1.17 (0.16)	**0.68 (0.03)**	1.00 (0.16)	0.81 (0.08)	6.25 (1.28)
	RMSE	2.36 (0.39)	**1.13 (0.09)**	2.04 (0.26)	1.24 (0.10)	1.88 (0.32)	1.27 (0.19)	8.37 (1.35)
	PICP	**1.00 (0.00)**	0.99 (0.01)	0.79 (0.09)	**1.00 (0.00)**	0.99 (0.00)	**1.00 (0.00)**	0.75 (0.06)
	MPIW	24.32 (1.22)	8.15 (0.21)	6.80 (1.29)	7.92 (0.67)	10.14 (1.35)	10.88 (1.09)	26.03 (2.53)

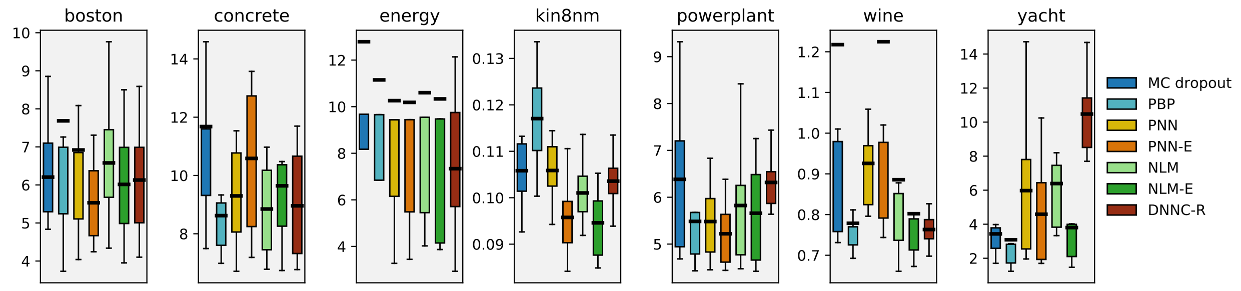
Table 4.4: Predictive results for UCI datasets with **gap splits**. We depict the mean $\overline{\text{LogS}}$, $\overline{\text{CRPS}}$, RMSE, PICP, and MPIW for all methods and datasets with standard errors in parentheses. The method with the best mean score per dataset is depicted in bold. For RMSE and $\overline{\text{CRPS}}$ lower values correspond to better performance, for $\overline{\text{LogS}}$ and PICP high values represent accurate predictions.



(a) Average logarithmic score ($\overline{\text{LogS}}$)



(b) Average continuous ranked probability score ($\overline{\text{CRPS}}$)



(c) Root mean square error (RMSE)

Figure 4.8: Predictive box-plots for all models for UCI datasets with **tail splits**. (a) $\overline{\text{LogS}}$: the higher the value the better performance of the model. (b) $\overline{\text{CRPS}}$: a small value corresponds to a good performance. (c) RMSE: the smaller the better. Note that the vertical black lines in each box-plot correspond to the mean.

random splits. Thus, it is likely that for *wine*, the incorporation of empirical marginal distribution in the DNNC-R approach still leads to meaningful predictive densities.

With the tail splits, we aim to gauge the methods' ability to predict uncertainty outside of data clusters. We observe more extreme results as for the gap splits. Ensemble methods perform best. We find that MC dropout and PBP perform similarly to the ensemble models for most datasets. This coincides with our observations from the toy experiments in Section 4.1. Contrary to the gap splits, the results for the tail splits are strongly influenced by outliers, as indicated in Figure 4.8. In particular, for *energy*, we observe a substantial decline in predictive performance. In line with the results from the toy data experiments in Section 4.1, a simple PNN seems to predict low variance outside the training data range. This makes sense as PNN cannot capture

		MC dropout	PBP	PNN	PNN-E	NLM	NLM-E	DNNC-R
boston	$\overline{\text{LogS}}$	**-3.53 (0.42)**	-5.03 (0.51)	-47.93 (29.33)	-4.33 (0.63)	-5.74 (0.76)	-3.85 (0.37)	-32.28 (15.91)
	$\overline{\text{CRPS}}$	3.19 (0.25)	3.82 (0.74)	3.69 (0.43)	**2.86 (0.18)**	3.47 (0.22)	3.08 (0.16)	3.57 (0.27)
	RMSE	6.21 (0.34)	7.68 (1.50)	6.92 (0.92)	**5.53 (0.32)**	6.58 (0.42)	6.01 (0.42)	6.13 (0.38)
	PICP	0.85 (0.04)	0.72 (0.03)	0.58 (0.04)	0.83 (0.03)	0.65 (0.03)	**0.86 (0.02)**	0.54 (0.03)
	MPIW	19.16 (2.36)	9.96 (0.77)	8.29 (0.80)	16.50 (2.76)	10.02 (0.90)	19.49 (3.02)	7.51 (0.28)
concrete	$\overline{\text{LogS}}$	-3.84 (0.10)	-3.97 (0.20)	-7.69 (1.99)	-3.69 (0.10)	-4.36 (0.29)	**-3.66 (0.14)**	-7.28 (0.92)
	$\overline{\text{CRPS}}$	5.91 (0.52)	**4.83 (0.34)**	5.27 (0.38)	5.31 (0.44)	4.86 (0.33)	5.59 (0.84)	5.49 (0.46)
	RMSE	11.68 (1.71)	**8.62 (0.50)**	9.30 (0.59)	10.59 (0.89)	8.85 (0.55)	9.65 (0.83)	8.96 (0.66)
	PICP	**0.88 (0.04)**	0.80 (0.03)	0.69 (0.03)	0.85 (0.02)	0.76 (0.03)	0.87 (0.02)	0.61 (0.04)
	MPIW	35.87 (4.67)	20.23 (0.38)	18.18 (1.69)	27.22 (1.79)	20.30 (2.02)	41.74 (11.12)	14.44 (0.91)
energy	$\overline{\text{LogS}}$	**-13.62 (3.57)**	-43.48 (37.86)	-796.52 (705.41)	-34.18 (17.29)	-119.61 (82.17)	-15.87 (9.51)	-74.83 (38.79)
	$\overline{\text{CRPS}}$	7.72 (3.74)	7.38 (4.01)	6.04 (2.88)	5.55 (2.78)	6.40 (3.04)	5.21 (2.90)	**4.51 (1.34)**
	RMSE	12.79 (4.84)	11.15 (4.94)	10.26 (4.08)	10.19 (4.15)	10.60 (4.27)	10.33 (4.83)	**7.34 (1.62)**
	PICP	0.61 (0.12)	0.61 (0.16)	0.38 (0.15)	0.59 (0.09)	0.42 (0.15)	**0.70 (0.12)**	0.36 (0.13)
	MPIW	11.27 (3.08)	10.02 (1.06)	3.37 (1.11)	6.55 (1.45)	5.41 (0.88)	12.14 (3.91)	4.32 (0.63)
kin8nm	$\overline{\text{LogS}}$	0.93 (0.04)	0.69 (0.05)	0.90 (0.05)	1.05 (0.04)	0.99 (0.04)	**1.08 (0.03)**	0.55 (0.05)
	$\overline{\text{CRPS}}$	0.06 (0.00)	0.06 (0.00)	0.06 (0.00)	**0.05 (0.00)**	**0.05 (0.00)**	**0.05 (0.00)**	0.06 (0.00)
	RMSE	0.11 (0.00)	0.12 (0.00)	0.11 (0.00)	0.10 (0.00)	0.10 (0.00)	**0.09 (0.00)**	0.10 (0.00)
	PICP	0.96 (0.01)	0.91 (0.01)	0.91 (0.01)	**0.97 (0.00)**	0.94 (0.01)	**0.97 (0.01)**	0.89 (0.01)
	MPIW	0.42 (0.02)	0.39 (0.00)	0.34 (0.01)	0.38 (0.01)	0.36 (0.01)	0.37 (0.01)	0.39 (0.01)
powerplant	$\overline{\text{LogS}}$	-3.40 (0.27)	-3.27 (0.25)	-3.39 (0.35)	**-3.05 (0.08)**	-3.31 (0.15)	-3.37 (0.15)	-4.77 (0.70)
	$\overline{\text{CRPS}}$	3.52 (0.55)	3.07 (0.43)	3.08 (0.33)	**2.87 (0.24)**	3.52 (0.44)	4.80 (1.02)	3.92 (0.22)
	RMSE	6.38 (1.05)	5.48 (0.70)	5.48 (0.52)	**5.22 (0.44)**	5.82 (0.89)	5.66 (0.66)	6.32 (0.39)
	PICP	0.88 (0.04)	0.88 (0.04)	0.89 (0.06)	0.95 (0.02)	0.91 (0.04)	**0.97 (0.01)**	0.74 (0.10)
	MPIW	19.94 (0.25)	15.74 (0.17)	17.86 (1.32)	20.00 (1.03)	27.13 (5.68)	56.94 (18.67)	19.76 (4.40)
wine	$\overline{\text{LogS}}$	-1.29 (0.06)	**-1.21 (0.05)**	-7.85 (3.21)	-1.38 (0.07)	-1.70 (0.13)	**-1.21 (0.04)**	-7.82 (3.96)
	$\overline{\text{CRPS}}$	0.56 (0.08)	**0.42 (0.02)**	0.52 (0.02)	0.66 (0.13)	0.68 (0.17)	0.44 (0.02)	0.44 (0.01)
	RMSE	1.22 (0.28)	0.78 (0.04)	0.93 (0.05)	1.22 (0.27)	0.89 (0.10)	0.80 (0.06)	**0.76 (0.01)**
	PICP	**0.91 (0.01)**	0.88 (0.01)	0.77 (0.03)	0.90 (0.01)	0.84 (0.02)	**0.91 (0.01)**	0.71 (0.02)
	MPIW	3.36 (0.36)	2.29 (0.02)	2.67 (0.45)	5.91 (2.07)	6.19 (3.01)	3.41 (0.38)	1.41 (0.06)
yacht	$\overline{\text{LogS}}$	-2.73 (0.39)	-7.92 (5.86)	-32.61 (20.77)	**-2.59 (0.38)**	-4.37 (1.40)	-2.78 (0.24)	NaN (NaN)
	$\overline{\text{CRPS}}$	1.89 (0.28)	**1.69 (0.65)**	3.51 (1.18)	2.62 (0.99)	3.78 (1.07)	2.74 (0.76)	6.10 (0.76)
	RMSE	3.43 (0.64)	**3.09 (1.05)**	5.97 (2.02)	4.59 (1.41)	6.39 (1.63)	3.80 (1.13)	10.47 (1.06)
	PICP	0.88 (0.07)	0.87 (0.07)	0.52 (0.08)	0.93 (0.05)	0.84 (0.10)	**0.98 (0.02)**	0.47 (0.07)
	MPIW	14.92 (2.77)	7.57 (0.84)	15.59 (11.25)	21.77 (12.22)	33.64 (17.24)	32.83 (12.41)	15.11 (1.68)

Table 4.5: Predictive results for UCI datasets with **tail splits**. We depict the mean $\overline{\text{LogS}}$, $\overline{\text{CRPS}}$, RMSE, PICP, and MPIW for all methods and datasets with standard errors in parentheses. The method with the best mean score per dataset is depicted in bold. For RMSE and $\overline{\text{CRPS}}$ lower values correspond to better performance, for $\overline{\text{LogS}}$ and PICP high values represent accurate predictions.

epistemic uncertainty and thus underestimates the variance outside for inputs outside the training range. For some datasets, NLM suffers from the same problem and has low predictive epistemic uncertainty, compare MPIW for *concrete* in Table 4.5. Further, the results for DNNC-R indicate that for some tail splits, the distribution of the response data is largely different for train and test sets. As DNNC-R relies heavily on the estimated marginal distribution of the targets, its predictive densities put low probability mass on targets outside the training range. We thus observe a low $\overline{\text{LogS}}$ for DNNC-R for most datasets. Especially for *yacht*, we encountered a $\overline{\text{LogS}}$ of minus infinity for DNNC-R.

4.3 DNNC with Standard Fit NN

The marginally calibrated deep distributional regression – outlined in Section 3.5 – requires the training of the underlying neural network with the transformation of the targets by $\Phi^{-1}(\hat{F_Y}(\boldsymbol{y}))$. In this section, we summarize our empirical findings that this transformation might not be necessary. If our findings hold, one could apply the marginally calibrated deep distributional regression as a post-calibration method. Concretely, we could use it for applications where the training process is costly and we want to avoid retraining the model with transformed targets. Further, we might use the same neural network for applications with different sub-problems for which the distribution of the response varies. Imagine we have trained a model that predicts the steering angle of an autonomous car based on image inputs. It could be possible to train one model with all available data and then employ the marginally calibrated method for different sub-problems – such as highway driving or city traffic – individually. We could train on all data that is relevant to the task and

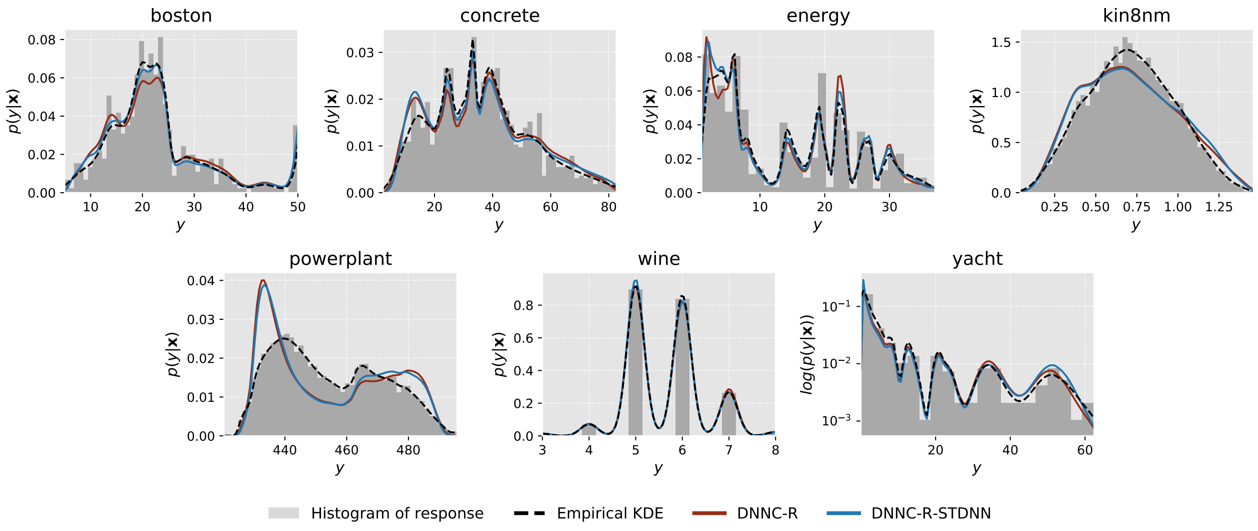


Figure 4.9: DNNC with standard fit: marginal calibration for UCI datasets.

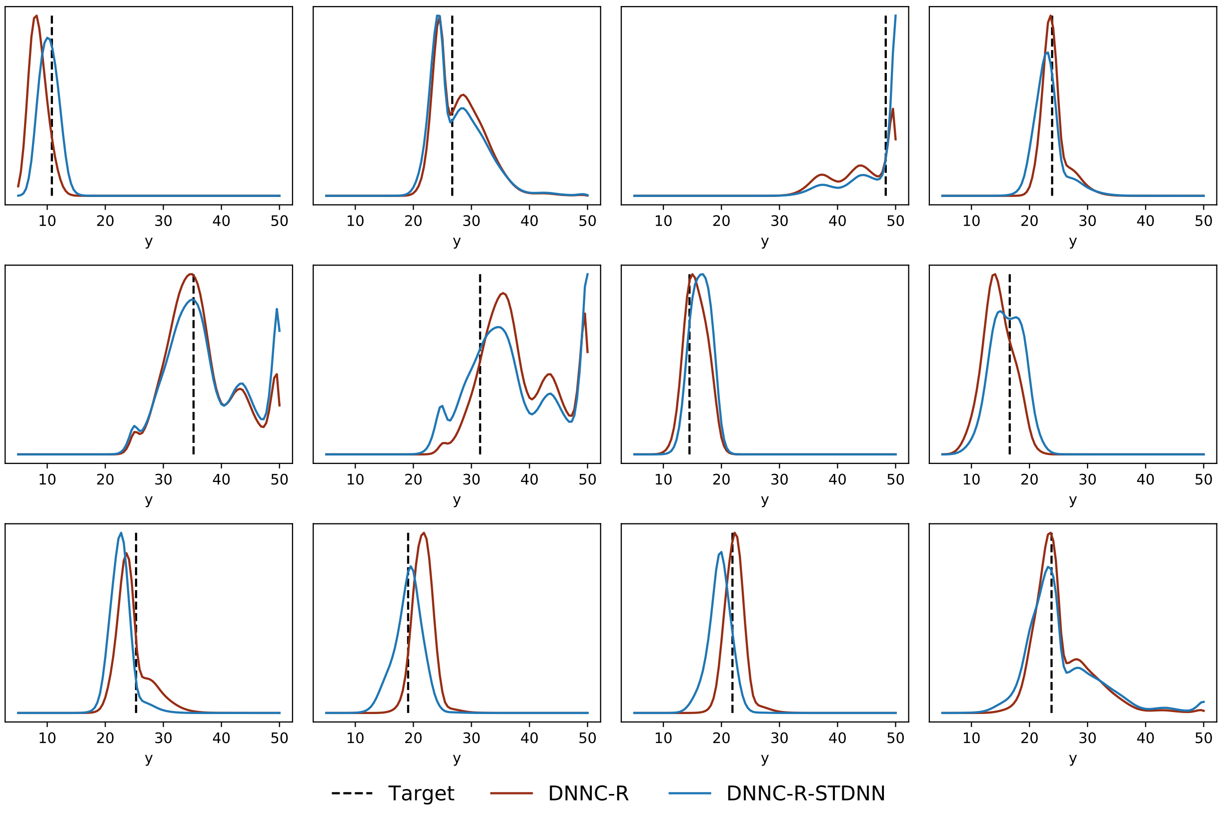


Figure 4.10: DNNC with standard fit: predictive densities for Boston Housing. We train both methods with the full data and show the predictive densities for twelve randomly chosen targets on the y-axis. The red lines indicate the densities of a fit where we transformed the targets by the empirical CDF: $\tilde{\boldsymbol{z}} = \Phi^{-1}(\hat{F}_Y(\boldsymbol{y}))$. In blue we depict densities where we fit the NN with standardized targets: $\boldsymbol{z} = \sigma_y^{-1}(\boldsymbol{y} - \mu_y)$. The true scalar targets are shown as dashed black lines.

still retain the marginal calibration property of the approach. Another use-case could be to easily compare multiple possible KDEs for the response in some regression problem.

Our results suggest that one could train the underlying neural network for regression in the most standard form. I.e, we fit the NN with standardized inputs $\frac{\boldsymbol{X}-\mu_x}{\sigma_x}$ and targets $\frac{\boldsymbol{y}-\mu_y}{\sigma_y}$ and linear activation function in the output layer. This section compares the copula approach (ridge prior) with and without transforming the targets by the marginal CDF. We refer to the former again by DNNC-R and to the latter by DNNC-R-STDNN. Interestingly, for both approaches, the targets are standard normally distributed.

Figure 4.9 displays the marginal calibration of both methods for all UCI datasets. The methods perform similarly. Especially, both methods struggle with the large datasets *kin8nm* and *power-plant*. For *boston*, DNNC-R-STDNN is slightly better marginally calibrated.

Figure 4.10 shows the predictive densities of both methods – trained on full data – for twelve randomly chosen targets of *boston*. Both approaches yield similar predictive densities that are flexible functions of the inputs.

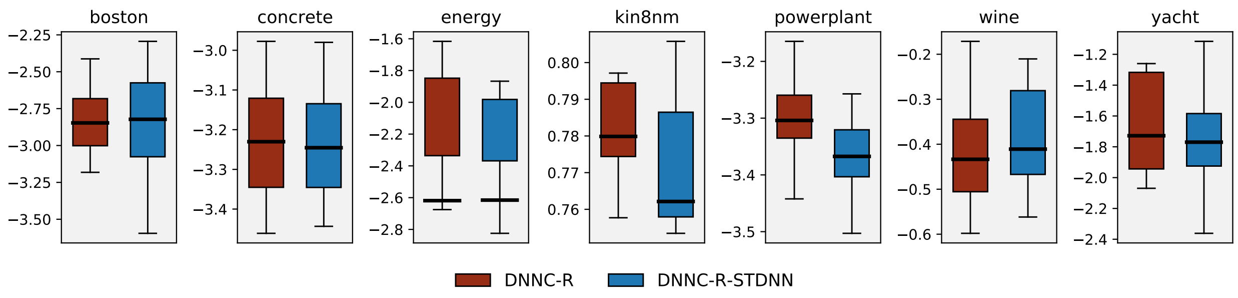
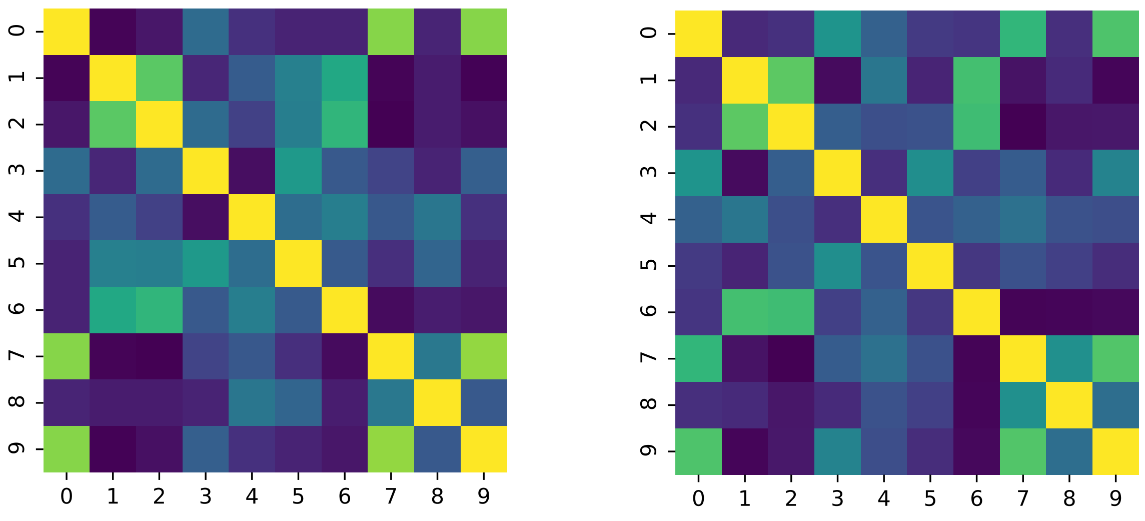


Figure 4.11: DNNC with standard fit: predictive $\overline{\text{LogS}}$ for UCI datasets. We show the results for ten random 90/10 splits.



(a) Targets: $\tilde{\boldsymbol{z}} = \Phi^{-1}(\hat{F}_Y(\boldsymbol{y}))$ (b) Targets: $\boldsymbol{z} = \sigma_y^{-1}(\boldsymbol{y} - \mu_y)$

Figure 4.12: DNNC with standard fit: identified dependency structures from NN hidden layer outputs. We show the dependence for ten randomly chosen targets and plot the matrix $R(\boldsymbol{X}, \boldsymbol{\theta})$. The neural network was trained to minimize the MSE and with a linear activation function in the output layer. We standardized the input values $\boldsymbol{x}$. (a) We transformed the targets with the empirical CDF: $\tilde{\boldsymbol{z}} = \Phi^{-1}(\hat{F}_Y(\boldsymbol{y}))$. (b) We fit the NN with standardized targets: $\boldsymbol{z} = \sigma_y^{-1}(\boldsymbol{y} - \mu_y)$. We see a similar dependence structures for both transformations of the targets.

We show the predictive $\overline{\text{LogS}}$ for ten random splits of the UCI datasets in Figure 4.11. The performance is similar whether we fit the underlying NN with or without transforming the targets by the empirical marginal CDF.

Regardless of the transformation of the targets, the *implicit* copula is similar for both approaches. This argument is supported by Figure 4.12, where we plot a heat-map of the matrix $R(\boldsymbol{X}, \boldsymbol{\theta})$ – which is at the heart of the copula density – for ten randomly selected targets. For both methods, the neural network is initialized randomly. During training, the last hidden layers learn different transformations of the inputs. Thus, we see similar but not identical dependence structures.

We have provided empirical evidence that it may be possible to use the marginally calibrated deep distributional regression with neural networks where inputs and targets are standardized to have zero mean and unit variance. This empirical finding is a topic for further research.

5 Conclusion

In this thesis, we benchmarked various methods that allow for uncertainty modeling in neural networks. We studied some basic characteristics with toy data, and obtained calibration and robust predictive performance results on standard UCI regression datasets. For the latter, we used a variety of metrics to judge the models' quality to predict in-between and out-of-range uncertainty. We summarize our main findings as:

- For applications with a large amount of data, modeling **aleatoric** uncertainty is relevant as epistemic uncertainty diminishes.
- Small datasets or out-of-distribution inputs require the ability to capture **epistemic** uncertainty arising from a lack of data.
- Good in-sample calibration does not necessarily translate into good predictive performance.
- The last-layer inference method NLM performs competitive for random splits of the data but struggles with gap and tail splits, reflecting the cost for the reduced computational complexity.
- DNNC-R is sensitive to the estimation of the marginal distribution of the response. Nevertheless, a meaningful KDE can lead to a good performance.

We provided empirical evidence that one can apply the DNNC approach to a NN that has been fitted with standardized inputs and outputs. This enables the use of DNNC without retraining a model with transformed targets. Further, one might combine the last hidden layer outputs of a single NN – trained with all available data – with different marginal distributions of the response for different sub-problems.

Some adaptations of methods that we benchmarked in this thesis would be compelling to study in more detail. First, it might be fruitful to use the heteroscedastic implicit copula of Smith and Klein (2021) that builds on a NN that learns mean and heteroscedastic aleatoric variance as functions of inputs. We further would like to gauge the performance of an adaption of the DNNC method that performs multinomial classification. This seems to be a promising application (compare the results obtained for the UCI dataset *wine* in Section 4.2.4). Second, we could place different priors on the neural linear model and perform approximate Bayesian inference with MCMC or VI. Third, benchmarking the last-layer methods (NLM and DNNC) in deeper and more complex architectures would be crucial to assess their usability for practical applications. Fourth, one might evaluate the performance of the models that output a mixture of Gaussians without approximating the predictive density with a univariate Gaussian.

We believe that specialized soft- and hardware will drive both research and adoption of Bayesian neural networks. For example, we think that novel ideas can be built on top of software libraries such as TensorFlow Probability (Abadi et al., 2015) or Edwards (Tran et al., 2016), enabling easier adaptions and modifications. Specialized hardware such as Gaussian Random Number Generators (Cai et al., 2018) may substantially accelerate the training of and inference in (variational) Bayesian neural networks.

It would be helpful to tackle the problem of quickly computing various models (or various sets of weights of a single model). This would enable the use of ensemble models in real-time applications and address a fundamental problem of using MCMC for BNNs. As outlined in Section 3.2.1, it is – given the currently available software – hardly practical to sample, e.g., 1,000 sets of weights and use them to predict the distribution of some outcome. A solution that allows for a fast evaluation of ensemble members would probably lead to extensive research on obtaining the MCMC samples of the weights in the first place, thus making MCMC for BNNs feasible.

We believe the study of uncertainty in machine learning is of vital importance and is likely to gain in popularity in the upcoming years and decades. Accurate uncertainty awareness of machine learning methods might further facilitate their adoption in our everyday life, as it enables us to both automate frequent and easy decisions and delegate difficult ones to human experts.

Bibliography

Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* URL: `https://www.tensorflow.org/`.

Betancourt, M. (2017). "A Conceptual Introduction to Hamiltonian Monte Carlo". In: *arXiv preprint.* arXiv: `1701.02434`.

Bishop, C. M. (1994). *Mixture density networks.* Tech. rep. NCRG/94/004, Neural Computing Research Group, Aston University.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning.* New York: Springer.

Blundell, C., J. Cornebise, K. Kavukcuoglu, and D. Wierstra (2015). "Weight uncertainty in neural networks". In: *Proceedings of the 32nd International Conference on Machine Learning*, pp. 1613–1622. arXiv: `1505.05424`.

Brach, K., B. Sick, and O. Dürr (2020). "Single Shot MC Dropout Approximation". In: *arXiv preprint.* arXiv: `2007.03293 [cs.LG]`.

Cai, R., A. Ren, N. Liu, C. Ding, L. Wang, X. Qian, M. Pedram, and Y. Wang (2018). "VIBNN: Hardware acceleration of Bayesian neural networks". In: *ACM SIGPLAN Notices* 53.2, pp. 476–488. DOI: `10.1145/3173162.3173212`. arXiv: `1802.00822`.

Carpenter, B., A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. A. Brubaker, J. Guo, P. Li, and A. Riddell (2017). "Stan: A Probabilistic Programming Language". In: *Journal of Statistical Software* 76.1, pp. 1–32. DOI: `10.18637/jss.v076.i01`.

Chen, T., E. Fox, and C. Guestrin (2014). "Stochastic Gradient Hamiltonian Monte Carlo". In: *Proceedings of the 31st International Conference on Machine Learning*, pp. 1683–1691.

Chollet, F. et al. (2015). *Keras.* URL: `httos://www.keras.io`.

Chollet, F. (2017). *Deep Learning with Python.* Manning Publications Co.

Chua, K., R. Calandra, R. McAllister, and S. Levine (2018). "Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models". In: *arXiv preprint.* arXiv: `1805.12114 [cs.LG]`.

Cowles, M. K. and B. P. Carlin (1996). "Markov Chain Monte Carlo Convergence Diagnostics: A Comparative Review". In: *Journal of the American Statistical Association* 91.434, pp. 883–904. DOI: `10.1080/01621459.1996.10476956`.

Denker, J. and Y. Lecun (1991). "Transforming Neural-Net Output Levels to Probability Distributions". In: *Advances in Neural Information Processing Systems 3*, pp. 853–859.

Depeweg, S., J. M. Hernández-Lobato, F. Doshi-Velez, and S. Udluft (2017). "Decomposition of Uncertainty in Bayesian Deep Learning for Efficient and Risk-sensitive Learning". In: *arXiv preprint.* arXiv: `1710.07283 [stat.ML]`.

Dua, D. and C. Graff (2017). *UCI Machine Learning Repository.* URL: `http://archive.ics.uci.edu/ml`.

Duane, S., A. D. Kennedy, B. J. Pendleton, and D. Roweth (1987). "Hybrid Monte Carlo". In: *Physics Letters B* 195.2, pp. 216–222. DOI: `10.1016/0370-2693(87)91197-X`.

Duchi, J., E. Hazan, and Y. Singer (2011). "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12.7, pp. 2121–2159.

Foong, A. Y. K., D. R. Burt, Y. Li, and R. E. Turner (2019a). "On the Expressiveness of Approximate Inference in Bayesian Neural Networks". In: *arXiv preprint.* arXiv: `1909.00719`.

Foong, A. Y. K., Y. Li, J. M. Hernández-Lobato, and R. E. Turner (2019b). "'In-Between' Uncertainty in Bayesian Neural Networks". In: *arXiv preprint.* arXiv: `1906.11537`.

Gal, Y. (2016). "Uncertainty in Deep Learning". PhD thesis. University of Cambridge.

Gal, Y. and Z. Ghahramani (2016). "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning". In: *Proceedings of The 33rd International Conference on Machine Learning*, pp. 1050–1059. arXiv: `1506.02142`.

Gelfand, A. E. and A. F. Smith (1990). "Sampling-Based Approaches to Calculating Marginal Densities". In: *Journal of the American Statistical Association* 85.410, pp. 398–409. DOI: `10.1080/01621459.1990.10476213`.

Gelman, A., J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, and D. B. Rubin (2013). *Bayesian Data Analysis.* Boca Raton: CRC Press.

Geman, S. and D. Geman (1984). "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-6.6, pp. 721–741. DOI: `10.1109/TPAMI.1984.4767596`.

Gneiting, T., F. Balabdaoui, and A. E. Raftery (2007). “Probabilistic forecasts, calibration and sharpness”. In: *Journal of the Royal Statistical Society B* 69.2, pp. 243–268.

Gneiting, T. and A. E. Raftery (2007). “Strictly proper scoring rules, prediction, and estimation”. In: *Journal of the American Statistical Association* 102.477, pp. 359–378. DOI: `10.1198/016214506000001437`.

Good, I. J. (1952). “Rational Decisions”. In: *Journal of the Royal Statistical Society B* 14.1, pp. 107–114.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. Cambridge, MA: MIT Press. URL: `http://www.deeplearningbook.org`.

Graves, A. (2011). “Practical Variational Inference for Neural Networks”. In: *Advances in Neural Information Processing Systems*. Vol. 24.

Hastie, T., R. Tibshirani, and J. Friedman (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer. DOI: `10.1198/jasa.2004.s339`.

Hastings, W. K. (1970). “Monte Carlo Sampling Methods Using Markov Chains and Their Applications”. In: *Biometrika* 57.1, pp. 97–109. DOI: `10.2307/2334940`.

Hernández-Lobato, J. M. and R. P. Adams (2015). “Probabilistic backpropagation for scalable learning of Bayesian neural networks”. In: *Proceedings of the 32nd International Conference on Machine Learning*, pp. 1861–1869.

Hinton, G. E. and D. van Camp (1993). “Keeping the Neural Networks Simple by Minimizing the Description Length of the Weights”. In: *Proceedings of the Sixth Annual Conference on Computational Learning Theory*. COLT ’93, pp. 5–13. DOI: `10.1145/168304.168306`.

Hoffman, M. D. and A. Gelman (2014). “The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo”. In: *Journal of Machine Learning Research* 15.1, pp. 1593–1623. arXiv: `1111.4246`.

Hora, S. C. (1996). “Aleatory and epistemic uncertainty in probability elicitation with an example from hazardous waste management”. In: *Reliability Engineering & System Safety* 54.2-3, pp. 217–223.

Ioffe, S. (2017). “Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models”. In: *arXiv preprint*. arXiv: `1702.03275 [cs.LG]`.

Ioffe, S. and C. Szegedy (2015). “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *Proceedings of the 32nd International Conference on Machine Learning*, pp. 448–456. arXiv: `1502.03167`.

Kendall, A. and Y. Gal (2017). "What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision?" In: *Advances in Neural Information Processing Systems* 30. arXiv: 1703.04977.

Kingma, D. P. and M. Welling (2014). "Auto-Encoding Variational Bayes". In: *arXiv preprint.* arXiv: 1312.6114 [stat.ML].

Kingma, D. P. and J. L. Ba (2015). "Adam: A method for stochastic optimization". In: *3rd International Conference on Learning Representations.* arXiv: 1412.6980.

Kiureghian, A. D. and O. Ditlevsen (2009). "Aleatory or epistemic? Does it matter?" In: *Structural Safety* 31.2, pp. 105–112. DOI: 10.1016/j.strusafe.2008.06.020.

Klein, N. and T. Kneib (2016). "Scale-Dependent Priors for Variance Parameters in Structured Additive Distributional Regression". In: *Bayesian Analysis* 11.4, pp. 1071–1106. DOI: 10.1214/15-BA983.

Klein, N., D. J. Nott, and M. S. Smith (2021). "Marginally Calibrated Deep Distributional Regression". In: *Journal of Computational and Graphical Statistics* 30.2, pp. 467–483. DOI: 10.1080/10618600.2020.1807996.

Klein, N. and M. S. Smith (2019). "Implicit Copulas from Bayesian Regularized Regression Smoothers". In: *Bayesian Analysis* 14.4, pp. 1143–1171. DOI: 10.1214/18-BA1138.

Korattikara, A., V. Rathod, K. P. Murphy, and M. Welling (2015). "Bayesian dark knowledge". In: *Advances in Neural Information Processing Systems* 28. arXiv: 1506.04416.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in neural information processing systems* 25, pp. 1097–1105.

Kullback, S. (1959). *Information Theory and Statistics.* New York: Wiley. DOI: 10.2307/3613211.

Kullback, S. and R. A. Leibler (1951). "On Information and Sufficiency". In: *The Annals of Mathematical Statistics* 22.1, pp. 79–86. DOI: 10.1214/aoms/1177729694.

Lakshminarayanan, B., A. Pritzel, and C. Blundell (2016). "Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles". In: *arXiv preprint.* arXiv: 1612.01474.

Lee, J., Y. Bahri, R. Novak, S. S. Schoenholz, J. Pennington, and J. Sohl-Dickstein (2017). "Deep Neural Networks as Gaussian Processes". In: *arXiv preprint.* arXiv: 1711.00165.

MacKay, D. J. C. (1992). "A Practical Bayesian Framework for Backpropagation Networks". In: *Neural Computation* 4.3, pp. 448–472.

Martin, G. M., D. T. Frazier, and C. P. Robert (2020). "Computing Bayes: Bayesian Computation from 1763 to the 21st Century". In: *arXiv preprint.* arXiv: 2004.06425.

Matheson, J. E. and R. L. Winkler (1976). “Scoring rules for continuous probability distributions”. In: *Management Science* 22.10, pp. 1087–1096.

Matthews, A. G. G., M. Van Der Wilk, T. Nickson, K. Fujii, A. Boukouvalas, P. León-Villagrá, Z. Ghahramani, and J. Hensman (2017). “GPflow: A Gaussian Process Library using TensorFlow”. In: *Journal of Machine Learning Research* 18.40, pp. 1–6.

Metropolis, N., A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller (1953). “Equation of State Calculations by Fast Computing Machines”. In: *The Journal of Chemical Physics* 21.6, pp. 1087–1092. DOI: 10.1063/1.1699114.

Mikolov, T., K. Chen, G. Corrado, and J. Dean (2013). “Efficient Estimation of Word Representations in Vector Space”. In: *arXiv preprint.* arXiv: 1301.3781 [cs.CL].

Minka, T. P. (2001). “A family of algorithms for approximate Bayesian inference”. PhD thesis. Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science. arXiv: arXiv:1011.1669v3.

Moberg, J., L. Svensson, J. Pinto, and H. Wymeersch (2019). “Bayesian Linear Regression on Deep Representations”. In: *arXiv preprint.* arXiv: 1912.06760 [cs.LG].

Murphy, K. P. (2012). *Machine Learning, a Probabilistic Perspective.* Cambridge, MA: MIT Press.

Nair, V. and G. E. Hinton (2010). “Rectified Linear Units Improve Restricted Boltzmann Machines Vinod Nair”. In: *Proceedings of the 27th International Conference on Machine Learning*, pp. 807–814.

Neal, R. M. (1992). *Bayesian training of backpropagation networks by the hybrid Monte Carlo method.* Tech. rep. CRG-TR-92-1, Dept. of Computer Science, University of Toronto.

Neal, R. M. (1993). *Probabilistic Inference Using Markov Chain Monte Carlo Methods.* Tech. rep. CRG-TR-93-1, Dept. of Computer Science, University of Toronto.

Neal, R. M. (1995). “Bayesian Learning for Neural Networks”. PhD thesis. Dept. of Computer Science, University of Toronto.

Neal, R. M. (2012). “MCMC using Hamiltonian dynamics”. In: *arXiv preprint.* arXiv: 1206.1901 [stat.CO].

Nelsen, R. B. (2006). *An Introduction to Copulas.* New York: Springer. DOI: 10.2307/2669568.

Nix, D. A. and A. S. Weigend (1994). “Estimating the mean and variance of the target probability distribution”. In: *Proceedings of 1994 IEEE International Conference on Neural Networks.* Vol. 1. IEEE, pp. 55–60. DOI: 10.1109/icnn.1994.374138.

Ober, S. W. and C. E. Rasmussen (2019). “Benchmarking the Neural Linear Model for Regression”. In: *arXiv preprint.* arXiv: 1912.08416 [stat.ML].

Ovadia, Y., E. Fertig, J. Ren, Z. Nado, D. Sculley, S. Nowozin, J. V. Dillon, B. Lakshminarayanan, and J. Snoek (2019). “Can You Trust Your Model’s Uncertainty? Evaluating Predictive Uncertainty Under Dataset Shift”. In: *arXiv preprint.* arXiv: `1906.02530 [stat.ML]`.

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. (2019). “Pytorch: An imperative style, high-performance deep learning library”. In: *arXiv preprint.* arXiv: `1912.01703`.

Pratt, H., F. Coenen, D. M. Broadbent, S. P. Harding, and Y. Zheng (2016). “Convolutional Neural Networks for Diabetic Retinopathy”. In: *Procedia Computer Science* 90, pp. 200–205. DOI: `https://doi.org/10.1016/j.procs.2016.07.014`.

Rasmussen, C. E. and C. K. I. Williams (2006). *Gaussian Processes for Machine Learning.* Cambridge, MA: MIT Press.

Riquelme, C., G. Tucker, and J. Snoek (2018). “Deep Bayesian Bandits Showdown: An Empirical Comparison of Bayesian Deep Networks for Thompson Sampling”. In: *arXiv preprint.* arXiv: `1802.09127 [stat.ML]`.

Rissanen, J. (1994). “Stochastic complexity and universal modeling”. In: *Proceedings of 1994 IEEE International Symposium on Information Theory*, pp. 3–. DOI: `10.1109/isit.1994.394968`.

Ruder, S. (2016). “An overview of gradient descent optimization algorithms”. In: *arXiv preprint.* arXiv: `1609.04747`.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). “Learning representations by back-propagating errors”. In: *Nature* 323.6088, pp. 533–536. DOI: `10.1038/323533a0`.

Salvatier, J., T. V. Wiecki, and C. Fonnesbeck (2016). “Probabilistic programming in Python using PyMC3”. In: *PeerJ Computer Science* 2, e55.

Scott, D. W. (1992). *Multivariate density estimation and visualization.* New York: Wiley. DOI: `10.1002/9780470316849`.

Shimazaki, H. and S. Shinomoto (2010). “Kernel bandwidth optimization in spike rate estimation”. In: *Journal of Computational Neuroscience* 29.1, pp. 171–182. DOI: `10.1007/s10827-009-0180-4`.

Sklar, A. (1959). “Fonctions de Répartition à n Dimensions et Leurs Marges”. In: *Publications de L’Institut de Statistique de L’Université de Paris* 8, pp. 229–231.

Smith, M. S. and N. Klein (2021). “Bayesian Inference for Regression Copulas”. In: *Journal of Business & Economic Statistics* 39.3, pp. 712–728. DOI: `10.1080/07350015.2020.1721295`.

Snoek, J., O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. M. A. Patwary, Prabhat, and R. P. Adams (2015). “Scalable Bayesian Optimization Using Deep Neural Networks”. In: *arXiv preprint.* arXiv: `1502.05700 [stat.ML]`.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014). “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56, pp. 1929–1958.

Sun, S., G. Zhang, J. Shi, and R. Grosse (2019). “Functional Variational Bayesian Neural Networks”. In: *arXiv preprint.* arXiv: `1903.05779`.

Sutskever, I., J. Martens, G. Dahl, and G. Hinton (2013). “On the importance of initialization and momentum in deep learning”. In: *Proceedings of the 30th International Conference on Machine Learning*, pp. 1139–1147.

Theano Development Team (2016). “Theano: A Python framework for fast computation of mathematical expressions”. In: *arXiv preprint.* arXiv: `1605.02688`.

Tran, D., A. Kucukelbir, A. B. Dieng, M. Rudolph, D. Liang, and D. M. Blei (2016). “Edward: A library for probabilistic modeling, inference, and criticism”. In: *arXiv preprint.*

Unger, D. A. (1985). “A method to estimate the continuous ranked probability score”. In: *Ninth Conf. on Probability and Statistics in Atmospheric Sciences*, pp. 206–213.

Waterhouse, S., D. MacKay, and A. Robinson (1996). “Bayesian Methods for Mixtures of Experts”. In: *Advances in Neural Information Processing Systems* 8.

Welling, M. and Y. W. Teh (2011). “Bayesian Learning via Stochastic Gradient Langevin Dynamics”. In: *Proceedings of the 28th International Conference on Machine Learning*, pp. 681–688.

Yao, Y., A. Vehtari, D. Simpson, and A. Gelman (2018). “Yes, but Did It Work?: Evaluating Variational Inference”. In: *arXiv preprint.* arXiv: `1802.02538 [stat.ML]`.

Zhang, R., C. Li, J. Zhang, C. Chen, and A. G. Wilson (2019). “Cyclical Stochastic Gradient MCMC for Bayesian Deep Learning”. In: *arXiv preprint.* arXiv: `1902.03932`.

A Experimental Setup

A.1 Settings for Toy Experiments

We used the following settings and hyperparameters for our toy experiments:

- **MC dropout**: we find the optimal dropout rate by grid search in $[0.005, 0.01, 0.05, 0.1]$ with a validation size of 20%. As dropout needs more training iterations to converge, we used 10x the epochs as for the other methods (Gal and Ghahramani, 2016).
- **NLM**: We choose $\tau^2 = 2$ for the prior.
- **PNN-E/NLM-E**: Ensembles models consist of five members as in the experimental setup of Lakshminarayanan et al. (2016).
- **DNNC**: we sample 10,000 iterations with MCMC and discard the first 5,000 (burn-in). We set the scale parameter $r = 2.5$ for the scale-dependent hyper-prior $p(\tau^2)$ as in Klein et al. (2021). We use the MATLAB code of Klein et al. (2021) for the KDE estimation and the MCMC sampling.

A.2 Software and Hardware

Software

We implemented all methods in Python3. For the neural networks we used TensorFlow (Abadi et al., 2015) and Keras (Chollet et al., 2015).

We ran all experiments with the Adam (Kingma and Ba, 2015) optimizer using the following default settings from Keras (Chollet et al., 2015):

- beta_1 = 0.9,
- beta_2 = 0.999,
- epsilon = 1e-07,

- amsgrad = False.

See also `https://keras.io/api/optimizers/adam/` for details on the default parameters used in Keras.

The Python and packages versions we used are depicted in Figure A.1.

```
Python implementation: CPython
Python version       : 3.7.10
IPython version      : 5.5.0
numpy                : 1.19.5
scipy                : 1.4.1
tqdm                 : 4.41.1
matplotlib           : 3.2.2
pandas               : 1.1.5
gpflow               : 2.2.1
properscoring        : 0.1
tensorflow           : 2.5.0
Theano-PyMC          : 1.1.2
```

Figure A.1: Software/package versions.

Hardware

We ran all our experiments on Google Colaboratory in an interactive Jupyter Notebook with GPU support, see Figure A.2 for details.

```
Compiler    : GCC 7.5.0
OS          : Linux
Release     : 5.4.109+
Machine     : x86_64
Processor   : x86_64
CPU cores   : 4
Architecture: 64bit
```

Figure A.2: Hardware overview.

A.3 Implementation Details

Gaussian process with ReLU kernel We implemented a Gaussian Process with GPflow (Matthews et al., 2017) and used the ReLU-Kernel from Foong et al. (2019b). The code can be found at `https://github.com/cambridge-mlg/expressiveness-approx-bnns`.

Monte Carlo dropout We implemented this method ourself.

Probabilistic backpropagation We used the Theano (Theano Development Team, 2016) implementation of Hernández-Lobato and Adams (2015). The code can be found at `https://github.com/HIPS/Probabilistic-Backpropagation`.

Probabilistic neural network We implemented this method ourself.

Deep ensemble We implemented this method ourself.

Neural linear model We implemented this method ourself.

Marginally calibrated deep distributional regression We implemented this method ourself based on the MATLAB code of Klein et al. (2021).

B UCI Results

B.1 In-sample UCI Results

Tables B.1, B.2, and B.3 depict the in-sample mean values for all metrics ($\overline{\text{LogS}}$, $\overline{\text{CRPS}}$, RMSE, PICP, and MPIW) with standard errors in parentheses for the random, gap, and tail splits, respectively. Figures B.1, B.2, and B.3 show the predictive $\overline{\text{LogS}}$, $\overline{\text{CRPS}}$, and RMSE values graphically via box-plots for the random, gap, and tail splits, respectively.

		MC dropout	PBP	PNN	PNN-E	NLM	NLM-E	DNNC-R
boston	$\overline{\text{LogS}}$	**-1.91 (0.04)**	-2.27 (0.01)	-2.00 (0.02)	-1.97 (0.01)	-1.95 (0.01)	-1.96 (0.01)	-2.18 (0.01)
	$\overline{\text{CRPS}}$	1.12 (0.05)	1.26 (0.01)	1.20 (0.02)	1.10 (0.02)	1.12 (0.02)	**1.09 (0.01)**	1.33 (0.02)
	RMSE	2.66 (0.09)	**2.34 (0.02)**	2.89 (0.06)	2.56 (0.05)	2.74 (0.06)	2.59 (0.05)	2.68 (0.04)
	PICP	**0.99 (0.00)**	0.96 (0.00)	0.96 (0.00)	**0.99 (0.00)**	0.98 (0.00)	**0.99 (0.00)**	0.96 (0.00)
	MPIW	10.97 (0.81)	10.02 (0.06)	8.55 (0.13)	10.19 (0.29)	9.24 (0.21)	10.22 (0.21)	12.17 (0.24)
concrete	$\overline{\text{LogS}}$	**-2.70 (0.01)**	-3.02 (0.00)	-2.85 (0.02)	-2.75 (0.01)	-2.77 (0.01)	-2.77 (0.01)	-2.97 (0.01)
	$\overline{\text{CRPS}}$	**2.12 (0.03)**	2.74 (0.01)	2.63 (0.04)	2.25 (0.01)	2.39 (0.02)	2.31 (0.02)	2.60 (0.01)
	RMSE	**4.11 (0.04)**	4.95 (0.02)	5.14 (0.07)	4.25 (0.02)	4.69 (0.04)	4.45 (0.03)	4.64 (0.02)
	PICP	**0.99 (0.00)**	0.95 (0.00)	0.97 (0.00)	**0.99 (0.00)**	0.98 (0.00)	**0.99 (0.00)**	0.95 (0.00)
	MPIW	18.90 (0.36)	20.67 (0.06)	19.32 (0.48)	19.87 (0.25)	20.02 (0.41)	21.29 (0.28)	21.28 (0.18)
energy	$\overline{\text{LogS}}$	-1.58 (0.03)	-2.34 (0.05)	-1.55 (0.07)	-1.43 (0.02)	**-1.31 (0.03)**	-1.39 (0.02)	-1.81 (0.03)
	$\overline{\text{CRPS}}$	0.95 (0.05)	1.41 (0.10)	0.98 (0.03)	0.92 (0.01)	0.90 (0.02)	**0.89 (0.01)**	0.98 (0.02)
	RMSE	2.98 (0.13)	3.11 (0.18)	2.96 (0.06)	2.93 (0.05)	2.95 (0.05)	2.80 (0.05)	**2.32 (0.06)**
	PICP	**1.00 (0.00)**	0.93 (0.01)	0.95 (0.01)	**1.00 (0.00)**	0.99 (0.00)	**1.00 (0.00)**	0.98 (0.00)
	MPIW	9.44 (0.64)	10.56 (0.10)	7.02 (0.42)	7.57 (0.15)	6.99 (0.25)	8.00 (0.22)	9.58 (0.24)
kin8nm	$\overline{\text{LogS}}$	1.16 (0.01)	0.92 (0.00)	1.19 (0.01)	1.26 (0.00)	1.27 (0.00)	**1.30 (0.00)**	0.85 (0.01)
	$\overline{\text{CRPS}}$	**0.04 (0.00)**	0.05 (0.00)	**0.04 (0.00)**	**0.04 (0.00)**	**0.04 (0.00)**	**0.04 (0.00)**	0.05 (0.00)
	RMSE	0.08 (0.00)	0.10 (0.00)	0.08 (0.00)	**0.07 (0.00)**	0.08 (0.00)	**0.07 (0.00)**	0.09 (0.00)
	PICP	0.98 (0.00)	0.96 (0.00)	0.95 (0.00)	**0.99 (0.00)**	0.97 (0.00)	**0.99 (0.00)**	0.96 (0.00)
	MPIW	0.37 (0.01)	0.41 (0.00)	0.31 (0.00)	0.35 (0.00)	0.32 (0.00)	0.33 (0.00)	0.45 (0.00)
powerplant	$\overline{\text{LogS}}$	-2.82 (0.00)	-2.82 (0.00)	-2.82 (0.01)	-2.79 (0.00)	-2.79 (0.00)	**-2.78 (0.00)**	-3.28 (0.01)
	$\overline{\text{CRPS}}$	2.27 (0.01)	2.22 (0.00)	2.26 (0.02)	2.19 (0.00)	2.18 (0.00)	**2.16 (0.00)**	3.04 (0.03)
	RMSE	4.14 (0.02)	4.06 (0.01)	4.14 (0.03)	4.03 (0.01)	4.01 (0.01)	**3.98 (0.01)**	4.58 (0.02)
	PICP	**0.97 (0.00)**	0.96 (0.00)	0.96 (0.00)	**0.97 (0.00)**	**0.97 (0.00)**	**0.97 (0.00)**	0.95 (0.00)
	MPIW	17.71 (0.19)	16.06 (0.02)	16.47 (0.24)	16.67 (0.08)	16.55 (0.18)	16.39 (0.10)	30.06 (0.45)
wine	$\overline{\text{LogS}}$	-0.75 (0.02)	-0.89 (0.00)	-0.73 (0.01)	-0.72 (0.00)	-0.74 (0.01)	-0.74 (0.01)	**-0.07 (0.00)**
	$\overline{\text{CRPS}}$	0.37 (0.04)	0.33 (0.00)	0.31 (0.00)	0.32 (0.01)	0.32 (0.01)	0.33 (0.01)	**0.27 (0.00)**
	RMSE	0.58 (0.01)	0.59 (0.00)	0.58 (0.00)	0.55 (0.00)	0.58 (0.00)	0.57 (0.00)	**0.53 (0.00)**
	PICP	**0.97 (0.00)**	0.95 (0.00)	0.96 (0.00)	**0.97 (0.00)**	0.95 (0.00)	**0.97 (0.00)**	0.96 (0.00)
	MPIW	3.40 (0.64)	2.34 (0.00)	2.27 (0.05)	2.73 (0.19)	2.45 (0.13)	2.83 (0.24)	2.28 (0.02)
yacht	$\overline{\text{LogS}}$	-1.42 (0.07)	-1.57 (0.01)	-1.48 (0.08)	**-1.11 (0.05)**	-1.28 (0.09)	-1.35 (0.07)	-1.28 (0.02)
	$\overline{\text{CRPS}}$	0.59 (0.03)	0.54 (0.01)	0.67 (0.05)	**0.43 (0.02)**	0.58 (0.05)	0.52 (0.03)	4.46 (0.07)
	RMSE	0.97 (0.06)	0.84 (0.01)	1.36 (0.08)	**0.81 (0.02)**	1.22 (0.06)	0.92 (0.03)	4.46 (0.14)
	PICP	0.99 (0.00)	0.99 (0.00)	0.95 (0.01)	**1.00 (0.00)**	**1.00 (0.00)**	**1.00 (0.00)**	0.90 (0.00)
	MPIW	7.41 (0.44)	6.79 (0.06)	6.37 (0.71)	5.95 (0.30)	7.24 (0.93)	7.45 (0.51)	25.48 (0.37)

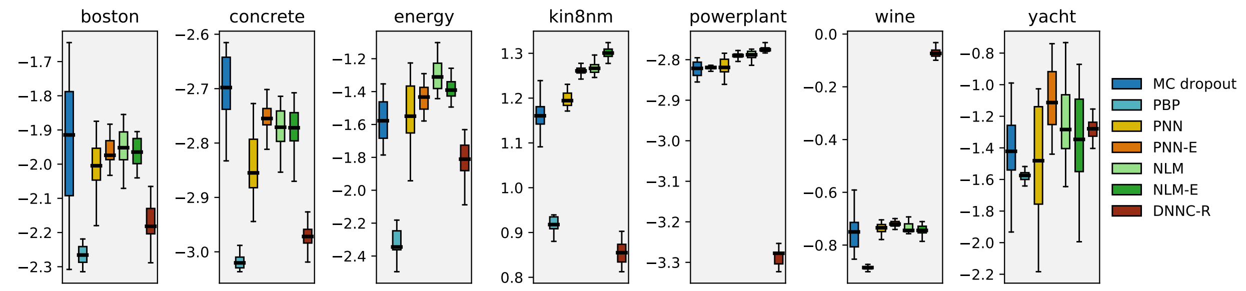
Table B.1: In-sample results for UCI datasets with **random splits**. We depict the mean $\overline{\text{LogS}}$, $\overline{\text{CRPS}}$, RMSE, PICP, and MPIW for all methods and datasets with standard errors in parentheses. The method with the best mean score per dataset is depicted in bold. For RMSE and $\overline{\text{CRPS}}$ lower values correspond to better performance, for $\overline{\text{LogS}}$ and PICP high values represent accurate predictions.

		MC dropout	PBP	PNN	PNN-E	NLM	NLM-E	DNNC-R
boston	$\overline{\text{LogS}}$	-2.23 (0.03)	-2.32 (0.02)	-2.04 (0.03)	-1.99 (0.02)	-2.01 (0.04)	**-1.96 (0.02)**	-2.17 (0.02)
	$\overline{\text{CRPS}}$	1.42 (0.04)	1.33 (0.02)	1.27 (0.04)	1.13 (0.03)	1.23 (0.04)	**1.10 (0.02)**	1.30 (0.02)
	RMSE	3.30 (0.17)	**2.46 (0.04)**	3.25 (0.09)	2.66 (0.09)	3.11 (0.10)	2.72 (0.08)	2.55 (0.07)
	PICP	**0.99 (0.00)**	0.96 (0.00)	0.96 (0.01)	**0.99 (0.00)**	0.98 (0.00)	**0.99 (0.00)**	0.96 (0.00)
	MPIW	14.60 (0.47)	10.63 (0.19)	9.43 (0.30)	10.27 (0.23)	10.57 (0.64)	10.40 (0.17)	11.89 (0.28)
concrete	$\overline{\text{LogS}}$	-3.29 (0.02)	-3.00 (0.02)	-2.88 (0.05)	-2.71 (0.02)	-2.75 (0.03)	**-2.69 (0.01)**	-2.97 (0.01)
	$\overline{\text{CRPS}}$	3.45 (0.07)	2.68 (0.06)	2.62 (0.10)	2.17 (0.04)	2.29 (0.05)	**2.12 (0.03)**	2.59 (0.04)
	RMSE	5.99 (0.18)	4.85 (0.09)	5.03 (0.16)	4.19 (0.08)	4.50 (0.12)	**4.12 (0.08)**	4.58 (0.07)
	PICP	**1.00 (0.00)**	0.96 (0.00)	0.95 (0.01)	0.99 (0.00)	0.99 (0.00)	0.99 (0.00)	0.96 (0.00)
	MPIW	36.43 (0.48)	20.58 (0.44)	19.68 (0.89)	19.17 (0.40)	20.57 (0.72)	19.58 (0.24)	21.15 (0.38)
energy	$\overline{\text{LogS}}$	-1.41 (0.12)	-2.20 (0.16)	-1.17 (0.22)	-1.08 (0.16)	**-0.99 (0.17)**	-1.09 (0.13)	-1.72 (0.11)
	$\overline{\text{CRPS}}$	0.77 (0.09)	1.27 (0.27)	0.77 (0.15)	0.68 (0.11)	0.72 (0.10)	**0.67 (0.09)**	1.03 (0.08)
	RMSE	3.09 (0.11)	3.42 (0.43)	3.11 (0.13)	2.90 (0.15)	2.98 (0.06)	2.87 (0.09)	**2.58 (0.21)**
	PICP	**1.00 (0.00)**	0.96 (0.02)	0.97 (0.01)	**1.00 (0.00)**	0.99 (0.00)	**1.00 (0.00)**	0.98 (0.00)
	MPIW	7.99 (0.71)	8.69 (1.25)	6.18 (1.74)	5.98 (0.65)	6.61 (1.10)	6.35 (0.60)	9.84 (0.85)
kin8nm	$\overline{\text{LogS}}$	1.15 (0.01)	0.87 (0.01)	1.15 (0.03)	1.25 (0.01)	1.23 (0.02)	**1.29 (0.01)**	0.85 (0.01)
	$\overline{\text{CRPS}}$	**0.04 (0.00)**	0.06 (0.00)	0.05 (0.00)	**0.04 (0.00)**	**0.04 (0.00)**	**0.04 (0.00)**	0.05 (0.00)
	RMSE	0.08 (0.00)	0.10 (0.00)	0.09 (0.00)	**0.07 (0.00)**	0.08 (0.00)	**0.07 (0.00)**	0.09 (0.00)
	PICP	0.98 (0.00)	0.96 (0.00)	0.96 (0.00)	**0.99 (0.00)**	0.98 (0.00)	**0.99 (0.00)**	0.96 (0.00)
	MPIW	0.37 (0.01)	0.44 (0.00)	0.33 (0.01)	0.35 (0.00)	0.36 (0.01)	0.34 (0.00)	0.45 (0.01)
powerplant	$\overline{\text{LogS}}$	-2.83 (0.03)	-2.83 (0.01)	-2.81 (0.01)	-2.79 (0.02)	-2.79 (0.02)	**-2.78 (0.01)**	-3.27 (0.01)
	$\overline{\text{CRPS}}$	2.29 (0.06)	2.24 (0.03)	2.23 (0.03)	2.19 (0.03)	2.19 (0.04)	**2.16 (0.02)**	3.19 (0.10)
	RMSE	4.19 (0.10)	4.10 (0.06)	4.11 (0.05)	4.06 (0.06)	4.03 (0.06)	**4.00 (0.05)**	5.06 (0.30)
	PICP	**0.97 (0.00)**	0.96 (0.00)	0.95 (0.01)	**0.97 (0.00)**	0.96 (0.00)	**0.97 (0.00)**	0.95 (0.00)
	MPIW	18.00 (0.63)	16.27 (0.23)	15.90 (0.59)	16.48 (0.34)	16.45 (0.64)	16.61 (0.30)	31.19 (0.82)
wine	$\overline{\text{LogS}}$	-0.77 (0.01)	-0.88 (0.01)	-0.73 (0.02)	-0.70 (0.01)	-0.72 (0.02)	-0.71 (0.01)	**-0.13 (0.01)**
	$\overline{\text{CRPS}}$	0.31 (0.00)	0.33 (0.00)	0.32 (0.01)	0.38 (0.08)	0.30 (0.00)	0.60 (0.29)	**0.26 (0.00)**
	RMSE	0.58 (0.01)	0.59 (0.00)	0.59 (0.01)	0.54 (0.00)	0.57 (0.01)	0.56 (0.01)	**0.51 (0.00)**
	PICP	**0.97 (0.00)**	0.95 (0.00)	0.96 (0.00)	**0.97 (0.00)**	0.96 (0.00)	**0.97 (0.00)**	0.96 (0.00)
	MPIW	2.52 (0.05)	2.34 (0.02)	2.47 (0.18)	3.90 (1.26)	2.28 (0.06)	7.41 (4.91)	2.21 (0.04)
yacht	$\overline{\text{LogS}}$	-2.43 (0.11)	-1.75 (0.03)	-1.70 (0.23)	-1.36 (0.07)	-1.33 (0.10)	-1.51 (0.06)	**-1.26 (0.06)**
	$\overline{\text{CRPS}}$	1.64 (0.09)	0.65 (0.02)	0.80 (0.15)	**0.53 (0.03)**	0.59 (0.03)	0.57 (0.03)	4.99 (0.59)
	RMSE	2.57 (0.36)	0.99 (0.02)	1.70 (0.38)	**0.96 (0.03)**	1.33 (0.07)	1.01 (0.04)	4.40 (0.40)
	PICP	**1.00 (0.00)**	0.99 (0.00)	0.94 (0.04)	**1.00 (0.00)**	**1.00 (0.00)**	**1.00 (0.00)**	0.89 (0.01)
	MPIW	23.09 (1.22)	8.16 (0.25)	7.06 (1.43)	7.46 (0.61)	6.73 (0.63)	8.12 (0.59)	27.27 (2.47)

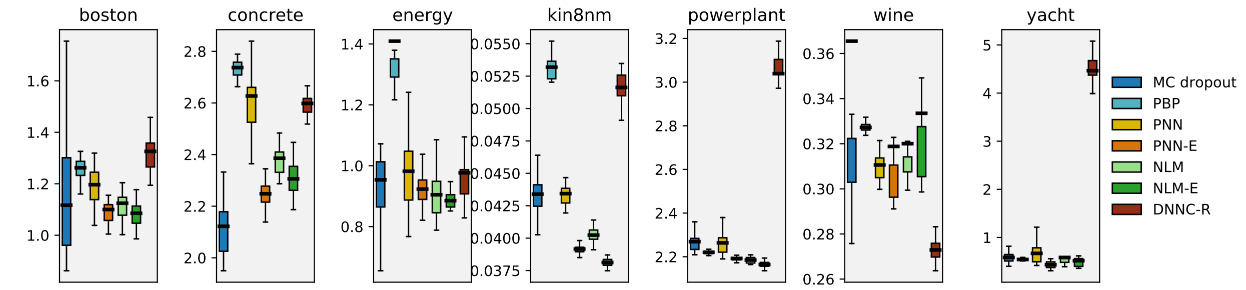
Table B.2: In-sample results for UCI datasets with **gap splits**. We depict the mean $\overline{\text{LogS}}$, $\overline{\text{CRPS}}$, RMSE, PICP, and MPIW for all methods and datasets with standard errors in parentheses. The method with the best mean score per dataset is depicted in bold. For RMSE and $\overline{\text{CRPS}}$ lower values correspond to better performance, for $\overline{\text{LogS}}$ and PICP high values represent accurate predictions.

		MC dropout	PBP	PNN	PNN-E	NLM	NLM-E	DNNC-R
boston	$\overline{\text{LogS}}$	-2.01 (0.05)	-2.11 (0.03)	-1.84 (0.04)	-1.79 (0.04)	**-1.76 (0.04)**	-1.78 (0.03)	-2.06 (0.02)
	$\overline{\text{CRPS}}$	1.12 (0.05)	1.08 (0.03)	1.04 (0.05)	0.94 (0.04)	0.97 (0.04)	**0.91 (0.04)**	1.17 (0.04)
	RMSE	2.56 (0.18)	**2.01 (0.06)**	2.55 (0.20)	2.21 (0.14)	2.33 (0.16)	2.19 (0.15)	2.43 (0.09)
	PICP	**1.00 (0.00)**	0.96 (0.00)	0.96 (0.01)	0.99 (0.00)	0.99 (0.00)	0.99 (0.00)	0.96 (0.00)
	MPIW	11.38 (0.59)	8.65 (0.24)	7.87 (0.57)	9.04 (0.53)	8.47 (0.58)	8.90 (0.39)	11.11 (0.42)
concrete	$\overline{\text{LogS}}$	-2.94 (0.11)	-2.97 (0.02)	-2.76 (0.02)	**-2.67 (0.02)**	-2.70 (0.04)	**-2.67 (0.02)**	-2.87 (0.02)
	$\overline{\text{CRPS}}$	2.60 (0.22)	2.58 (0.05)	2.40 (0.05)	2.07 (0.05)	2.23 (0.08)	**2.06 (0.05)**	2.36 (0.04)
	RMSE	4.69 (0.34)	4.69 (0.08)	4.78 (0.10)	**3.98 (0.10)**	4.43 (0.16)	4.02 (0.12)	4.27 (0.05)
	PICP	**1.00 (0.00)**	0.95 (0.00)	0.96 (0.01)	0.99 (0.00)	0.98 (0.00)	0.99 (0.00)	0.95 (0.00)
	MPIW	26.28 (2.78)	19.76 (0.36)	17.28 (0.35)	18.50 (0.33)	18.73 (0.67)	19.10 (0.23)	19.38 (0.47)
energy	$\overline{\text{LogS}}$	-1.47 (0.13)	-2.14 (0.15)	-1.44 (0.17)	-1.21 (0.14)	**-1.18 (0.12)**	-1.24 (0.11)	-1.20 (0.18)
	$\overline{\text{CRPS}}$	0.79 (0.11)	1.16 (0.16)	0.93 (0.10)	0.84 (0.12)	0.76 (0.10)	0.79 (0.10)	**0.61 (0.09)**
	RMSE	2.10 (0.38)	2.31 (0.34)	2.84 (0.28)	2.72 (0.31)	2.42 (0.37)	2.33 (0.25)	**1.25 (0.19)**
	PICP	**1.00 (0.00)**	0.96 (0.01)	0.95 (0.03)	**1.00 (0.00)**	0.99 (0.00)	**1.00 (0.00)**	0.96 (0.00)
	MPIW	8.68 (1.13)	9.93 (1.31)	6.63 (0.69)	6.91 (0.94)	7.21 (0.91)	8.14 (1.08)	5.71 (0.77)
kin8nm	$\overline{\text{LogS}}$	1.18 (0.02)	0.97 (0.01)	1.24 (0.01)	1.31 (0.01)	1.29 (0.01)	**1.33 (0.01)**	0.91 (0.02)
	$\overline{\text{CRPS}}$	**0.04 (0.00)**	0.05 (0.00)	**0.04 (0.00)**	**0.04 (0.00)**	**0.04 (0.00)**	**0.04 (0.00)**	0.05 (0.00)
	RMSE	0.08 (0.00)	0.09 (0.00)	0.08 (0.00)	**0.07 (0.00)**	**0.07 (0.00)**	**0.07 (0.00)**	0.08 (0.00)
	PICP	**0.99 (0.00)**	0.96 (0.00)	0.96 (0.00)	**0.99 (0.00)**	0.98 (0.00)	**0.99 (0.00)**	0.96 (0.00)
	MPIW	0.38 (0.01)	0.39 (0.00)	0.31 (0.01)	0.33 (0.00)	0.32 (0.01)	0.32 (0.01)	0.42 (0.01)
powerplant	$\overline{\text{LogS}}$	-2.77 (0.01)	-2.80 (0.01)	-2.77 (0.01)	-2.76 (0.01)	-2.75 (0.01)	**-2.74 (0.01)**	-3.25 (0.06)
	$\overline{\text{CRPS}}$	2.17 (0.02)	2.18 (0.02)	2.17 (0.02)	2.14 (0.02)	2.11 (0.02)	**2.10 (0.02)**	2.96 (0.14)
	RMSE	3.98 (0.05)	3.97 (0.04)	3.99 (0.05)	3.93 (0.05)	**3.87 (0.04)**	**3.87 (0.04)**	4.36 (0.07)
	PICP	**0.97 (0.00)**	0.96 (0.00)	0.96 (0.00)	**0.97 (0.00)**	**0.97 (0.00)**	**0.97 (0.00)**	0.96 (0.00)
	MPIW	16.78 (0.38)	15.70 (0.17)	15.26 (0.15)	16.37 (0.23)	16.01 (0.54)	16.13 (0.29)	29.14 (1.96)
wine	$\overline{\text{LogS}}$	-0.73 (0.01)	-0.84 (0.01)	-0.68 (0.02)	-0.66 (0.01)	-0.69 (0.02)	-0.66 (0.01)	**0.02 (0.01)**
	$\overline{\text{CRPS}}$	0.30 (0.01)	0.31 (0.00)	0.30 (0.01)	0.29 (0.01)	0.36 (0.06)	0.29 (0.01)	**0.24 (0.00)**
	RMSE	0.55 (0.01)	0.56 (0.00)	0.56 (0.02)	0.52 (0.01)	0.55 (0.01)	0.53 (0.01)	**0.48 (0.01)**
	PICP	**0.98 (0.00)**	0.95 (0.00)	0.96 (0.00)	**0.98 (0.00)**	0.96 (0.00)	**0.98 (0.00)**	0.96 (0.00)
	MPIW	2.45 (0.09)	2.24 (0.02)	2.24 (0.15)	2.55 (0.16)	3.26 (0.99)	2.32 (0.12)	1.99 (0.03)
yacht	$\overline{\text{LogS}}$	-1.65 (0.22)	-1.56 (0.14)	-1.34 (0.18)	**-1.27 (0.16)**	-1.40 (0.29)	-1.74 (0.17)	-1.31 (0.08)
	$\overline{\text{CRPS}}$	0.85 (0.15)	0.55 (0.06)	0.70 (0.14)	**0.53 (0.08)**	0.82 (0.21)	0.75 (0.11)	3.68 (0.60)
	RMSE	1.24 (0.24)	**0.83 (0.09)**	1.64 (0.44)	0.99 (0.23)	1.74 (0.49)	1.02 (0.12)	4.27 (0.49)
	PICP	**1.00 (0.00)**	**1.00 (0.00)**	0.95 (0.02)	**1.00 (0.00)**	**1.00 (0.00)**	**1.00 (0.00)**	0.92 (0.01)
	MPIW	12.09 (2.25)	7.03 (0.77)	5.68 (1.45)	7.32 (1.18)	10.46 (3.04)	11.43 (1.81)	23.20 (3.74)

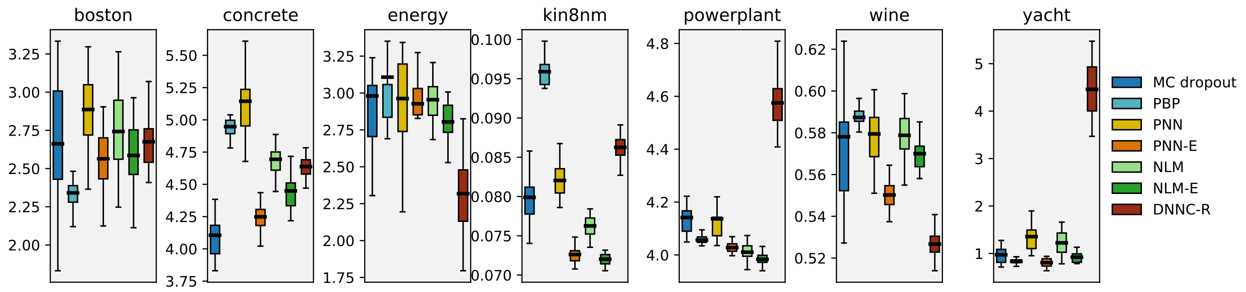
Table B.3: In-sample results for UCI datasets with **tail splits**. We depict the mean $\overline{\text{LogS}}$, $\overline{\text{CRPS}}$, RMSE, PICP, and MPIW for all methods and datasets with standard errors in parentheses. The method with the best mean score per dataset is depicted in bold. For RMSE and $\overline{\text{CRPS}}$ lower values correspond to better performance, for $\overline{\text{LogS}}$ and PICP high values represent accurate predictions.



(a) Average logarithmic score ($\overline{\text{LogS}}$)

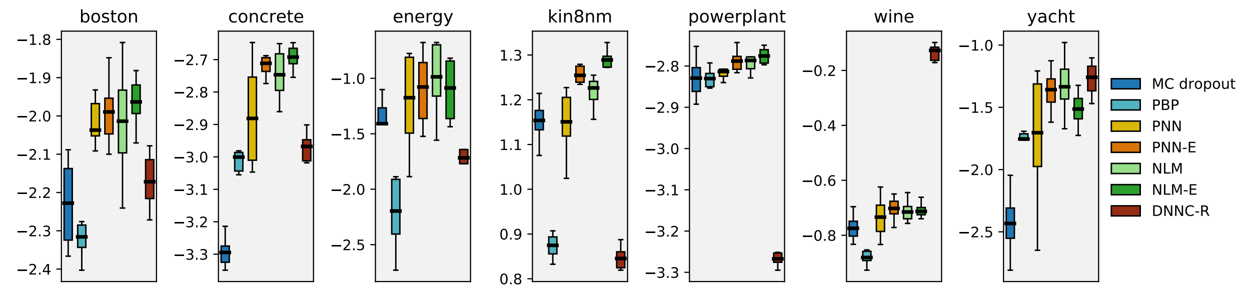


(b) Average continuous ranked probability score ($\overline{\text{CRPS}}$)

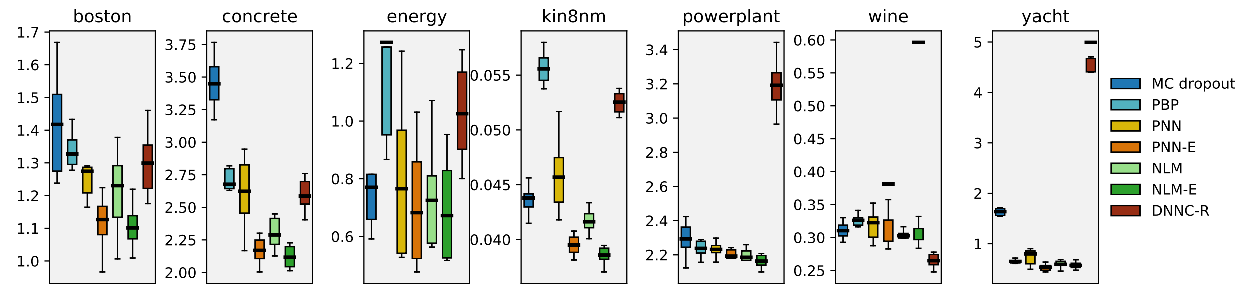


(c) Root mean square error (RMSE)

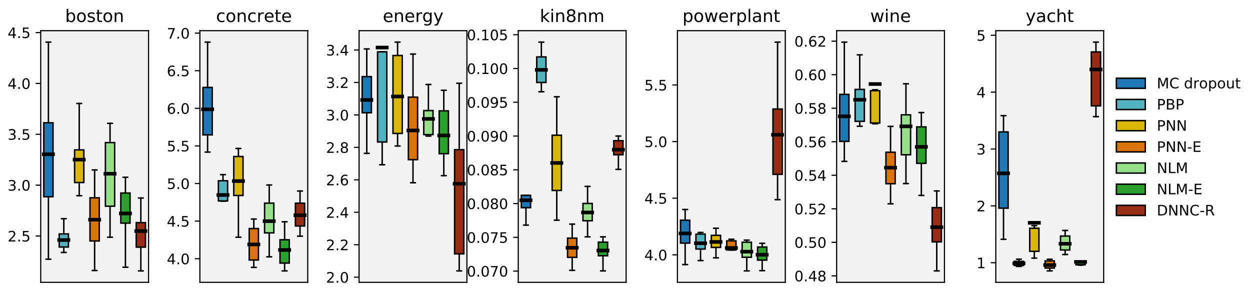
Figure B.1: In-sample box-plots for all models for UCI datasets with **random splits**. (a) $\overline{\text{LogS}}$: the higher the value the better performance of the model. (b) $\overline{\text{CRPS}}$: a small value corresponds to a good performance. (c) RMSE: the smaller the better. Note that the vertical black lines in each box-plot correspond to the mean.



(a) Average logarithmic score ($\overline{\text{LogS}}$)

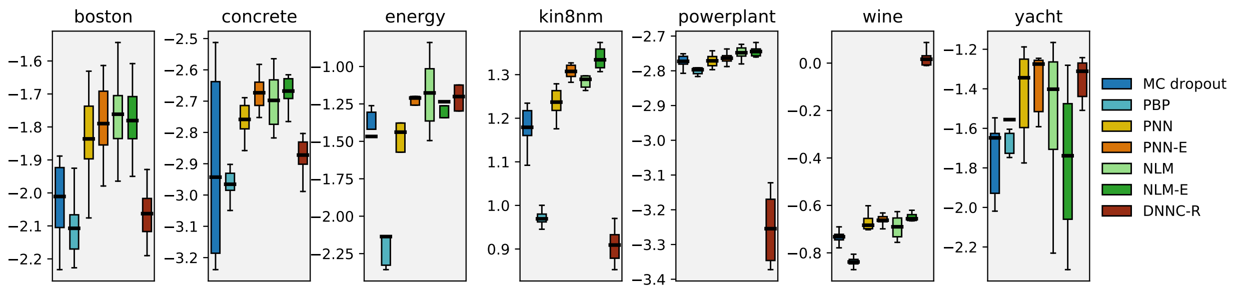


(b) Average continuous ranked probability score ($\overline{\text{CRPS}}$)

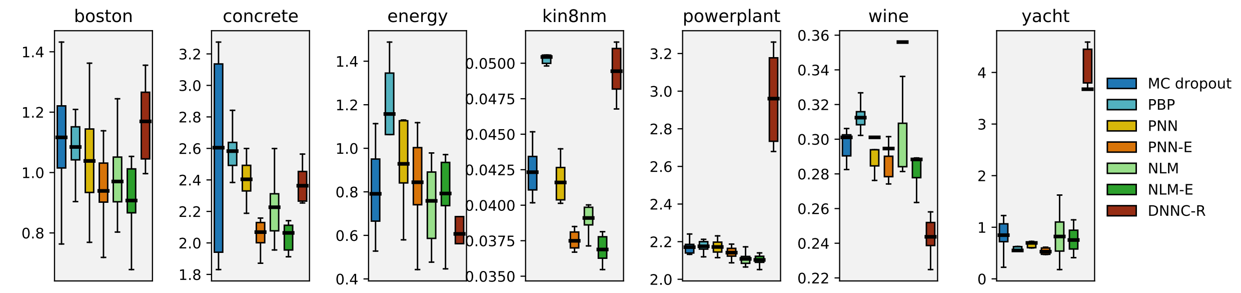


(c) Root mean square error (RMSE)

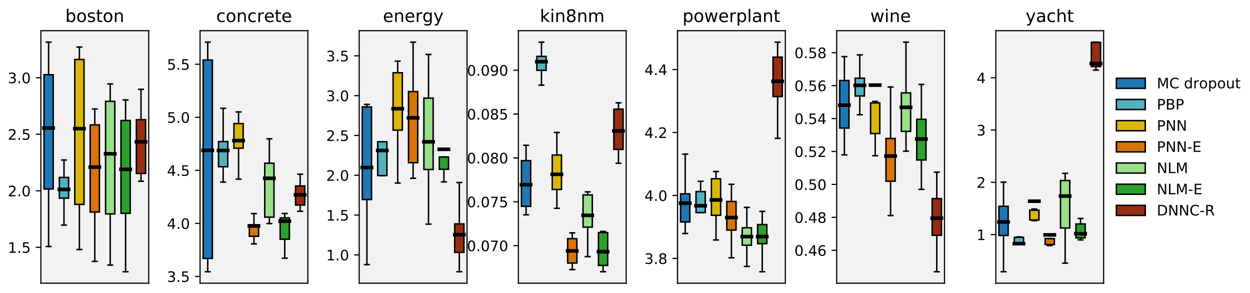
Figure B.2: In-sample box-plots for all models for UCI datasets with **gap splits**. (a) $\overline{\text{LogS}}$: the higher the value the better performance of the model. (b) $\overline{\text{CRPS}}$: a small value corresponds to a good performance. (c) RMSE: the smaller the better. Note that the vertical black lines in each box-plot correspond to the mean.



(a) Average logarithmic score ($\overline{\text{LogS}}$)



(b) Average continuous ranked probability score ($\overline{\text{CRPS}}$)



(c) Root mean square error (RMSE)

Figure B.3: In-sample box-plots for all models for UCI datasets with **tail splits**. (a) $\overline{\text{LogS}}$: the higher the value the better performance of the model. (b) $\overline{\text{CRPS}}$: a small value corresponds to a good performance. (c) RMSE: the smaller the better. Note that the vertical black lines in each box-plot correspond to the mean.

Declaration of Authorship

I, Per Joachims, hereby confirm that I have authored this master's thesis independently and without the use of other than the indicated sources. All passages literally or in general matter taken out of publications or other sources are marked as such.

Berlin, June 30 2021

Per Joachims