UiO : Department of Mathematics
University of Oslo

Combining Variational Bayes and GMJMCMC for Scalable Inference on Bayesian Generalized Nonlinear Models

Philip Sebastian Hauglie Sommerfelt

Master's Thesis, Spring 2023

Abstract

We change the approach for computing posterior distributions in Bayesian Generalized Nonlinear Models. We replace MCMC with variational Bayes, and approximate the posterior distribution with mean-field, or through utilization of normalizing flows. Step by step, we go through the theory behind BGNLM, variational inference and normalizing flows. We also show the calculations needed to understand the new implementation, and provide a Python framework for training and testing BGNLMs. Through a series of applications we demonstrate that we are able to make accurate predictions, and get easily obtainable measures for the uncertainty of the predictions.

Acknowledgements

I have received a lot of support and encouragement when writing this thesis. Especially, I would like to thank Aliaksandr Hubin for immense support, and for lending me his expert knowledge throughout the process. He has provided invaluable insight, and assisted me through longer days towards the end.

In addition, I would like to thank Terje Kvernes for providing me with the computational resources needed. Every bit of code used in this thesis ran smoothly on a powerful GPU managed by him.

Lastly, I offer my sincere thanks to family and friends for their encouragement and patience. I would not have been able to write this thesis without them.

Contents

CHAPTER 1

Introduction

According to Breiman (2001), there are two broad approaches to data analysis: the data modelling culture, which focuses on understanding the data's generating process, and the algorithmic modelling culture, which emphasizes accurate predictions and utilizes techniques like neural networks or random forests. However, as datasets become increasingly large and complex, there is a growing need for models that not only make accurate predictions but also provide clear and understandable explanations for those predictions. One reason for this is the General Data Protection Regulation (European Parliament 2016), which requires organizations to explain decisions made by automated systems to individuals affected by them.

To address this need, Hubin, Storvik and Frommlet (2021) introduced a class of regression models called Bayesian Generalized Nonlinear Models (BGNLM), which provide flexible models with high interpretability. BGNLMs are based on generalized linear models (Nelder and Wedderburn 1972), where the distribution of the observations is assumed to be from the exponential family, and the mean parameter is a nonlinear function of the covariates. In BGNLM, the specific modeling of this nonlinear dependency incorporates ideas related to neural networks. While neural networks can be difficult to interpret, BGNLMs are designed to retain good predictive abilities while remaining relatively simple, interpretable, and transparent.

However, as with most Bayesian approaches, BGNLMs can be computationally expensive, restricting their usage to relatively small problems with few observations and covariates. The fitting algorithm proposed by Hubin, Storvik and Frommlet (2020) is a special case of Markov Chain Monte Carlo (MCMC) called Genetically Modified Mode Jumping Markov Chain Monte Carlo (GMJMCMC). To improve computation times for datasets with many observations, Lachmann (2021) implemented a subsampling GMJMCMC. Nevertheless, BGNLMs still struggle to scale well to datasets with many covariates.

To address this challenge, in this thesis, we propose a new fitting algorithm that combines ideas from variational inference and normalizing flows.

In Chapter 2, we provide a detailed description of BGNLMs, GMJMCMC, variational inference, and normalizing flows. In Chapter 3, we present the algorithm used for inference, and in Chapter 4, we apply our new implementation to different datasets. In Chapter 5, we conclude and discuss suggestions for further research.

Notation

Throughout the thesis, we will use bold letters or symbols, such as $\mathbf{x}$ or $\boldsymbol{\theta}$ to denote vectors or sets, while x and y are generally used for scalars. Bold capital letters, such as $\mathbf{W}$ are used to denote matrices. We will use $p(\cdot)$ or $q(\cdot)$ to denote arbitrary distribution functions. If we are concerned with the parameters of these functions, we use subscript to denote the parameters, e.g $p_{\boldsymbol{\theta}}(\cdot)$, unless specified otherwise.

CHAPTER 2

Background

2.1 Generalized Linear Models (GLM)

In order to provide some context, it is appropriate to start with an introduction to the Generalized Linear Model (Nelder and Wedderburn 1972, GLM). Despite its simplicity, it is a very powerful tool for data analysis and is widely used in many fields. There are entire books written on the GLM, but we will be very brief and provide a high-level overview of some useful concepts.

The GLM extends the linear regression model to handle response variables that have a non-normal distribution. It is a flexible model that can be used for a wide range of data types, including binary, count, and continuous data. A key component of the GLM is the exponential family. The exponential family is a collection of probability distributions that can be written on the form

$$f(y|\boldsymbol{\xi}) = A(y)\, exp\Big[\eta(\boldsymbol{\xi})B(y) - C(\boldsymbol{\xi})\Big],$$

where A is a function that depends only on y, $\boldsymbol{\xi}$ is a set of parameters, η is a vector-valued function of $\boldsymbol{\xi}$, B is a vector valued function of y, and C is a function of $\boldsymbol{\xi}$, typically referred to as the normalizing constant.

These distributions include the normal, binomial, Poisson, and exponential distributions, among others. In a GLM, the response variable is assumed to come from a distribution within this family.

Another important component of the GLM is the link function. The link function maps the linear predictor to the mean of the response variable:

$$h(\mu) = \mathbf{X}\boldsymbol{\beta}^T,$$

where $\mathbf{X} \in \mathbb{R}^{n \times p}$ is a matrix of covariates, and $\boldsymbol{\beta} \in \mathbb{R}^p$ is a set of coefficients. There are several commonly used link functions, depending on the distribution of the response variable. These include the logistic, log, and identity link functions. The GLM can then be written as

$$Y|\mu, \phi \sim f(y|\mu, \phi), \tag{1a}$$

$$h(\mu) = \beta_0 + \sum_{j=1}^{p} \beta_j x_j. \tag{1b}$$

Where $f(\cdot|\mu, \phi)$ is the density or mass of a probability distribution from the exponential family with mean μ and dispersion parameter ϕ, and $h(\mu)$ is the link function relating the mean to the covariates.

Bayesian GLMs and variable selection

The GLM can also be put into a Bayesian setting. This involves using Bayesian methods to estimate the model parameters $\boldsymbol{\beta}$, and possibly the dispersion parameter ϕ. In the context of variable selection, it is common to introduce a binary $\gamma_j \in \{0, 1\}$ to indicate whether or not the corresponding β_j is non-zero and should be included in the model. In such a setting, prior distributions are placed on the model parameters $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ and possibly ϕ, which are then updated using the data.

Bayesian GLMs provide several advantages over frequentist GLMs. For example, Bayesian GLMs allow for the incorporation of prior knowledge or belief into the analysis. They also provide a full posterior distribution for the parameters, which can be used for parameter estimation, model comparison, and prediction.

2.2 (Bayesian) Neural Networks

Since we will need a description of neural networks later, it is appropriate to give a brief introduction here.

A neural network is a type of machine learning model inspired by the structure and function of the human brain. It consists of layers of interconnected nodes, or neurons, that are trained on input data to make predictions or classifications. The standard approach to training a neural network involves defining an objective function depending on the target variable, and using gradient-based optimization methods to iteratively adjust the weights of the neurons in order to minimize the objective function. This process is known as *backpropagation* (Linnainmaa 1970; Rumelhart, Hinton and Williams 1986), and it allows the network to learn patterns and relationships in the input data.

In the standard approach, one uses a fixed set of weights and biases in the neural network. The output of each layer is determined by the input to that layer multiplied by the weights, plus the bias term, passed through an activation function. We can write an arbitrary layer in the network as

$$u_j^{(l)} = \sigma^{(l)}\Big(b_j^{(l)} + \sum_{i=1}^{n^{(l-1)}} u_i^{(l-1)} w_{ij}^{(l)}\Big), \quad j = 1, ..., n^{(l)},$$

where $n^{(l-1)}$ is the dimension of the previous layer, $\mathbf{u}^{(l-1)}$ is the output of the previous layer, $\mathbf{w}_j^{(l)}$ is the weights and $b_j^{(l)}$ is the bias and $\sigma^{(l)}$ is the activation function.

In the Bayesian approach to neural networks, the weights of the neurons are treated as random variables with prior distributions. One then uses Bayes' theorem to compute the posterior distribution over the weights and biases given the training data. The goal is to compute the posterior distribution over the weights given the observed data, which can be used for prediction and uncertainty quantification. This approach allows for more robust and interpretable predictions, as well as the ability to incorporate prior knowledge into the model. The output of each layer is now a distribution over possible values, rather than a single deterministic value. We can write this mathematically as

$$p(\mathbf{u}^{(l)}|\mathbf{u}^{(l-1)}, \boldsymbol{\xi}) = \int_{\mathbf{w}} \int_{\mathbf{b}} p(\mathbf{u}^{(l)}|\mathbf{u}^{(l-1)}, \mathbf{w}^{(l)}, \mathbf{b}^{(l)}) p(\mathbf{w}^{(l)}, \mathbf{b}^{(l)}|\boldsymbol{\xi}) d\mathbf{b} d\mathbf{w}.$$

Where $p(\mathbf{u}^{(l)}|\mathbf{u}^{(l-1)}, \boldsymbol{\xi})$ is the posterior distribution over the output of layer l, given the input to that layer and a set of prior hyperparameters $\boldsymbol{\xi}$, $p(\mathbf{u}^{(l)}|\mathbf{u}^{(l-1)}, \mathbf{w}^{(l)}, \mathbf{b}^{(l)})$ is the likelihood of the output given the input and the weights and biases, and $p(\mathbf{w}^{(l)}, \mathbf{b}^{(l)}|\boldsymbol{\xi})$ is the prior distribution over the weights and biases.

In practice, Bayesian neural networks are computationally expensive and difficult to implement, but they have shown promise in various applications.

2.3 Bayesian Generalized Nonlinear Models (BGNLM)

We will now turn our attention to what is the main model in this thesis: Bayesian Generalized Nonlinear Models. It was developed by Hubin, Storvik and Frommlet (2021), with the purpose of providing flexible models with high interpretability. This class is based on the GLM where the distribution of the observations is assumed to come from the exponential family and the mean parameter is a nonlinear function of the input variables. More specifically, we model the relationship between between p explanatory variables and a response variable based on n samples from a data set. For $i = 1, ..., n$, let Y_i denote the response data and let $\mathbf{x}_i = (x_{i1}, ..., x_{ip})$ be the corresponding vector of covariates.

The model framework is similar to the GLM, with the added freedom of including a flexible class of nonlinear transformations to the covariates. These nonlinear transformations $F_j(\mathbf{x}, \boldsymbol{\alpha}_j)$ for $j = 1, .., q$ are called features and will be properly presented shortly. The BGNLM is defined through:

$$Y|\mu, \phi \sim f(y|\mu, \phi), \tag{2a}$$

$$h(\mu) = \beta_0 + \sum_{j=1}^{q} \gamma_j \beta_j F_j(\mathbf{x}, \boldsymbol{\alpha}_j). \tag{2b}$$

Where $f(\cdot|\mu, \phi)$ is the density or mass of a probability distribution from the exponential family with mean μ and dispersion parameter ϕ, and $h(\mu)$ is a link function relating the mean to the features. The features enters the model with coefficients $\beta_j \in \mathbb{R}$ for $j = 1, ..., q$. The formulation presented includes all possible q features but uses a binary variable $\gamma_j \in \{0, 1\}$ to indicate whether or not the corresponding features are to be included in the model.

Features

To make sense of the model definition in Equation (2b), a specification of the features $F_j(\mathbf{x}, \boldsymbol{\alpha}_j)$ $j = 1, .., q$ is required. In addition, we shall define the process defining the full hierarchy of the features.

A feature is defined as a nonlinear transformation, transforming one or more of the covariates through a function or a series of functions. The model firstly needs to specify which functions $\mathcal{G} = g_1, ..., g_k$ to consider. In principle, any function can be part of $\mathcal{G}$ as long as it has $\mathbb{R}$ as domain and range within $\mathbb{R}$. However, continuous and differentiable functions will make the optimisation easier.

The feature generating process begins with the input variables $\mathbf{x}$ as features, i.e. $F_j(\mathbf{x}_i) = x_{ij}$ for $j \in 1, ..., p$. In a recursive setting, we denote the set of features included in the model at a given level of recursion as A and the number of features in it as $|A|$. Assume that at a certain point, a set of features $F_k(\cdot, \boldsymbol{\alpha}_k), k \in A$ is generated. Then, we define the following transformations to generate new features for the next step:

$$F_j(\mathbf{x}, \boldsymbol{\alpha}_j) = \begin{cases} g_j(\boldsymbol{\alpha}_{j,0}^{out} + \sum_{k \in A_j} \boldsymbol{\alpha}_{j,k}^{out} F_k(\mathbf{x}, \alpha_k^{in})) & \text{projection,} \\ g_j(F_k(\mathbf{x}, \boldsymbol{\alpha}_k^{in})) & \text{modification,} \\ F_k(\mathbf{x}, \boldsymbol{\alpha}_k^{in}) F_l(\mathbf{x}, \boldsymbol{\alpha}_l^{in}) \ k, l \in A & \text{multiplication.} \end{cases} \tag{3}$$

The first transformation called projection has similar definition to that used in neural networks, but the activation function g_j is now selected from $\mathcal{G}$. The linear combination is taken over a subset of features $F_k(\cdot, \boldsymbol{\alpha}_k), k \in A_j$ where $A_j \subseteq A$ and $|A_j| > 1$. Note that we differ between the parameters that defines the current projection, $\boldsymbol{\alpha}_j^{out}$ and the parameters that is contained in the previously defined features nested inside the projection, $\boldsymbol{\alpha}_j^{in}$.

The modifications and multiplications are included to allow for more parsimonious models. Note that $\boldsymbol{\alpha}_j^{out} = \emptyset$ for both. Modifications allow for nonlinear transformations of existing features, while multiplications corresponds to interactions in the language of statistics. The latter is allowed to select the same feature more than once. As noted in Hubin, Storvik and Frommlet (2021), both of these transformations can be seen as a special case of projections. Modifications are a projection where $|A_j| = 1$, and multiplications can be seen as a special case of two projections with the $\exp(x)$ and $\log(x)$ transformations. This allows limiting BGNLM to include only modifications and multiplications.

Feature properties

The depth, d_j, of a feature F_j is determined by the minimum number of nonlinear transformations applied recursively to generate it. For example, if a feature F_j is defined as $F_j(\mathbf{x}, \boldsymbol{\alpha}_j) = h(u(v(x_1)) + w(x_2))$, for some nonlinear functions h, u, v, w, then its depth is 3. Conversely, if a multiplication operation is applied, the depth is defined as one plus the sum of the depths of the operands. For example, $F_k(\mathbf{x}, \boldsymbol{\alpha}_k) = x_2 u(x_1)$ has depth d_k of 2, using that the depth of a linear component is zero. Hubin, Storvik and Frommlet (2021) shows that the number of features grows super-exponentially with depth, and in practice limiting the depth to be small even for problems with few covariates.

The local width, lw_j, of a feature is the number of previously defined features used to generate a new feature. The value of lw_j depends on the type of operation used: $|A_j|$ for a projection, 1 for a modification, and 2 for a multiplication. The operations count, oc_j, of a feature is the total number of algebraic operations used in its representation. For instance, $F_j(\mathbf{x}, \boldsymbol{\alpha}_j) = \mathbf{x}$ has $oc_j = 0$, while $F_j(\mathbf{x}, \boldsymbol{\alpha}_j) = v(u(x))$ has $oc_j = 2$.

Three strategies for optimizing α parameters

In the context of the general projection transformation, the $\boldsymbol{\alpha}_j$ parameters must be determined. Hubin, Storvik and Frommlet (2021) proposes three strategies for optimization. These strategies aim to find $\boldsymbol{\alpha}_j$ values that result in high explanatory power for $F_j(\mathbf{x}, \boldsymbol{\alpha}_j)$, independent of the other features involved in the model. The strategies are as follows:

Strategy 1, (optimize then transform, naive) is the simplest method for determining $\boldsymbol{\alpha}_j$.

$$h(\mu) = \alpha_{j,0}^{out} + \sum_{k \in A_j} \alpha_{j,k}^{out} F_k(\mathbf{x}, \boldsymbol{\alpha}_k^{in}).$$

The $\boldsymbol{\alpha}_j^{in}$ parameters are fixed from the nested features, and the maximum likelihood estimates for $\boldsymbol{\alpha}_j^{out}$ are calculated by using Model (1) directly, without considering the nonlinear transformation $g_j(\cdot)$. This approach has several benefits. The nonlinear transformation $g_j(\cdot)$ is not involved in the calculation of $\boldsymbol{\alpha}_j^{out}$, allowing for easy application to multiple nonlinear transformations simultaneously. Additionally, non-differentiable functions, such as decision tree characteristic functions or the ReLU function, can be used. Maximum likelihood estimation for generalized linear models creates a convex optimization problem, and the resulting $\boldsymbol{\alpha}_j^{out}$ values are unique. However, fixing $\boldsymbol{\alpha}_j^{in}$ and neglecting the activation function $g_j(\cdot)$ may result in a feature-generating process that is not optimal in terms of prediction accuracy.

Strategy 2, (transform then optimize, concave) In Strategy 1, the weights $\boldsymbol{\alpha}_j^{out}$ are estimated based on $\boldsymbol{\alpha}_j^{in}$. Now, the optimization is performed after the transformation $g_j(\cdot)$ is applied. This means that the weights are calculated as maximum likelihood estimates for the following model:

$$h(\mu) = g_j\Big(\boldsymbol{\alpha}_{j,0}^{out} + \sum_{k \in A_j} \boldsymbol{\alpha}_{j,k}^{out} F_k(\mathbf{x}, \boldsymbol{\alpha}_k^{in})\Big).$$

If $h^{-1}(g_j(\cdot))$ is a concave function, this strategy creates a simple optimization, with uniquely defined estimates. However, if gradient based optimizers are desired, this restricts $h^{-1}(g_j(\cdot))$ to be continuous and differentiable in relevant regions which excludes certain non-linear functions from the model. Otherwise, gradient-free optimization techniques must be utilized.

Strategy 3, (transform then optimize, deep) Similar to Strategy 2, parameters are estimated as maximum likelihood estimates using Model (2b). However, in this strategy, the outer $\boldsymbol{\alpha}_j^{out}$ and nested $\boldsymbol{\alpha}_j^{in}$ are jointly estimated. This means that the optimization is performed with respect to parameters across all layers. All involved nonlinear functions must be continuous and differentiable in relevant regions to enable the use of gradient-based optimizers. One major disadvantage of this strategy is that previous parameter specifications cannot be utilized; all parameters must be recomputed. Additionally, even if all g_j-functions are concave, there is no assurance of finding a unique global optimum of the feature. If gradient-free optimizers are used, the problem becomes extremely computationally demanding. In addition, different local optima define features with the same structural configuration.

Bayesian model specifications

The feature generating process described above gives rise to a extremely large and flexible feature space that is prone to overfitting. In order to avoid this, we will use a Bayesian approach with priors that favours a simple structure. We assume that the $\boldsymbol{\alpha}$ parameters are deterministic and specified through one of the strategies presented above. A more general setting with priors on $\boldsymbol{\alpha}$s is discussed in Hubin, Storvik and Frommlet (2021) but not included here.

We will also mainly use the same priors as presented in the original paper. They start by defining three hard constrains in order to avoid problems with overfitting.

Constraint 1. The depth of any feature is less than or equal to D.

Constraint 2. The width of any feature is less than or equal to L.

Constraint 3. The number of features in a model is less than or equal to Q.

The first constraint provides a finite feature space, while the second and third constraints further limits the number of features and models.

In order to incorporate model (2b) into a Bayesian framework, it is necessary to assign prior probabilities to all parameters. For ease of notation, the symbol $p(\cdot)$ is used to represent a general prior, with its arguments specifying the relevant parameters.

The unique structure of a particular model is determined by the vector $\boldsymbol{\gamma} = (\gamma_1, ..., \gamma_q)$. Our first step will be to establish the prior probabilities for $\boldsymbol{\gamma}$:

$$p(\boldsymbol{\gamma}) \propto \mathrm{I}(|\boldsymbol{\gamma}| \leq Q) \prod_{j=1}^{q} p(\gamma_j).$$

Here, the number of features included in the model, $|\boldsymbol{\gamma}| = \sum_{j=1}^{q} \gamma_j$, is limited by the maximum allowed number of features per model, Q. The factors $p(\gamma_j)$ are used to assign lower prior probabilities to more complex features. Specifically, we use

$$p(\gamma_j) = a^{\gamma_j \mathrm{c}(F_j(\cdot, \boldsymbol{\alpha}_j))}, \tag{4}$$

with $0 < a < 1$ and $\mathrm{c}(F_j(\cdot, \boldsymbol{\alpha}_j)) \geq 0$ being a non-decreasing measure of the complexity of feature j.

This means that if two models differ in just one feature, with one of them being larger, then the prior probability of the larger model will be less than that of the smaller model. The larger the model, the more it will be penalized. The parameter a and the complexity measure $\mathrm{c}(F_j(\cdot\,, \boldsymbol{\alpha}_j))$ hence play a crucial role in determining the quality of the model prior. For example, if a is chosen as e^{-1} and $\mathrm{c}(F_j(\cdot\,, \boldsymbol{\alpha}_j))$ as $\log q_{d_j}$, where d_j represents the depth of F_j, then for $\gamma_j = 1$, the result would be

$$a^{\mathrm{c}(F_j(\cdot\,, \boldsymbol{\alpha}_j))} = \frac{1}{q_{d_j}}.$$

The contribution of a feature to the prior probability of a model will then be inversely proportional to the total number of features having the same depth. This means that more complex features with higher depths will have smaller prior probabilities. This resembles the Bonferroni correction in multiple testing (Bogdan, Ghosh and Tokdar 2008; Scott and Berger 2006).

However, computing the number of features q_d in BGNLM involves nontrivial recursions and can be challenging. To avoid this, we consider an alternative approach based on the geometric distribution, as suggested by Fritsch and Ickstadt (2009). This approach corresponds to penalizing on the number of operations involved in each feature. That is, we use the operations count oc_j of a feature as a complexity measure for BGNLM, which is a ungenerous property that grows smoothly with increased complexity.

The choice of parameter a remains a question. We will borrow from Hubin, Storvik and Frommlet (2021) and mainly use $a = e^{-2}$ for prediction and $a = e^{-log\, n}$ for model identification, inspired by modifications of AIC and BIC, respectively.

In order to finish constructing the Bayesian model, the priors for the components of $\boldsymbol{\beta}$ where $\gamma_j = 1$, and, if necessary, the prior for the dispersion parameter ϕ, need to be specified. We will mainly rely on a Gaussian prior for all β_j which is conjugate for the Gaussian likelihood, resulting in a closed form for the posterior in those cases. However, Hubin, Storvik and Frommlet (2021) considers different approaches, including using Jeffrey's prior and mixtures of g-priors.

Bayesian inference

Posterior marginal probabilities for the model structures are given by

$$p(\boldsymbol{\gamma}|\mathbf{y}) = \frac{p(\boldsymbol{\gamma})p(\mathbf{y}|\boldsymbol{\gamma})}{\sum_{\boldsymbol{\gamma}' \in \mathcal{M}} p(\boldsymbol{\gamma}')p(\mathbf{y}|\boldsymbol{\gamma}')},$$

where $p(\mathbf{y}|\boldsymbol{\gamma})$ is the marginal likelihood of $\mathbf{y}$ for a specific $\boldsymbol{\gamma}$ in the space of possible models $\mathcal{M}$. The posterior inclusion probability for a feature $F_j(\mathbf{x}, \boldsymbol{\alpha}_j)$ is

$$p(\gamma_j = 1|\mathbf{y}) = \sum_{\boldsymbol{\gamma}:\gamma_j=1} p(\boldsymbol{\gamma}|\mathbf{y}).$$

Since the posterior inclusion probability contains s sum over 2^q possible models, an integral of high dimension over the coefficients $\boldsymbol{\beta}$ and an integral over the hyperparameters $\boldsymbol{\eta}$ it is not possible to compute exactly.

Hubin, Storvik and Frommlet (2021) circumvented these issues by splitting the problem into two different problems. The main points at issue in their approach is to calculate the marginal likelihoods $p(\mathbf{y}|\boldsymbol{\gamma})$ for a specific model, and to search through the space of possible models $\boldsymbol{\gamma} \in \mathcal{M}$. For efficient search through the space of models they suggest a special case of Markov Chain Monte Carlo (MCMC) which will be presented shortly.

Based on the results of the computations, the posterior marginal probabilities can then be estimated as

$$\hat{p}(\boldsymbol{\gamma}|\mathbf{y}) = \frac{p(\boldsymbol{\gamma})\hat{p}(\mathbf{y}|\boldsymbol{\gamma})}{\sum_{\boldsymbol{\gamma}' \in \mathcal{M}*} p(\boldsymbol{\gamma}')\hat{p}(\mathbf{y}|\boldsymbol{\gamma}')} \mathrm{I}(\gamma \in \mathcal{M}^*), \tag{5}$$

where the model space is restricted to a appropriate subset of the model space, $\mathcal{M}^* \subset \mathcal{M}$. $\hat{p}(\mathbf{y}|\boldsymbol{\gamma})$ is an estimate (or exact calculation) of the marginal likelihood given model $\boldsymbol{\gamma}$. The marginal likelihood can be written as

$$p(\mathbf{y}|\boldsymbol{\gamma}) = \int_{\boldsymbol{\eta}_m} p(\mathbf{y}|\boldsymbol{\eta}, \boldsymbol{\gamma}) p(\boldsymbol{\eta}|\boldsymbol{\gamma}) d\boldsymbol{\eta},$$

where $\boldsymbol{\eta}$ for a given model is the set of regression coefficients $\{\beta_j, j : \gamma_j = 1\}$ for the features to be included, and possibly, the dispersion parameter ϕ. If we assume that the $\boldsymbol{\alpha}_j$'s are fixed and estimated according to one of the strategies presented, the BGNLM (2b) becomes equal to the GLM (1b) where exact calculations of the marginal likelihoods are available through utilization of conjugate priors. If other priors are to be considered, the marginal likelihoods can be substituted with numerical approximations such as simple Laplace approximations (Tierney and Kadane 1986) or integrated nested Laplace approximations (Rue, Martino and Chopin 2009).

2.4 Genetically Modified MJMCMC (GMJMCMC)

We will in this section give a high level overview of the algorithmic approach used by Hubin, Storvik and Frommlet (2021), and extended by Lachmann (2021), to calculate the posterior (5). However, as this thesis is not necessarily concerned with such methods, we will be very brief. More details are presented in Hubin, Storvik and Frommlet (2021) and Lachmann (2021).

We start with a short introduction to MCMC.

MCMC and the Metropolis-Hastings Algorithm

It is a common problem in Bayesian statistics that the posterior distribution is complicated and difficult to sample from. Different methods have been suggested over the years, and some of the most used methods belongs to the class of algorithms called Markov Chain Monte Carlo. A Markov chain is defined as a sequence of random variables $X_1, X_2, ...$ where the distribution or mass of X_t only depends on the previous state X_{t-1}. It can be shown that if such a process is recurrent, irreducible and aperiodic, there exists a stationary distribution $\pi(\cdot)$ such that the probability of being in a given state remains unchanged over time (Hastings 1970). The idea behind MCMC is to construct a Markov chain with stationary distribution equal to the posterior of interest and then obtain samples of this distribution through recording the states of the chain.

One of the most common MCMC algorithms is the Metropolis-Hastings algorithm. It is a widely used MCMC algorithm that generates samples from $\pi(\cdot)$ using a proposal distribution $q(x^*|x)$. The algorithm accepts proposed new samples x^* based on the current sample x with a probability calculated using the Metropolis-Hastings ratio:

$$r_{mh}(x, x^*) = \min\left\{1, \frac{\pi(x^*)q(x|x^*)}{\pi(x)q(x^*|x)}\right\}.$$

If the proposed new sample x^* is rejected, the algorithm then stays in state x.

The practical implementation of the algorithm requires careful consideration of various factors to optimize its performance, such as the design of an appropriate proposal distribution. The proposal distribution should closely resemble the target distribution while maintaining a high acceptance rate to avoid the algorithm getting stuck and unable to escape the current location.

Mode Jumping MCMC (MJMCMC)

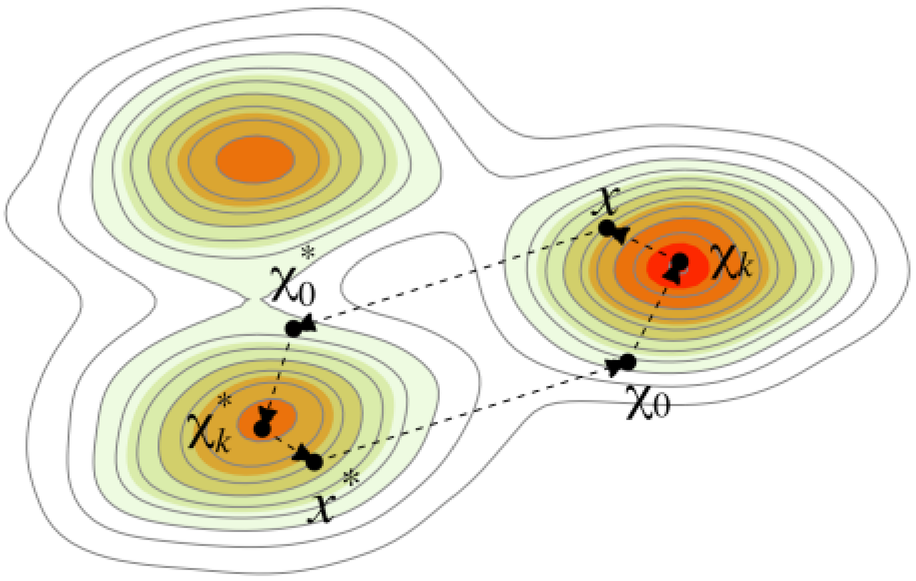


Figure 2.1: Illustration of a mode jumping proposals. Figure is reprinted from Lachmann (2021)

Using standard MCMC algorithms to sample from complicated multi-modal distributions can be difficult to do. This is because the proposal distribution needs to strike a balance between thoroughly exploring modes and being able to transition to other modes. If the proposal distribution makes mostly small steps, it will explore the current location thoroughly but may not escape it, while a proposal that suggests large jumps may end up in points of low probability with respect to the target distribution.

To address this problem, Tjelmeland and Hegstad (2001) introduced Mode Jumping proposals. Starting from the current state x, they generate the new proposal via a two intermediate states X_0^* and X_k^*. In order to be able to calculate the acceptance probability, two *backwards* intermediate states are also visited to get the reverse path (see Figure 2.1). They recommend using mode jumping proposals only a fraction of the time, with regular Metropolis-Hastings kernels generating the remaining proposals. They demonstrate the effectiveness of the algorithm in exploring complicated target distributions with multiple modes through various examples.

While the original MJMCMC algorithm was designed for continuous variables, Hubin, Storvik and Frommlet (2020) adapted it for use with discrete binary variables, as is the context for the problem of variable selection. In addition, MJMCMC requires all features defining the model to be predifined, which is simply too computationally demanding for a BGLNM.

GMJMCMC

Exploring the entire feature space of a BGNLM using MJMCMC is not straightforward due to two main issues. Firstly, the model space of size 2^q increases exponentially with the number of features q. Secondly, q grows super-exponentially with the depth of the features. As a result, it is typically not feasible to predefine the features as it would require a large amount of computing time and memory. To address these problems, Hubin, Storvik and Frommlet (2021) used a modification of MJMCMC they called Genetically Modified MJMCMC (GMJMCMC). This algorithm embeds MJMCMC into a genetic programming framework.

To initialize the chain, they start by performing marginal testing on the covariates, to obtain a subset, $\mathcal{S}_0$. This subset can be thought of as the first population. Generation of subsequent populations are then done in an iterative procedure, where features with low marginal probability in each iteration are replaced to obtain the next generation (Algorithm 1). Each $\mathcal{S}_t$ contains a different set of features and forms a different search space. This results in a dynamic evolution of the population, allowing for different parts of the full model space to be explored without predefining features.

Algorithm 1 GMJMCMC

```
Require: $\mathcal{S}_0$
  Run MJMCMC within the search space of $\mathcal{S}_0$ for N$_{init}$ iterations and
  initialize $\mathcal{S}_1$
  for $t = 1, .., T - 1$ do
      Run MJMCMC within the search space of $\mathcal{S}_t$ for N$_{expl}$ iterations.
      Generate a new population $\mathcal{S}_{t+1}$.
  end for
  Run MJMCMC within the search space of $\mathcal{S}_T$ for N$_{final}$ iterations.
```

MCMC with data subsampling

MCMC algorithms such as Metropolis-Hastings are extremely useful for sampling from complicated posterior distributions, but as the amount of data is increasing, so does computational time. With the trend being ever increasing data sets, both in terms of observations and variables, it seems that such traditional methods are deemed to be replaced. Quiroz, Kohn et al. 2019, who were the first to propose subsampling for MCMC, note that this is unfortunate. Since although MCMC samplers might be slow, they are guaranteed to converge towards the true posterior.

In an attempt to speed up MCMC algorithms, two main paths are being explored. Distributed MCMC that works by running multiple chains in parallel, with each chain using just a partitioning of the data, and subsampling MCMC. The problem with the former approach is how to combine the result of each chain to make inference on the complete data set. Subsampling MCMC however, aims to estimate the likelihood for all the data based on a only a subsample in each step in the chain. This approach is similar to batch methods used in e.g Stochastic Gradient Decent (Robbins 1951).

In Hubin, Storvik and Frommlet (2021) they implement a distributed version of GMJMCMC. While, in his Master's thesis, Lachmann (2021) utilizes subsampling techniques on GMJMCMC. We will in this thesis attempt to suggest another way for computation of the posterior (5). The method is popularly called variational inference, and some important topics regarding our implementation will be discussed in the remainder of this chapter.

2.5 Variational Inference

Variational Bayesian methods, or mean-field methods, was first applied to neural networks (Hinton and Camp 1993; Peterson and Anderson 1987) and later extended to a more general setting (Jordan et al. 1999). It has become an increasingly popular technique in machine learning, particularly in deep learning, due to its ability to scale to large datasets and high-dimensional models.

In variational inference, the problem of computing a difficult posterior distribution is transformed into an optimization problem that can be solved using numerical methods. The main idea is to approximate the true posterior distribution with a simpler distribution that belongs to a family of known parametric distributions. This simpler distribution is popularly called the *variational distribution*, and the parameters of this distribution are optimized to minimize the distance between the true posterior distribution and the variational distribution.

The optimization problem is formulated as a minimization of some discrepancy measure between the true posterior and the variational distribution. The most common choice, and the one we use, is the Kullback-Leibler (KL) divergence (Csiszar 1975). However, other measures such as *f-divergence* (Rényi 1961) and *integral probability measures* (Sriperumbudur et al. 2009) has been applied.

KL divergence

The KL divergence can be thought of as a loss function that measures the amount of information lost when the variational distribution is used to approximate the true posterior distribution. By minimizing the KL divergence, we will tune the variational parameters to produce a distribution that is as close as possible to the true posterior.

The variational parameters, which we will denote $\boldsymbol{\theta}$, are parameters or latent variables and are treated equally in all settings. Let $p(\mathbf{x})$ be the marginal (or joint) distribution of some variable $\mathbf{x}$ (target, not to be confused with data), and let $q_\theta(\mathbf{x})$ be a variational distribution parametrized by $\boldsymbol{\theta}$. The reversed KL divergence, used to fit the approximation to the target distribution, is then defined as:

$$\mathrm{KL}[q_\theta(\mathbf{x})||p(\mathbf{x})] = \int q_\theta(\mathbf{x}) \log \frac{q_\theta(\mathbf{x})}{p(\mathbf{x})} d\mathbf{x}.$$

By rewriting the KL divergence, we obtain:

$$
\begin{aligned}
\mathrm{KL}[q_\theta(\mathbf{x})||p(\mathbf{x})] &= \int q_{\boldsymbol{\theta}}(\mathbf{x}) \log \Big[q_{\boldsymbol{\theta}}(\mathbf{x}) - \log p(\mathbf{x}) \Big] d\mathbf{x} \\
&= \mathbb{E}_{q_{\boldsymbol{\theta}}(\mathbf{x})} \Big[\log q_{\boldsymbol{\theta}}(\mathbf{x}) - \log p(\mathbf{x}) \Big]
\end{aligned}
$$

Minimizing the above expression will then be done with respect to $\boldsymbol{\theta}$. It is typically done in an iterative procedure, using gradient decent or other optimization techniques.

Mean-field variational inference

Mean-field approximations are the most traditional and widely used technique in variational inference for computationally efficient approximation of complex posterior distributions. This approach is based on the mean-field assumption, which assumes that the posterior distribution can be factorized into a product of independent distributions. Specifically, for a set of parameters, $\boldsymbol{\theta}$, and target variable $\mathbf{x}$, the variational distribution can be written as:

$$
q_{\boldsymbol{\theta}}(\mathbf{x}) = \prod_{i=1}^{D_{\mathbf{x}}} q_{\theta_i}(x_i),
$$

where $D_{\mathbf{x}}$ denotes the dimension of $\mathbf{x}$.

This assumption simplifies the optimization problem, as we only need to optimize each individual distribution in the product, rather than the entire posterior approximation. That is, minimization of the KL-divergence between the approximate posterior distribution and the target distribution is done by optimizing the parameters θ_j in each individual distribution in the product. This optimization can be done using gradient descent or other optimization algorithms.

Although mean-field variational inference is a powerful technique, it also has several drawbacks. For instance, it assumes that the target distribution is factorizable. This assumption may not hold for complex models with strong dependencies between the parameters. In addition, the quality of the approximation depends heavily on the choice of the variational distribution. If this distribution is not flexible enough to capture the true posterior distribution, the approximation may be poor.

Various methods has been used to produce increasingly flexible variational distributions. In the next sections, we will discuss a class of popular such techniques.

2.6 Normalizing Flows

The term normalizing flows was first coined by Tabak and Vanden-Eijnden (2010) and Tabak and Turner (2013) in the context of classification and density estimation. It has since then seen a lot of development and interest. A nice overview of different methods can be found in Papamakarios, Nalisnick et al. (2021). For ease of notation, we here let $p(\cdot)$ or $q(\cdot)$ denote arbitrary distributions. Later, when the parameters of these distributions are relevant, we will go back to using subscript for parameters.

The concept of normalizing flows involves creating flexible probability distributions over continuous random variables. Consider vector $\mathbf{x} \in \mathbb{R}^D$, and suppose the aim is to define the joint distribution of $\mathbf{x}$. Using the flow-based modeling approach we will transform a real vector $\mathbf{u} \in \mathbb{R}^D$, sampled from simple *base distribution* $q(\mathbf{u})$, into $\mathbf{x}$ through a transformation T

$$\mathbf{x} = T(\mathbf{u}) \quad \text{where} \quad \mathbf{u} \sim q(\mathbf{u}). \tag{7}$$

The transformation T must be a *diffeomorphism* and hence differentiable and invertible and T^{-1} must be also be differentiable. The density of $\mathbf{x}$ is then well-defined and can be obtained by change of variables

$$p(\mathbf{x}) = q(\mathbf{u})\, |\det J_T(\mathbf{u})|^{-1} \quad \text{where} \quad \mathbf{u} = T^{-1}(\mathbf{x}).$$

$J_T(\mathbf{u}) \in \mathbb{R}^{D\times D}$ is the Jacobian, the matrix of all partial derivatives, of T. Equivalently, we can write $p(\mathbf{x})$ in terms of the Jacobian of T^{-1}

$$p(\mathbf{x}) = q(T^{-1}(\mathbf{x}))\, |\det J_{T^{-1}}(\mathbf{x})|. \tag{8}$$

The log-density is then

$$\log p(\mathbf{x}) = \log q(\mathbf{u}) - \log \left|\det J_T(\mathbf{u})\right|. \tag{9}$$

A useful property of the differentiable and invertible transformations is that they are composable, and that the resulting composistion is differentiable and invertible. This means that we can chain together multiple transformations $T_1, ..., T_K$ to obtain $T = T_1 \circ \cdots \circ T_K$ where each T_k transforms $\mathbf{z}_{k-1}$ into

$\mathbf{z}_k$, assuming $\mathbf{z}_0 = \mathbf{u}$ and $\mathbf{z}_K = \mathbf{x}$. Thus, the term "flow" refers to the path taken from a set of samples from $q(\mathbf{u})$ as they undergo a sequence of transformations $T_1, ..., T_K$. The term "normalizing" stems from the fact that the inverse flow, through $T_K^{-1}, ..., T_1^{-1}$, transforms a set of samples from $p(\mathbf{x})$ into a set of samples from the designated density $q(\mathbf{u})$, effectively "normalizing" them into a proper density (Papamakarios, Nalisnick et al. 2021).

In terms of its abilities, a flow-based model offers two functions: generating samples from the model with Equation (7) and determining the model's density through Equation (8). Generating samples requires the capacity to sample from $q(\mathbf{u})$ and compute the forward transformation T. On the other hand, evaluating the model's density necessitates computing the inverse transformation T^{-1} and its Jacobian determinant, as well as determining the density $q(\mathbf{u})$.

Constructing a flow

As discussed previously, the normalizing flows are composable, meaning we can construct a flow by composing a finite number of transformations T_k

$$T = T_1 \circ \cdots \circ T_K.$$

The objective is to utilize simple transformations as basic components, each having an easily invertible Jacobian determinant, to form a more sophisticated transformation with greater expressiveness than any of its individual components. The forward and inverse evaluations, as well as the computation of the Jacobian determinant, will be restricted to the sub-flows. Respectively, with $\mathbf{z}_0 = \mathbf{u}$ and $\mathbf{z}_K = \mathbf{x}$, the forward and backward evaluations are:

$$\mathbf{z}_k = T_k(\mathbf{z}_{k-1}) \quad \text{for } k = 1, ..., K,$$
$$\mathbf{z}_{k-1} = T_k^{-1}(\mathbf{z}_k) \quad \text{for } k = K, ..., 1.$$

The Jacobian log-determinant is calculated as

$$\log|\det J_T(\mathbf{z}_0)| = \log\left|\prod_{k=1}^{K} \det J_{T_k}(\mathbf{z}_{k-1})\right| = \sum_{k=1}^{K} \log|\det J_{T_k}(\mathbf{z}_{k-1})|.$$

In practical terms, we implement either T_k or T_k^{-1} using a neural network, parameterized by $\boldsymbol{\phi}_k$ which we represent as f_{ϕ_k}. This means that we can use the model f_{ϕ_k} to carry out either T_k, where it takes $\mathbf{z}_{k-1}$ as input and

$$\begin{tikzcd}
\mathbf{z}_0 \arrow[r] & \boxed{T_1} \arrow[r, "\mathbf{z}_1"] \arrow[d] & \boxed{T_2} \arrow[r, "\mathbf{z}_2"] \arrow[d] & \cdots \arrow[r, "\mathbf{z}_{K-1}"] & \boxed{T_K} \arrow[r] \arrow[d] & \mathbf{z}_K \\
& \log|\det J_{T_1}(\mathbf{z}_0)| + & \log|\det J_{T_2}(\mathbf{z}_1)| & + \cdots + & \log|\det J_{T_K}(\mathbf{z}_{K-1})| & = \log|\det J_T(\mathbf{z}_0)|
\end{tikzcd}$$

Figure 2.2: This figure is inspired by Papamakarios, Nalisnick et al. (2021) and illustrates a flow composed of K transformations.

produces $\mathbf{z}_k$ as output, or T_k^{-1}, where it takes $\mathbf{z}_k$ as input and produces $\mathbf{z}_{k-1}$ as output. Regardless, we must ensure that the model is reversible and has a tractable Jacobian determinant. Ensuring that f_{ϕ_k} is reversible and explicitly computing its inverse are not equivalent concepts. Although the inverse of f_{ϕ_k} is certain to exist in some implementation, precise computation can be costly or impractical.

As discussed above, the forward transformation T is utilized for sampling, while the inverse transformation T^{-1} is utilized for density evaluation. If the inverse of f_{ϕ_k} is not efficient, either density evaluation or sampling will be slow or impractical. We should also specify what we mean with "tractable Jacobian determinant". Although we can always compute the Jacobian matrix of a differentiable function, calculating its determinant explicitly can be computationally expensive. For flow-based models, the computation of the Jacobian determinant should be no more linear time with respect to the input dimension.

Normalizing flows provide a powerful framework for generative modeling by transforming a simple distribution to a complex one via a series of invertible and tractable transformations. By using neural networks as building blocks, we can construct a normalizing flow with high expressiveness and computational efficiency. Composable transformations allow for both efficient density evaluation and sampling. However, designing transformations with tractable Jacobian determinants is not always straightforward, and it requires careful consideration to balance between expressiveness and computational efficiency.

Inference for Flow-based Models

Similarly to fitting any probabilistic model, fitting a flow-based model $q_\omega(\mathbf{x})$ to a target distribution $p(\mathbf{x})$ can be done by minimizing the discrepancy between them. This minimization will be done with respect to the model's parameters $\boldsymbol{\omega} = \{\boldsymbol{\phi}, \boldsymbol{\psi}\}$ where $\boldsymbol{\phi}$ are the parameters of the flow T_ϕ and $\boldsymbol{\psi}$ are the parameters of the base distribution $p_\psi(\mathbf{u})$.

Our discrepancy measure is the KL-divergence, and we will here separate between forward- and reversed KL-divergence. Fitting a model using forward KL-divergence is equivalent to maximum likelihood estimation, while the reverse KL-divergence is commonly used by Bayesians in the context of variational inference.

Forward KL-divergence

The forward KL-divergence between the target distribution $p(\mathbf{x})$ and the flow model $q_\omega(\mathbf{x})$ can be written as

$$\begin{aligned}\mathcal{L}(\boldsymbol{\omega}) &= \mathrm{KL}[p(\mathbf{x})||q_\omega(\mathbf{x})] \\ &= -\mathbb{E}_{p(\mathbf{x})}\Big[\log q_\omega(\mathbf{x})\Big] + \text{constant} \\ &= -\mathbb{E}_{p(\mathbf{x})}\Big[\log p_\phi(T_\phi^{-1}(\mathbf{x})|\boldsymbol{\psi}) + \log|\det J_{T_\phi^{-1}}(\mathbf{x})|\Big] + \text{constant}.\end{aligned}$$

Using the forward KL-divergence is well suited for situations where we can obtain samples from the target distribution, but are unable to evaluate the the density $p(\mathbf{x})$. If we are able to sample $\{\mathbf{x}_i\}_{i=1}^N$ from $p(\mathbf{x})$, we can estimate the expectation above by Monte Carlo:

$$\mathcal{L}(\boldsymbol{\omega}) \approx -\frac{1}{N}\sum_{i=1}^{N}\Big[\log p_\psi(T_\phi^{-1}(\mathbf{x}_i)) + \log|\det J_{T_\phi^{-1}}(\mathbf{x}_i)|\Big] + \text{constant}.$$

Minimizing this Monte Carlo estimate is then equivalent to fitting the flow-based model to the samples through maximum likelihood estimation, and the parameters can e.g. be optimized by gradient-based methods.

When using the forward KL-divergence for inference, we need to compute the inverse flow, T_ϕ^{-1}, its Jacobian determinant and the density $q_\psi(\mathbf{u})$, as well as computing the derivative of all three if we are using gradient-based optimization.

Reversed KL-divergence

The standard way of performing variational inference is through minimization of the reversed KL-divergence. Here, the target density $p(\mathbf{x})$ will hence be a posterior distribution of interest. We have the following expression:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\omega}) &= \mathrm{KL}\Big[q_{\boldsymbol{\omega}}(\mathbf{x})||p(\mathbf{x})\Big] \\ &= \mathbb{E}_{q_{\boldsymbol{\omega}}(\mathbf{x})}\Big[\log q_{\boldsymbol{\omega}}(\mathbf{x}) - \log p(\mathbf{x})\Big] \\ &= \mathbb{E}_{p_{\psi}(\mathbf{u})}\Big[\log p_{\psi}(\mathbf{u}) - \log|\mathrm{det} J_{T_{\phi}}(\mathbf{u})| - \log p_{\phi}(T(\mathbf{u})))\Big].\end{aligned}$$

Where we have used a change of variables in order to express the expectation with respect to $\mathbf{u}$. In order to use the reversed KL-divergence, we need to evaluate the target density. However, since the target density is the posterior, we let $p(\mathbf{x}) = p^*(\mathbf{x})/C$, where $p^*(\mathbf{x})$ is likelihood$\times$prior and $C = \int p^*(\mathbf{x})d\mathbf{x}$ is the intractable normalizing constant, and rewrite the reverse KL-divergence as

$$\mathcal{L}(\boldsymbol{\omega}) = \mathbb{E}_{p_{\psi}(\mathbf{u})}\Big[\log p_{\psi}(\mathbf{u}) - \log|\mathrm{det} J_{T_{\phi}}(\mathbf{u})| - \log p^*(T_{\phi}(\mathbf{u})))\Big] + \text{constant}.$$

In practice, we minimize $\mathcal{L}(\boldsymbol{\omega})$ with a gradient-based method. Since we are taking expectation with respect to to the base distribution, $p_{\psi}(\mathbf{u})$, we can easily use Monte Carlo to obtain an unbiased estimate of the gradient of $\mathcal{L}(\boldsymbol{\omega})$ with respect to $\boldsymbol{\phi}$. Let $\{\mathbf{u}\}_{i=1}^N$ be samples from $p_{\psi}(\mathbf{u})$. The gradient with respect to $\boldsymbol{\phi}$ can then be estimated as

$$\nabla_{\phi}\mathcal{L}(\boldsymbol{\omega}) \approx -\frac{1}{N}\sum_{i=1}^{N}\Big[\nabla_{\phi}\log|\mathrm{det} J_{T_{\phi}}(\mathbf{u}_i)| + \nabla_{\phi}\log p^*(T_{\phi}(\mathbf{u}_i)\Big].$$

Relationship between forward and reverse KL-divergence

As an alternative, one can think of the target $p(\mathbf{x})$ as the base distribution and the inverse flow as inducing a distribution $q^*_{\phi}(\mathbf{u})$. Intuitively, $q^*_{\phi}(\mathbf{u})$ is the distribution that $\mathbf{x}$ will follow when passed through the inverse flow T^{-1}. Since the target distribution and the base distribution uniquely determines each other when given the flow transformation, the induced distribution $q^*_{\phi}(\mathbf{u})$ is equal to the base $q_{\psi}(\mathbf{u})$ if and only if the target $p(\mathbf{x})$ is equal to the flow $q_{\omega}(\mathbf{u})$. Therefore, we can think of fitting the flow model to the target as fitting the induced distribution to the base and vice versa.

In Papamakarios, Pavlakou and Murray 2018, they indeed show that

$$\mathrm{KL}[q_{\omega}(\mathbf{x})||p(\mathbf{x})] = \mathrm{KL}[q_{\psi}(\mathbf{u})||q^*_{\phi}(\mathbf{u})],$$

which means that fitting the induced distribution $q^*_\phi(\mathbf{u})$ to the base $q_\psi(\mathbf{u})$ through forward KL-divergence (maximum likelihood) is equivalent to fitting the flow model to the target via reversed KL-divergence.

2.7 Autoregressive Flows

In the next sections, we will be discussing the autoregressive flows. They are one of the most widely used classes of normalizing flows due to its effectiveness in density estimation and its simplicity of implementation.

We will mainly be concerned with efficient construction of the flow components, f_{ϕ_k}. We simplify the notation and drop $\boldsymbol{\phi}$ from f_{ϕ_k} and call it f_k as it should be clear what we mean by this. We will also denote the input of the model as $\mathbf{z}$ and the output as $\mathbf{z}'$ regardless of whether the model implements the forward or the inverse flow.

In an autoregressive flow, f_k has the following form:

$$z'_i = \tau(z_i; \mathbf{h}_i) \quad \text{where} \quad \mathbf{h}_i = c_i(\mathbf{z}_{<i}),$$

where τ is referred to as the transformer and c_i as the i-th conditioner. The transformer is required to be a strictly monotonic function of the input z_i. It is parameterized by $\mathbf{h}_i$, and specifies how the flow changes z_i to give output z'_i. The conditioners determines the parameters of the transformer. They take as input only the indices of the input less than i, giving rise to the autoregressive structure. Each conditioner can in principle be implemented as an arbitrary function of $\mathbf{z}_{<i}$ that outputs $\mathbf{h}_i$. However, if each $c_i(\mathbf{z}_{<i})$ is a different model it would scale very poorly with dimensionality D. It is therefore common practice to share parameters across conditioners, or to combine the conditioners into a single model.

Since the transformer is monotonic, it is also invertible. Given output $\mathbf{z}'_i$ we can compute the input $\mathbf{z}$ through

$$z_i = \tau^{-1}(z'_i; \mathbf{h}_i) \quad \text{where} \quad \mathbf{h}_i = c_i(\mathbf{z}_{<i}).$$

Since z'_i does not depend on $\mathbf{z}_{>i}$, the partial derivative of z'_i with respect to z_j is zero for $j > i$. A key property of these transformations is therefore that the Jacobian is lower triangular. This makes the computation of the Jacobian determinant easily tractable, as the determinant of any triangular matrix is equal to the product of the diagonal.

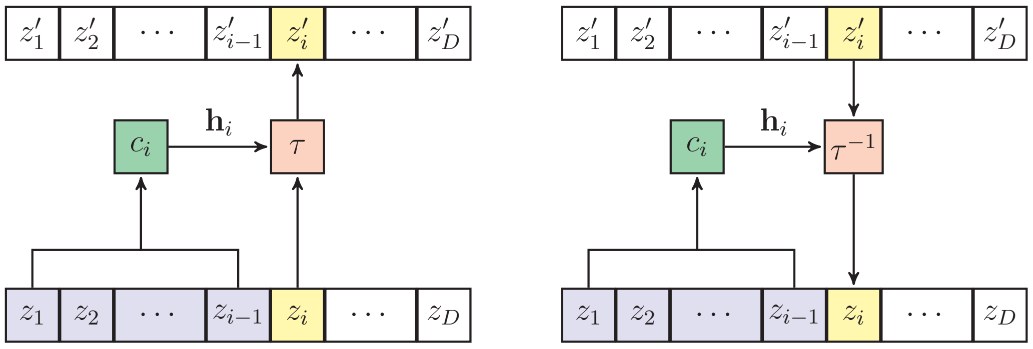


Figure 2.3: Illustration of the i-th step of a normalizing flow. **Left:** Forward flow. **Right:** Inverse flow.

The Jacobian of the transformer is

$$J_{f_k}(\mathbf{z}) = \begin{bmatrix} \frac{\partial \tau}{\partial z_1}(z_1; \mathbf{h}_1) & & \mathbf{0} \\ & \ddots & \\ \mathbf{L}(\mathbf{z}) & & \frac{\partial \tau}{\partial z_D}(z_D; \mathbf{h}_D) \end{bmatrix},$$

and the log determinant can then be computed as

$$\log \left| \det J_{f_k(\mathbf{z})} \right| = \log \left| \prod_{i=1}^{D} \frac{\partial \tau}{\partial z_i}(z_i; \mathbf{h}_i) \right| = \sum_{i=1}^{D} \log \left| \frac{\partial \tau}{\partial z_i}(z_i; \mathbf{h}_i) \right|.$$

Implementing the autoregressive flow comes down to choice of transformer and conditioner. Any type of transformer can in practice be paired with any type of conditioner and numerous combinations are represented in the literature. For our implementation, we follow the *Inverse Autoregressive Flow* (IAF) from Kingma, Salimans, Jozefowicz et al. (2017). They suggest pairing an *affine transformer* with a *masked conditioner*.

Affine Transformers

Perhaps the simplest transformers used within autoregressive flows belongs to the class of affine functions. We will restrict our transformer τ to be on this form:

$$\tau(z_i; \mathbf{h}_i) = \alpha_i z_i + \beta_i, \quad \text{where} \quad \mathbf{h}_i = \{\alpha_i, \beta_i\}.$$

It can be thought of as a location-scale transformation, where β_i defines the location and α_i the scale. The transformation is invertible if and only if $\alpha_i \neq 0$, which can be guaranteed by letting $\alpha_i = \exp \tilde{\alpha}_i$, where $\tilde{\alpha}_i$ is an unconstrained parameter. The derivative of an affine transformer with respect to z_i, is α_i, and the log determinant is:

$$\log \left|\det J_{f_k}(\mathbf{z})\right| = \sum_{i=1}^{D} \log |\alpha_i| = \sum_{i=1}^{D} \tilde{\alpha}_i.$$

While affine transformers have analytical tractability, their expressivity is limited. To illustrate why, let $\mathbf{z}$ follow a Gaussian distribution. Then, each z'_i conditioned on $\mathbf{z}_{<i}$ will also follow a Gaussian distribution. That is, a single affine transformation of a multivariate Gaussian results in a distribution whose conditionals $p(z'_i|\mathbf{z}'_{<i})$ are also Gaussian by necessity. This problem is often addressed by stacking multiple layers of affine transformers, but the expressive powers of the final flow still remains unknown (Papamakarios, Nalisnick et al. 2021).

Masked conditioners

As mentioned above, it is common practice to implement conditioners that shares parameters, and that is exactly what makes masked conditioners attractive. This approach uses a single, typically feed forward neural network that takes input $\mathbf{z}$ and outputs the whole sequence $(\mathbf{h}_1, ..., \mathbf{h}_D)$ in one pass, only requiring obedience with respect to the autoregressive structure: output $\mathbf{h}_i$ can only depend on $\mathbf{z}_{<i}$.

In constructing such a network, one takes an arbitrary neural network and removes connections until there is no path from input z_i to outputs $(\mathbf{h}_1, ..., \mathbf{h}_i)$. This is done trough a technique called *masking*, where each weight matrix is multiplied with a binary matrix of the same size. The connections that are to be removed will correspond to a zero-entry in the mask matrix, and all other connections will remain unmodified. The masked network will have the same architecture and size as the original network, retaining the computational properties.

A key advantage of masked autoregressive flows is that they are efficient to evaluate. Given $\mathbf{z}$, the parameters $(\mathbf{h}_1, ..., \mathbf{h}_D)$ are computed in a singe neural network pass where each dimension can be computed in parallel via $z'_i = \tau(z_i, \mathbf{h}_i)$.

A main disadvantage is however that the inverse is not as efficient to evaluate. This is because parameters $\mathbf{h}_i$ that are needed to obtain the inverse $z_i = \tau^{-1}(z'_i, \mathbf{h}_i)$ cannot be computed until $(z_1, ..., z_{i-1})$ have been obtained. That is, we must compute $\mathbf{h}_1$ to obtain z_1, $\mathbf{h}_2$ to obtain z_2 and so on until z_D have been obtained. Despite computational issues related to the inversion, the masked conditioner remains one of the most used technique for implementing autoregressive flows. Especially, it is useful for situations where the dimension of the data is not too large or where inverting the flow is not needed. Examples of autoregressive models with masking include IAF (Kingma, Salimans, Jozefowicz et al. 2017), MAF (Papamakarios, Pavlakou and Murray 2018) and NAF (Cao, Titov and Aziz 2019). Masking has also been used in non-flow autoregressive models such as MADE (Germain et al. 2015). The former is the transformer in our implementation and the latter is our choice for conditioner. They will also be the topics in the following sections.

2.8 Masked Autoencoders for Distribution Estimation (MADE)

Masked autoencoders for distribution estimation was first introduced by Germain et al. (2015). The original paper has nothing to do with normalizing flows, but it is an easy to implement autoregressive model and is widely used.

We are here assuming a Gaussian model. Given a set of variables $\{\mathbf{z}\}_{i=1}^{D}$, the goal of the autoencoder is to learn the hidden statistical structure that generated them. Borrowing the notation from Germain et al. (2015), this autoencoder can be written as a neural network in the following way:

$$\begin{aligned}\mathbf{h}(\mathbf{z}) &= \mathbf{g}(\mathbf{b} + \mathbf{W}\mathbf{z}),\\ \mathbf{z}' &= \mathbf{c} + \mathbf{V}\mathbf{h}(\mathbf{z}),\end{aligned}$$

where $\mathbf{h}(\mathbf{z})$ is a representation of the hidden structure we wish to learn, $\mathbf{W}$ and $\mathbf{V}$ are matrices, $\mathbf{b}$ and $\mathbf{c}$ are vectors and $\mathbf{g}$ is a nonlinear activation function.

In order to satisfy the autoregressive property, we will need to modify the autoencoder. Since output z'_i can only depend on inputs $\mathbf{z}_{<i}$ it means that inputs $\mathbf{z}_{>i}$ can not be used to compute z'_i. MADE solves this problem with masking. In short, masking corresponds to setting at least one connection in matrix $\mathbf{W}$ or $\mathbf{V}$ to 0. One way to do this is to elementwise-multiply each matrix with a binary matrix called a mask matrix. The entries of the mask matrix are zero if we wish to remove the corresponding connection. We now write

$$\begin{aligned}\mathbf{h}(\mathbf{z}) &= \mathbf{g}(\mathbf{b} + (\mathbf{W} \odot \mathbf{M}^{\mathbf{W}})\mathbf{z}),\\ \mathbf{z}' &= \mathbf{c} + (\mathbf{V} \odot \mathbf{M}^{\mathbf{V}})\mathbf{h}(\mathbf{z}),\end{aligned} \tag{10}$$

where $\odot$ denotes elementwise multiplication.

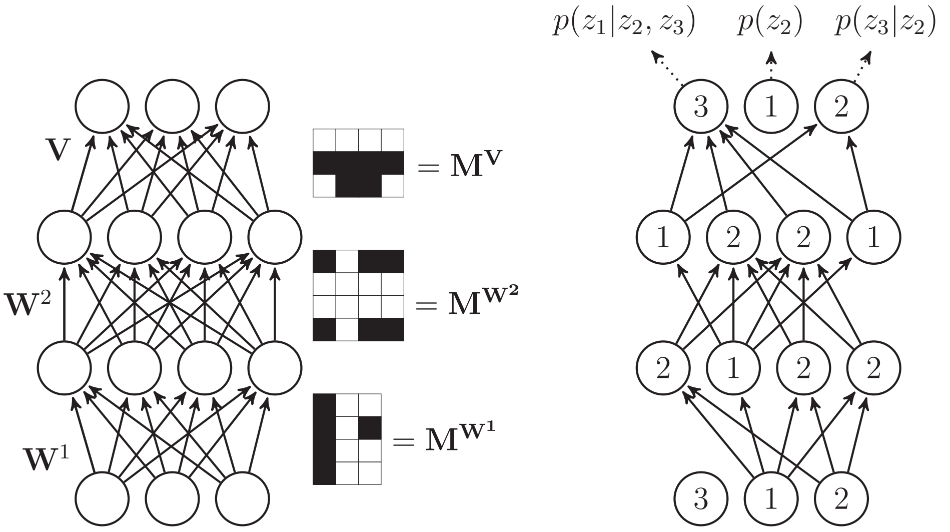


Figure 2.4: This figure is borrowed from Germain et al. (2015). **Left:** Standard three hidden layer autoencoder. **Right**: MADE. The network has the same structure as the autoencoder, but a set of connections is removed.

The problem at hand is how to create the masks matrices. Germain et al. (2015) start by assigning an integer m between 1 and $D - 1$ to each neuron in the hidden layer. The k-th hidden unit's number, $m(k)$, defines the maximum number of inputs connected to that neuron. The reason for disallowing $m(k) = 1$ and $m(k) = D$ is to make sure that hidden units are not constant or depend on all input units and hence not be able to model any of the conditionals $p(z_d|\mathbf{z}_{<d})$. The mask matrices that governs the connections between all layers except the last one follows these constraints, and are encoded in the following way:

$$M^{\mathbf{W}}_{k,d} = \mathbf{1}_{m(k)\geq d} = \begin{cases} 1 & \text{if } m(k) \geq d, \\ 0 & \text{otherwise,} \end{cases}$$

for $d \in \{1, ..., D\}$ and $k \in \{1, ..., K\}$. In the last layer of connections, we need to make sure that the d-th output unit is only connected to $\mathbf{z}_{<d}$.

Therefore, we must make sure that hidden units that are connected to the d-th output unit have $m(k) < d$ and hence connected to at most $d-1$ input units. The output mask matrix can hence be encoded as

$$M^{\mathbf{V}}_{d,k} = \mathbf{1}_{d \geq m(k)} = \begin{cases} 1 & \text{if } d \geq m(k), \\ 0 & \text{otherwise.} \end{cases}$$

2.9 Inverse Autoregressive Flows (IAF)

Inverse Autoregressive Flows was first introduced by Kingma, Salimans, Jozefowicz et al. (2017). It is well known to scale well to high dimensional latent spaces, and plays a central role in our implementation in the next chapter. We will here go through their reasoning and borrow from their notation.

We again assume a Gaussian model. Let $\mathbf{y} = \{y_i\}_{i=1}^{D}$ be a variable modeled by a computationally efficient Gaussian version of an autoregressive conditioner, such as MADE. We denote the output of this conditioner $[\boldsymbol{\mu}(\mathbf{y}), \boldsymbol{\sigma}(\mathbf{y})]$ as a function of $\mathbf{y}$, which elements $[\mu_i(\mathbf{y}_{<i}), \sigma_i(\mathbf{y}_{<i})]$ are the predicted mean and standard deviation of the i-th element of $\mathbf{y}$. Due to the autoregressive structure, the Jacobian is lower triangular with zeros on the diagonal. That is, $\frac{\partial \boldsymbol{\mu}_i}{\partial y_j} = \frac{\partial \boldsymbol{\sigma}_i}{\partial y_j} = 0$ for $j \geq i$.

When sampling from such a model, one can transform a noise vector $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$ into the corresponding vector $\mathbf{y}$: $y_0 = \mu_0 + \sigma_0 \odot \epsilon_0$, and for $i > 0$, $y_i = \mu_i(\mathbf{y}_{<i}) + \sigma_i(\mathbf{y}_{<i}) \cdot \epsilon_i$. This is often referred to as the local reparametrization trick (LRT), explained in detail by Kingma, Salimans and Welling (2015). Since traditional variational inference requires sampling from the posterior, this setup is not relevant for direct use in application. However, the inverse transformations are interesting for inference through normalizing flows. Assume $\sigma_i > 0$ for all i. Inverting the given transformation yields

$$\boldsymbol{\epsilon} = \frac{\mathbf{y} - \boldsymbol{\mu}(\mathbf{y})}{\boldsymbol{\sigma}(\mathbf{y})},$$

where the subtraction and division are elementwise.

A key property of the inverse transformation is again the simple Jacobian determinant. Due to the autoregressive structure, we have $\frac{\partial \boldsymbol{\mu}_i}{\partial y_j} = \frac{\partial \boldsymbol{\sigma}_i}{\partial y_j} = 0$ for $j \geq i$, resulting in $\frac{\partial \epsilon_i}{\partial y_i} = 0$ for $j \geq i$, and the diagonal elements are $\frac{\partial \epsilon_i}{\partial y_i} = \sigma_i$. Hence the log-determinant is simply

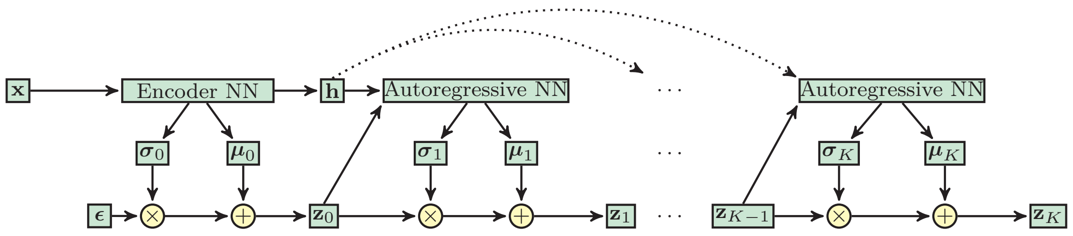


Figure 2.5: Illustration of an Inverse Autoregressive Flow. Here, $\mathbf{x}$ is transformed into $\mathbf{z}_K$ through a series of autoregressive neural networks.

$$\log \det \left| \frac{d\boldsymbol{\epsilon}}{d\mathbf{y}} \right| = \sum_{i=1}^{D} -\log \sigma_i(\mathbf{y}).$$

Constructing the IAF

To initialize a chain of flows, Kingma, Salimans, Jozefowicz et al. (2017) uses a encoder neural network that takes input $\mathbf{x}$ and outputs $\boldsymbol{\mu}_0$, $\boldsymbol{\sigma}_0$ and $\mathbf{h}$. Here, they use $\mathbf{h}$ as an additional input to each step in the flow. We then draw a random sample $\boldsymbol{\epsilon} \sim N(0, \mathbf{I})$ and initialize the chain with

$$\mathbf{z}_0 = \boldsymbol{\mu}_0 + \boldsymbol{\sigma}_0 \odot \boldsymbol{\epsilon}.$$

Then, following previous logic, constructing the flow is done by composing K of the following transformations

$$\mathbf{z}_k = \boldsymbol{\mu}_k + \boldsymbol{\sigma}_k \odot \mathbf{z}_{k-1}, \tag{11}$$

where a different neural network that takes input $\mathbf{z}_{k-1}$ and $\mathbf{h}$, and outputs $\boldsymbol{\mu}_k$ and $\boldsymbol{\sigma}_k$ is used at each step (see Figure 2.5). These neural networks are autoregressive with respect to $\mathbf{z}_{k-1}$, such that the Jacobian $\frac{d\mathbf{z}_k}{d\mathbf{z}_{k-1}}$ is triangular with $\boldsymbol{\sigma}_k$ on the diagonal.

The Jacobian log-determinant of the k-th step is

$$\log \left| \frac{\partial \mathbf{z}_k}{\partial \mathbf{z}_{k-1}} \right| = \sum_{i=1}^{D} \log \sigma_i \ ,$$

and, still assuming a Gaussian model, we can get the log density of the last iterate through (9):

$$\log p(\mathbf{z}_K) = -\sum_{i=1}^{D}\left(\frac{1}{2}\epsilon_i^2 + \frac{1}{2}\log(2\pi) + \sum_{k=0}^{K}\log\sigma_{k,i}\right).$$

The flexibility of this distribution, and its potential to closely fit the true posterior will increase with the depth of the chain and with the expressiveness of the autoregressive neural networks within each step.

For numerical stability, Kingma, Salimans, Jozefowicz et al. (2017) suggest the following setup in each iteration:

$$\begin{aligned}\mathbf{m}_k, \mathbf{s}_k &\leftarrow \text{AutoregressiveNN}[k](\mathbf{z}_k, \mathbf{h}),\\ \boldsymbol{\sigma}_k &= \text{sigmoid}(\mathbf{s}_k),\\ \mathbf{z}_k &= \boldsymbol{\sigma}_k \odot \mathbf{z}_{k-1} + (1-\boldsymbol{\sigma}_k)\odot \mathbf{m}_k.\end{aligned}$$

2.10 Optimization algorithms

A key part of any machine learning model is the technique used for optimization of its parameters. Over the years, many different such algorithms have been suggested, and a thorough investigation of the most influential can be found in Bottou, Curtis and Nocedal (2018). Since it is such an important topic and has great influence on our model, we will end this chapter with a short overview.

Any optimization problem can be formulated as finding the maximum or minimum value of some objective function, f. In our case, as we are trying to find the values for some parameter set, ξ, that minimizes some loss function, we can formalize optimization problem as

$$\text{minimize}_{\xi}\ f(\xi).$$

There are different ways to approach such problems, depending on the function f. Searching for an optimum is however generally done in an iterative procedure.

Gradient free optimizers

If the gradient of f is not tractable or costly to compute, we can use gradient free optimizers. Included in this category is Greedy Search, Simulated Annealing, and Genetic Algorithms, amongst others. Such methods typically have slow convergence rate and low computational cost in each iteration.

If the possible number combinations is too large, we will not be able to find an optimal solution within a reasonable amount of time. However, if we are able to evaluate the objective function at any time, these methods will work and are guaranteed to converge towards a local optimum.

First-order methods

If the gradient of f with respect to ξ is tractable, first-order methods such as gradient decent (GD) will often speed up the convergence. However, since such methods requires calculation of the gradient in each iteration they also come with a higher computational cost in each iteration.

In traditional GD, a step proportional to the negative gradient is taken in each iteration. One starts with a initial guess ξ_0 and updates the guess according to

$$\xi_{t+1} = \xi_t - \eta \, \nabla_\xi f(\xi_t),$$

where, η is referred to as the *learning rate*.

Due to potentially high computational cost for computing the gradient, Robbins (1951), came up with Stochastic Gradient Descent (SGD). Here, the gradient is replaced with an unbiased estimate, $\widetilde{\nabla}_\xi f(\xi_t)$. This produces noisy steps in the path towards the minimum, and SGD might take more steps to converge. However, as the computation of the stochastic gradients are generally much faster, this approach often leads to less time to converge if the full gradient is costly to compute. In the original paper, they suggested using only one observation for each iteration, but this can lead to too much variance causing slow convergence. Typically, the gradient is updated using a batch of observations in each iteration.

Careful consideration should also be given to choosing the right learning rate. If it is too small, convergence will be slow. On the other hand, if the learning rate is too large, one might make too big steps and risk "jumping" over the optimum. To deal with these issues, momentum and acceleration methods have been developed (Rumelhart, Hinton and Williams 1986). Such methods has a dynamic learning rate, depending on the steepness of the gradient, allowing for larger updates if the gradient is very steep and smaller updates if the gradient is less steep.

If ξ contains many different parameters, as is the case in in deep learning, the gradient may vary in many different magnitudes. This makes the problem of defining a single global learning rate for all parameters very challenging, and the need for methods with different learning rates for different parameters is apparent. Hinton, Srivastava and Swersky (2018) was the first to propose such methods, and they called it RMSprop.

Recent developments have lead to the Adaptive moment estimation (Kingma and Ba 2017, Adam). It works by combining ideas related to momentum and RMSprop, and has shown to generalize well to a wide range of machine learning applications.

Second-order methods

If the second-order derivative (i.e the Hessian) of f is tractable, one can use second order methods to further speed up convergence rate. Such methods are scale-invariant, meaning we do not have to scale the input variables. However, as they generally require a lot of computation in each iteration and scale very poorly with the dimension of the parameter space, they are seldom used for machine learning purposes.

The most prominent of second-order optimization methods is Newton's Methods (Battiti 1992). In each iteration, these methods takes a step according to

$$\begin{aligned} \xi_{t+1} &= \xi_t + \eta_t s_t, \\ \text{where } s_t \text{ satisfies} : \nabla^2 f(\xi_t) s_t &= -\nabla f(\xi_t). \end{aligned}$$

In practical terms, this means that such methods even require inversion of the Hessian. Therefore, Bottou, Curtis and Nocedal (2018) highlights that if the Hessian is not positive-definite, second-order methods might not even work where first-order methods do.

CHAPTER 3

Contribution and further specifications

In this chapter we will combine the theory from the previous chapter. The BGLM (1b) and BGNLM (2b) will be trained using a combination of a genetic algorithm and variational inference, and we show the calculations needed to implement a mean-field and a flow-based approximate posterior of the parameters. In the next chapter we present results.

In order to evaluate the capabilities of a variational inference approach applied to BGNLM, we have developed a Python library (**GitHub link**). Here, the implemented methods for training and testing a BGNLM with mean-field approximations or with a flow-based approximations can be found.

3.1 The genetic algorithm

To explore the vast space of different features in a BGNLM, we use a genetic algorithm similar to that found in Hubin, Storvik and Frommlet (2021).

The algorithm begins with all p covariates as features, making the first generation equivalent to a Bayesian generalized linear model (1b). We refer to this first population as $\mathcal{S}_0$. For subsequent populations after $\mathcal{S}_0$, we allow the size of each population, q^*, to be greater than p, which speeds up the exploration of different features.

The process of choosing which features should be part of the next generation is done in two steps. First, we estimate the marginal inclusion probabilities and retain all members of $\mathcal{S}_t$ with an inclusion probability above some threshold ρ_{del}. Members with inclusion probability less than ρ_{del} are retained with a low probability proportional to their marginal inclusion probability. The features that are removed are replaced with new features generated randomly by projection, modification, multiplication through (3), or by a selecting a input variable. The generated features that are already present or linearly dependent with a feature in $\mathcal{S}_t$ will not be included in the next population, $\mathcal{S}_{t+1}$. Thus, the members of the next population are all the high probability features from $\mathcal{S}_t$ plus the newly generated features.

When replacing features, the replacement is generated randomly by the projection transformation with probability P_{pr}, modification transformation with probability P_{mo}, or multiplication transformation with probability P_{mu}, or by an input variable with probability P_{in}, where $P_{pr}+P_{mo}+P_{mu}+P_{in} = 1$. This implementation allows us to exclude a certain transformation by setting P_{pr}, P_{mu}, P_{mo}, or P_{in} to zero. If a projection or a modification is added, a nonlinearity is chosen from $\mathcal{G}$ with probabilities $P_{\mathcal{G}}$.

Algorithm 2 GMSVI

```
Require: $\mathcal{S}_0$, $\rho_{del}$, T
  Estimate the marginal inclusion probabilities, $\boldsymbol{\kappa}$, using VI on $\mathcal{S}_0$
  Evaluate the loss of $\mathcal{S}_0$
  Replace features $F_k : \kappa_k > \rho_{del}$, but keep with some probability.
  Generate extra features to obtain $\mathcal{S}_1$
  for $t = 1, .., T$ do
      Estimate $\boldsymbol{\kappa}$ using VI on $\mathcal{S}_t$
      Evaluate the loss of $\mathcal{S}_t$
      if $t < T$ then
          Replace features $F_k : \kappa_k > \rho_{del}$ to obtain $\mathcal{S}_{t+1}$
      end if
  end for
  Make inference on model in generation with the smallest loss
```

Within each population there are two main questions: what features in the population are relevant for prediction, and how good are the predictions made by the current population (evaluated on a validation set). The validation loss of every generation is recorded, and in the end inference is done on the best generation.

3.2 The model in each generation

In the genetic algorithm explained above, we essentially fix the a subset of features in each generation $t \in 1, ..., T$ with a population of features

$\mathcal{S}_t = \{F_{t_1}, ..., F_{t_{q^*}}\} \subseteq \{F_1, ..., F_q\}$. This makes each population equal to a Bayesian GLM with $F_{t_1}, ..., F_{t_{q^*}}$ acting as covariates. We will, for ease of notation, denote $\mathbf{v}_j$ as the values of feature F_j.

In practice, we use the $\mathbf{v}_j$'s as inputs to a Bayesian neural network with one hidden layer and aim to estimate the posterior weights in the hidden layer, $\boldsymbol{\beta} = (\beta_{t_1}, ..., \beta_{t_{q^*}})$. We let binary model vector, denoted $\boldsymbol{\gamma} = (\gamma_{t_1}, ..., \gamma_{t_{q^*}})$, be independent Bernoulli random variables with the probability of success parameters $\boldsymbol{\kappa} = (\kappa_{t_1}, ..., \kappa_{t_{q^*}})$.

The hidden layer of our neural network consists of a linear predictor. This linear predictor of the current population will be denoted as $\mathbf{u}$ and takes the following form

$$u_i = b + \sum_{j=1}^{q^*} v_{ij}\gamma_j\beta_j, \qquad i = 1, ..., n,$$

where n is the number of observations in the training set.

This model implementation is essentially a simplified version of the LBBNN from Skaaret-Lund, Hubin and Storvik (2023), where we use a single hidden layer with n nodes. We will make use of the calculations provided in their paper throughout this section.

Bayesian Inference

One of the one main motivations behind Bayesian methods is that we are able to take into account parameter and model uncertainty. By doing inference through a posterior predictive distribution, we will average over all possible parameters $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$.

For a new observation, $\tilde{\mathbf{y}}$, and given data, $\mathcal{D}$, the predictive distribution is

$$p(\tilde{\mathbf{y}}|\mathcal{D}) = \sum_{\gamma} \int_{\beta} p(\tilde{\mathbf{y}}|\boldsymbol{\beta}, \boldsymbol{\gamma}, \mathcal{D}) p(\boldsymbol{\beta}, \boldsymbol{\gamma}|\mathcal{D}) d\boldsymbol{\beta}.$$

However, since the posterior $p(\boldsymbol{\beta}, \boldsymbol{\gamma}|\mathcal{D})$ is intractable, we replace this with an approximation $q_{\boldsymbol{\theta}}(\boldsymbol{\beta}, \boldsymbol{\gamma})$, with $\boldsymbol{\theta}$ denoting the parameters of the approximation.

The mean-field approximation

The first and simplest option for approximate posterior is the mean-field Gaussian. It is commonly used in Bayesian neural networks, and is usually defined over the weights. In an arbitrary generation, this becomes

$$q_{\boldsymbol{\theta}}(\boldsymbol{\beta}) = \prod_{j=1}^{q^*} \mathcal{N}(\tilde{\mu}_j, \tilde{\sigma}_j^2).$$

where the goal is to estimate $\tilde{\mu}_j$ and $\tilde{\sigma}_j$ for all j.

This is extended to include binary inclusion variables in Hubin and Storvik (2019):

$$\begin{aligned} q_{\boldsymbol{\theta}}(\boldsymbol{\beta}|\boldsymbol{\gamma}) &= \prod_{j=1}^{q^*} \Big[\gamma_j \mathcal{N}(\tilde{\mu}_j, \tilde{\sigma}_j^2) + (1-\gamma_j)\delta(\beta_j)\Big], \\ q_{\tilde{\kappa}_j}(\gamma_j) &= \text{Bernoulli}(\tilde{\kappa}_j), \end{aligned}$$

where $\delta(\cdot)$ is Dirac's delta function with zero mass everywhere except at zero where it has a "spike".

However, this distribution will typically not be flexible enough to approximate the true posterior, and the need a more flexible distribution is apparent. This is why we continue to follow Skaaret-Lund, Hubin and Storvik (2023) and adopt the Multiplicative Normalizing Flow (Louizos and Welling 2017, MNF).

The flow approximation

We now introduce a latent variational distribution $q_{\boldsymbol{\omega}}(\mathbf{z})$ in order to model dependencies. Here, $\boldsymbol{\omega} = \{\boldsymbol{\phi}, \boldsymbol{\psi}\}$, where $\boldsymbol{\phi}$ and $\boldsymbol{\psi}$ are the parameters of the flow and the base distribution, respectively. For a illustration of the difference between the mean-field and flow model, see Figure 3.1.

The variational posterior of the flow models is given

$$\begin{aligned} q_{\boldsymbol{\theta}}(\boldsymbol{\beta}|\boldsymbol{\gamma}) &= \prod_{j=1}^{q^*} \Big[\gamma_j \mathcal{N}(z_j\tilde{\mu}_j, \tilde{\sigma}_j^2) + (1-\gamma_j)\delta(\beta_j)\Big], \\ q_{\tilde{\kappa}_j}(\gamma_j) &= \text{Bernoulli}(\tilde{\kappa}_j). \end{aligned}$$

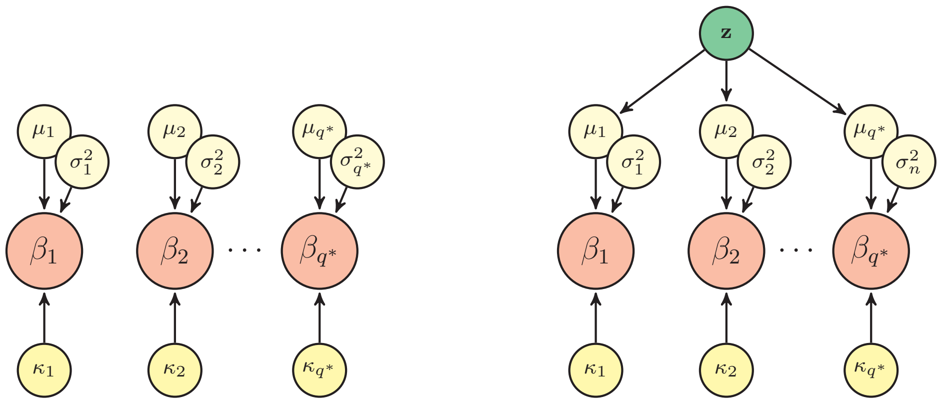


Figure 3.1: Illustration of difference between mean-field and flow based approximate posterior.

This is a simplified version of Skaaret-Lund, Hubin and Storvik (2023), which is again similar to MNF from Louizos and Welling (2017). The main difference from MNF is the introduction of the binary variable, $\boldsymbol{\gamma}$. We will transform the variational posterior with normalizing flows by applying transformations to the weights through a latent $\mathbf{z}$.

The flow is following the inverse autoregressive flow (Kingma, Salimans, Jozefowicz et al. 2017, (11)) with numerically stable updates. The chain of flows is initialized by $\mathbf{z}_0$ which is derived from the input features, and each flow component, f_{ϕ_k}, is a MADE (Germain et al. 2015, (10)), to retain an autoregressive structure.

Following this setup, we will sample a $\mathbf{z} = \mathbf{z}_K$ from the last iterate of the flow, which log density is given as

$$\log q_{\boldsymbol{\omega}}(\mathbf{z}) = \log q_{\psi}(\mathbf{z}_0) - \sum_{k=1}^{K} \log \left| \det \frac{\partial f_{\phi_k}}{\partial z_{k-1}} \right|,$$

where the base, $q_{\psi}(\mathbf{z}_0)$, is another Gaussian distribution.

Calculation of the KL-divergence

For the mean-field model, we are comparing two normal densities, leading to a relatively straightforward computation of the KL-divergence:

$$\mathrm{KL}(q_{\boldsymbol{\theta}}(\boldsymbol{\beta},\boldsymbol{\gamma})||p(\boldsymbol{\beta},\boldsymbol{\gamma})) = \sum_{j=1}^{q^*}\Big[\tilde{\kappa}_j\Big(\log\frac{\sigma_j}{\tilde{\sigma}_j} - \frac{1}{2} + \log\frac{\tilde{\kappa}_j}{\kappa_j} + \frac{\tilde{\sigma}_j^2 + (\tilde{\mu}_j - \mu_j)^2}{2\sigma_j^2}\Big) \\ + (1-\tilde{\kappa}_j)\log\frac{1-\tilde{\kappa}_j}{1-\kappa_j}\Big],$$

where σ_j, κ_j and μ_j are samples from their respective priors.

For the flow model, things get a bit more complicated. First, we need to marginalize out $\mathbf{z}$ from the joint posterior.

$$q_{\boldsymbol{\theta}}(\boldsymbol{\beta},\boldsymbol{\gamma}) = \int q(\boldsymbol{\beta},\boldsymbol{\gamma},\mathbf{z})d\mathbf{z}.$$

However, as this is not tractable, we will use the approximation found in Skaaret-Lund, Hubin and Storvik (2023), first suggested by Louizos and Welling (2017). They start by turning it into a log density:

$$\log q_{\boldsymbol{\theta}}(\boldsymbol{\beta},\boldsymbol{\gamma}) = \log q_{\boldsymbol{\theta}}(\boldsymbol{\beta},\boldsymbol{\gamma}|\mathbf{z}) + \log q_{\boldsymbol{\omega}}(\mathbf{z}) - \log q_{\boldsymbol{\omega}}(\mathbf{z}|\boldsymbol{\beta},\boldsymbol{\gamma}),$$

leading to the following expression for the KL-divergence

$$\mathrm{KL}[q_{\boldsymbol{\theta}}(\boldsymbol{\beta},\boldsymbol{\gamma})||p(\boldsymbol{\beta},\boldsymbol{\gamma})] = \mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{\beta},\boldsymbol{\gamma},\mathbf{z})}\Big[\mathrm{KL}[q_{\boldsymbol{\theta}}(\boldsymbol{\beta},\boldsymbol{\gamma},\mathbf{z})||p(\boldsymbol{\beta},\boldsymbol{\gamma})] \\ + \log q_{\boldsymbol{\omega}}(\mathbf{z}) - \log q_{\boldsymbol{\omega}}(\mathbf{z}|\boldsymbol{\beta},\boldsymbol{\gamma})\Big].$$

Since $q_{\boldsymbol{\omega}}(\mathbf{z}|\boldsymbol{\beta},\boldsymbol{\gamma})$ is intractable and difficult to compute numerically, we will also follow Ranganath, Tran and Blei (2016) and introduce an auxilliary distribution $r(\mathbf{z}|\boldsymbol{\beta},\boldsymbol{\gamma})$ to reach the following upper bound on the KL divergence:

$$\mathrm{KL}[q_{\boldsymbol{\theta}}(\boldsymbol{\beta},\boldsymbol{\gamma})||p(\boldsymbol{\beta},\boldsymbol{\gamma})] \leq \mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{\beta},\boldsymbol{\gamma},\mathbf{z})}\Big[\mathrm{KL}[q_{\boldsymbol{\theta}}(\boldsymbol{\beta},\boldsymbol{\gamma}|\mathbf{z})||p(\boldsymbol{\beta},\boldsymbol{\gamma})] \\ + \log q_{\boldsymbol{\omega}}(\mathbf{z}) - \log r(\mathbf{z}|\boldsymbol{\beta},\boldsymbol{\gamma})\Big], \tag{12}$$

where we have that

$$\mathrm{KL}(q_{\boldsymbol{\theta}}(\boldsymbol{\beta},\boldsymbol{\gamma})||p(\boldsymbol{\beta},\boldsymbol{\gamma})) = \sum_{j=1}^{q^*}\Big[\tilde{\kappa}_j\Big(\log\frac{\sigma_j}{\tilde{\sigma}_j} - \frac{1}{2} + \log\frac{\tilde{\kappa}_j}{\kappa_j} + \frac{\tilde{\sigma}_j^2 + (\tilde{\mu}_j z_j - \mu_j)^2}{2\sigma_j^2}\Big) + (1-\tilde{\kappa}_j)\log\frac{1-\tilde{\kappa}_j}{1-\kappa_j}\Big]$$

and

$$\log q_{\boldsymbol{\omega}}(\mathbf{z}) = \log q_{\psi}(\mathbf{z}_0) - \sum_{k=1}^{K}\log\left|\det\frac{\partial f_{\phi_k}}{\partial \mathbf{z}_{k-1}}\right|.$$

For the last term, $r(\mathbf{z}|\boldsymbol{\beta},\boldsymbol{\gamma})$, we will use an inverse normalizing flow to make this distribution flexible. We will utilize a setup leading to a Gaussian distribution:

$$r(\mathbf{z}_B|\boldsymbol{\beta},\boldsymbol{\gamma}) = \prod_{j=1}^{q^*}\mathcal{N}(\nu_j,\tau_j^2),$$

where the dependence on $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ is defined similarly to Skaaret-Lund, Hubin and Storvik (2023):

$$\begin{aligned}\boldsymbol{\nu} &= \mathbf{d}_1 \odot \tanh(\mathbf{e}^T(\boldsymbol{\beta}\odot\boldsymbol{\gamma})),\\ \log\boldsymbol{\tau}^2 &= \mathbf{d}_2 \odot \tanh(\mathbf{e}^T(\boldsymbol{\beta}\odot\boldsymbol{\gamma})).\end{aligned}$$

Here, $\mathbf{d}_1$, $\mathbf{d}_2$ and $\mathbf{e}$ are trainable parameters with dimensionality q^*, $\odot$ denotes the elementwise multiplication and tanh is the hyperbolic tangent function.

This implementation hence requires that we must use two normalizing flows; one to get from $\mathbf{z}_0$ to $\mathbf{z} = \mathbf{z}_K$ and one to get from $\mathbf{z} = \mathbf{z}_K$ to $\mathbf{z}_B$. Finally, for the last term in (12) we get

$$\log r(\mathbf{z}|\boldsymbol{\beta},\boldsymbol{\gamma}) = \log r(\mathbf{z}_B|\boldsymbol{\beta},\boldsymbol{\gamma}) + \sum_{t=K}^{B}\log\left|\det\frac{\partial \mathbf{z}_t}{\partial \mathbf{z}_{t-1}}\right|.$$

For the biases, we assume that they are independent of the weights and of each other and use the standard normal prior with a mean-field Gaussian approximate posterior for both models. The KL-divergence can be calculated as

$$\mathrm{KL}(q(b)||p(b)) = \log \frac{\sigma_b}{\tilde{\sigma}_b} - \frac{1}{2} + \frac{\tilde{\sigma}_b^2 + (\tilde{\mu}_b - \mu_b)^2}{2\sigma_b^2},$$

where σ_b and μ_b are samples from their respective priors.

The local reparametrization trick

One downside of our novel implementation is that each optimization step in training requires sampling of all γ_j's and β_j's, to compute the linear predictor $\mathbf{u}$. Due to the binary nature of the γ_j's, we will also need a continuous relaxation. In Hubin and Storvik (2019), relaxations of γ_js has been proposed trough the Concrete distribution (Maddison, Mnih and Teh 2016), which was replaced with the local reparametrization trick (LRT) in Skaaret-Lund, Hubin and Storvik (2023). LRT is more computationally attractive, but it is important to note that this is also an approximation.

The general idea behind the LRT is that if we have a sum of independent Gaussian random variables, the sum will also be Gaussian. In our case, we have a mixture of independent Gaussians, but the central limit theorem still holds, as long as Lindeberg's condition (Lindeberg 1922) is satisfied.

Sampling with LRT corresponds to drawing a random sample ϵ_i from the standard normal and approximating u_i through

$$u_i = \mathbb{E}(u_i) + \sqrt{\mathrm{Var}(u_i)} \cdot \epsilon_i, \quad \text{for} i = 1, ..., n.$$

In our mean-field approximation, we can compute this means and variances as

$$\begin{aligned}\mathbb{E}(u_i) &= \mathbb{E}\Big[b + \sum_j v_j \gamma_j \beta_j\Big] = \tilde{\mu}_b + \sum_j v_{ij} \tilde{\kappa}_j \tilde{\mu}_j,\\ \mathrm{Var}(u_i) &= \mathrm{Var}\Big[b + \sum_j v_{ij} \gamma_j \beta_j\Big] = \tilde{\sigma}_b^2 + \sum_j v_{ij}^2 \tilde{\kappa}_j \Big(\tilde{\sigma}_j^2 + (1 - \tilde{\kappa}_j)\tilde{\mu}_j^2\Big).\end{aligned}$$

For the flow, the mean and the variance of u_i can be computed as

$$\mathbb{E}(u_i) = \tilde{\mu}_b + \sum_j v_{ij} \tilde{\kappa}_j \tilde{\mu}_j z_j,$$

$$\mathrm{Var}(u_i) = \tilde{\sigma}_b^2 + \sum_j v_{ij}^2 \tilde{\kappa}_j \Big(\tilde{\sigma}_j^2 + (1 - \tilde{\kappa}_j) \tilde{\mu}_j^2 z_j \Big).$$

Here, it is important to note that $\mathbf{z}$ affects both the mean and variance of the approximation. In Louizos and Welling (2017) it only influences the mean.

Algorithm 3 SVI

```
Require: $\mathbf{v}, \mathbf{y}_{train}$, $\boldsymbol{\mu}$, $\boldsymbol{\sigma}$, $\boldsymbol{\kappa}$, $\mu_b$, $\sigma_b$.
  $\tilde{\boldsymbol{\mu}}, \tilde{\boldsymbol{\sigma}}, \tilde{\boldsymbol{\kappa}}, \tilde{\mu}_b, \tilde{\sigma}_b \leftarrow$ Initialize.
  for $i = 1, .., $epochs do
      for $j = 1, ..., $batches do
          $\mathbf{v}^*, \mathbf{y}^*_{train} \leftarrow$ Sample a batch.
          $\mathbf{u}^* \leftarrow$ Sample linear predictor
          Evaluate the loss of the linear predictor.
          Calculate the gradient of the KL-divergence w.r.t $\tilde{\mu}, \tilde{\sigma}, \tilde{\kappa}, \tilde{\mu}_b, \tilde{\sigma}_b$.
          Update hyperparameters using Adam.
      end for
  end for
```

3.3 Considerations

Before presenting the results, we must discuss some considerations that are to be made when training a BGNLM.

First, it is important to consider which nonlinear functions should be part of $\mathcal{G}$. Composing a diverse set of transformations will allow for exploration of lavish models. However, as we have used the same datasets as Hubin, Storvik and Frommlet (2021), we mainly used the same nonlinear functions.

Second, it is important to consider whether or not to include all transformations (3). Including projections create more flexible models, but will increase training time due to the additional optimization of the weight parameters within each projection, $\boldsymbol{\alpha}$. Excluding projections also puts our model into a fully Bayesian framework, which can be desirable.

Third, we must choose how many generations to run. If we train for too many generations the algorithm will be very time consuming, and if we use too few generations we might not be able too consider some important features.

Fourth, we must choose how many features to include in each generation. We found that it was beneficial to include as many features as possible as this will allow for fast exploration of different features. However, having too many features will possibly make feature generation time consuming as we require that the features in each population are not collinear.

In addition to these, there are some important considerations to be made regarding the algorithmic properties.

Scaling/normalizing

We will in regard to scaling/normalizing try to follow existing literature. However, there are a lot to say about scaling/normalizing input data in neural networks. We will therefore merely present our approach without to much discussion, and refer to the literature for explanations.

Bottou, Curtis and Nocedal (2018) highlights that first-order optimization methods are not scale invariant. In practical terms, this means that if the input variables are not scaled properly, it may lead to very noisy gradients causing optimization to be slow or even fail to converge. Thus, we must specify a scaling strategy.

In a simple linear regression, it can be shown (Lecun et al. 1998) that the Hessian of the mean squared error with respect to the regression parameters, is exactly the covariance matrix of the input data. This means that if the Hessian is close to the identity, the optimization problem becomes much easier (Lecun et al. 1998). Therefore, one might desire that the input data has covariance matrix equal to the identity, and the process for doing this is known as *data whitening*. However, as this is often costly and might even lead to poor performance (Wadia et al. 2021), it is common practice in neural networks to *standardize* the input variables to have zero mean and unit standard deviation. This will at least give some benefits of identity covariance, and enhance the performance of first-order methods such as Adam.

The most natural choice in our implementation is hence to normalize the features to have mean equal to zero and standard deviation equal to one, and we will mainly use this strategy. That is, we normalize all the original covariates, $\mathbf{x}_j$, as well as all the transformed values $\mathbf{v}_j$ in each generation. This strategy corresponds to what is known as Layer Normalization (Ba, Kiros and Hinton 2016) in neural networks. Another prominent strategy known as Batch Normalization (Ioffe and Szegedy 2015) could also be considered. However, this is more costly and are shown to generally not improve performance in fully connected neural networks such as ours (Ba, Kiros and Hinton 2016).

It is important to note that this approach may cause bad performance. If the covariates, $\mathbf{x}$, are normalized before transforming through a nonlinear function, and the outputs of the nonlinear functions are also normalized, we may loose important information along the way.

Tuning parameters/initialization

There are a lot of tuning parameters in this model that can be changed in order to enhance performance. We must consider batch sizes, epochs, learning rate- and parameter initialization, prior parameters, etc. If we are using the flow-based approximate posterior, we must also consider number of flow components and the depth and width of each component.

To be clear, careful considerations was given to these things when testing our implementation, and a lot of tuning was done. Hence, we will present the different settings used in different applications when providing the results in the next chapter.

However, some things are unchanged throughout the next chapter. Namely, the prior for $\boldsymbol{\beta}$, and the initialization of $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ and b. The prior for β_j, $j = 1, ..., q^*$ is the standard normal in all settings, and for initialization we used, for $j = 1, ..., q^*$:

$$
\begin{aligned}
&\tilde{\mu_j} \sim \text{uniform}(-0.01, 0.01), \\
&\tilde{\sigma_j} = \log(\rho_j + 1), \quad \text{where } \rho_j \sim \text{uniform}(-5, -4), \\
&\tilde{\kappa_j} = \text{sigmoid}(\lambda_j), \quad \text{where } \lambda_j \sim \text{uniform}(1.5, 2.5), \\
&\tilde{\mu_b} \sim \text{uniform}(-0.01, 0.01), \\
&\tilde{\sigma_b} = \log(\rho_b + 1), \quad \text{where } \rho_b \sim \text{uniform}(-5, -4).
\end{aligned}
$$

As parameter initialization can have great influence on the results, we must admit that this could need some more attention. It will however be left to future work.

CHAPTER 4

Applications and Results

4.1 Simulation Studies

In testing of our novel algorithms, we will start with three simulation studies. We will here mainly be interested in variable selection and model detection. That is, we will be concerned with the accuracy of our approximate posteriors rather than accuracy of predictions.

In all studies, we test both algorithms using $N = 100$ simulation runs. To evaluate the performance, we report estimates for the power (Pow), the false discovery rate (FPR) and the estimated number of false positives (FP). These measures are defined as follows:

$$\text{Pow} = \frac{1}{N}\sum_{l=1}^{N} \mathrm{I}(\hat{\gamma}_{j*}^{l} = 1); \quad \text{FDR} = \frac{1}{N}\sum_{l=1}^{N} \frac{\sum_j \mathrm{I}(\gamma_j = 0, \hat{\gamma}_j^l = 1)}{\sum_j \mathrm{I}(\hat{\gamma}_j^l = 1)},$$

$$\text{FP} = \frac{1}{N}\sum_{l=1}^{N}\sum_{j \neq j*} \mathrm{I}(\hat{\gamma}_j^l = 1).$$

Here, $\hat{\gamma}_j^l$ is used to denote if the algorithm chose to include variable j. We here used a threshold of 0.3, meaning that all variables with marginal inclusion probability above 0.3 are regarded as positives.

Independent Normal Data with Varying Noise

The model in the first two studies is a simple Bayesian GLM with normal response and identity link function. The goal is here to evaluate our novel algorithms in their ability to perform variable selection. The model is defined as

$$\mathbf{X} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}),$$
$$\mathbf{y} \sim \mathcal{N}(\mathbf{X}\boldsymbol{\beta}^T, \sigma^2\mathbf{I}).$$

We used a fixed $\boldsymbol{\beta}$:

$$\boldsymbol{\beta} = (0, 0, 0, 0, 0, 1.5, -4, 3, -0.2, 1, 0, 0, 0, 0, 0, -2, 1.3, 0.3, -0.8, 3),$$

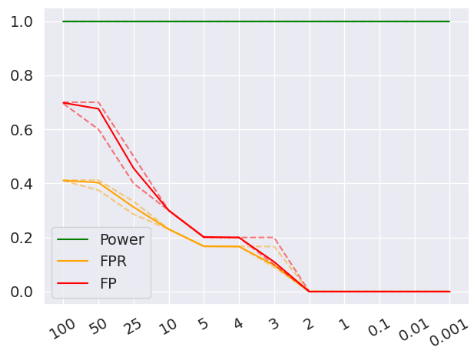
and let the noise, σ^2, vary. $\mathbf{X}$ is here simulated from the standard normal with $n = 15,000$ observations and $p = 20$ covariates.

The goal was then to determine which $\mathbf{x}_j$ corresponded to a nonzero coefficient β_j. That is, for this $\boldsymbol{\beta}$, we were trying to evaluate how well our algorithms did in estimating $\boldsymbol{\gamma}$:

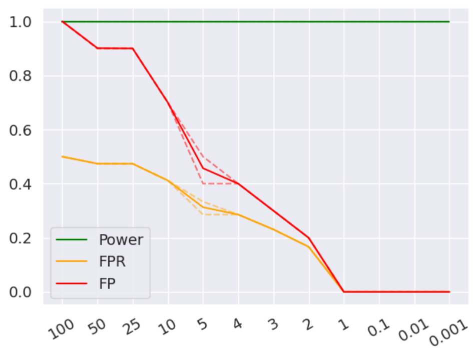
$$\boldsymbol{\gamma} = (0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1).$$

In addition, we were interested in finding credible intervals for $\boldsymbol{\beta}$ to see if they contain the true values. The latter can be found in the Appendix.

For the model prior, we used $\kappa_j = e^{-2} \approx 0.135$ for all j. For both algorithms, we used a batch size of 1000 and 300 epochs. The flow-based approximate posterior was computed using $K = 2$ components with 2 hidden layers of size 75 for each component.



(a) Flow



(b) Mean-field

Figure 4.1: Pow, FP and FPR for mean-field and flow-based approximate posterior for different values of σ^2. The dashed lines represents the best and worst values between the 100 runs.

The figure shows that both models did well in terms of model recovery in this rather simple study. The model with flow-based approximate posterior was however able to recover the model for higher noise level, and with overall fewer false positives. It is also important to note that both has perfect power for all noise levels, indicating that our models included all the correct variables throughout.

Correlated data

For our next simulation study, we used the same GLM as in the first study. However, we now fixed the noise at $\sigma^2 = 1$. To make it harder for our algorithms to separate between which covariates that contributed to the response, we made one of the "false" covariates, depend on one of the "true" covariates. To be specific, we defined a dependence of $\mathbf{x}_6$ on $\mathbf{x}_3$ in the following way:

$$\mathbf{x}_3 = \alpha\mathbf{x}_6 + (1-\alpha)\mathbf{x}_3,$$

for $0 < \alpha < 1$.

Since both algorithms were able to detect the true model for $\sigma^2 = 1$ in the previous study, we expected they would do it again in this study. Indeed, the power remained constant at 1 as before, and we only report the expected number of false positives (FP). As expected, all false positives now came from a false detection of $\mathbf{x}_3$. This is why the scale of the y-axis is changed in Figure 4.2 (y-axis here goes from 0 to 0.1).

For both algorithms, we used a batch size of 1000 and 300 epochs. Here, we tried different Ks for the flow approximate posterior to see if more components improved the results. However, results were similar for $K = 5$ and $K = 10$ and $K = 50$, and we only show results for $K = 2$ components with 2 hidden layers of size 75 for each component.

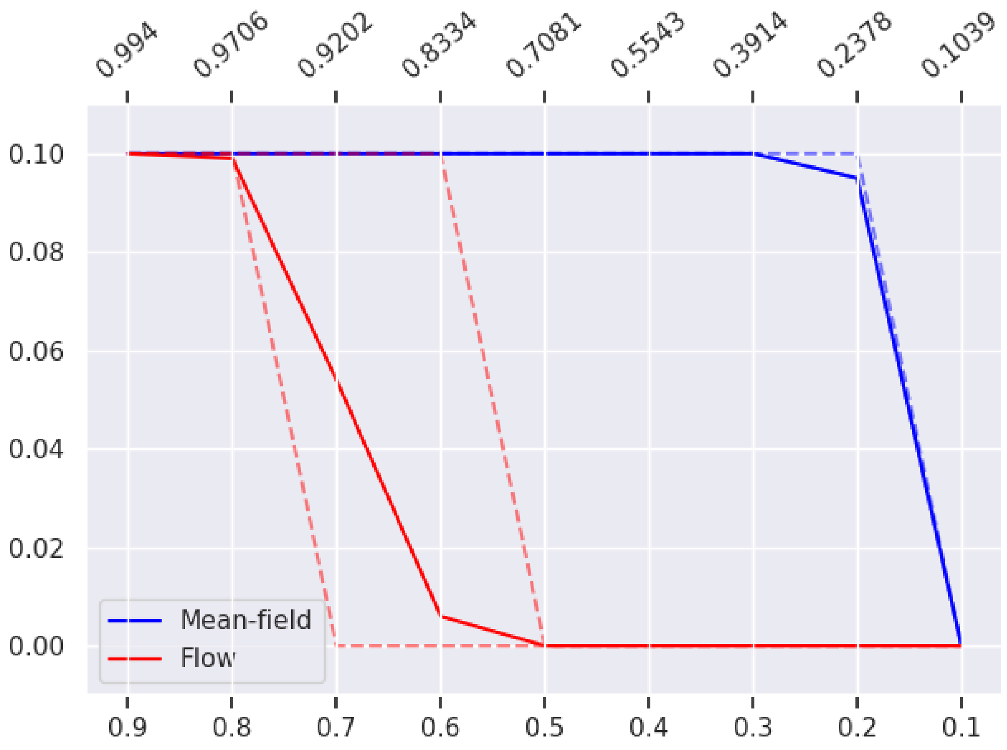


Figure 4.2: False Positives for correlated data. Upper x-axis is Pearson's correlation coefficient between $\mathbf{x}_3$ and $\mathbf{x}_6$ for different values of α (lower x-axis). Dashed lines represents the best and worst of the 100 runs.

As is evident from Figure 4.2, the mean-field approximate posterior implementation did not do a very good job of excluding $\mathbf{x}_3$ here, and was only able to detect the true model for a rather low correlation ($\alpha = 0.1$). However, as it is based on an assumption of independent covariates, this result is not very surprising.

In contrast, the results for the algorithm with flow-based approximate posterior are rather uplifting. It was able to detect the true model almost every time for $\alpha = 0.6$ and almost half the time for $\alpha = 0.7$. This study indicates that our flow-based approximations are working as intended, as we are able to perform variable selection even though two variables are correlated.

ART Study

In further evaluating our algorithms, we will take advantage of the simulation design from the ART study, which was created to assess fractional polynomials models using a large breast cancer dataset (Schmoor, Olschewski and Schumacher 1996). The ART study includes six continuous predictors $(x_1, x_3, x_5, x_6, x_7, x_{10})$ and four categorical predictors (x_2, x_4, x_8, x_9). The categorical predictors consist of an ordered three-level variable (x_4), an unordered three-level variable (x_9), and two binary variables (x_2 and x_8).

The correlation structure and predictor distribution in the ART study are considered realistic and accurate, and Royston and Sauerbrei (2008) provides further details on the design. For comparison, we use the frequentist multivariate fractional polynomials (MFP), which was fitted with the R package `mfp` (Heinze, Ambler and Benner 2022) and GMJMCMC modified to include fractional polynomials (Hubin and De Bin 2022). The values in Figure 4.3 are borrowed from their (unpublished/submitted for publication) paper. The inclusion of this study is also inspired by their work. We ran all methods 100 times, and report the median result.

The model used to compute the response in the original simulation study is given by

$$y = x_1^{0.5} + x_1 + x_3 + x_{4a} + x_5^{-0.2} + \log(x_6 + 1) + x_8 + x_{10} + \epsilon,$$

where x_{4a} represents the second level of the categorical predictor x_4, and $\epsilon \sim \mathcal{N}(0, \sigma^2)$. The instances used in the study are available online.

However, at noted in Hubin and De Bin (2022), this model is limited in its ability to fully investigate the properties of the algorithms being evaluated. For instance, the model misspecification of $x_5^{-0.2}$ makes it impossible to evaluate how frequently the algorithm selects the true model.

The model is therefore modified by introducing a fractional polynomial effect of -1 for x_5 and changing the effect of x_3 from linear to a fractional polynomial of degree 2 with powers of -0.5. This is done to make the search for the true model more challenging. The new model is given by

$$y = |x_1|^{0.5} + x_1 + |x_3|^{-0.5} + |x_3|^{-0.5}\log(|x_3| + \varepsilon) + x_{4a} + x_5^{-1} + \log(|x_6| + \varepsilon) + x_8 + x_{10} + \epsilon,$$

where again x_{4a} is the second level of x_4 and $\epsilon \sim \mathcal{N}(0, \sigma^2)$. Here, we used a small number, $\varepsilon = 10^{-5}$, for numerical stability.

The definition of fractional polynomials allows for transformations that belong to either $\mathcal{G}_0 = \{x\}$, $\mathcal{G}_1 = \{x^{-2}, x^{-1}, x^{0.5}, \log x, x^{0.5}, x^3\}$ or $\mathcal{G}_2 =$

$\{x^{-2}\log x, x^{-1}\log x, x^{-0.5}\log x, \log x\log x, x^{0.5}\log x, x\log x, x^2\log x, x^3\log x\}$.

We did not allow for interactions or projections, or modifications of already transformed variables. To be more specific, we set $P_{mu} = 0$, $P_{pr} = 0$, $P_{mo} = 1/2$ and $P_{in} = 1/2$. If, at a given time, the algorithm chose modification, it was only allowed to modify the features that were not already modifications. A transformation from either $\mathcal{G}_0$, $\mathcal{G}_1$ or $\mathcal{G}_2$ (with $P_{\mathcal{G}_0} = P_{\mathcal{G}_1} = P_{\mathcal{G}_2} = 1/3$) was then selected.

For the model prior, κ_k was chosen to be $\kappa_k = \exp(-\log n)$ for $k : F_k \in \mathcal{G}_0$, $\kappa_k = \exp(-(1+\log 2)\log n)$ for $k : F_k \in \mathcal{G}_1$ and $\kappa_k = \exp(-(1+\log 4)\log n)$ for $k : F_k \in \mathcal{G}_2$. Hence, this prior heavily penalizes the depth of features.

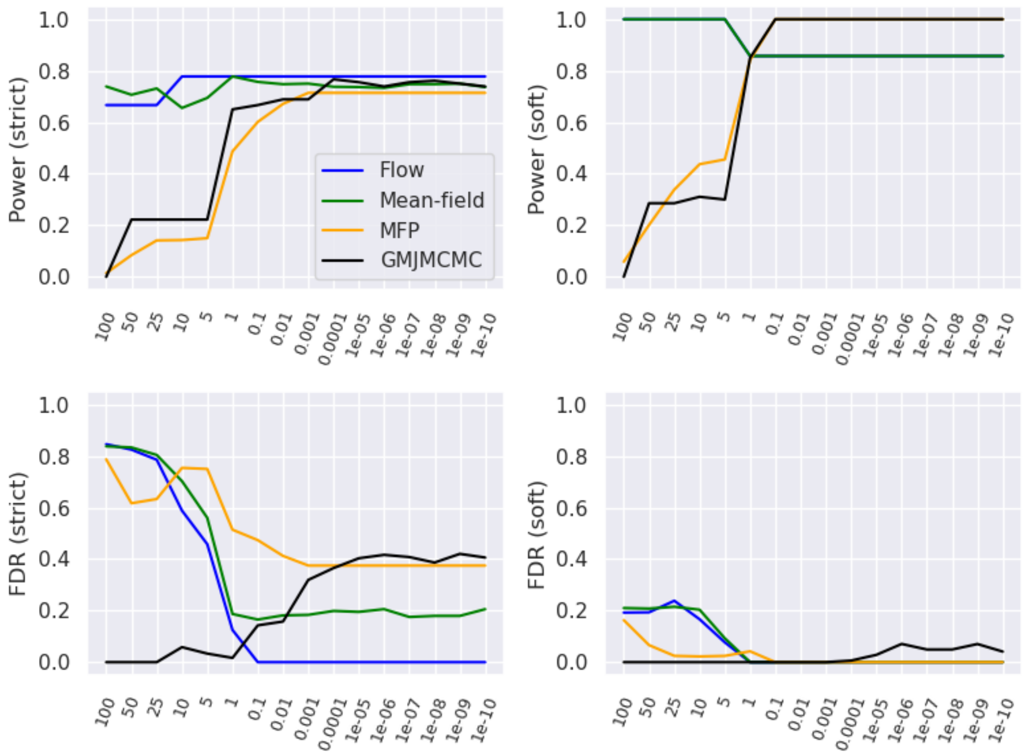


Figure 4.3: FDR and power on art study with varying noise for different algorithms. Plotted values are the median results of 100 runs.

This time, the performance metrics were slightly modified. We still report power and false positive rate, but they are now separated into two categories:

strict and soft. Strict is used to indicate whether or not the algorithms were able to detect the actual fractional polynomial used to simulate y, with right variable and right transformations, while soft indicates if the correct variable x was detected, but not the correct transformation.

From Figure 4.3 it is interesting to note the difference between our implementations and GMJMCMC. For higher noise, GMJMCMC does not include any variables (low power and low FPR), while BGNLM_FLOW and BGNLM_MF includes too many variables (high power and high FPR). For less noise, GMJMCMC were able to detect the right variables but included the incorrect transformations, while BGNLM_FLOW included most of the right variables, with right transformations. Overall, both BGNLM_FLOW and BGNLM_MF are doing quite well in this study; strict power is high, and strict FDR is low as desired. In fact, BGNLM_FLOW rarely had a false detection for lower noise.

The only obvious weakness of our implementations is (soft) power for less noise. After inspecting the results, we found that this is due to the inability to detect x_3 as a true variable. For higher noise, we were unable to dislike anything, resulting in a lot of false positives. For less noise, we filtered out the false variables and transformations, but were unable to retain x_3 as a positive. We suspect that this is a side effect of our normalization strategy. Since we are normalizing variables before and after transforming, it leads to a weakened signal coming from the transformations of x_3 when generating the response.

4.2 Real Data Applications

We will in this section apply our novel algorithms on real world datasets. Here, we use the same datasets as in Hubin, Storvik and Frommlet (2021). This allows us to compare our variational approximate posteriors with posteriors computed by GMJMCMC, and evaluate whether or not our methods are able to get similar results.

Binary Classification

For binary classifications, we consider a BGNLM (1b) with response variable coming from a Bernoulli distribution. We use the logit link function and dispersion parameter $\phi = 1$. This is hence comparable to classical logistic regression. We use the model prior corresponding to (4) with $a = e^{-2}$ and $\mathrm{c}(F_j(\cdot,\cdot)) = \log(oc_j + 1) - \varepsilon$, for $j = 1, ..., q^*$. By using this prior, we practically force all original covariates to be in the model by assigning them a prior probability close to 1 ($oc = 0$).

Predictions are made by applying the logit function to the linear predictor:

$$\hat{y}_i^* = \mathrm{I}(\frac{1}{1+e^{u_i}} \geq 0.5),$$

where u_i is sampled directly using the LRT described in the previous chapter.

We here used, in both applications, nonlinear functions form $\mathcal{G} = \{\mathrm{gauss}(x), \mathrm{sigmoid}(x), \sin(x), \cos(x), \tanh(x), \tan^{-1}(x), \log(|x| + \varepsilon), \exp(x), |x|^{7/2}, |x|^{5/2}, |x|^{1/3}\}$, selected with uniform probability. In addition, we allowed for both multiplications and projections and let $P_{mu} = P_{mo} = P_{pr} = P_{new} = 1/4$.

We compare our implementations with various classification algorithms: tree-based (TXGBOOST) and linear gradient boosting (LXGBOOST), penalized likelihood (LASSO and (RIDGE), deep fully connected neural networks (DEEPNETS), random forests (RFOREST), naive Bayes classifier (NBAYES), logistic regression (LR) and GMJMCMC (BGNLM and BGNLM_PRL). Here, BGNLM_PRL (Hubin, Storvik and Frommlet 2021) is implemented using distributed data GMJMCMC. To properly compare, the same data were used for training and testing for all algorithms, and the algorithms with a stochastic component were trained $N = 100$ times. We report the median of different statistics, as well as the minimum and maximum values across the 100 runs.

To evaluate the different algorithms, we use the accuracy (ACC), false positive rate (FPR) and false negative rate (FNR) on the test set. In a given run, these are defined as

$$\mathrm{ACC} = \frac{\sum_i \mathrm{I}(\hat{y}_i^* = y_i^*)}{n_p}; \quad \mathrm{FPR} = \frac{\sum_i \mathrm{I}(\hat{y}_i^* = 1, y_i^* = 0)}{\sum_i \mathrm{I}(y_i^* = 0)},$$
$$\mathrm{FNR} = \frac{\sum_i \mathrm{I}(\hat{y}_i^* = 0, y_i^* = 1)}{\sum_i \mathrm{I}(y_i^* = 1)},$$

where n_p is the size of the test data and y_i^* is the i-th test sample.

Wisconsin Breast Cancer data set

This data set was first introduced by Mangasarian, Street and Wolberg (1995). It contains digitized images of fine needle aspirates (FNA) of breast mass from 569 patients (357 benign and 212 malignant tissues). Ten real-valued characteristics are considered: radius, texture, perimeter, area, smoothness, concavity, concave points, symmetry and fractal dimension. For each of these characteristics, the mean, standard deviation and the mean of the three largest values per image were computed, resulting in 30 explanatory variables.

To properly compare our implementation with Hubin, Storvik and Frommlet 2021, we used the same samples for training and testing. In their paper, a randomly selected 25% of the images was selected as training data while the reminding 75% were used as a test set.

Since the size of the training data is relatively small, we found it beneficial to use only one batch trained for 1000 epochs in each generation. Both BGNLM_MF and BGNLM_FLOW was trained for 15 generations, where the number of features in each population was 130 and the maximum operations count of any feature was 10.

Algorithm	ACC	FNR	FPR
BGNLM_MF	0.9789 (0.9765,0.9836)	0.0503 (0.0377,0.0503)	0.0037 (0.0037,0.0075)
BGNLM_FLOW	0.9765 (0.9742,0.9812)	0.0503 (0.0440,0.0629)	0.0037 (0.0000,0.0112)
BGNLM_PRL	0.9742 (0.9695,0.9812)	0.0479 (0.0479,0.0536)	0.0111 (0.0000,0.0184)
RIDGE	0.9742 (-,-)	0.0592 (-,-)	0.0037 (-,-)
BGLM	0.9718 (0.9648,0.9765)	0.0592 (0.0536,0.0702)	0.0074 (0.0000,0.0148)
BGNLM	0.9695 (0.9554,0.9789)	0.0536 (0.0479,0.0809)	0.0148 (0.0037,0.0326)
DEEPNETS	0.9695 (0.9225,0.9789)	0.0674 (0.0305,0.1167)	0.0074 (0.0000,0.0949)
LR	0.9671 (-,-)	0.0479 (-,-)	0.0220 (-,-)
LASSO	0.9577 (-,-)	0.0756 (-,-)	0.0184 (-,-)
LXGBOOST	0.9554 (0.9554,0.9554)	0.0809 (0.0809,0.0809)	0.0184 (0.0184,0.0184)
TXGBOOST	0.9531 (0.9484,0.9601)	0.0647 (0.0536,0.0756)	0.0326 (0.0291,0.0361)
RFOREST	0.9343 (0.9038,0.9624)	0.0914 (0.0422,0.1675)	0.0361 (0.0000,0.1010)
NBAYES	0.9272 (-,-)	0.0305 (-,-)	0.0887 (-,-)

Table 4.1: Accuracy (ACC), false negative rate (FNR), and false positive rate (FPR) of various classification algorithms. The values in parentheses represent the smallest and largest values from the 100 runs, respectively. TThe contents of this table is taken from Hubin, Storvik and Frommlet 2021 and the results from 100 runs of BGNLM_MF and BGNLM_FLOW were added.

The above table shows that NBAYES and RFOREST was the worst performers. NBAYES predicted too many many false positives and RFOREST too many false negatives. All of the algorithms based on linear features are among the best performing methods, indicating that nonlinearities are not of primary importance in this dataset. Nevertheless, all versions of BGNLM are amongst the best performing algorithms. BGNLM_MF performed best in terms of median accuracy, while BGNLM_FLOW and BGNLM_PRL did slightly worse over the 100 runs. Interestingly, BGNLM_MF shows great consistency in this application with median FPR equal to its best, and median FNR equal to its worst.

Spam Classification

The Spambase dataset (Cranor and LaMacchia 1998) consists of emails classified as either spam or non-spam. The emails were collected from a variety of sources and manually labeled by humans as either spam or non-spam. The dataset contains a total of 4, 601 instances, with 1, 813 spam and 2, 788 non-spam emails. Each email instance in the dataset is represented by 57 features, which include things like the frequency of certain words, the use of all capital letters, and the presence of certain characters such as exclamation points and dollar signs. The data were randomly split into a training set of 1536 emails, with the remanding 3065 emails being used for testing.

Both BGNLM_FLOW and BGNLM_MF was trained using a batch size of 512 and 600 epochs for 15 generations. The size of each population was set to 200, and maximum operations count for all features was 10.

Algorithm	ACC	FNR	FPR
TXGBOOST	0.9465 (0.9442,0.9481)	0.0783 (0.0745,0.0821)	0.0320 (0.0294,0.0350)
RFOREST	0.9328 (0.9210,0.9413)	0.0814 (0.0573,0.1174)	0.0484 (0.0299,0.0825)
BGNLM_MF	0.9303 (0.9251,0.9362)	0.1042 (0.0857,0.1143)	0.0478 (0.0414, 0.0531)
BGNLM_FLOW	0.9297 (0.9225,0.9368)	0.1076 (0.0941,0.1227)	0.0468 (0.0383, 0.0584)
DEEPNETS	0.9292 (0.9002,0.9357)	0.0846 (0.0573,0.1465)	0.0531 (0.0310,0.0829)
BGNLM_PRL	0.9251 (0.9139,0.9377)	0.0897 (0.0766,0.1024)	0.0552 (0.0445,0.0639)
BGNLM	0.9243 (0.9113,0.9328)	0.0927 (0.0808,0.1116)	0.0552 (0.0465,0.0658)
LR	0.9194 (-,-)	0.0681 (-,-)	0.0788 (-,-)
BGLM	0.9178 (0.9168,0.9188)	0.1090 (0.1064,0.1103)	0.0528 (0.0523,0.0538)
LASSO	0.9171 (-,-)	0.1077 (-,-)	0.0548 (-,-)
RIDGE	0.9152 (-,-)	0.1288 (-,-)	0.0415 (-,-)
LXGBOOST	0.9139 (0.9139,0.9139)	0.1083 (0.1083,0.1083)	0.0591 (0.0591,0.0591)
NBAYES	0.7811 (-,-)	0.0801 (-,-)	0.2342 (-,-)

Table 4.2: Accuracy, false negative rate (FNR), and false positive rate (FPR) of various classification algorithms. The values in parentheses represent the smallest and largest values from the 100 runs, respectively. The contents of this table is taken from Hubin, Storvik and Frommlet 2021 and the results from 100 runs of BGNLM_MF and BGNLM_FLOW were added.

Table 4.2 shows the results for the different algorithms. Once again, NBAYES was the worst performer due to too many false positives. However, this time TXGBOOST and RFOREST performed the best, and the models with linear features are amongst the worst. This indicates that nonlinearities are important for this dataset. Both BGNLM_FLOW and BGNLM_MF are however still amongst the top-performing algorithms and have similar performance.

Prediction of metric outcome

For prediction of a metric outcome, we consider a BGNLM with Gaussian response and identity link function. We assumed homoscedasticity and let $\phi = \sigma^2$ where σ^2 is the variance of $\mathbf{y}$.

Predictions are now made according to

$$\hat{y}_i^* = u_i,$$

where u_i is sampled directly using the LRT described in the previous chapter.

Here, we used both $a = e^{-2}$ and $a = e^{-\log n}$ for the model priors in Equation (4). In Table 4.3, these are marked as AIC and BIC, respectively. The complexity measure, $c(F_j(\cdot,\cdot))$, is still $\log(oc_j + 1) - \varepsilon$, resulting in prior probability close to 1 for linear terms. Nonlinearities were selected from $\mathcal{G} = \{\text{sigmoid}(x), \log(|x|+1), \exp(-|x|), |x|^{7/2}, |x|^{5/2}, |x|^{1/3}\}$ with uniform probability, and we used all transformations with $P_{mu} = P_{mo} = P_{pr} = P_{new} = 1/4$.

We still compare our implementations with different algorithms, all with the same training- and testing data. However, since the Bayes classifier and logistic regression are not well suited for this task, we replace these with Bayesian linear regression, fitted with variational inference (VARBAYES) (Carbonetto and Stephens 2012) and a simple Gaussian regression (GR). Methods with a stochastic component were again run $N = 100$ times, and we report the median, worst and best results of three different statistics. The statistics are now the root mean squared error (RMSE), mean absolute error (MAE) and Pearson's correlation coefficient between the data and the predicted values (CORR). These are defined as follows:

$$\text{RMSE} = \sqrt{\frac{\sum_i^{n_p}(\hat{y}_i^* - y_i^*)^2}{n_p}}; \quad \text{MAE} = \frac{\sum_i^{n_p}|\hat{y}_i^* - y_i^*|}{n_p},$$

$$\text{CORR} = \frac{\sum_i^{n_p}(\hat{y}_i^* - \bar{\hat{y}}^*)(y_i^* - \bar{y}^*)}{\sqrt{\sum_i^{n_p}(\hat{y}_i^* - \bar{\hat{y}}^*)^2}\sqrt{\sum_i^{n_p}(y_i^* - \bar{y}^*)^2}},$$

Abalone Age

The abalone age dataset (Nash et al. 1994) has served as a benchmark dataset for prediction algorithms for almost three decades. It contains information about abalone, a type of sea snail, and their physical measurements. The dataset consists of $4,177$ instances, with each instance representing a single abalone. There are eight features for each abalone, including measurements of the shell length, diameter, height, and weight, as well as the sex of the abalone. The target variable is the age of the abalone, which is typically determined by counting the number of rings on their shells. Many of the covariates in the dataset are correlated (see the Appendix for correlation matrix). Intuitively, it is easy to see that weight is positively correlated with height etc.

We again use the same data for training and testing across all algorithms. A randomly selected 3177 instances were used for training, and the remaining 1000 were used for testing. Both BGNLM_MF and BGNLM_FLOW was trained using a batch size of 500 for 600 epochs. We ran for 15 generations, and the number of features in each population was set to 50, with a maximum operations count of 10 for each feature. The flow-based approximate posterior was again computed using $K = 2$ components with two hidden layers of 75 nodes for each component.

Algorithm	RMSE	MAE	CORR
BGNLM_FLOW (AIC)	1.9561 (1.9375,1.9755)	1.4218 (1.4113,1.4371)	0.7815 (0.7761,0.7841)
BGNLM_PRL (BIC)	1.9573 (1.9334,1.9903)	1.4467 (1.4221,1.4750)	0.7831 (0.7740, 0.7895)
BGNLM_FLOW (BIC)	1.9622 (1.9456,1.9986)	1.4273 (1.4149,1.4589)	0.7799, (0.7703,0.7838)
BGNLM (BIC)	1.9690 (1.9380,2.0452)	1.4552 (1.4319,1.5016)	0.7803 (0.7616,0.7882)
BGNLM_PRL (AIC)	1.9720 (1.9328,2.0081)	1.4548 (1.4377,1.4903)	0.7795 (0.7693,0.7893)
BGNLM_MF (BIC)	1.9894 (1.9494,2.0085)	1.4661 (1.4299,1.4817)	0.7746 (0.7698,0.7847)
BGNLM_MF (AIC)	1.9896 (1.9630,2.0071)	1.4672 (1.4417,1.4836)	0.7744 (0.7698,0.7813)
BGNLM (AIC)	2.0046 (1.9573,2.0560)	1.4821 (1.4471,1.5209)	0.7707 (0.7566,0.7831)
RFOREST	2.0352 (2.0020,2.0757)	1.4924 (1.4650,1.5259)	0.7633 (0.7530,0.7712)
BGLM	2.0758 (-,-)	1.5381 (-,-)	0.7522 (-,-)
LASSO	2.0765 (-,-)	1.5386 (-,-)	0.7514 (-,-)
VARBAYES	2.0779 (-,-)	1.5401 (-,-)	0.7516 (-,-)
GR	2.0801 (-,-)	1.5401 (-,-)	0.7500 (-,-)
LXGBOOST	2.0880 (2.0879,2.0880)	1.5429 (1.5429,1.5429)	0.7479 (0.7479,0.7479)
TXGBOOST	2.0881 (2.0623,2.1117)	1.5236 (1.4981,1.5438)	0.7526 (0.7461,0.7590)
RIDGE	2.1340 (-,-)	1.5649 (-,-)	0.7347 (-,-)
DEEPNETS	2.1466 (1.9820,3.5107)	1.5418 (1.3812,3.1872)	0.7616 (0.6925,0.7856)

Table 4.3: Root mean squared error (RMSE), mean absolute error (MAE), correlation (CORR) of various regression algorithms. The values in parentheses represent the smallest and largest values of the 100 runs, respectively. The contents of this table is taken from Hubin, Storvik and Frommlet 2021 and the results from 100 runs of BGNLM_MF and BGNLM_FLOW were added.

DEEPNETS was the worst performer for this dataset with remarkably varying results. However, Hubin, Storvik and Frommlet (2021) admits that not much effort was put into tuning the DEEPNETS. Again, BGNLM_FLOW did slightly better than BGNLM_PRL over the 100 runs. In contrast to GMJMCMC, our flow implementation worked better with modified-AIC prior, with the worst run of BGNLM_FLOW (AIC) comparable to the median of BGNLM_PRL (AIC). In addition, BGNLM_FLOW had overall lower MAE, while BGNLM_PRL reports predictions that are more correlated with the test set.

The mean-field approximate posterior did well but not great in this application. Overall, BGNLM_MF did better than the BGNLM implemented without using distributed data and with AIC-like prior. However, it was not able to predict as well as the flow model, presumably due to high correlations in the dataset.

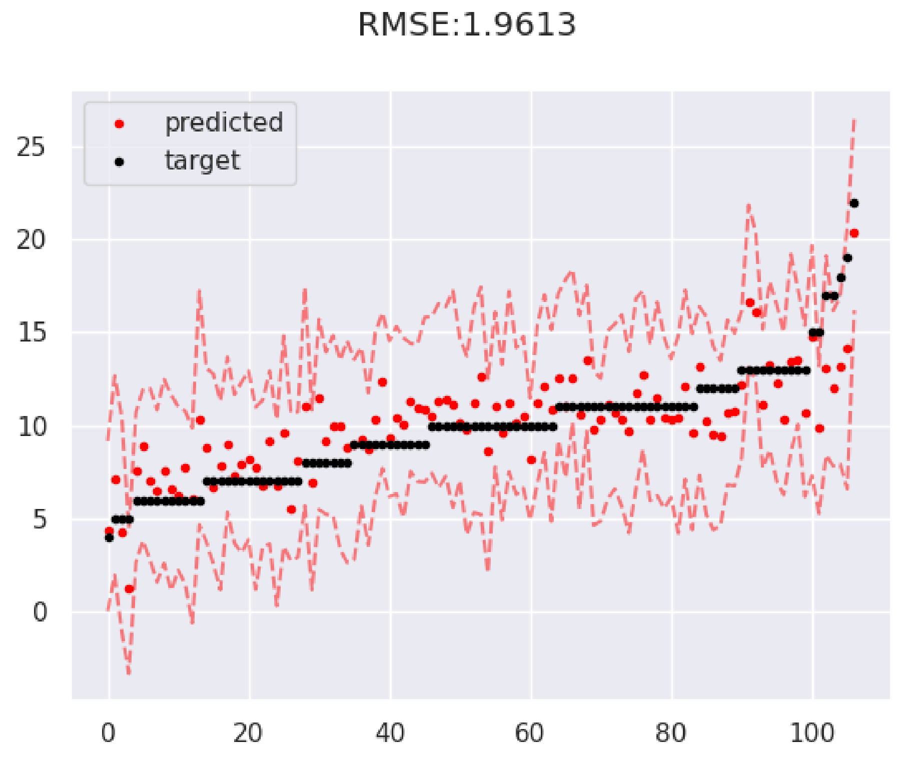


Figure 4.4: Predictions of abalone age (y-axis). Plot shows 100 randomly sampled predictions and the respective test values from an arbitrary run of BGNLM_FLOW (AIC). Dashed lines are 95% credible predictive intervals.

In Figure 4.4 we illustrate the ability to evaluate the uncertainty of predictions made by our models. Mean and variance is available through calculations shown in previous chapter, and predictions are sampled using the local reparametrization trick.

CHAPTER 5

Conclusion

In this thesis, we have implemented a new method to make inference on BGLMs and BGNLMs. We have shown that this method was able to do Bayesian model selection and model averaging. In addition, we have performed feature generation through a genetic algorithm. Our implementation of mean-field and flow-based approximate posteriors are able to estimate the marginal inclusion probabilities of predictive features as well as give uncertainty measures for the regression coefficients in both BGLM and BGNLM. Through a series of applications we have shown that our implementation is competitive with GMJMCMC (Hubin, Storvik and Frommlet 2020), and even outperforms GMJMCMC on multiple real world datasets. All though we were not successful in testing our implementations on a larger dataset, and leave that to future work, we are certain that the proposed method is able to scale.

Yet, there are multiple limitations to consider. First, we tried to recover Kepler's third law, which was one of the applications in Hubin, Storvik and Frommlet (2021). It seems our implementations fail when the signal coming from nonlinear structures are weak. One solution to this could be to implement a second-order optimization technique. However, this would severely increase training times. Second, we must admit that our implementation is not fully Bayesian since the $\boldsymbol{\alpha}$-parameters inside the projections are not estimated in a Bayesian manner. Rather, we used the practical approach of fixing parameters within projections before applying a nonlinearity. It is a computationally efficient strategy but not entirely Bayesian, and we and suggest improving upon this by implementing the other suggested strategies.

A topic for future research involves comparing the computation times of the different implementations with GMJMCMC. Since computation times are highly dependent on hardware and efficient implementation, we were not able to properly do this. Another topic for future research is related to the priors. All though the model prior was listed as possible weakness in Hubin, Storvik and Frommlet (2021), we adopted their approach inspired by modifications of AIC and BIC. They worked well for the applications presented here but there are other priors that should be considered. We will hence extend the suggestion of modifying the model prior, possibly with a hyperprior on a. Another topic for future research involves the parameter initialization. We used a strategy similar to zero-initialization. The approach again worked fine for the applications here, but other strategies should also be considered.

In the Appendix we have included some plots that did not make it to Chapter 4. These include credible intervals for regression coefficients in the two first simulation studies, and correlation plot for the Abalone dataset.

All code for running and plotting the different applications can be found on https://github.com/sebsommer/BGNLM. At the time of writing this is not a finished Python library, but we aim to finalize the development in short time.

The Appendix

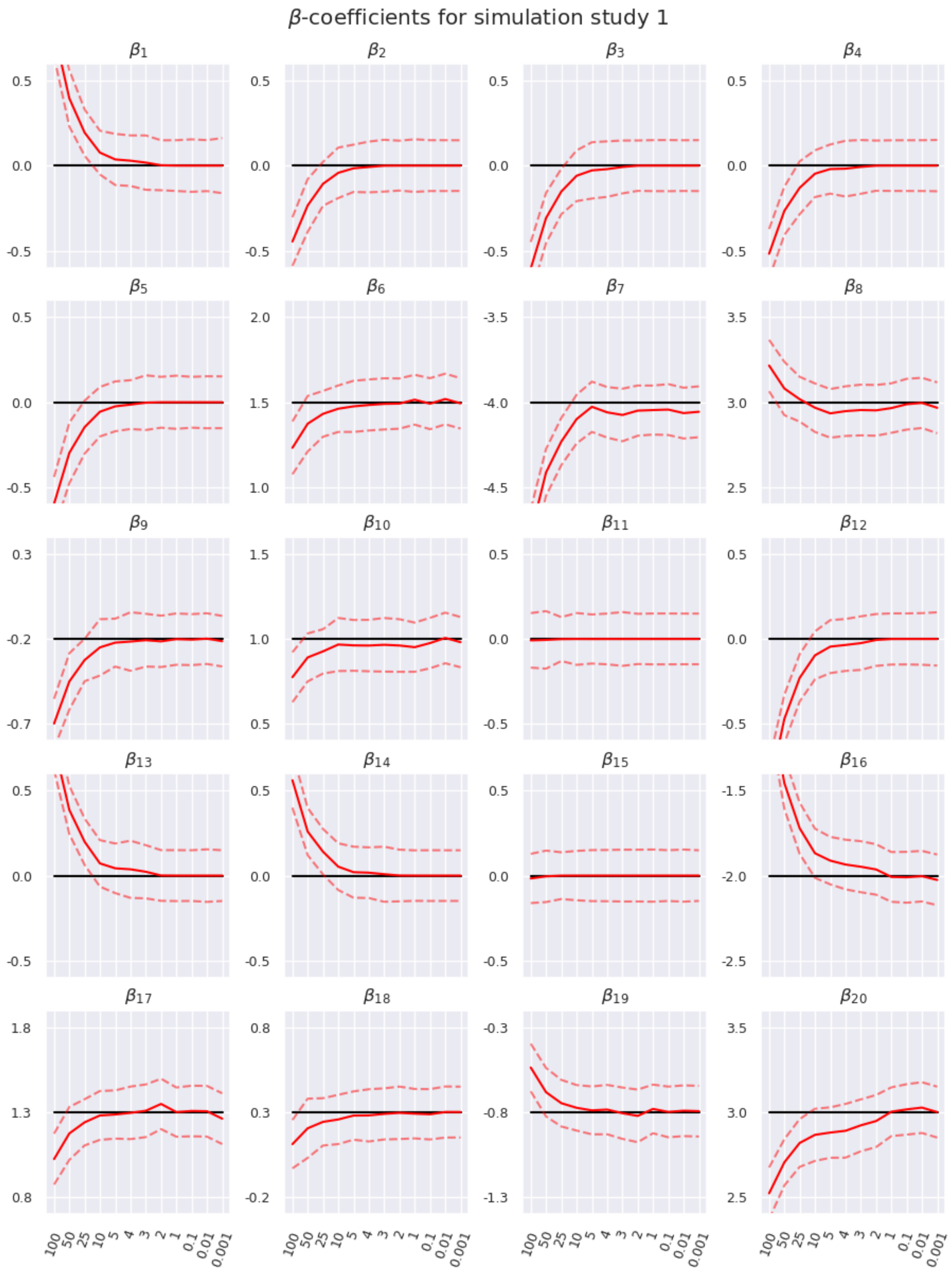


Figure 5.1: Uncertainty plots for β-parameters for an arbitrary run of simulation study 1 using flow-based approximate posterior. Black lines are the true values.

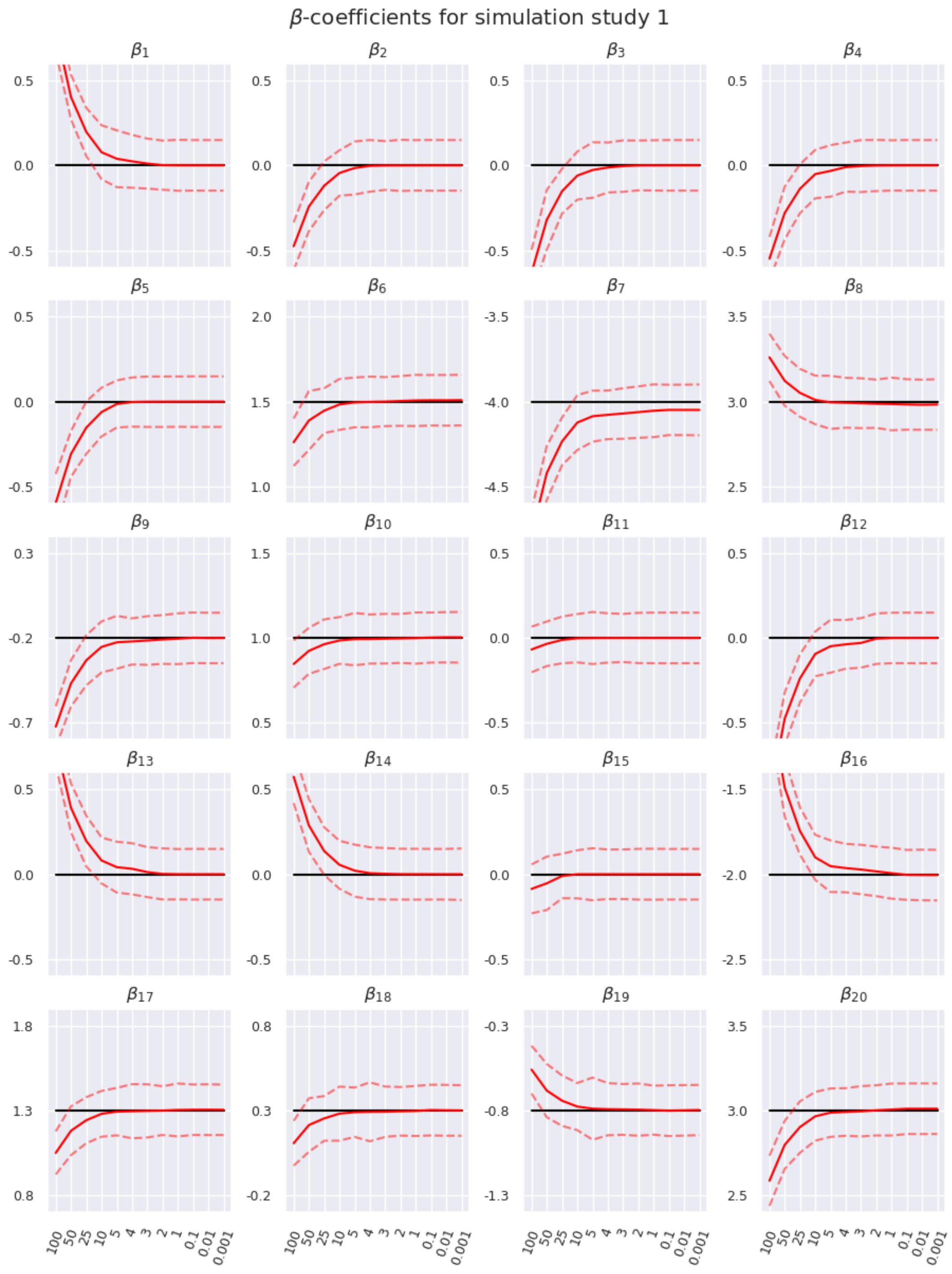
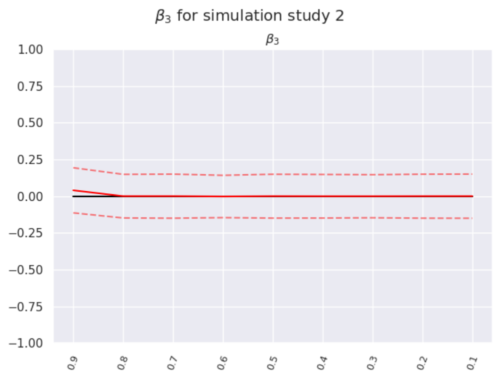
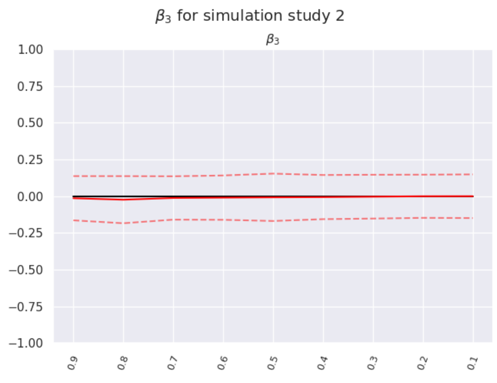


Figure 5.2: Uncertainty plots for β-parameters for an arbitrary run of simulation study 1 using mean field approximate posterior. Black lines are the true values.

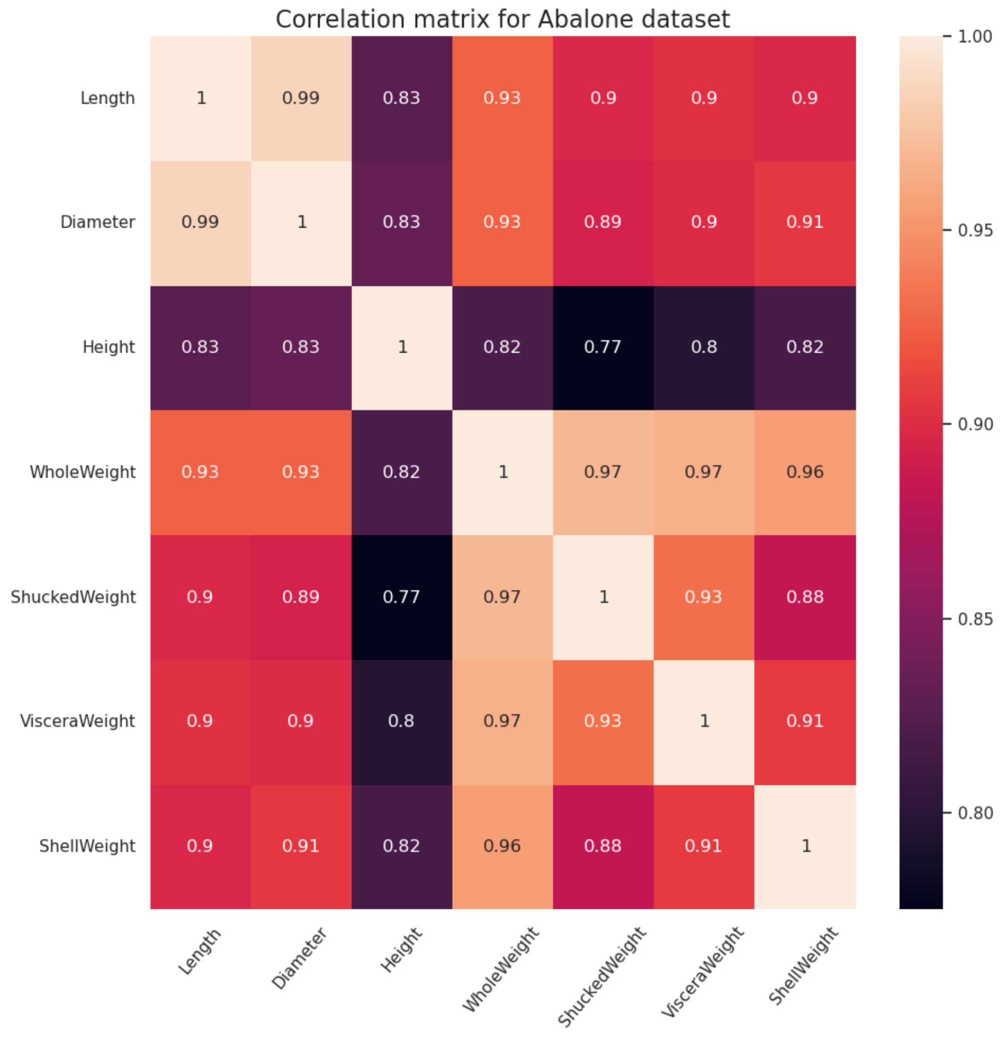


(a) Flow



(b) Mean-field

Figure 5.3: Uncertainty plots for β_3 for an arbitrary run of simulation study 2. Black lines are the true values. With γ_3 fixed we are still able to recover the true value of β_3 due to strong signal (low noise).



	Length	Diameter	Height	WholeWeight	ShuckedWeight	VisceraWeight	ShellWeight
Length	1	0.99	0.83	0.93	0.9	0.9	0.9
Diameter	0.99	1	0.83	0.93	0.89	0.9	0.91
Height	0.83	0.83	1	0.82	0.77	0.8	0.82
WholeWeight	0.93	0.93	0.82	1	0.97	0.97	0.96
ShuckedWeight	0.9	0.89	0.77	0.97	1	0.93	0.88
VisceraWeight	0.9	0.9	0.8	0.97	0.93	1	0.91
ShellWeight	0.9	0.91	0.82	0.96	0.88	0.91	1

Figure 5.4: Correlation plot for abalone data set. Categorical variable 'Sex' is excluded.

Bibliography

Adya, M. and Collopy, F. (1998). 'How effective are neural networks at forecasting and prediction? A review and evaluation'. In: *Journal of Forecasting* vol. 17, no. 5-6, pp. 481–495.

Ba, J. L., Kiros, J. R. and Hinton, G. E. (2016). *Layer Normalization.* arXiv: 1607.06450 [stat.ML].

Battiti, R. (Mar. 1992). 'First- and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method'. In: *Neural Computation* vol. 4.

Bogdan, M., Ghosh, J. K. and Tokdar, S. T. (2008). 'A comparison of the Benjamini-Hochberg procedure with some Bayesian rules for multiple testing'. In: *Beyond Parametrics in Interdisciplinary Research: Festschrift in Honor of Professor Pranab K. Sen.* Institute of Mathematical Statistics, pp. 211–230.

Bottou, L., Curtis, F. E. and Nocedal, J. (2018). *Optimization Methods for Large-Scale Machine Learning.* arXiv: 1606.04838 [stat.ML].

Breiman, L. (Oct. 2001). 'Random Forests'. In: *Machine Learning* vol. 45, pp. 5–32.

Cao, N. D., Titov, I. and Aziz, W. (2019). *Block Neural Autoregressive Flow.* arXiv: 1904.04676 [stat.ML].

Carbonetto, P. and Stephens, M. (2012). 'Scalable Variational Inference for Bayesian Variable Selection in Regression, and Its Accuracy in Genetic Association Studies'. In: *Bayesian Analysis* vol. 7, no. 1, pp. 73–108.

Cranor, L. F. and LaMacchia, B. A. (Aug. 1998). 'Spam!' In: *Commun. ACM* vol. 41, no. 8, pp. 74–83.

Csiszar, I. (1975). 'I-Divergence Geometry of Probability Distributions and Minimization Problems'. In: *The Annals of Probability* vol. 3, no. 1, pp. 146–158.

Dinh, L., Sohl-Dickstein, J. and Bengio, S. (2017). *Density estimation using Real NVP*. arXiv: 1605.08803 [cs.LG].

European Parliament (2016). ‘General Data Protection Regulation’. In: vol. 119, no. 1, pp. 1–88.

Fritsch, A. and Ickstadt, K. (2009). ‘Improved criteria for clustering based on the posterior similarity matrix’. In: *Bayesian Analysis* vol. 4, no. 2, pp. 367–391.

Germain, M. et al. (2015). *MADE: Masked Autoencoder for Distribution Estimation*. arXiv: 1502.03509 [cs.LG].

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

Hansen, L. and Sargent, T. J. (May 2001). ‘Robust Control and Model Uncertainty’. In: *American Economic Review* vol. 91, no. 2, pp. 60–66.

Hastings, W. K. (Apr. 1970). ‘Monte Carlo Sampling Methods using Markov Chains and their Applications’. In: *Biometrika* vol. 57, no. 1, pp. 97–109.

Heinze, G., Ambler, G. and Benner, A. (2022). ‘mfp: Multivariable Fractional Polynomials’.

Hinton, G., Srivastava, N. and Swersky, K. (2018). ‘Lecture6a: Neural Networks for Machine Learning’.

Hinton, G. E. and Camp, D. van (1993). ‘Keeping the neural networks simple by minimizing the description length of the weights’. In: *Annual Conference Computational Learning Theory*.

Hinton, G. E., Srivastava, N., Krizhevsky, A. et al. (2012). *Improving neural networks by preventing co-adaptation of feature detectors*. arXiv: 1207.0580 [cs.NE].

Hubin, A. and De Bin, R. (July 2022). ‘On a genetically modified mode jumping MCMC approach for multivariate fractional polynomials’. In.

Hubin, A. and Storvik, G. (2019). *Combining Model and Parameter Uncertainty in Bayesian Neural Networks*. arXiv: 1903.07594 [stat.ML].

Hubin, A., Storvik, G. and Frommlet, F. (Mar. 2020). ‘A Novel Algorithmic Approach to Bayesian Logic Regression (with Discussion)’. In: *Bayesian Analysis* vol. 15, no. 1.

Hubin, A., Storvik, G. and Frommlet, F. (2021). *Flexible Bayesian Nonlinear Model Configuration*.

Ioffe, S. and Szegedy, C. (2015). *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. arXiv: 1502.03167 [cs.LG].

Jordan, M. I. et al. (1999). ‘An Introduction to Variational Methods for Graphical Models’. In: *Machine Learning* vol. 37, pp. 183–233.

Kanter, J. M. and Veeramachaneni, K. (2015). ‘Deep feature synthesis: Towards automating data science endeavors’. In: *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pp. 1–10.

Kingma, D. P. and Ba, J. (2017). *Adam: A Method for Stochastic Optimization.* arXiv: 1412.6980 [cs.LG].

Kingma, D. P., Salimans, T., Jozefowicz, R. et al. (2017). *Improving Variational Inference with Inverse Autoregressive Flow.* arXiv: 1606.04934 [cs.LG].

Kingma, D. P., Salimans, T. and Welling, M. (2015). *Variational Dropout and the Local Reparameterization Trick.* arXiv: 1506.02557 [stat.ML].

Lachmann, J. (2021). ‘Subsampling Strategies for Bayesian Variable Selection and Model Averaging in GLM and BGNLM’. In.

Lecun, Y. et al. (1998). ‘Gradient-based learning applied to document recognition’. In: *Proceedings of the IEEE* vol. 86, no. 11, pp. 2278–2324.

Lindeberg, J. W. (1922). ‘Eine neue Herleitung des Exponentialgesetzes in der Wahrscheinlichkeitsrechnung’. In: *Mathematische Zeitschrift* vol. 15, pp. 211–225.

Linnainmaa, S. (1970).

Louizos, C. and Welling, M. (2017). *Multiplicative Normalizing Flows for Variational Bayesian Neural Networks.* arXiv: 1703.01961 [stat.ML].

Maddison, C. J., Mnih, A. and Teh, Y. W. (2016). ‘The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables’. In: *CoRR* vol. abs/1611.00712. arXiv: 1611.00712.

Mangasarian, O. L., Street, W. N. and Wolberg, W. H. (1995). ‘Breast Cancer Diagnosis and Prognosis via Linear Programming’. In: *Operations Research* vol. 43, no. 4, pp. 570–577.

Nalisnick, E., Anandkumar, A. and Smyth, P. (2015). *A Scale Mixture Perspective of Multiplicative Noise in Neural Networks.* arXiv: 1506.03208 [stat.ML].

Nash, W. et al. (Jan. 1994). ‘7he Population Biology of Abalone (Haliotis species) in Tasmania. I. Blacklip Abalone (H. rubra) from the North Coast and Islands of Bass Strait’. In: *Sea Fisheries Division, Technical Report No* vol. 48.

Nelder, J. A. and Wedderburn, R. W. M. (1972). ‘Generalized Linear Models’. In: *Journal of the Royal Statistical Society: Series A (General)* vol. 135, no. 3, pp. 370–384. eprint: https://rss.onlinelibrary.wiley.com/doi/pdf/10.2307/2344614.

Papamakarios, G., Nalisnick, E. et al. (2021). *Normalizing Flows for Probabilistic Modeling and Inference.* arXiv: 1912.02762 [stat.ML].

Papamakarios, G., Pavlakou, T. and Murray, I. (2018). *Masked Autoregressive Flow for Density Estimation.* arXiv: 1705.07057 [stat.ML].

Peterson, C. and Anderson, J. R. (1987). ‘A Mean Field Theory Learning Algorithm for Neural Networks’. In: *Complex Systems*, no. 1, pp. 995–1019.

Quiroz, M., Kohn, R. et al. (2019). ‘Speeding Up MCMC by Efficient Data Subsampling’. In: *Journal of the American Statistical Association* vol. 114, no. 526, pp. 831–843.

Quiroz, M., Villani, M. et al. (2018). *Subsampling MCMC - An introduction for the survey statistician.* arXiv: `1807.08409 [stat.ME]`.

Ranganath, R., Tran, D. and Blei, D. M. (2016). *Hierarchical Variational Models.* arXiv: `1511.02386 [stat.ML]`.

Razi, M. A. and Athappilly, K. (2005). ‘A comparative predictive analysis of neural networks (NNs), nonlinear regression and classification and regression tree (CART) models’. In: *Expert Syst. Appl.* vol. 29, pp. 65–74.

Refenes, A. N., Zapranis, A. and Francis, G. (1994). ‘Stock performance modeling using neural networks: A comparative study with regression models’. In: *Neural Networks* vol. 7, no. 2, pp. 375–388.

Rényi, A. (1961). ‘On measures of entropy and information’. In: *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics.* Vol. 4. University of California Press, pp. 547–562.

Rezende, D. J. and Mohamed, S. (2016). *Variational Inference with Normalizing Flows.* arXiv: `1505.05770 [stat.ML]`.

Robbins, H. E. (1951). ‘A Stochastic Approximation Method’. In: *Annals of Mathematical Statistics* vol. 22, pp. 400–407.

Royston, P. and Sauerbrei, W. (2008). ‘Multivariable Model-Building: A Pragmatic Approach to Regression Analysis based on Fractional Polynomials for Modelling Continuous Variables’. In.

Rue, H., Martino, S. and Chopin, N. (2009). ‘Approximate Bayesian inference for latent Gaussian models by using integrated nested Laplace approximations’. In: *Journal of the Royal Statistical Society Series B* vol. 71, no. 2, pp. 319–392.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986). ‘Learning representations by back-propagating errors’. In: *Nature* vol. 323, pp. 533–536.

Schmoor, C., Olschewski, M. and Schumacher, M. (1996). ‘Randomized and Non-Randomized Patients in Clinical Trials: Experiences with Comprehensive Cohort Studies’. In: *Statistics in Medicine* vol. 15, no. 3, pp. 263–271.

Scott, J. and Berger, J. (July 2006). ‘An exploration of aspects of Bayesian multiple testing’. In: *Journal of Statistical Planning and Inference* vol. 136, pp. 2144–2162.

Skaaret-Lund, L., Hubin, A. and Storvik, G. (2023). *Sparsifying Bayesian neural networks with latent binary variables and normalizing flows.*

Sriperumbudur, B. K. et al. (2009). *On integral probability metrics, phi-divergences and binary classification.* arXiv: 0901.2698 [cs.IT].

Tabak, E. and Turner, C. (Feb. 2013). 'A family of nonparametric density estimation algorithms'. English (US). In: *Communications on Pure and Applied Mathematics* vol. 66, no. 2, pp. 145–164.

Tabak, E. G. and Vanden-Eijnden, E. (2010). 'Density estimation by dual ascent of the log-likelihood'. In: *Communications in Mathematical Sciences* vol. 8, no. 1, pp. 217–233.

Tierney, L. and Kadane, J. B. (1986). 'Accurate Approximations for Posterior Moments and Marginal Densities'. In: *Journal of the American Statistical Association* vol. 81, no. 393, pp. 82–86.

Tjelmeland, H. and Hegstad, B. K. (2001). 'Mode Jumping Proposals in MCMC'. In: *Scandinavian Journal of Statistics* vol. 28, no. 1, pp. 205–223. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1467-9469.00232.

Wadia, N. S. et al. (2021). *Whitening and second order optimization both make information in the dataset unusable during training, and can reduce or prevent generalization.* arXiv: 2008.07545 [cs.LG].