Bachelor's Thesis

Creating and Evaluating Stochastic Regression Models on the Basis of Heterogeneous Sensor Networks.

by

Stanislav Arnaudov

Chair of Pervasive Computing Systems/TECO,
Institute of Telematics
Department of Informatics

Handover Date: 31.09.2018

Supervisors: Dr. Johannes Riesterer[1]
Dr. Sebastian Lerch[2,3]

[1]Chair for Pervasive Computing Systems / TECO, Institute of Telematics,
Karlsruhe Institute of Technology
[2]Institute for Stochastics, Karlsruhe Institute of Technology
[3]Heidelberg Institute for Theoretical Studies

Statement of Authorship

I hereby declare that I am the sole author of this bachelor thesis and that I have not used any sources other than those listed in the bibliography and identified as references. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, 31.09.2018

Abstract

This thesis aims to better understand Bayesian machine learning models and their practical use on real world data. We examine two models that incorporate uncertainty in their predictions - Bayesian Neural Networks and Mixture Density Networks. The used data comes from air-pollution sensors. The quality of 3 of the sensors is known to be high but for the rest of them the quality of measurement is unknown. We aim to build a model that can predict the air pollution at some sensor at a given time. Consideration of the uncertainty in the predicted value is crucial as it allows the precise evaluation of the generated models. We compare the models through evaluation with proper scoring rules. As the quality of the majority of sensors is from unknown quality we try to find out which of the sensors are most relevant for a better prediction through a feature importance technique. We leverage the capabilities of Tensorflow, Edward and GPFlow machine learning libraries in order to build probabilistic regression models that can be further evaluated.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Probabilistic regression . . . 1
- 1.3 Data . . . 4
- 1.4 Models . . . 6

2 Related work **7**

3 Probabilistic Modeling **8**
- 3.1 Bayesian regression modeling . . . 8
- 3.2 Bayesian Neural Networks . . . 10
  - 3.2.1 Neural networks . . . 10
  - 3.2.2 Neural networks in Bayesian settings . . . 11
  - 3.2.3 Variational inference and Kullback-Leibler divergence . . . 12
- 3.3 Mixture Density Networks . . . 14

4 Model evaluation methods **16**
- 4.1 Probabilistic Forecast . . . 16
  - 4.1.1 Prediction spaces . . . 17
  - 4.1.2 Dispersion and sharpness . . . 17
- 4.2 Proper Scoring rules . . . 18
  - 4.2.1 Logarithmic score (LS) . . . 18
  - 4.2.2 Continuously ranked probability score (CRPS) . . . 19
  - 4.2.3 Dawid–Sebastiani score (DSS) . . . 19
  - 4.2.4 Computation of scoring rules . . . 20
- 4.3 Rank Histogram . . . 21
- 4.4 Feature Importance . . . 23

5 Results **24**
- 5.1 Data used . . . 24
- 5.2 Evaluation of the models . . . 24

6 Implementation **24**
- 6.1 Preparing the data . . . 24
  - 6.1.1 Download module . . . 25
  - 6.1.2 Preprocess module . . . 25
  - 6.1.3 Description module . . . 26
  - 6.1.4 Preprocess LU BW . . . 27
  - 6.1.5 Loader module . . . 27
- 6.2 Models implementations . . . 27
  - 6.2.1 Mixture density networks . . . 28
  - 6.2.2 Bayesian neural networks . . . 30
- 6.3 Model training . . . 31
- 6.4 Models evaluation . . . 31
- 6.5 Auxiliary scripts . . . 31
  - 6.5.1 Run Generation pipeline . . . 31
  - 6.5.2 Clean `env` . . . 32

7 Conclusion **33**
7.1 Summery 33
7.2 Discussion 33

List of Figures

1 General system 2
2 Forecast sample draws 3
3 Point estimate and distribution forecast 4
4 Probabilistic regression example model 5
5 Mixture Density Network Architecture 16
6 Continuous Rank Probability Score Intuition 20
7 Rank Histogram Example 22

1 Introduction

1.1 Motivation

There are a lot of cases where a network of sensors is used to measure a certain quantity over time in different places. Some of the measurements can have better *quality* than other as not all sensors are created equal. With quality we mean the reliability that the measured value indeed reflects the reality. The difference in the quality of the sensors makes the measured data *heterogeneous* and brings certain difficulties in the modeling process. For a machine learning application only some of the data maybe relevant for building a reliable predictive model. One of our goals is to asses which part of the input is useful data and to what extend.

From another point of view we can also set another goal. We consider the case where only a few of the sensors are known to produce reliable measurements. In this sense we try to expand a network of *homogeneous* and high reliability sensors with sensors of unknown quality and thus making the network heterogeneous. In this case we increase the spatial resolution of the network and thus we bring the possibility to make more accurate predictions on the basis of data coming from the whole network. On the other hand, heterogeneity in quality of the different sources in the network introduces uncertainty in the measurements. It thus becomes necessary to build appropriate model that predicts the measured values from given sensor but also respects the inherent uncertainty in the data and also models it properly.

To concertize the problem - we use air pollution data from a network of sensors around Stuttgart. The Landesanstalt für Umwelt, Messungen und Naturschutz Baden-Württemberg (LU-BW) has provided data from 3 high quality air pollution sensors. The other part of the data is from *DIY* (*Do it yourself*) sensors that provide their measurements publicly under luftdaten.info. Those are the unreliable parts of the network as described in the previous paragraph. We can make no statements about the quality of the data coming form the DIY sensors. We do not know how are they calibrated and the sensors are deployed from anonymous builders. Furthermore the sensors are cheap. All of this effectively means that we have unreliable features in our input data. We aim to find out which of the features (and with that which of the sensors) are "better" than the others. This is done through a *feature importance* technique described in Section 4.4. We also investigate whether or not we can predict the air pollution values measured by one of the LUBW-sensors based on the combined data coming from all other sensors. If reliable predictions are possible, this would suggest that one can theoretically use the cheap network of sensors as an substitute for the few high quality sensors. Figure 1 shows the general overview of the system we want to develop.

1.2 Probabilistic regression

Regression models make real valued predictions given an input data point and based on training data. Each time a prediction is made, one would desire to have a certain measurement for the uncertainty of the prediction. If this is the case and the prediction is given in a form of some distribution we say that the forecast is probabilistic. This quantification of the uncertainty allows for optimal decision making. Models that produce such probabilistic forecasts have

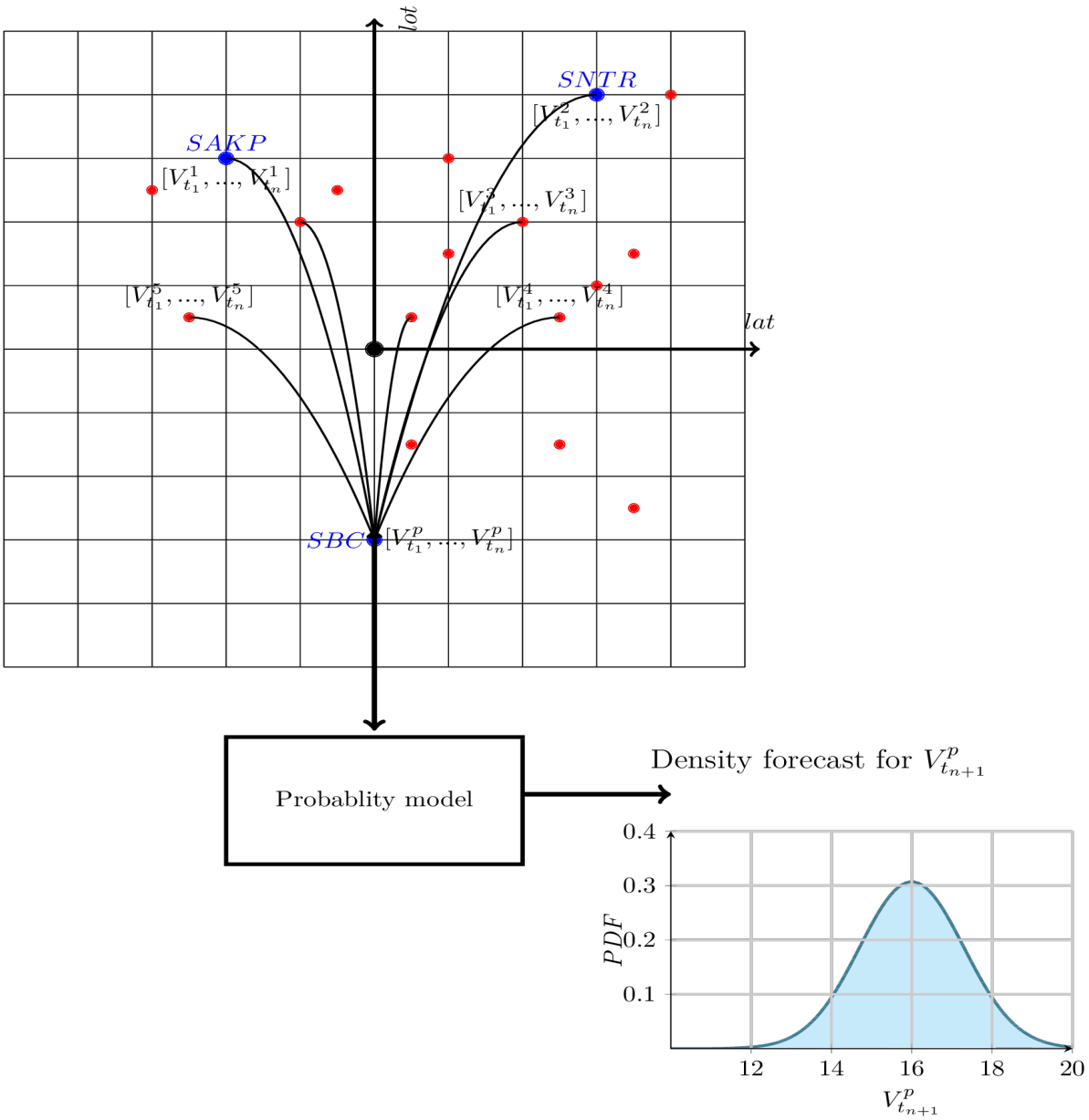

Figure 1: General view of the system we aim to build. The red dots represent unreliable DYI sensors. Data from all the sensors is used in order to make prediction for one (SBC) of the LUBW-Stations (shown in blue). The prediction made is for some future moment $\boldsymbol{t+1}$ and it is made on the basis of the past - the values of all moments till $\boldsymbol{t+1}$ from all the stations. The prediction is generated in form of a probability distribution generated by the probabilistic regression model.

gained recent popularity in areas such as meteorology, seismology and economics (see Section 2 for concrete examples).

Probability forecasts are not uncommon for classification models. Through the use of soft-max layer (see [DL90]) neural networks can assign each class they recognize a certain probability. This thesis is centered however around probabilistic regression models. *Probabilistic regression models* produce a probability distribution as their prediction rather than a point estimate. One can then draw samples form this distribution, make statements about the drawn samples and reason about the nature of the possible values that could be predicted. Illustration of this is given in Figure 2. Another way of thinking about probability forecasts is that they define a certain region where the actual value can be. The region however is a probability distribution. The model tells us then what the probability of each realising value given the input by evaluation is. We say that the model describes the conditional probability of the output given the input (more details in Section 3). Figure 4 illustrates these considerations about probabilistic regression models.

Predicting distributions rather than point estimates also plays a role in the evaluation of the predictive capabilities of a given model. Because the prediction is distribution, its evaluation requires different measures of predictive performance. Figure 3 illustrates the apparent difference between point estimate and distribution prediction and how can one can reason much more if a distribution for the prediction is present. In our evaluation of the models we employ proper scoring rules (see [GR07]). They evaluate the error between a predicted probability distribution and a realized observation. We give more details on the proper scoring rules in Section 4.2.

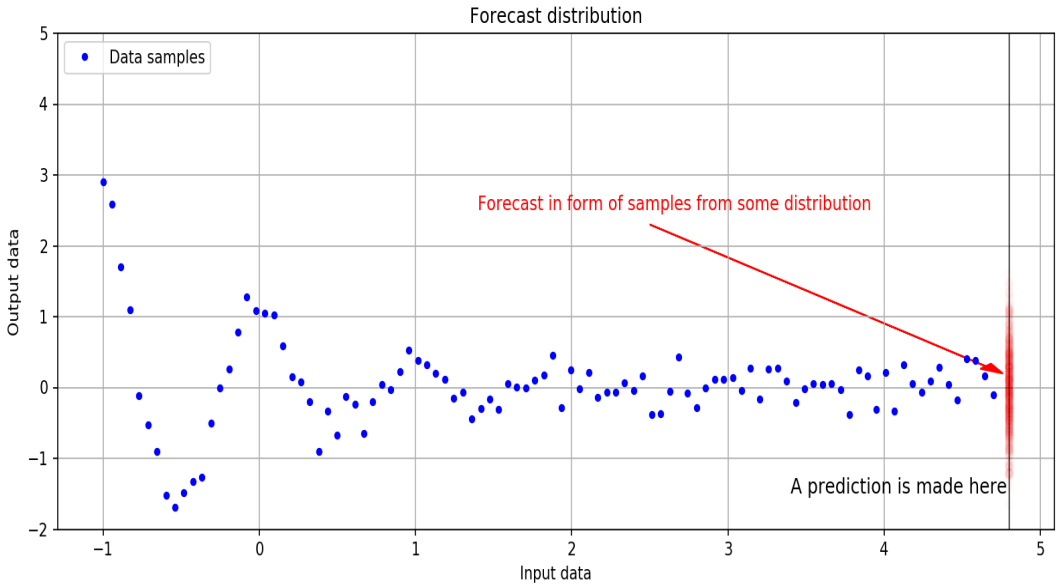

Figure 2: Illustration of what samples from forecast could look like. The blue dots are observations. On their basis a prediction is made. The red dots are drawn samples from the predicted distribution. We can see that those are concentrated near the most probable value of the observation at this place.

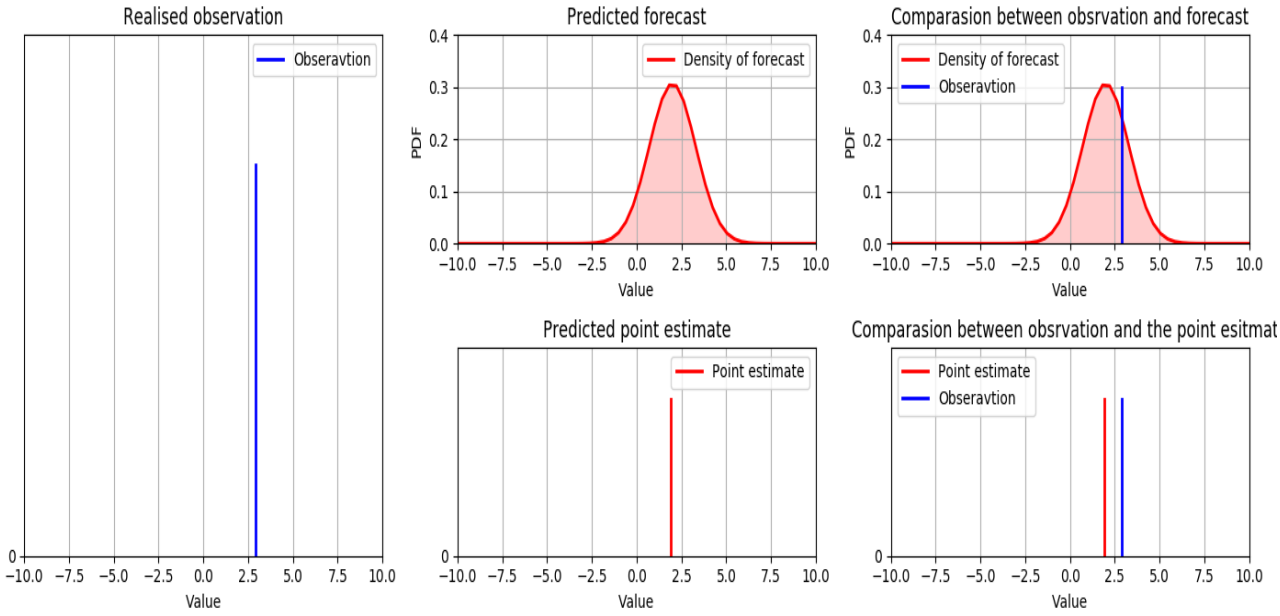

Figure 3: The left most graph shows actual observation that is to be predicted. In the middle there are two prediction made by two different models - a probabilistic forecast (on the top) and point estimate(on the bottom). With the right graphs one can clearly see how there is not a whole lot to compare between the point estimate and the observation other than the distance between the two. With the probability forecast however we can evaluate where and how exactly fits the observation in the predicted distribution.

The quality of probabilistic forecasts also have two other important characteristics:

- Calibration - This is measure of the compatibility between the forecast and the observation. It depends both on the observation and on the forecast in form of probability distribution.
- Sharpness - Concentration of the forecasts. This property is purely of the distribution.

We look at those again in detail in Section 3.

The statistical framework for inferring probability distributions is *Bayesian Inference*. A good theoretical and practical overview of Bayesian inference can be found in [Tip04]. In Bayesian inference Bayes theorem (see [Bay63]) plays a central role in determining the probability for a hypothesis given some evidence. Bayesian inference takes into consideration our prior beliefs about the nature of some data and then combines those beliefs with the actual observed data (denoted as *evidence*) and generated *posterior* probability about the values of the output. We discuss Bayesian reasoning in Section 3.1.

1.3 Data

As previously stated, the used data comes from two different types from sources.

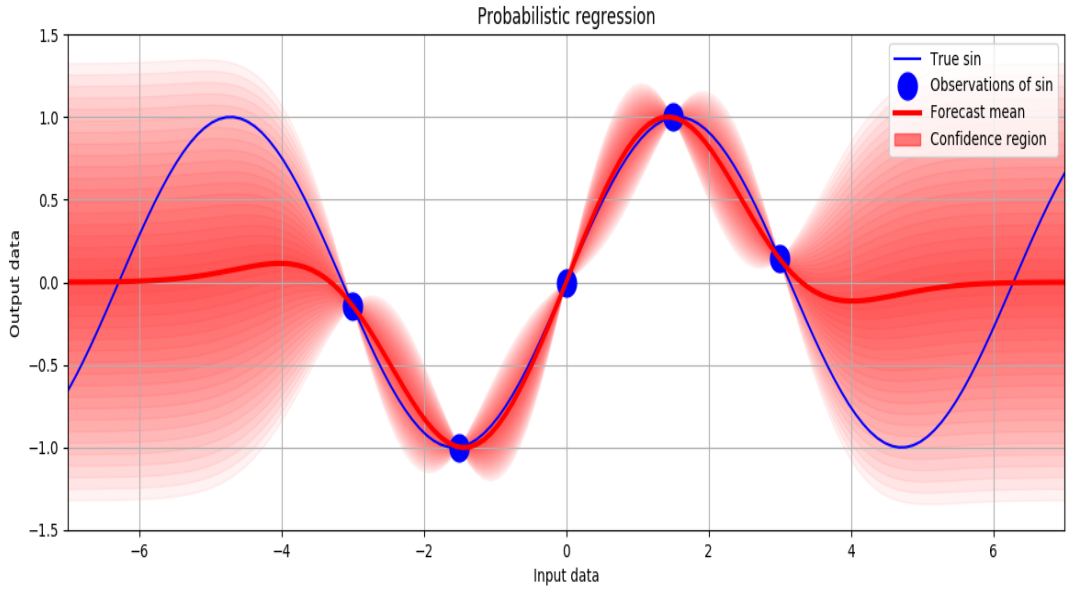

Figure 4: A probabilistic regression model (in red) that tries to model the **sin**-function (in blue). The big blue circles are the observations of the function that the models becomes as training set. On the basis of those observations we construct a probabilistic model that can evaluate the whole input space. The model knows about the given observations of the modeled function so it predicts their values with high certainty. The places where there is no observed data however, the model cannot make precise prediction and the distributions is wider. We notice however that true value of the function lies in the distribution so in sense the model predicts the function somewhat accurately.

- LU-BW (Landesanstalt für Umwelt, Messungen und Naturschutz Baden-Württemberg) - Organization whose activities include air measurements and high quality sensors deployment. LUBW has gathered data from 3 high quality sensors in Stuttgart for the year 2017. The data is to be used for research proposes only and is not publicly available.

- luftdaten.info - a public network of sensors where everyone can provide data from sensor from certain type. There we can find air pollution data from *sds011*-sensors (see [sds] for specifications). Those are cheap DIY sensors and give no guarantees for the quality of the measured data. Measurements of wide networks of such sensors are publicly available under [dat].

All of the sensors measure two types of air pollution data. The different naming of those values relates to the size of the measured particles. The two data types are:

- *PM2.5* - Particular matter up to 2.5 micrometers.

- *PM10* - Particular matter up to 10 micrometers.

We found that those values are highly correlated - correlation of about 0.95 on average (see Section 5.1 for details). For this reason the built models use either PM2.5 or PM10. That is, with PM2.5 data of all other sensors, we predict the PM2.5 values of a particular sensor or respectively with PM10 data we predict PM10 values.

The format of the data coming straight from the networks was not convenient for direct training of the models. The sensors from *luftdaten* provide measurements for each minute over the whole year while the ones of LUBW integrate their measurements over thirty minutes. Because of this inconsistency we had to "synchronize" the both data sources in order to produce the final data sets. We've also found numerous problems with the data from *luftdaten*. Some of them are:

- Not all sensors are located around Stuttgart.

- Some of the sensors have a lot of missing days without measurements.

- Some of the sensors have missing measurement in random minutes of the day.

The raw data was extensively prepossessed with different techniques. In Section 6 we describe all of the steps taken to achieve that.

1.4 Models

The models of interested that we are going to look at are *Bayesian Neural Networks* (*BNNs*, see Section 3.2) and *Mixture Density networks* (*MDNs*, see Section 3.3). Here we give brief overview of the both models and we discuss them in detail in the respective sections.

BNNs are relatively new type of model proposed in the 1990s and studied in depth in [Mac92], [Nea96]. [Gal16] provides general survey and study on of uncertainty quantification in deep learning. More recently BNNs have seen a surge in popularity as they combine Neural Networks with Probabilistic Models.The inferred distributions over their wights allows them to offer a probability distributions for their estimates. This makes BNNs attractive to the machine

learning community as those properties give them robustness to over-fitting and ability to learn from small datasets. BNNs place prior distribution on their weights. The posterior distributions are estimated either by variational inference ([BCKW15],[PBJ12]) or by sampling techniques ([VSL00]). In our work we use variational inference with *Kullback-Leibler divergence* and give details about the method in Section 3.2.3.

With MDNs we hope to construct a model similar to mixture of Gaussian experts similar to [RG02]. The mixture of Gaussian experts is a combination of several Gaussian Process regression models and a gating network. Given an example data point the gating network assigns certain probabilities for each of the Gaussian process models. On their turn they generate several probability distributions which are weighted with the probabilities from the gating networks. The end result again is a probability distributions given by the sum of the individual ones. MDNs model this by creating a single network that not only outputs weights for the individual mixture components but also their parameters. MDNs are very similar to standard Neural Networks with the only difference being that final layer is mapped to a mixture of distributions. In our case the mixture model is a weighted sum of Gaussian process regressors. MDNs produce especially good results when the input data is such that a single input can correspond to multiple output values. In those cases, the predicted distribution models the corresponding output values by giving some probability on each of them. We investigate what results one can get when this approach is used for simpler data. As MDNs are almost standard neural networks, they are trained with the standard *backpropagation* method, described for example in [BNVP01].

In order to have a baseline with which we can compare the build models we also construct simple empirical model. The empirical modeling technique for prediction of values based on empirical observations. In this cases there is no consideration of the input of the data but rather a simple looking at the past values of the output value. The past values of the predicted measurement are treated as samples from a random variable. We describe how we infer a probability distribution for a prediction in Section 6.2. Similar empirical models are commonly used for simple predictions of meteorological data (see [KVS11] and [EvOHS15]).

2 Related work

Heterogeneous sensor networks are not uncommon. [WC07] looks at the deployment and the topology control specifically of Heterogeneous Wireless Sensor Networks. [KCS] surveys a variety of clustering algorithms aimed at increasing the network's scalability by recognizing the sensors with similar energy requirements. [US14] investigates the kinds of security threats that arise with the use of heterogeneous sensor networks. In the context of machine learning [LZLQ17] tries to use deep learning techniques and adapt them to the settings of heterogeneous sensor data.

A lot of attention has been brought to probabilistic forecasting([GK14]). When it comes to history, [Sti75] describes the transition from point estimates to distribution prediction. Nowadays the technique is widely used in variety of cases as it is further developed ([EG12]). Common areas where probabilistic forecasts are employed are whether and climate prediction ([Col07]). The uses are however not limited only to that. [Krz01] looks how probabilistic models could be used for flood risk assessment, [Tim] deals with economic and financial risk management, [Pin13] tries to make predictions about the availability of renewable energy

resources. Distribution forecasting could be even used to make statements about the outcome of elections ([MHW12]).

The other focus of our work are BNNs. As models that tie Bayesian reasoning and neural networks ([Sch15] presents good overview of NN and deep learning) together, they have also gained popularity. [LV01] first gives brief overview of the Bayesian approach to neural networks and then surveys different use cases assessing the performance. Fields like astrophysics [BXP+16] and particle physics [PCB15] also profit from the predictive capabilities of BNNs. As medicine is one of the fields that gets much of the attention of the machine learning research, [KP17] applies BNNs in order to quantify the uncertainties and apply them in ischemic stroke lesion segmentation. BNNs are also continuing to be developed. [FBV17] explores the Bayesian approach in Recurrent Neutral Networks developing *Bayesian Recurrent Neural Networks*.

MDNs have also found popularity as one another alternative to combine neural networks with distribution forecast models. [SDD98] treat MDNs as a general framework for identifying stochastic processes and apply them in modeling stock exchange index data. [ZS14] bring MDNs to use in order to improve deep neural networks for acoustic modeling. MDNs have their own variant of recurrent models as described in [MDD03]. We like to note that MDNs have similar structure to other models such as *Adaptive mixture of local experts* ([JJNH91]) and *Infinite Mixtures of Gaussian Process Experts* ([RG02]) but are different in their way of modeling data (as explained in Section 3.3).

The use of proper scoring rules is not uncommon in probability forecast setting. [JKL17] discusses the **scoringRules** package for R (programming language) and its usage details by presenting case studies where proper scoring rules find application. [Cle] uses probability integral transform (see Section 4.1.2) in order to evaluate the UK Monetary Policy Committee's inflation density forecasts.

3 Probabilistic Modeling

3.1 Bayesian regression modeling

In probabilistic modeling we try to estimate parameters of given model by first defining some prior on them. The prior distribution describes, how likely values of the parameter are without considering any observed data. Then a *conditional likelihood* for the output based on the parameters and the input data is defined. This posterior probability distribution describes the probability of observing certain data given the parameters of the model and under the assumption that the models has generated the data . The two distributions are then combined in order to produce the final posterior on the parameters of the model. A very detailed look at Bayesian models in the context of Gaussian processes can be found in [RW05]. We can summarize:

- **Prior** - uncertainty in form of probability distribution *before* observing the data.
- **Posterior** - uncertainty probability distribution *after* observing the data.

Now we formally write down everything following [Gal16]. Let

$$\boldsymbol{X} = \{\boldsymbol{x_1}, ..., \boldsymbol{x_N}\}, \forall \boldsymbol{i} : \boldsymbol{x_i} \in \mathbb{R}^M \tag{3.1}$$

be the set of input data. The corresponding output is given by the set $\boldsymbol{Y} = \{\boldsymbol{y_1}, ..., \boldsymbol{y_N}\}$. The dataset could also be seen as tuples of feature vector $\boldsymbol{x_i}$ and an output value $\boldsymbol{y_i}$. We can then write the dataset as $\{(\boldsymbol{x_i}, \boldsymbol{y_i})\}_{i=1}^{N}$. As we deal with regression models we assume that $\forall \boldsymbol{i} : \boldsymbol{y}_\text{i} \in \mathbb{R}$ and there exists some function $\boldsymbol{f} : \boldsymbol{X} \to \mathbb{R}$ with $\boldsymbol{f_{true}}(\boldsymbol{x_i}) = \boldsymbol{y_i}, \forall \boldsymbol{i} \in \{\boldsymbol{1}, ..., \boldsymbol{n}\}$. We want to find a model with appropriate parameters that is likely to have generated the data. In fact, through choice of parameters, we want the data to be as probable as possible according to the defined model. We model the mapping function between $\boldsymbol{X}$ and $\boldsymbol{Y}$ through a separate function $\boldsymbol{f_m}(\boldsymbol{x}; \boldsymbol{\omega})$ parameterized by $\boldsymbol{\omega}$. We want to find a "good" set of parameters $\boldsymbol{\omega}$ such that the function $\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\omega})$ is *likely* to have generated the output data. We define our prior believes about the parameters through the distribution $\boldsymbol{p}(\boldsymbol{\omega})$. The generalized likelihood distribution is then $\boldsymbol{p}(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\omega})$. That is, how likely is output $\boldsymbol{y}$ given model parameters $\boldsymbol{\omega}$ and concrete input $\boldsymbol{x}$. Our assumption here is that the defined model generates the data. The output of a regression model is typically set to be a Gaussian likelihood: $\boldsymbol{p}(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\omega}) = \boldsymbol{\mathcal{N}}(\boldsymbol{y}; \boldsymbol{f_m}(\boldsymbol{x}; \boldsymbol{\omega}), \boldsymbol{\tau^{-1}I})$ where $\boldsymbol{\tau}$ is the model precision. This can be interpreted as adding small amount of noise to the output as modeled in [Gal16].

We can now write out the posterior by using the Bayes' theorem:

$$\boldsymbol{p}(\boldsymbol{\omega}|\boldsymbol{X}, \boldsymbol{Y}) = \frac{\boldsymbol{p}(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{\omega})\boldsymbol{p}(\boldsymbol{\omega})}{\boldsymbol{p}(\boldsymbol{Y}|\boldsymbol{X})} \tag{3.2}$$

This distribution models the most probable function parameters given the observed data. If we choose the parameters from this distribution the observed data will be most likely. For prediction of new data point $\boldsymbol{x^*}$ we use $\boldsymbol{p}(\boldsymbol{y^*}|\boldsymbol{x^*}, \boldsymbol{X}, \boldsymbol{Y}) = \int \boldsymbol{p}(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{\omega})\boldsymbol{p}(\boldsymbol{\omega})\boldsymbol{d\omega}$. This integration is called *inference*([Gal16])). The quantity

$$\boldsymbol{p}(\boldsymbol{Y}|\boldsymbol{X}) = \int (\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{\omega})\boldsymbol{p}(\boldsymbol{\omega})\boldsymbol{d\omega} \tag{3.3}$$

is called marginal likelihood or sometimes "model evidence". The first name comes from the fact that we are marginalizing the likelihood over $\boldsymbol{\omega}$. This factor is independent of the parameters so no matter how we choose them, it does not change. Marginalization plays central part in Bayesian modeling and in the prefect case we would want to marginalize over all possible parameter values $\boldsymbol{\omega}$ with respective weight $\boldsymbol{p}(\boldsymbol{\omega})$. In complex models this is not possible and we resort to approximations.

"Learning" in this Bayesian setting boils down to finding the posterior distribution $\boldsymbol{p}(\boldsymbol{\omega}|\boldsymbol{X}, \boldsymbol{Y})$ over the parameters. In the general case this cannot be done analytically (a closed formula cannot be given) and thus one must attempt to approximate the distribution.

Inference provides us with the full probability distribution over the parameters and for that reason we are not dealing with point estimates of the parameters. By extension we could also provide some distribution when we evaluate the model on unseen data. In order to "generate" a sample from the resulting distribution for some unseen input data point $\boldsymbol{x^*}$ we first need to sample $\boldsymbol{\omega_s}$ from the parameter distribution and then evaluate the model $\boldsymbol{f_e}$ at point $\boldsymbol{x^*}$ parameterized by the drawn sample $\boldsymbol{f_e}(\boldsymbol{x^*}; \boldsymbol{\omega_s})$.

The models that we look next are nothing more than concrete choices for $\boldsymbol{f_e}(\boldsymbol{x}; \boldsymbol{\omega})$

3.2 Bayesian Neural Networks

3.2.1 Neural networks

First we give brief overview of conventional neural networks.

Neural networks (or artificial neural networks) are computing systems that model certain function structure. Neural networks consist of artificial neurons that have a certain number of inputs. They first compute weighted linear sum of their inputs and then apply non-linearity (in form of **tanh**, logistic or other type of *sigmoid* function). Several stacked neurons are considered network layers. Several consecutive layers where the output of one layer is the input of the next are considered a *neural network*.

Formally the neural network is nothing more than a parameterized function with specific structure that allows it to estimate almost every possible function when given the right parameters. Let $\boldsymbol{W_i} \in \mathbb{R}^{\boldsymbol{A_i} \times \boldsymbol{B_i}}$ and $\boldsymbol{b_i}\mathbb{R}^{\boldsymbol{B_i}}$ be weight matrices and biases respectively and let $\boldsymbol{o_i}\mathbb{R}^{\boldsymbol{B_i}}$ denote the output of the $\boldsymbol{i}^{\text{th}}$ layer of the network. For an input $\boldsymbol{x}\mathbb{R}^{\boldsymbol{A_1}}$ we then have:

$$\boldsymbol{o_1} = \mathbf{tanh}((\boldsymbol{W_1 x}) + \boldsymbol{b_1}) \tag{3.4}$$
$$\boldsymbol{o_2} = \mathbf{tanh}((\boldsymbol{W_2 o_1}) + \boldsymbol{b_2}) \tag{3.5}$$
$$\vdots \tag{3.6}$$

In general, for a network with $\boldsymbol{m}$ layers where the $\boldsymbol{m}^{\text{th}}$ layer is the output layer, the outputs of the individual layers can be given as:

$$\boldsymbol{o_{i+1}} = \mathbf{tanh}((\boldsymbol{W_{i+1} o_i}) + \boldsymbol{b_{i+1}}) \tag{3.7}$$
$$\boldsymbol{o_m} = (\boldsymbol{W_m o_{m-1}}) + \boldsymbol{b_m} \tag{3.8}$$

where $\boldsymbol{o_m}$ is the final output of the network. The forward evaluation of the network is just a cascade of applications of liner functions in form of matrix multiplication and additions of bias followed up by application of some non-linear function. For sake of simplicity let us collapse the notation of the output of the network to a single function of the input $\boldsymbol{x}$ and the collection of parameters (weight matrices and bias-terms) $\boldsymbol{\Omega}$. With the evaluation of the netowork can be given with

$$\boldsymbol{o_m} = \boldsymbol{f_{net}}(\boldsymbol{x}; \boldsymbol{\Omega}) \tag{3.9}$$

In order to be useful we have to "teach" the neural network our desired mapping. As in previous section we have a set of input data $\boldsymbol{X} = \{\boldsymbol{x_1}, ..., \boldsymbol{x_N}\}, \forall \boldsymbol{i} : \boldsymbol{x_i} \in \mathbb{R}^M$ and the corresponding output for each $\boldsymbol{x_i}$ is contained in the set for the output data $\boldsymbol{Y} = \{\boldsymbol{y_1}, ..., \boldsymbol{y_N}\}, \forall \boldsymbol{i} : \boldsymbol{y_i} \in \mathbb{R}$. Ideally we would want the following to hold for the neural network

$$\boldsymbol{f_{net}}(\boldsymbol{x_i}; \boldsymbol{\Omega}) = \boldsymbol{y_i} \tag{3.10}$$

In other words, the network have *learned* the exact mapping between $\boldsymbol{X}$ and $\boldsymbol{Y}$ and can predict the output value based on the input one. This is done by choosing appropriate parameter set $\boldsymbol{\Omega}$. In general, finding parameters realising the exact mapping will not be possible but we want to at least approximate this case. This leads us naturally to a definition of measurement of just how wrong the network is. We call this quantity *error function* (also called *loss function*). The error function is a function of the parameter space and it is closer to zero if the given

parameters of the network are better in modeling the output. There are lot of possible choices for loss function but the most common is the *squared error difference*:

$$\boldsymbol{E}(\boldsymbol{\omega}) = \sum_{\boldsymbol{i=0}}^{\boldsymbol{N}} (\boldsymbol{f_{net}}(\boldsymbol{x_i}; \boldsymbol{\omega}) - \boldsymbol{y_i})^{\boldsymbol{2}} \tag{3.11}$$

Here $\boldsymbol{\omega}$ is some choice for the parameters for the networks, namely $\boldsymbol{m}$ weight matrices and $\boldsymbol{m}$ bias terms for network with $\boldsymbol{m}$ layers. Minimizing this *error function* is what we call *learning* the data or *training* the network. The actual method trough which learning occurs is backpropagation. This involves calculating the derivatives of the loss function with the respect to each parameters of the network $\frac{\boldsymbol{\partial E}}{\boldsymbol{\partial W_i}}$. The corresponding parameters are then updated through some gradient descent (see [Rud16] for overview) technique. Some state of the art optimization methods are *LM-BFGS* (Limited Memory Broyden–Fletcher–Goldfarb–Shanno algorithm, [BLNZ95]), Adam optimization algorithm (see [KB14]) and *SGD* (Stochastic gradient descent,[RM51]). To note is that the last motioned method - the SGD - does not calculate the gradient of the loss function using all of the training examples rather it uses only a small subset of the training set.

We would like to note that neural networks are often considered as universal function approximator. This means that given enough complexity of their structure, they can model arbitrary non-linear functions (see [HSW89]).

3.2.2 Neural networks in Bayesian settings

We now focus on what Bayesian neural networks are.

BNNs place a prior distribution over the parameters of a neural network. As described in previous section, once we have distributions on the parameters of a model, we have effectively defined a distribution of functions[1]. Usually, for given weight matrix $\boldsymbol{W_i}$ and bias term $\boldsymbol{b_i}$ in layer $\boldsymbol{i}$ of the network, BNNs place standard Gaussian prior distributions over the matrices, $\boldsymbol{p(W_i) = \mathcal{N}(0, I)}$ and a point estimate for the $\boldsymbol{b}_\text{i}$ is assumed. In this work, however, the standard prior is also placed on the bias vectors so we define the distributions $\boldsymbol{p(b_i) = \mathcal{N}(0, I)}$.

As at their core BNNs are neural networks, we will use the notations from above. The output of the network for a given input $\boldsymbol{x}$ and parameters $\boldsymbol{\Omega}$ is

$$\boldsymbol{f_{net}}(\boldsymbol{x}; \boldsymbol{\Omega}) \tag{3.12}$$

At this moment we have defined what the basic structure of our considered model is (neural network) and know what the priors of the parameters over the model are. As in the previous section we now define a likelihood for each data point. We consider set of data $\boldsymbol{\{(x_i, y_i), i \in \{1, ..., n\}\}}$ where each data point has some features $\boldsymbol{x_i} \in \mathbb{R}^M$ of dimension $\boldsymbol{M}$ and real valued output $\boldsymbol{y_i} \in \boldsymbol{\mathcal{R}}$. We define the likelihood for the BNNs as:

$$\boldsymbol{p(y_i|\Omega, x_i, \sigma^2) = \mathcal{N}(y_i|f_{net}(x_i; \Omega), \sigma^2)} \tag{3.13}$$

where $\boldsymbol{\sigma}^2$ is again some known small variance that can be considered as "noise" of the output ([Gal16]).

[1]Each time when we draw a sample from the parameter space, we have defined a "new" function that models the given data. In this sense, we have sampled a new function from some distribution.

With this overview we can make several general statements about BNN. A BNN is a probability model that utilizes a neural network as universal function approximator. The distributions on the network's parameter allow it to generate a distributional output rather than a point estimate. This the exact same Bayesian setting as described in Section 3.1.

The final piece of having some useful model is to have notion what "learning" means in the case of BNN. As this is Bayesian probability model we have to find the posterior distribution over the parameters based on the example data:

$$\boldsymbol{p(\omega|Y,X) = \frac{p(Y|X,\omega)p(\omega)}{p(Y|X)}} \tag{3.14}$$

This distribution describes the most likely parameters given the example data. Direct computation of the posterior is not possible for networks of any practical sizes. The problems come from the term in the denominator as it involves integrating over the whole parameter space:

$$\boldsymbol{p(Y|X) = \int_{\Omega} p(Y|\omega,X)p(\omega|X)d\omega} \tag{3.15}$$

In the next section we describe a technique that aims to estimate the posterior distribution without actually having to approximate the above integral. Once we have the posterior distribution (or a sufficient estimate of it) we could use it to produce the predicted distribution for unseen data samples. This is done through *inference* as described in Section 3.1.

3.2.3 Variational inference and Kullback-Leibler divergence

As previously stated, it is impractical to compute the full posterior. In fact the true posterior is intractable and cannot be evaluated analytically. We therefore rely on an estimation technique that transforms the Bayesian inference into a optimization problem - *variational inference* (first proposed in [JGJS99], see an overview in [BKM16], a detailed look in [FR12]). We define a *variational* distribution $\boldsymbol{q_\theta(\omega)}$, parameterized by $\boldsymbol{\theta}$. We assume that this distribution is easy to evaluate. Furthermore we would really like this distribution to be able to approximate other distributions based on the choice of parameters $\boldsymbol{\theta}$. If those assumptions are met, we then try to choose such parameters for $\boldsymbol{q_\theta(\omega)}$ that it maximally resembles the posterior $\boldsymbol{p(\omega|Y,X)}$. For that we also need some measure between two distributions that gives us how "similar" they are. This is slightly analogous to the choice of loss function that is to be minimized. In this case we use the *Kullback–Leibler* (KL) divergence proposed by [KL51]. KL is defined as

$$\boldsymbol{KL(q_\theta(\omega), p(\omega|X,Y)) = \int q_\theta(\omega) \log \frac{q_\theta(\omega)}{p(\omega|X,Y)} d\omega} \tag{3.16}$$

KL is a measure of how one probability distributions differs from another. A Kullback–Leibler divergence of $\mathbf{0}$ implies the two distributions are identical and it does not have a maximum as it is unbounded. We aim to minimize this function with respect to $\boldsymbol{\theta}$. Our problem then becomes:

$$\boldsymbol{\theta^*} = \boldsymbol{\arg\min_\theta KL(q_\theta(\omega), p(\omega|X,Y))} \tag{3.17}$$

$$= \boldsymbol{\arg\min_\theta \int q_\theta(\omega) \log \frac{q_\theta(\omega)}{p(\omega|X,Y)} d\omega} \tag{3.18}$$

$$= \boldsymbol{\arg\min_\theta KL(q_\theta(\omega), p(\omega)) - \mathbb{E}_{q_\theta(\omega)}[\log\, p(Y|w)]} \tag{3.19}$$

The last cost function is known the *variational free energy* ([NH98],[BCKW15]) or the *expected lower bound* ([FR12], [SJJ96]). Thus the function that we want to optimize is

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{Y}) = \boldsymbol{KL}(q_{\boldsymbol{\theta}}(\boldsymbol{\omega})||p(\boldsymbol{\theta})) + \mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{\omega})}[\log(p(\boldsymbol{Y}|\boldsymbol{\omega}))] \tag{3.20}$$

This loss function is a sum of a data-dependent part and a prior-dependent part. The function embodies a trade-off between satisfying the complexity of the data - $\boldsymbol{X}$ and $\boldsymbol{Y}$ - and satisfying the simplicity of the prior $\boldsymbol{p(\omega)}$([BCKW15]).

Gradient calculation of this function is not a trivial task. As we do use standard Gaussian priors on the parameter $\boldsymbol{\omega}$, the first term in the function (the KL divergence) is traceable and analytically solvable ([PBJ12]). For the actual implementation of the optimization we use [AAB+15] (more details in Section 6.2). The other term however, is not that well behaved. Several methods have been proposed to optimize the function with backpropagation techniques. The concrete implementation that we ([TKD+16]) use, utilizes either *Stochastic Gradient Vitiation Bayes* (*SGVB*) as described in [KW13] or the method described in [PBJ12].

To give brief overview of SGVB - for the approximate posterior $\boldsymbol{q_\theta(\omega)}$ we reparameterize the random variable $\boldsymbol{z} \sim \boldsymbol{q_\theta(\omega)}$ by the use of some differential transformation $\boldsymbol{g_\omega(\epsilon, x)}$ and a noise variable $\boldsymbol{\epsilon}$.

$$\boldsymbol{z} = \boldsymbol{g_\omega(\epsilon, x)}, \quad \boldsymbol{\epsilon} \sim \boldsymbol{p(\epsilon)} \tag{3.21}$$

With those definitions we can now estimate the exception of some function $\boldsymbol{f(x)}$ with respect to $\boldsymbol{q_\theta(\omega)}$:

$$\mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{\omega})}[\boldsymbol{f(x)}] = [\boldsymbol{f(g_\omega(\epsilon, x))}] \simeq \frac{\boldsymbol{1}}{\boldsymbol{L}} \sum_{\boldsymbol{l=1}}^{\boldsymbol{L}} \boldsymbol{f(g_\omega(\epsilon^{(l)}, x))} \tag{3.22}$$

We could apply this technique to our cost function and we get a tractable differentiable estimate of it

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{Y}) \simeq \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{Y})^{\boldsymbol{E}} = \boldsymbol{KL}(q_{\boldsymbol{\theta}}(\boldsymbol{\omega})||p_{\boldsymbol{\theta}}(\boldsymbol{z})) + \frac{\boldsymbol{1}}{\boldsymbol{L}} \sum_{\boldsymbol{l=1}}^{\boldsymbol{L}} (\log(p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{\omega}))) \tag{3.23}$$

Now that we have a function for which the gradients could be given analytically, we could optimize it with respect to the parameter $\boldsymbol{\theta}$. This is done typically through Stochastic Gradient Descent (see Section 3.2.1). In this case we do not calculate the loss on the whole dataset but rather on s sub-sample of it. Lets say we take $\boldsymbol{M}$ datapoints from dataset with $\boldsymbol{N}$ datapoints. Then essentially we are making the estimate:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{Y}) \simeq \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{Y}^{\boldsymbol{M}})_{\boldsymbol{M}}^{\boldsymbol{E}} = \frac{\boldsymbol{N}}{\boldsymbol{M}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{M}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{y_i})^{\boldsymbol{E}} \tag{3.24}$$

It is known that SGD leads to faster convergence with less computations ([RM51]). It is also proven and observed ([SW96]) that SGD converges to the result of regular gradient descent. For those reasons, the usage of SGD is almost always advisable.

There is also a somewhat simpler approach of estimating the gradient of the loss function described in [PBJ12]. Under some regularity conditions, a gradient of the form $\boldsymbol{\nabla_\theta \mathbb{E}_q f}$ where $\boldsymbol{f}$ is some untraceable function, $\boldsymbol{q}$ is some distribution and we take the gradient with respect to $\boldsymbol{\theta}$) could be rewritten as:

$$\boldsymbol{\nabla_\theta \mathbb{E}_q f(\theta)} = \boldsymbol{\nabla_\theta} \int_{\boldsymbol{\theta}} \boldsymbol{f(\theta) q(\theta) d\theta} \tag{3.25}$$

$$= \int_{\boldsymbol{\theta}} \boldsymbol{f(\theta) \nabla_\theta q(\theta) d\theta} \tag{3.26}$$

$$= \int_{\boldsymbol{\theta}} \boldsymbol{f(\theta) q(\theta) \nabla_\theta \ln(q(\theta)) d\theta} \tag{3.27}$$

The last integral can be then estimated through Monte Carlo integration ([HH64]). This approach could be used to replace the $\mathbb{E}_{q_\theta(\omega)}[\log(p(Y|\omega))]$ in the cost function. We can then again apply stochastic gradient descent in order to spare some computations and accelerate convergence. For more detail on the method see [PBJ12].

Other techniques for posterior estimation rely on sampling from the posterior. This can be done for example by *Gibbs sampling* ([GG84]), *Metropolis-Hastings* ([MRR+53], [Has70]) or *Hybrid Monte Carlo* ([DKPR87]). These are all so called *Markov Chain Monte Carlo* (*MCMC*) methods and are not uncommon techniques. We however don't consider them here.

3.3 Mixture Density Networks

In this section we describe an alternative approach in generating a probability model for given data set. The first thing to consider in the mixture density networks approach is the way we model each data point. We try to describe each data point $(\boldsymbol{x}, \boldsymbol{y}) \in \boldsymbol{X} \times \boldsymbol{Y}$ with a feature vector $\boldsymbol{x} \in \mathbb{R}^{\boldsymbol{R}}$ and a value $\boldsymbol{y} \in \mathbb{R}$ directly through some mixture model.

$$\boldsymbol{p(y|x)} = \sum_{\boldsymbol{m=1}}^{\boldsymbol{M}} \boldsymbol{\alpha_m(x)\phi_m(y|x)} \tag{3.28}$$

where $\boldsymbol{M}$ is the number of mixture components and $\boldsymbol{\phi_m(y|x)}$ is the conditional density of the $\boldsymbol{m^{th}}$ kernel. The parameters (or weights) $\boldsymbol{\alpha_m(x)}$ are called *mixing coefficients* ([MB94]). To note is that everything in the model depends on the concrete input data $\boldsymbol{x}$. In this sense the parameters for the mixture change for every data point. We choose mixture of Gaussians as with appropriate weights it can model any arbitrary distribution. This means that theoretically MDNs with Gaussian mixture can be universal distribution estimators [2]. Thus all of the density functions become:

$$\boldsymbol{\phi_m(y|x)} = \frac{\boldsymbol{1}}{\boldsymbol{2\pi^{c/2}\sigma_m(x)^c}} \boldsymbol{\exp\{-\frac{\|y-\mu_m(x)\|^2}{2\sigma_m(x)^2}\}} \tag{3.29}$$

Again, the parameters of each mixture component, $\boldsymbol{\mu_m(x)}$ and $\boldsymbol{\sigma_m^2(x)}$, depend on the input data. The last equation assumes that the components of the output vector are statistically independent within each component of the distribution, and can be described by a common variance ([MB94]). As we deal with regression in this thesis, the output vector is one dimensional and thus the assumption holds true.

With this setup, the model provides general and arbitrary conditional density function $\boldsymbol{p(y|x)}$ for the output. We treat the parameters of the mixture - the mixing coefficients $\boldsymbol{\alpha}_{\mathrm{m}}\boldsymbol{(x)}$, the means $\boldsymbol{\mu_m(x)}$ and the variances $\boldsymbol{\sigma_m(x)}$ - as continuous functions of the input $\boldsymbol{x}$. It is clear that with the proper parameter functions we can model each data point. The "finding" of this function is achieved through a conventional neural network which takes $\boldsymbol{x}$ as an input. This combination of mixture model and a neural network is what *Mixture Density Networks* (MDNs) are. It is therefore said that MDNs are nothing more than neural network plus a mixture of Gaussians. An overview is given in Figure 5

Here we again use the nice property of neural networks as universal function estimators. Before we proceed further however we must place some restrictions on the outputs of

[2] Similar to regular neural networks which are universal real valued function estimators

the network. Let $\boldsymbol{z}_m^\alpha$, $\boldsymbol{z}_m^\mu$ and $\boldsymbol{z}_m^\sigma$ be the outputs of some neural network $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\omega})$ with some parameter set $\boldsymbol{\omega}$ where $\boldsymbol{z}_m^\alpha$ affect the mixing coefficients $\boldsymbol{\alpha}_m$ and $\boldsymbol{z}_m^\mu$ and $\boldsymbol{z}_m^\sigma$ affect the means and variances of the mixture components respectively.

Now we give the exact relationships between the outputs and the mixture's parameters. The mixing coefficients of the mixture obviously must sum up to one as they are to be treated as probabilities:

$$\sum_{m=1}^{M} \boldsymbol{\alpha}_m(\boldsymbol{x}) = \mathbf{1} \tag{3.30}$$

We can achieve that by placing *softmax* function (popularized by [Bri90]) on the respective outputs $\boldsymbol{z}_m^\alpha$ of the network

$$\boldsymbol{\alpha}_m = \frac{\boldsymbol{e}^{z_m^\alpha}}{\sum_{j=1}^{M} \boldsymbol{e}^{z_j^\alpha}} \tag{3.31}$$

where $\boldsymbol{z}_i^\alpha$ are the outputs of the networks "responsible" for the mixing coefficients $\boldsymbol{\alpha}_i$

The variances of the Gaussian distributions represent *scales* and for that they must be positive. We ensure this through the use of exponential function

$$\boldsymbol{\sigma}_m = \boldsymbol{e}^{z_m^\sigma} \tag{3.32}$$

This also avoids the problem when some of the variances goes to zero. The means $\boldsymbol{\mu}_m(\boldsymbol{x})$ are to be seen as *location* parameters and in order not to introduce bias in the system we model them directly with the network's corresponding outputs

$$\boldsymbol{\mu}_m = \boldsymbol{z}_m^\mu \tag{3.33}$$

With those definitions we can summarize - for a network with $\boldsymbol{M}$ mixtures the output would be $\{\boldsymbol{\mu}_m, \boldsymbol{\sigma}_m^2, \boldsymbol{\pi}_m\}, \boldsymbol{m} \in \{\mathbf{1}, ..., \boldsymbol{M}\}$. With the outputs we can define conditional probability distribution for the data

$$\boldsymbol{p}(\boldsymbol{Y}|\boldsymbol{X}) = \sum_{m=1}^{M} \boldsymbol{\pi}_m(\boldsymbol{X}) \boldsymbol{\mathcal{N}}(\boldsymbol{y}|\boldsymbol{\mu}_m(\boldsymbol{X}), \boldsymbol{\sigma}_m^2(\boldsymbol{X})) \tag{3.34}$$

Now we just have to give a loss function to be minimized. Starting from probability perspective, we want to maximize the likelihood of the the observed data $\{(\boldsymbol{x}_n, \boldsymbol{y}_n)\}_1^N$, This is given by a product over the likelihoods of each data point:

$$\boldsymbol{\mathcal{L}} = \prod_{n=1}^{N} \boldsymbol{p}(\boldsymbol{x}_n, \boldsymbol{y}_n) = \prod_{n=1}^{N} \boldsymbol{p}(\boldsymbol{x}_n) \boldsymbol{p}(\boldsymbol{y}_n|\boldsymbol{x}_n) \tag{3.35}$$

The term $\boldsymbol{p}(\boldsymbol{x}_n)$ does not depend on the networks weights and can be spared. As for the $\boldsymbol{p}(\boldsymbol{y}_n|\boldsymbol{x}_n)$ - this is where the mixture model comes into play. The quantity to be maximized thus becomes

$$\tilde{\boldsymbol{\mathcal{L}}} = \prod_{n=1}^{N} \sum_{m=1}^{M} \boldsymbol{\alpha}_m(\boldsymbol{x}_n) \boldsymbol{\mathcal{N}}(\boldsymbol{y}_n; \boldsymbol{\mu}_m(\boldsymbol{x}_n), \boldsymbol{\sigma}_m^2(\boldsymbol{x}_n)) \tag{3.36}$$

Maximizing $\tilde{\boldsymbol{\mathcal{L}}}$ is equivalent to minimizing

$$\boldsymbol{E} = \sum_{n=1}^{N} \boldsymbol{E}^n \tag{3.37}$$

where $\boldsymbol{E^n}$ is the contribution of the $\boldsymbol{n^{th}}$ data point and

$$\boldsymbol{E^n} = -\log\Big(\sum_{\boldsymbol{m=1}}^{\boldsymbol{M}} \boldsymbol{\alpha_m(x_n)\mathcal{N}(y_n; \mu_m(x_n), \sigma_m^2(x_n))}\Big) \tag{3.38}$$

This function $\boldsymbol{E}$ can be minimized with standard backpropagation technique ([BNVP01]) as its derivatives are fully tractable. Exact expressions for the derivatives of $\boldsymbol{E}$ are derived and given in [MB94] but we don't consider them here. Because the used implementation of MDNs ([MvN+17]) uses *tensorflow* ([AAB+15]) internally, the automatic differentiation engine does the necessary work. More details in Section 6.2

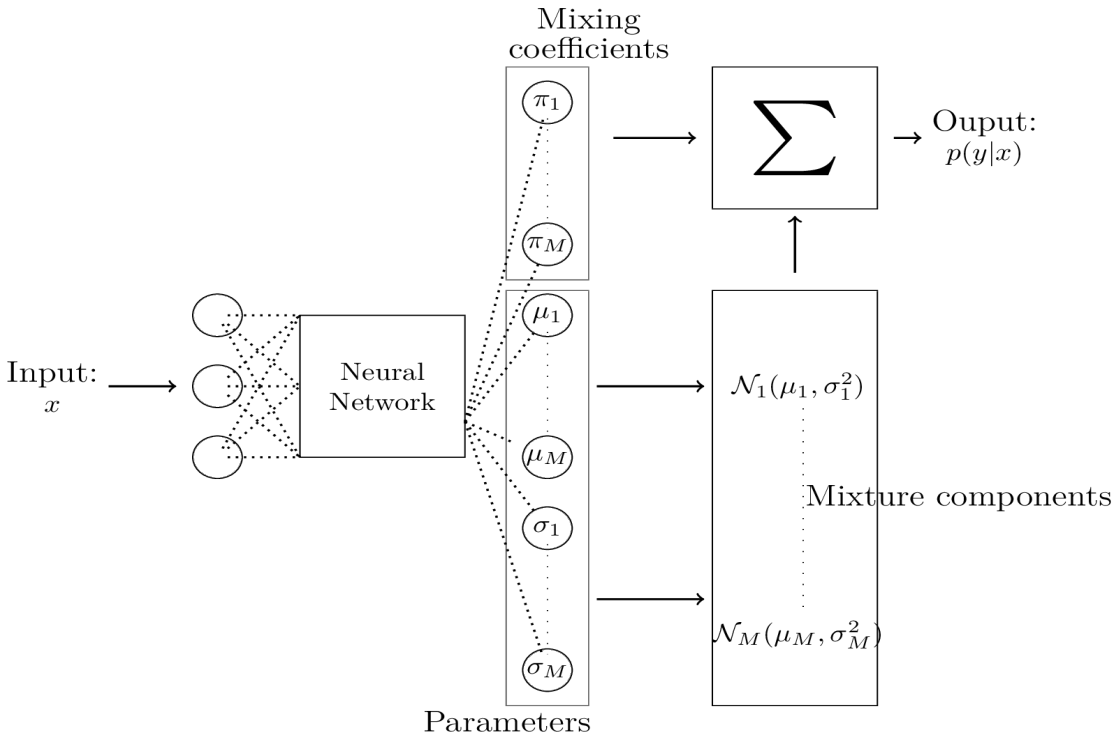

Figure 5: Structure of a Mixture Density Network. The Input $\boldsymbol{x}$ first gets passed through a neural network. The outputs of that are treated as parameters for a mixture model. The mixture model is a weighted linear sum of different distributions - normal distributions in our case. To note is that the mixing coefficients (the weights for the sum) are also generated from the network.

4 Model evaluation methods

In this section we give formal explanation to our approach in evaluating the models we build.

4.1 Probabilistic Forecast

First we look at more concretely how the considered stochastic models differ form classical ones.

4.1.1 Prediction spaces

Classical regression models produce a point estimate $\boldsymbol{y_i^e}$ for each sample data point $\boldsymbol{x_i}$ with an actual realization $\boldsymbol{y_i}$. In this thesis however, we are interested in regression models whose output is probabilistic forecast $\boldsymbol{F}$ that can be identified with with the associated *cumulative distribution function* (*CDF*). This introduces the notion of prediction spaces proposed in [GR13]. A prediction space is a probability space tailored to the study of distributional forecasts ([GK14]). Formally looked we have to consider the realized observation also as distributions even thou in practice this is a single value. In the most general case, the elements of the prediction space can be seen as tuples $\boldsymbol{(F, Y)}$ where $\boldsymbol{F}$ is the probabilistic forecast and $\boldsymbol{Y}$ is the true distribution. $\boldsymbol{F}$ is CDF-valued quantity which caries some information about the training data, the models' parameters, certain assumptions that are considered met and other model specific properties. Let $\boldsymbol{\mathcal{A}}$ denote all of the possible information that is available to make a forecast and let $\boldsymbol{\mathcal{L}}$ be some conditional distribution. We can say that the forecast $\boldsymbol{F}$ is ideal relative to the information encoded by $\boldsymbol{\mathcal{A}}$ if $\boldsymbol{F = \mathcal{L}(Y|A)}$. This means that the forecast utilizes the information completely.

In our concrete case, the model - BNN or MDN - is the mechanism which generates the forecast. The models are trained on the training set and thus can "gather" information about the data. When making a prediction about new data point, the models use the gathered data and produce a probabilistic forecast in form of either CDF or a set of samples drawn from the predicted distribution.

4.1.2 Dispersion and sharpness

Dispersion and sharpness are properties of predictive forecasts that describe the relationship between the observed value and the prediction. In order to formalize them however, we need to introduce the notion of probability integral transform (see [DGT98],[GBR], [Daw84]).

Again, let $\boldsymbol{F}$ be the predicted distribution for observation $\boldsymbol{Y}$. The probability random transform (PIT) is the random variable $\boldsymbol{Z_F = F(Y)}$. Intuitively the PIT is the value of the predictive CDF.

With that we can formally define what dispersion and calibration are. In the following definitions $\boldsymbol{F}$ and $\boldsymbol{G}$ are two forecasts and their PITs are respectively $\boldsymbol{Z_F}$ and $\boldsymbol{Z_G}$. We say that

- the forecast $\boldsymbol{F}$ is marginally calibrated if $\mathbb{F}(\boldsymbol{y}) = \mathbb{P}(\boldsymbol{Y \leq y})\, \forall \boldsymbol{y} \in \mathbb{R}$.
- the forecast $\boldsymbol{F}$ is probabilistically calibrated if its PIT $\boldsymbol{Z_F}$ has standard uniform distribution.
- the forecast $\boldsymbol{F}$ is overdispersed if $\boldsymbol{var(Z_F) < \frac{1}{12}}$ and underdispersed if $\boldsymbol{var(Z_{\mathrm{F}}) > \frac{1}{12}}$.
- the forecast $\boldsymbol{F}$ is more dispersed if $\boldsymbol{var(Z_F) < var(Z_G)}$.

As mentioned in Section 1.2 calibration has to do with both the observation and the forecast. According to [GK14], if a forecast is ideal relative to some information set then it is both marginally and probabilistically calibrated.

As explained in [GK14] the sharpness of a given forecast describes the concentration of the distribution without consideration of the actual observed value. When it comes to forecasts for a real-valued variable, sharpness can be assessed in terms of the associated prediction intervals. The mean widths of these intervals should be as short as possible.

In next section we'll explain a certain class of scoring rules that can evaluate a predicted forecast against the observed value with respect to the dispersion and sharpness properties i.e. they assign "better" score to forecast with "better" dispersion and sharpness.

4.2 Proper Scoring rules

Proper scoring rules allow us to measure the predictive capability of a given forecast distribution. Those rules can consider the sharpness as well as the concentration of the distribution. The generated scores are measurement of differences between a desired and generated distributions and thus we strive to minimize those scores. Proper scoring rules are explained in great detail in [GBR]. A scoring rule assigns a numerical score $\boldsymbol{S(F, y)}$ to each pair $\boldsymbol{(F, y)}$ of probabilistic forecast F in a form of probability distribution and an observed value $\boldsymbol{y} \in \mathbb{R}$. In general a scoring rule is some function $\boldsymbol{S : \mathcal{F} \times \mathbb{R} \to \mathbb{R}}$ where $\boldsymbol{\mathcal{F}}$ is some class of probability distributions. A scoring rules generally evaluates a predicted distribution against an realized observation. Let the desired distribution be $\boldsymbol{G \in \mathcal{F}}$. Then we write

$$\boldsymbol{S(F, G) = \mathbb{E}_G[S(F, Y)]} \tag{4.1}$$

for the expected score between $\boldsymbol{G}$ and the forecast $\boldsymbol{F}$. The rule $\boldsymbol{S : \mathcal{F} \times \mathbb{R} \to \mathbb{R}}$ where $\boldsymbol{\mathcal{F}}$ is proper if the following property holds

$$\boldsymbol{S(G, G) \leq S(F, G), \quad \forall G, F \in \mathcal{F}} \tag{4.2}$$

This means that a proper scoring rules yields a minimal score for the true distribution of the observation. We consider three proper scoring rules. The following section discuss those.

4.2.1 Logarithmic score (LS)

The most straight forward scoring rule is probably the *logarithmic score*. This scoring rule well established ([Goo52]). For a given density function $\boldsymbol{f^p(x)}$ that belongs to a predicted forecast, the LS is given by

$$\boldsymbol{LS(f^p, y) = -\log\ f^p(y)} \tag{4.3}$$

where $\boldsymbol{y}$ is the realized observation. The LS is lower (i.e. better) if according to the predicted forecast, the true observation is more probable. The LS would be at its lowest if the observation $\boldsymbol{y}$ is the maximum of the density $\boldsymbol{f^p}$. In this case, the actual observation will be the most probable value for the prediction according to the generated forecast. It is clear that sharper distributions will have lower LS given the observations falls into them. More concentrated distributions have higher values for their densities and thus are more certain for given observation.

The LS is the basis for a class of scoring rules called *local scoring rules*. Those are discussed in detail in [EG12].

A possible disadvantage of the LS is that it requires the density function of the forecast in order to be calculated. If the forecast is not given in density form, the probability density function must be approximated. For simple distributions this may not be a problem but for complicated ones, a considerable amount of samples from the distribution may be required. This could make the LS impractical for some probability models.

4.2.2 Continuously ranked probability score (CRPS)

The mentioned disadvantage of the LS is addressed by the so called *continuously ranked probability score* (CRPS). The CRPS uses the predictive cumulative density function (CDF) of the forecast to give it score. CRPS is denied as

$$\boldsymbol{CRPS}(\boldsymbol{F}, \boldsymbol{y}) = \int_{-\infty}^{\infty} (\boldsymbol{F}(\boldsymbol{x}) - \mathbb{1}\{\boldsymbol{y} \leq \boldsymbol{x}\})^2 \boldsymbol{dx} \tag{4.4}$$

$$= \mathbb{E}_{\boldsymbol{F}}|\boldsymbol{Y} - \boldsymbol{y}| - \frac{\mathbf{1}}{\mathbf{2}}\mathbb{E}_{\boldsymbol{F}}|\boldsymbol{Y} - \boldsymbol{Y'}| \tag{4.5}$$

where $\boldsymbol{Y}$ and $\boldsymbol{Y'}$ are are independent random variables with CDF $\boldsymbol{F}$ and finite first moment ([GR07], [MW76]) and $\mathbb{1}\{\boldsymbol{y} \leq \boldsymbol{x}\}$ is the probability for $\boldsymbol{y}$ to be less than or equal to $\boldsymbol{x}$. In plain words the definition says that the CRPS is a quadratic measure of the difference between the forecast's CDF and the empirical CDF of the observation. In contrast to the LS, the CRPS considers the distribution as a whole and doesn't focus on one specific point. CRPS is well studied and it finds application in a lot of cases (see [GR11] for weighted variant and [GGBJ07] for usage for circular variables). Figure 6 gives visual intuition of the CRPS. From the figure one can clearly see that if the variance of the predicted distribution is big, the CDF will have values different form **0** and **1** in bigger interval and thus the CRPS will be higher. This shows how the CRPS too encourages forecasts to be sharper with their distribution.

A useful property of the CRPS is that it reduces to the *mean absolute error* (*MAE*) if the forecast is deterministic. This allows us to compare probabilistic forecasts with point estimates in consistent manner ([GK14]).

4.2.3 Dawid–Sebastiani score (DSS)

For complex distributions even the CRPS can be hard to evaluated as it requires some estimate of the integral in Equation 4.4. The *Dawid–Sebastiani score* (*DSS*), introduced in [DS99], relies only on the first two central moments of a given predicted forecast $\boldsymbol{F}$. DSS is defined as

$$\boldsymbol{DSS}(\boldsymbol{F}, \boldsymbol{y}) = \frac{(\boldsymbol{y} - \boldsymbol{\mu_F})^2}{\boldsymbol{\sigma_F^2}} + \mathbf{2}\log(\boldsymbol{\sigma_F}) \tag{4.6}$$

Of the three rules the DSS is most simple to calculate as it requires only two simple metrics of the forecast. The only thing we need is the ability to draw samples from the distribution of the forecast and we could estimate the mean and variance.

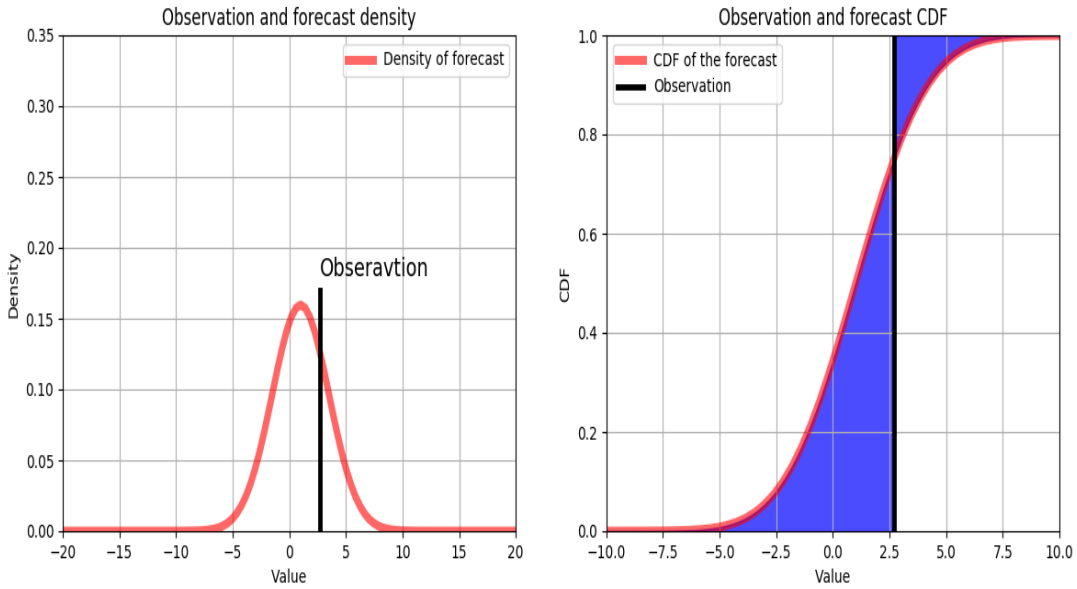

Figure 6: The square of the blue area is the value of the CRPS. Notice how if the predicted CDF were a step function (the case when the prediction is deterministic point estimate), the CRPS will be nothing more than the squared differnece between the observation and the prediction.

4.2.4 Computation of scoring rules

In this thesis we deal with sets of data and we need to evaluate the scoring rules on certain number of data points - on the test or train set. For this we take the average over the scores of all examples in a set. Let $\{\boldsymbol{x_n}, \boldsymbol{y_n}\}_{n=1}^{N}$ be our set of interest and some model have generated set of forecasts $\{\boldsymbol{F_n}\}_{n=1}^{N}$ where forecast $\boldsymbol{F_i}$ is based on the features $\boldsymbol{x_i}$ and tries to predict the observation $\boldsymbol{y_i}$. The overall score $\bar{\boldsymbol{S_N}}$ of the scoring rule $\boldsymbol{S}$ then is

$$\boldsymbol{S_N} = \frac{\mathbf{1}}{\boldsymbol{N}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} \boldsymbol{S(F_i, y_i)} \tag{4.7}$$

This is the final metric on which we compare several probability models.

When it comes to calculating the rules themselves on concrete observations there are explicit forms for the three rules if the predicted forecast is normal distribution. Those are given in Table 4.2.4 and taken from [GK14].

Scoring rule $\boldsymbol{S}$	$\boldsymbol{S(\mathcal{N}(\mu, \sigma^2), y)}$
Logarithmic score	$\frac{(y-\mu)^2}{2\sigma^2} + \log(\sigma) + \frac{1}{2}\log(2\pi)$
Continuous ranked probability score	$\sigma(\frac{y-\mu}{\sigma}(2\Phi(\frac{y-\mu}{\sigma}) - 1) + 2\phi(\frac{y-\mu}{\sigma}) - \frac{1}{\pi})$
Dawid–Sebastiani score	$\frac{(y-\mu)^2}{\sigma^2} + 2\log(\sigma)$

Table 1: Closed forms of the scoring rules for normal distribution

All of the examined model in the thesis can generate samples from their predicted forecasts. The DSS can thus be trivially calculated just through the empirical mean and variance of the generated samples. The predicted density function of the MDN is a sum of density functions of the mixture components (see Section 3.3) so it can easily be evaluated with the LS. For the BNN we use density estimation methods described in [Sil86] and [Sco15] to calculate the LS. Empirical CDFs are trivially computable with sample data. For sample set $\{\boldsymbol{x}_i\}_i = \mathbf{1}^n$ the empirical CDF is

$$\bar{\boldsymbol{F}}_n(\boldsymbol{t}) = \frac{\mathbf{1}}{\boldsymbol{n}} \sum_{i=1}^{n} \mathbf{1}_{x_i \leq t} \tag{4.8}$$

wehre (4.9)

$$\mathbf{1}_{a \leq t} = \begin{cases} \mathbf{1} & \text{, if } \boldsymbol{a} \leq \boldsymbol{t} \\ \mathbf{0} & \text{, if } \boldsymbol{a} > \boldsymbol{t} \end{cases} \tag{4.10}$$

With this function we can also calculate the CRPS in straightforward manner following the definition in Equation 4.4.

4.3 Rank Histogram

Rank histograms (sometimes called *verification rank histogram*) are a tool for evaluating ensemble forecasts ([Ham01]). In our work however we adapt them to evaluate forecasts in form of probability distributions. The underlying assumption is that the ensemble member forecasts are distributed so as to delineate ranges. Let $[\boldsymbol{y}_1^p, ..., \boldsymbol{y}_M^p]$ be a *sorted* ensemble where each $\boldsymbol{y}_i^p$ is a predicted value for the true observation $\boldsymbol{y}^t$. The rank of $\boldsymbol{y}^t$ according to the ensemble is an integer $\boldsymbol{k}$ such that

$$\boldsymbol{rank}(\boldsymbol{y}^t) = \begin{cases} \mathbf{0}, & \text{if } \boldsymbol{y}^t < \boldsymbol{y}_1^p \\ \boldsymbol{k}, & \text{if } \boldsymbol{y}_k^p \leq \boldsymbol{y}^t < \boldsymbol{y}_{k+1}^p \\ \boldsymbol{k+1}, & \text{if } \boldsymbol{y}_{k+1}^p < \boldsymbol{y}^t \end{cases} \tag{4.11}$$

In our case the sorted ensemble is just a sorted sequence of drawn samples from the distribution predicted by the given model. We calculated the rank of each observation and then create a histogram with certain number of bins. To be noted is that the number of bins is considerably less than the number of samples. The amount of samples drawn from each model for each observation is in the order of tens of thousand while the number of bins for the rank histogram is no more than 20.

For a well calibrated forecast, the rank histogram should be uniform. U-shaped histograms indicate underdispersed predictive distributions, whereas hump or inverse-U-shaped histograms correspond to overdispersed predictive distributions ([GK14], [DGT98], [Ham01]). If the histogram is uniform we can say that the observation behaves as a sample from the forecast. This is essentially what a good forecast should exhibit. Figure 7 illustrates the said about rank histograms.

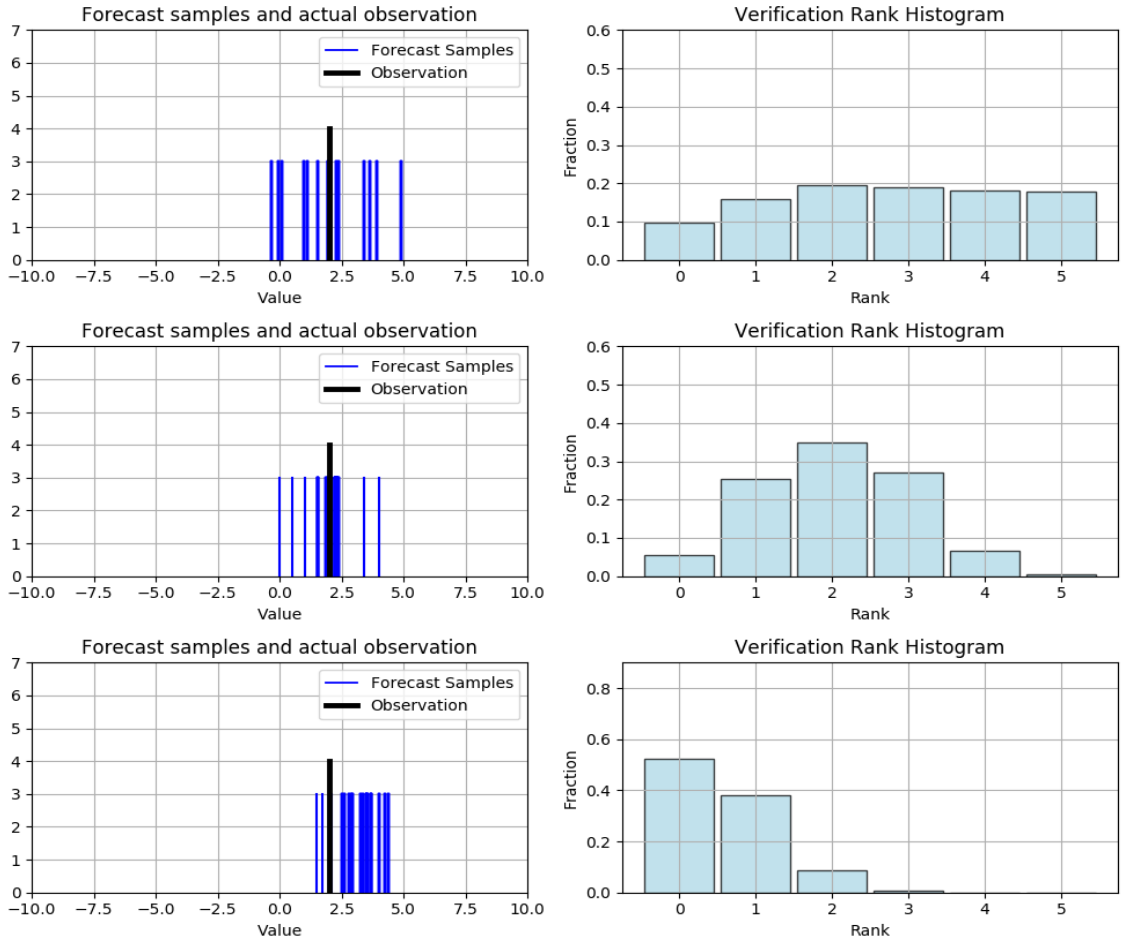

Figure 7: Histograms illustrating different types of distributions with respect to the actual observations. To note is that on left we give only one example of samples drawn form the distribution in respect to a given observation. We assume that the sample sets for the other observations are similar and that this will produce the respective histograms on the right. A rank histogram cannot be given for a single observation and a single set of drawn samples.

4.4 Feature Importance

In order to asses the relative importance of the features used by the build models we use a method called *permutation importance*. It is first described in [Bre01] and it was applied to random forests. The techniques is also successfully applied in assessing the feature importance of ensemble methods for forecasting ([RL18]). We closely follow the approach of [RL18] by randomly shuffling each feature in the test and training sets and then look at the mean value of a given scoring rule. For each feature we examine the difference in the mean value of the scoring rules between the permuted and regular datasets.

We give formal definition of the described above. We consider a set of data $\{\boldsymbol{x_n}, \boldsymbol{y_n}\}_{\boldsymbol{n=1}}^{\boldsymbol{N}}$ with feature vectors $\boldsymbol{x_n} \in \mathbb{R}^M$ and realized values $\boldsymbol{y_n} \in \mathbb{R}$. We define a matrix $\boldsymbol{X}$ with the features of all examples as

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x_1} \\ \boldsymbol{x_2} \\ \vdots \\ \boldsymbol{x_N} \end{bmatrix} \tag{4.12}$$

In this notation, the $\boldsymbol{i^{th}}$ row of the matrix are all of the features of the $\boldsymbol{i^{th}}$ example and the $\boldsymbol{i^{th}}$ column of the matrix are all of the $\boldsymbol{i^{th}}$ features of all examples. We further define $\boldsymbol{F|x}$ as the conditional forecast distribution given a vector of features of a input data point. We assume that $\boldsymbol{F|x}$ is predicted by some model. In essence we evaluate the importance of the used feature only in the context of the considered model. In this sense we examine which of the features have higher information value for the predictions of the given model.

Now we give formal definition for the permutation of the features. Let $\boldsymbol{X_s}$ denote the $\boldsymbol{s^{th}}$ column of the input data $\boldsymbol{X}$. Then the permuted set $\boldsymbol{X}^{\mathrm{perm_j}}$ for the $\boldsymbol{j^{th}}$ feature is given by

$$\boldsymbol{X}^{\mathrm{perm_j}} = \begin{cases} \boldsymbol{X_s}, & \text{if } \boldsymbol{j \neq s} \\ \boldsymbol{\pi(X_{(s)})}, & \text{if } \boldsymbol{j = s} \end{cases} \tag{4.13}$$

where $\boldsymbol{p(x)}$ is some random permutation of the elements of the vector $\boldsymbol{x}$. Now we can give definition for the quantity that we call feature importance.

$$\textit{Importance}(\boldsymbol{j}) = \frac{\boldsymbol{1}}{\boldsymbol{N}} \sum_{\boldsymbol{n=1}}^{\boldsymbol{N}} (\boldsymbol{SR(F|X^{perm_j,(n)}, y_n) - SR(F|X^{(n)}, y_n)}) \tag{4.14}$$

where $\mathrm{X^{(n)}}$ is the feature vector of the $\boldsymbol{n^{th}}$ datapoint (i.e. the $\boldsymbol{n^{th}}$ row of the matrix $\boldsymbol{X}$) and SR is some scoring rule - CRPS, DSS or LS in our case.

5 Results

5.1 Data used

5.2 Evaluation of the models

6 Implementation

In order to enforce separation between the code and all of the data that the code act upon, we place all of the generated data in the `env` directory of the project. This directory is added to the `.gitignore` file so it the implantation in no way relies on it. The contents of the folder are fully generated with the execution of the implemented scripts. In the next section we explain exactly what is the generated content.

6.1 Preparing the data

In Section 1.3 we mentioned that the public data that we use is less than ideal. We've also outline some of the problems that we found in the data. Now we like to explain how we deal with these problems. We've written several python scripts dealing with the prepossessing of the data.

The raw data files are located under [dat]. They are organized in folders by day. The folder for a forgiven day contains different files from different sensor types. We are only interested in the air pollution data form the sds011 sensors. Each data file for this type of sensor is a text file with *comma separated values* (*CSV*) where each row has measurements for PM2.5 and PM10 (see Section 1.3), the location of the sensor, the time of day when the measurement was taken as well as the sensor id. Our final goal in the data preprocessing in to generate a DataFrame where each column is labeled with sensor id and either P1 (for PM2.5) or P2 (for PM10) and each row contains measurements of the corresponding integrated values for a certain period of time. The periods of time in which we integrate of course must be synchronized. In end effect we generate three different DataFrames from the data - with integration over whole days, over twelve hours and over one hour time periods.

We try to automate everything from downloading the data, extracting the relevant pieces of it, sanitizing it and finally generating a single *DataFrame* object of *pandas* ([McK10]). As data prepossessing is not the focus of this thesis we don't explain each script in absolute detail but rather explain what happens in it.

All of the scripts we explain in this section require a configuration file. This file is `config.json`. In it we set certain properties for the scripts so that we could easily change them if necessary. In each of the scripts there is a `_read_config` method that loads the relevant for the script properties from the configuration file.

6.1.1 Download module

The amount of files under [dat] is too big for the files to be downloaded manually. Therefore the first step of the process was to write a script that downloads the relevant files. This is the functionality provided by the `download_module.py` file. In the configuration file there is a section with properties specifically for this script. Through those we can specify:

- `base_url` - the base link where the files are located. In our case `https://archive.luftdaten.info/`
- `start_date`, `end_date` - those specify the considered period of time. Only data files from this period will be downloaded. We've download files for the whole year of 2017.
- `sensor_type` - this fields specifies the type of information we want to download. We are dealing with air pollution and the considered sensor type is sds011 (see Section 1.3).
- `files_list_file` - optional field specifying a file where the links and filenames of all downloaded files can be saved
- `list_files` - a boolean value showing weather or not the script should save the links of the downloaded files as a list in the file given in `files_list_file`

The script downloads the desired files in the directory given by the `raw_down_dir` property of the configuration file. Internally the script uses python bindings for *wget* ([wge]) in order to preform the actual downloading from the Internet. Another library that comes into use is *BeautifulSoup* ([bs]). With it we parse the *HTML* of the web pages and extract *desired* the URLs.

6.1.2 Preprocess module

Central part of the preprocessing pipeline is the script in the `preprocess_module.py` file. It aims to select only the data files from the "good" sensors, from those then further filter the data and then for each sensor that is reliable enough, to create a DataFrame in form of a CSV-file. The final CSV-files will be stored in the directory specified by the `data_files_dir` property in the configuration file. The `preprocess_module` script first generates a file with the ids of the sensors that are actually worth processing. In this first phase we perform two checks while iterating over all of the downloaded files:

- For how many days a given sensor provides measurements. If a certain threshold is passed (given by the `min_sensor_cnt` property in the configuration), the sensors is considered as *saturated.*.
- Is a given sensor "around" Stuttgart. In the configuration file we specify we specify a center (property `center`) and a radius (property `radius`) and consider sensors only in the defined circle.

If a given sensors passes both of those checks, its name is added to the file specified by the `good_sensors_list_file` property. Furthermore, every data file of these "good" sensors is added to a file specified by the `good_sensors_data_files_list` property so that later we can

process only those.

The next step is to process the individual raw files and store their information in proper form in CSV-files. In this prepossessing for each file we do:

- We load the raw data file in memory in form of a pandas DataFrame.
- We transform the time column in more convenient format and then perform check if there are any duplicates in the DataFrame with respect to the time of the measurement. If there are, we collapse them to a single entry by the strategy specified by `duplicates_resolution` in the configuration. This could be *MEAN*, *MEDIAN*, *MAX* or *MIN*.
- We then perform the integration over the desired time period specified by `day_integration_period` in the configuration. To note is that we are still working of file per file basis and the processed DataFrames contain a day worth of measurements. For the biggest integration period here is a day.
- We rename the columns of the DataFrame to reflect the id of the corresponding sensor. The columns with measurements are then named `P1_<id>` or `P2_<id>` respectively for the integrated value for PM2.5 or PM10.
- Here we make one final check in this phase. After the values in the DataFrames have been integrated over specific time intervals, there are still can be missing values. If those count of those missing values is higher than `missing_data_cnt_threshold` in the configuration, the name of the sensors is added to a list in the file specified by `bad_missing_data_sensors` property. If the missing values are below the mentioned threshold, then the missing values are interpolated by the strategy given by `missing_data_resolutio` property. This could either be *MEAN* (the missing values are filled with the average of the rest of the values) or *linear* (the missing values are linearly interpolated based on the rest of the data).
- After all of that we append the data of the resulting DataFrame to a CSV-file with name `end_data_frame_<id>` in the direcotry specified by `data_files_dir`.

We do this for every raw data file and at the end we have a CSV-file for each sensor with accumulated data from the DataFrames. The final step is to go over these files one more time and sort the data by time.

6.1.3 Description module

The goal of this script is to provide some information about the generated data in from of plots and several DataFrames with statistics as well as to perform one final sanitation on the data. The script is implemented in the file `description_module.py`. It iterates over the generated from the previous step DataFrames and counts the missing values. Its important that in this step the missing values are in the context of the whole year we are considering. Depending on the integration intervals there must be exactly certain amount of values in each DataFrame. For integration over a day the exact count is 356, for integration over 12 hours - 730, for integration over each hour of the year - 8760 (365 days time 24 hours per day). If the number of missing values in the corresponding context are grater than the value of the

`missing_entries_threshold` property then the name of the sensor is added to a list in the file specified by `reindexed_frames_file`. If, however, the amount of missing values is under this threshold, the missing values are filled with the average value of the rest of the data. This is done of course both for the `P1` and `P2` column. As this is also our data description script, we generate a DataFrame with different statistics about the P1 and P2 values for each sensor. We wanted to know exactly how this last sanitation check on the data changes it so the DataFrames with the statistics are generated two times - once before the filling of the missing values and one time after the fact. We can then compare exactly what has changed about the data. At the end the scripts also generates plots of the P1 and P2 values of each sensor. Two kinds of plots are generated. One with the pure values as they appear in the final DataFrame. The other plot shows a moving average of 100 consecutive values. All of those description files are saved in the directory specified by the `description_files_dir` property in the configuration.

6.1.4 Preprocess LU BW

Although the data provided from LU-BW is in much better form than the publicly available data, it also needs certain preprocessing. The LU-BW comes in form of several spreadsheets in form of a *xlsx* file. The script in `process_lu_bw.py` takes this file as an input and generates three DataFrames for each LU-BW station. The measurements in the spreadsheets are in half an hour basis so an integration in the desired intervals is required. The script performs this integration and fills all of the missing values with the mean of the rest of the data. Similar to the previous section, a description file with statistics about the data before and after the missing values filling is generated as well as plots with the integrated values and a moving average of them. The DataFrames are saved in CSV-files in a separate folder in the folder with the rest of the data files. The plots and the description file are saved in the directory with the description files.

6.1.5 Loader module

The final step is to put the individual DataFrames of the sensors in one big DataFrame that can be conveniently used for the training of the models. This is what the script in `loader_module.py` performs. It gathers all the DataFrames defined by the CSV-files in the folder given by `folder` property and appends their columns to the columns to the final big DataFrame. The LU-BW DataFrames are also loaded. The final DataFrame is saved in the main `env` directory with name specified by `final_df_name`. In the section for this script of the configuration file there is a property with name `ignored_sensors_files`. This is meant to be a list of files, each containing a list of sensor ids. The sensors given in those files will be ignored and won't be loaded in the final DataFrame.

6.2 Models implementations

In the first two sections we explain the classes we've implemented for the two considered models. In the next two sections we describe the scripts that we use for training and evaluating the models. The major libraries we use are build on top of Tensorflow ([AAB+15]). In our explanations we assume basic knowledge of the principles and structures behind Tensorflow. A

good introduction can be found in [tfn].

6.2.1 Mixture density networks

For the implementation of the MDNs we follow the guidance of [Dut]. We use the *GPFlow* ([MvN+17]) library that internally uses Tensorflow. The library simplifies the creation of Gaussian process models. Although we don't unitize it fully, the features it provides make it really easy to define MDN.

The code implementing the MDN is in the `mdn_model.py` file. In it we define a class - `Mdn` - representing a mixture density network. As previously stated, we follow the approach of [Dut]. Our `Mdn` class inherits from the `Model` class in the `gpflow.models.model` package. This base class makes it possible to integrate our newly defined model in the GPFlow library and later trained. In order to explain the structure and the functionality of our class we take a bottom-up approach. We first describe the individual small parts and then how they come together.

In the constructor of the class we define the basic attributes of the models that we'll need in the other functions. For the construction of and Mdn-object we need:

- An arbitrary name for the new model in form of a string in the argument `model_id`
- The training data for the model - `X` are the input feature vectors (as explained in Section 4.4) and `Y` is the output.
- The definitions of the inner layers of the network. This is nothing more than an array of numbers specifying how many neurons there are each of the hidden layers. The number of hidden layers is given by the length of this array. In the constructor this is the `inner_dims` argument.
- Activation function that will be used as non-linearity in the neural network (see Section 3.2.1), by default this is the **tanh** function. This is given through the argument `activation`.
- The number of mixture components that the MDN will model. As simple integer value in the `num_mixtures` argument with default value of **5**.
- Optional file name of a file from which the class is supposed to load the model.

In the constructor itself we save the provided information in class attributes. Some of the more interesting ones are:

- `dims` - a two dimensional array that for each layers contains the dimensions of its input and output data. This makes it easier later to define the network in straight forward manner. To note is how the output dimension of the last layer is defined as in Equation 3.7.

After defining everything we either load the model (discussed later) from a file or construct it with the `_create_network` method.

The `_create_network` does nothing more than defining the weight matrices and bias vectors. The appropriate dimensions are known thanks to the `dims` array. The weight matrices are assigned random values and we expose restriction on the bias vectors to be positive in order introduce some regularization on the network (suggested by [Dut], see [HNGMW18]). All of the network's parameters are in the form of `Param` objects from GPFlow. Those encapsulate raw vector variables from Tensorflow that later could be learned and are managed by GPFlow. The generated parameters are put into class attributes for later use.

With the weights and biases defined we can describe the `_eval_network` method. In it we iterate over the weight matrices and biases and construct a Tensorflow graph defining the cascade of matrix multiplication, bias addition, non-linear function application as described in Section 3.2.1. The method takes one input argument that is the input to the network. At the end of the matrix multiplications, we split the intermediate output in three parts - mixing coefficients, means and variances. We place the described in Section 3.3 restrictions on them with the `softmax` and `exp` functions of Tensorflow. The end result of the method is a Tensorflow operation that when evaluated with some produces the output of the whole MDN as described in Section 3.3. To note is that the `_eval_network` is decorated with the `@params_as_tensors` decorator which transforms the arguments of the method as tensors automatically on invocation of the method.

The `_eval_network` method is used by `eval_network` to actually provide output of the MDN to a user. `eval_network` is defined with `@autoflow` decorator which automatically evaluates the returned tensorflow operations.

The most important method to be implemented so that GPFlow can train the model is `_build_likelihood`. It essentially defines the loss function to be optimized. In the method we first evaluate the network with the given input in order to generate the mixing coefficients, means and variances. Next we use these in order to build the sum over the log-likelihood over each example as described in Section 3.3 and given in Equation 3.37 and Equation 3.38. This is essentially all what we need to do as the automatic differentiation engine of Tensorflow does the gradient calculation.

The `fit` method is the place where we do the actual training of the model. We use the `ScipyOptimizer` class provided by GPFlow as a wrapper of *scipy* ([JOP+]) function for optimizing functions that uses the LM-BFGS method (see Section 3.2.1). The fit method takes the maximum number of iterations and a optional callback function. We perform the training in three steps. First we optimize the function while we allow changes only in the weight matrices, then we optimize only by changing the bias vectors and finally we optimize with changes in both of the parameter groups. This is done to introduce a little bit more regularization in the training of the network. After each step we invoke the callback if provided, so that the outside user can monitor the changes in network's performance after each step.

We've also implemented a `save` method that can make a trained model persistent by saving in storage memory. The method uses a `Saver` object provided by GPFlow that can serialize parameters and parameter lists. In order to fully reconstruct the model later we only need the weight matrices and bias vectors. For this reason those are the only objects that we serialize and save.

The later loading of a given model is done through the `load` method. Its purpose is to load the weight matrices and bias vectors from a given file through the deserialization

provided by the same `Saver` object use for saving. The `load` method essentially does the same thing as the `_create_network` method, it's just that now we load the objects from a file. The load method is in fact invoked instead of `_create_network` by the constructor if a file name is provided.

6.2.2 Bayesian neural networks

With the implementation of BNNs we again don't reinvent the wheel and use the *Edward* ([TKD+16]) library. Edward is another library built on top of Tensorflow and it provides a lot of functionality that makes Bayesian learning easier to implement. We follow the official guide ([Tra]) to BNNs from the authors of Edward. The class implementing BNNs is Bnn and is located in the `bnn_model.py`. We now look at the individual functions of our class in order to explain it.

The construction is trivial as the only thing it does is save the given name of the model in an class attribute.

First we'll look at the methods `generate_prior_vars` and `generate_latent_vars`. These methods are very similar. For given input and output dimensions for the whole network and layer definitions (analogous to Section 6.2.1) they both define weight matrices and bias vectors as Normal distributions (i.e. we can later sample wight matrices and bias vectors from the defines objects) with the appropriate dimensions. The difference between the methods is the context in which they define the distributions. `generate_prior_vars` defines the prior distributions over the wights and biases. These are all standard normal distributions and correspond to the $\boldsymbol{p}$ distributions in Section 3.2.3. `generate_latent_vars` defines distributions with trainable Tensorflow variables that later will be optimized. These distributions build the variational distribution (see Section 3.2.3) that we will perform inference and will try to approach the posterior. The individual variational variables are assigned random value at the start.

The `_neural_network` method does essentially the same thing as the `_eval_network` from the Mdn class. For a given input and two lists of weight matrices and bias vectors it creates an Tensorflow operation that evaluates to the output of the network defined by the weights and biases and with an input the given input. The non-linearity function that we use is `tanh`.

With all these methods explained we can now look at the central part of the Bnn class. The `build` method is what builds the whole Tensorflow graph for the structure of the BNN and the operations we will need for the inference. First, with the usage of `generate_prior_vars` and `generate_latent_vars`, we generate the prior and latent distributions we will need for the network. Then we define a tensor placeholder for the input data, operation that defines a normal distribution centered at the network's output and small variance (see Section 3.1) and a placeholder for the actual observed output data. To note is that here we use the prior distributions while we define the network operation. The next step is to define the operation necessary for the evaluation of the BNN with new data. For that we define two placeholders - one for the input with which we want to evaluate the trained network and one for the number of samples we want to draw from the posterior distribution (i.e. how many network functions we will generate and evaluate, see Section 3.2.2). After the placeholders will define operation that evaluates to a stacked evaluations of networks defined with draws from the latent variables

defining the variational distribution. To note is that we evaluate each network on the whole input. The final shape of the output of this operation will be $\boldsymbol{N} \times \boldsymbol{s}$ where $\boldsymbol{N}$ is the count of provided examples to evaluate and $\boldsymbol{s}$ is the number of samples we are supposed to draw from the BNN. The final thing in the `build` method is to initialize all the variables in the Tensorflow graph with execution of initialization operation and define a `Saver` object so that later we can save the trained model to storage memory.

The training of the model happens in the `fit` method. In it the Bayesian inference functionality of Edward comes into play. We first define a dictionary the represents a mapping between prior variable and its variational counterpart. This is the format that Edward requires. After that come some calculation necessary for implementing batching over the data. We then define the type of inference we will use for training the model. In our case, Kullback-Leibler divergence (see Section 3.2.3). Edward provides convenient interface for this definition through the class `ed.KLqp`. Once the inference is defined and initialized we create three in one another loops over the data. We feed certain amount of examples to the `update` method. As we operate on batches of data, we can also present given batch several times in a row to the `update` method. This is the job of the inner most loop. The middle loop iterates over all the batches. The outermost loop counts the epochs. One Epoch is when the entire dataset is passed through the inference algorithm once. One thing to note is that every one thousand epochs we invoke a callback if given. This way the outside user can monitor the models during the inference process.

With the `evaluate` methods we can evaluate data with some trained model. The method accepts the input data and samples to be drawn for each data point, feeds them to the write placeholders in the graph and evaluates the Tensorflow operation.

The `save` and `load` methods of the class are similar to the ones in the Mdn class. One key difference is that here we use the `Saver` object from Tensorflow and not from GPFlow. This means that by serialization we are saving the whole Tensorflow graph and the values of all variables in it. By the loading of the model later, we only need to load the graph and not build anything at all (i.e. not call the `build` method).

6.3 Model training

6.4 Models evaluation

6.5 Auxiliary scripts

Bash script that are aimed at automating some of the trivial tasks.

6.5.1 Run Generation pipeline

File:`run_generation_pipeline.sh`

The script performs the whole DataFrame generation with the current `config.json` from start to finish. The steps that are automatically executed:

(i) Extracts the integration period `P` for the data

(ii) Creates a log file with name `LOG_<P>.txt` that will contain the whole output to the standard output of the scripts in the next steps.

(iii) Runs `preprocess_module.py` with the right flags to generate cache files containing the senors that must be preprocessed.

(iv) Runs `preprocess_module.py` with the right flags to perform the actual preprocessing of the raw files.

(v) Runs `preprocess_lu_bw.py` to preprocess the LU-BW data.

(vi) Runs `description_module.py` to perform the final checks on the data and generate plots and statistics about it

(vii) Counts the lines in each of the generated CSV-files for the data form *luftdaten.info*. All of these numbers must be the same if everything is ok.

(viii) Counts the lines in each of the generated CSV-files for the data form LU-BW. All of these numbers must be the same if everything is ok.

(ix) Runs `loader_module.py` to generate the final big DataFrame with all of the data.

(x) Copies the final DataFrame in the `data_frames` folder of the `env` directory.

(xi) Archives all of the generated files during the whole process including the log file, the `config.json` file, all of the generated data files, all of the generated description files.

The result of running this script is a self-contained archive containing the generated data as well as description of how it was generated. This archive can later be inspected in order to validate the correctness of the data. One can also examine the log file and the configuration file and know what has happened exactly in the generation process.

Some of the steps are placed in bash scripts of their own. These scripts are with self expiatory names and we don't go over details about them rather just mentiom them: `preprocess_new_cache.sh` (step 3.), `preprocess_with_cached.sh` (step 4.), `save_final_dfs.sh` (step 10.), `archive_data_files.sh` (step 11.).

6.5.2 Clean `env`

File:`clean.sh`

The scripts deletes all files generated during the generation process expect the final data frame and the created archive. This prepares the `env` directory for a future execution of the preprocess pipeline.

7 Conclusion

7.1 Summery

7.2 Discussion

References

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. (Pages 13, 16, 27).

[Bay63] T. Bayes. An essay towards solving a problem in the doctrine of chances. *Phil. Trans. of the Royal Soc. of London*, 53:370–418, 1763. (Page 4).

[BCKW15] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra. Weight Uncertainty in Neural Networks. *ArXiv e-prints*, May 2015. (Pages 7, 13).

[BKM16] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe. Variational Inference: A Review for Statisticians. *ArXiv e-prints*, January 2016. (Page 12).

[BLNZ95] R. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995. (Page 11).

[BNVP01] K. Bertels, L. Neuberg, S. Vassiliadis, and D.G. Pechanek. On chaos and neural networks: The backpropagation paradigm. *Artificial Intelligence Review*, 15(3):165–187, May 2001. (Pages 7, 16).

[Bre01] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001. (Page 23).

[Bri90] John S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In Françoise Fogelman Soulié and Jeanny Hérault, editors, *Neurocomputing*, pages 227–236, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg. (Page 15).

[bs] Beautiful soup. https://www.crummy.com/software/BeautifulSoup/. Accessed: 2018-08-21. (Page 25).

[BXP+16] Y. Bai, Y. Xu, J. Pan, J.Q. Lan, and W.W. Gao. Application of bayesian neural networks to energy reconstruction in eas experiments for ground-based tev astrophysics. *Journal of Instrumentation*, 11(07):P07006, 2016. (Page 8).

[Cle] Michael P. Clements. Evaluating the bank of england density forecasts of inflation*. *The Economic Journal*, 114(498):844–866. (Page 8).

[Col07] Mat Collins. Ensembles and probabilities: A new era in the prediction of climate change. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, 365(1857):1957–1970, 2007. (Page 7).

[dat] Luftdaten archiv. https://archive.luftdaten.info/. Accessed: 2018-08-21. (Pages 6, 24, 25).

[Daw84] A. P. Dawid. Statistical theory: the prequential approach (with discussion). *J. R. Statist. Soc. A*, 147:278–292, 1984. (Page 17).

[DGT98] Francis Diebold, Todd A Gunther, and Anthony S Tay. Evaluating density forecasts with applications to financial risk management. *International Economic Review*, 39(4):863–83, 1998. (Pages 17, 21).

[DKPR87] Simon Duane, A. D. Kennedy, Brian J. Pendleton, and Duncan Roweth. Hybrid monte carlo. *Physics Letters B*, 195(2):216 – 222, 1987. (Page 14).

[DL90] John S. Denker and Yann LeCun. Transforming neural-net output levels to probability distributions. In *Proceedings of the 3rd International Conference on Neural Information Processing Systems*, NIPS'90, pages 853–859, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc. (Page 3).

[DS99] A. Philip Dawid and Paola Sebastiani. Coherent dispersion criteria for optimal experimental design. *Ann. Statist.*, 27(1):65–81, 03 1999. (Page 19).

[Dut] Vincent Dutordoir. Mixture density networks in gpflow. https://www.prowler.io/blog/mixture-density-networks-in-gpflow-a-tutorial. Accessed: 2018-08-20. (Pages 28, 29).

[EG12] Werner Ehm and Tilmann Gneiting. Local proper scoring rules of order two. *Ann. Statist.*, 40(1):609–637, 02 2012. (Pages 7, 18).

[EvOHS15] J. M. Eden, G. J. van Oldenborgh, E. Hawkins, and E. B. Suckling. A global empirical system for probabilistic seasonal climate prediction. *Geoscientific Model Development*, 8(12):3947–3973, 2015. (Page 7).

[FBV17] M. Fortunato, C. Blundell, and O. Vinyals. Bayesian Recurrent Neural Networks. *ArXiv e-prints*, April 2017. (Page 8).

[FR12] Charles W. Fox and Stephen J. Roberts. A tutorial on variational bayesian inference. *Artificial Intelligence Review*, 38(2):85–95, Aug 2012. (Pages 12, 13).

[Gal16] Yarin Gal. *Uncertainty in Deep Learning*. PhD thesis, University of Cambridge, 2016. (Pages 6, 8, 9, 11).

[GBR] Tilmann Gneiting, Fadoua Balabdaoui, and Adrian E. Raftery. Probabilistic forecasts, calibration and sharpness. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 69(2):243–268. (Pages 17, 18).

[GG84] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):721–741, Nov 1984. (Page 14).

[GGBJ07] E. P. Grimit, T. Gneiting, V. J. Berrocal, and N. A. Johnson. The continuous ranked probability score for circular variables and its application to mesoscale forecast ensemble verification. *Quarterly Journal of the Royal Meteorological*

Society, 132(621C):2925–2942, 2007. (Page 19).

[GK14] Tilmann Gneiting and Matthias Katzfuss. Probabilistic forecasting. *Annual Review of Statistics and Its Application*, 1(1):125–151, 2014. (Pages 7, 17, 18, 19, 20, 21).

[Goo52] I. J. Good. Rational decisions. *Journal of the Royal Statistical Society. Series B (Methodological)*, 14(1):107–114, 1952. (Page 18).

[GR07] Tilmann Gneiting and Adrian E Raftery. Strictly proper scoring rules, prediction, and estimation. *Journal of the American Statistical Association*, 102(477):359–378, 2007. (Pages 3, 19).

[GR11] Tilmann Gneiting and Roopesh Ranjan. Comparing density forecasts using threshold- and quantile-weighted scoring rules. *Journal of Business Economic Statistics*, 29(3):411–422, 2011. (Page 19).

[GR13] Tilmann Gneiting and Roopesh Ranjan. Combining predictive distributions. *Electron. J. Statist.*, 7:1747–1782, 2013. (Page 17).

[Ham01] Thomas M. Hamill. Interpretation of rank histograms for verifying ensemble forecasts. *Monthly Weather Review*, 129(3):550–560, 2001. (Page 21).

[Has70] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970. (Page 14).

[HH64] J. M. Hammersley and D. C. Handscomb. *General Principles of the Monte Carlo Method*, pages 50–75. Springer Netherlands, Dordrecht, 1964. (Page 14).

[HNGMW18] Tobias Hinz, Nicolás Navarro-Guerrero, Sven Magg, and Stefan Wermter. Speeding up the hyperparameter optimization of deep convolutional neural networks. *International Journal of Computational Intelligence and Applications*, 17(02):1850008, 2018. (Page 29).

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989. (Page 11).

[JGJS99] Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. An introduction to variational methods for graphical models. *Machine Learning*, 37(2):183–233, Nov 1999. (Page 12).

[JJNH91] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural Comput.*, 3(1):79–87, March 1991. (Page 8).

[JKL17] A. Jordan, F. Krüger, and S. Lerch. Evaluating probabilistic forecasts with scoringRules. *ArXiv e-prints*, September 2017. (Page 8).

[JOP+] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed <today>]. (Page 29).

[KB14] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *ArXiv e-prints*, December 2014. (Page 11).

[KCS] Vivek Katiyar, Narottam Chand, and Surender Soni. Clustering algorithms for heterogeneous wireless sensor network: A survey. *International Journal of Applied Engineering Research*, page 2010. (Page 7).

[KL51] S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 03 1951. (Page 12).

[KP17] Yongchan Kwon and Myunghee Paik. Uncertainty quantification for ischemic stroke lesion segmentation using bayesian neural networks. 12 2017. (Page 8).

[Krz01] Roman Krzysztofowicz. The case for probabilistic forecasting in hydrology. *Journal of Hydrology*, 249(1):2 – 9, 2001. (Page 7).

[KVS11] Oliver Krueger and Jin-Song Von Storch. A simple empirical model for decadal climate prediction. *Journal of Climate*, 24(4):1276–1283, 2011. (Page 7).

[KW13] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2013. (Page 13).

[LV01] Jouko Lampinen and Aki Vehtari. Bayesian approach for neural networks—review and case studies. *Neural Networks*, 14(3):257 – 274, 2001. (Page 8).

[LZLQ17] Zuozhu Liu, Wenyu Zhang, Shaowei Lin, and Tony Q. S. Quek. Heterogeneous sensor data fusion by deep multimodal encoding. *IEEE Journal of Selected Topics in Signal Processing*, 11:479–491, 2017. (Page 7).

[Mac92] David J. C. MacKay. A practical bayesian framework for backpropagation networks. *Neural Computation*, 4(3):448–472, 1992. (Page 6).

[MB94] Christopher M. Bishop. Mixture density networks. 01 1994. (Pages 14, 16).

[McK10] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010. (Page 24).

[MDD03] Tatiana Miazhynskaia, Georg Dorffner, and Engelbert J. Dockner. Risk management application of the recurrent mixture density network models. In Okyay Kaynak, Erkki Alpaydin, Ethemand Oja, and Lei Xu, editors, *Artificial Neural Networks and Neural Information Processing — ICANN/ICONIP 2003*, pages 589–596, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. (Page 8).

[MHW12] Jacob M. Montgomery, Florian M. Hollenbach, and Michael D. Ward. Ensemble predictions of the 2012 us presidential election. *PS: Political Science; Politics*, 45(4):651–654, 2012. (Page 7).

[MRR+53] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953. (Page

14).

[MvN+17] Alexander G. de G. Matthews, Mark van der Wilk, Tom Nickson, Keisuke. Fujii, Alexis Boukouvalas, Pablo Le'on-Villagr'a, Zoubin Ghahramani, and James Hensman. GPflow: A Gaussian process library using TensorFlow. *Journal of Machine Learning Research*, 18(40):1–6, apr 2017. (Pages 16, 28).

[MW76] James E. Matheson and Robert L. Winkler. Scoring rules for continuous probability distributions. *Management Science*, 22(10):1087–1096, 1976. (Page 19).

[Nea96] Radford M. Neal. *Bayesian Learning for Neural Networks*. Springer-Verlag, Berlin, Heidelberg, 1996. (Page 6).

[NH98] Radford M. Neal and Geoffrey E. Hinton. *A View of the Em Algorithm that Justifies Incremental, Sparse, and other Variants*, pages 355–368. Springer Netherlands, Dordrecht, 1998. (Page 13).

[PBJ12] J. Paisley, D. Blei, and M. Jordan. Variational Bayesian Inference with Stochastic Search. *ArXiv e-prints*, June 2012. (Pages 7, 13, 14).

[PCB15] Harrison B. Prosper Pushpalatha C. Bhat. Bayesian neural networks. 2015. (Page 8).

[Pin13] Pierre Pinson. Wind energy: Forecasting challenges for its operational management. *Statist. Sci.*, 28(4):564–585, 11 2013. (Page 7).

[RG02] CE. Rasmussen and Z. Ghahramani. Infinite mixtures of gaussian process experts. Max-Planck-Gesellschaft, 2002. (Pages 7, 8).

[RL18] S. Rasp and S. Lerch. Neural networks for post-processing ensemble weather forecasts. *ArXiv e-prints*, May 2018. (Page 23).

[RM51] Herbert Robbins and Sutton Monro. A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407, 09 1951. (Pages 11, 13).

[Rud16] S. Ruder. An overview of gradient descent optimization algorithms. *ArXiv e-prints*, September 2016. (Page 11).

[RW05] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005. (Page 8).

[Sch15] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. cited By 1624. (Page 8).

[Sco15] David Scott. *Multivariate Density Estimation: Theory, Practice, and Visualization*. 03 2015. (Page 21).

[SDD98] Christian Schittenkopf, Georg Dorffner, and Engelbert Dockner. Identifying stochastic processes with mixture density networks. 07 1998. (Page 8).

[sds] Sds-011 specifications. https://nettigo.pl/attachments/398. Accessed:

2018-08-21. (Page 6).

[Sil86] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman Hall, London, 1986. (Page 21).

[SJJ96] L. K. Saul, T. Jaakkola, and M. I. Jordan. Mean Field Theory for Sigmoid Belief Networks. *eprint arXiv:cs/9603102*, February 1996. (Page 13).

[Sti75] S. M. Stigler. The transition from point to distribution estimation, 1975. (Page 7).

[SW96] A. Shapiro and Y. Wardi. Convergence analysis of gradient descent stochastic algorithms. *Journal of Optimization Theory and Applications*, 91(2):439–454, Nov 1996. (Page 13).

[tfn] Introduction to tensorflow. https://www.tensorflow.org/guide/low_level_intro. Accessed: 2018-08-21. (Page 28).

[Tim] Allan Timmermann. Density forecasting in economics and finance. *Journal of Forecasting*, 19(4):231–234. (Page 7).

[Tip04] Michael E. Tipping. *Bayesian Inference: An Introduction to Principles and Practice in Machine Learning*, pages 41–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. (Page 4).

[TKD+16] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016. (Pages 13, 30).

[Tra] Dustin Tran. Getting started. http://nbviewer.jupyter.org/github/blei-lab/edward/blob/master/notebooks/getting_started.ipynb. Accessed: 2018-08-20. (Page 30).

[US14] Prachi Uplap, , and Preeti Sharma. Review of heterogeneous/homogeneous wireless sensor networks and intrusion detection system techniques. In *Int. Conf. on Recent Trends in Information, Telecommunication and Computing , ITC*. Institute of Doctors Engineers and Scientists, ACEEE (A Computer division of IDES), 2014. (Page 7).

[VSL00] Aki Vehtari, Simo Särkkä, and Jouko Lampinen. On mcmc sampling in bayesian mlp neural networks. 1:317–322 vol.1, 02 2000. (Page 7).

[WC07] Chun-Hsien Wu and Yeh-Ching Chung. Heterogeneous wireless sensor network deployment and topology control based on irregular sensor model. In Christophe Cérin and Kuan-Ching Li, editors, *Advances in Grid and Pervasive Computing*, pages 78–88, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. (Page 7).

[wge] Gnu wget. https://www.gnu.org/software/wget/. Accessed: 2018-08-21. (Page 25).

[ZS14] Heiga Zen and Andrew Senior. Deep mixture density networks for acoustic modeling in statistical parametric speech synthesis. In *Proceedings of the IEEE In-*

ternational Conference on Acoustics, Speech, and Signal Processing (ICASSP), pages 3872–3876, 2014. (Page 8).