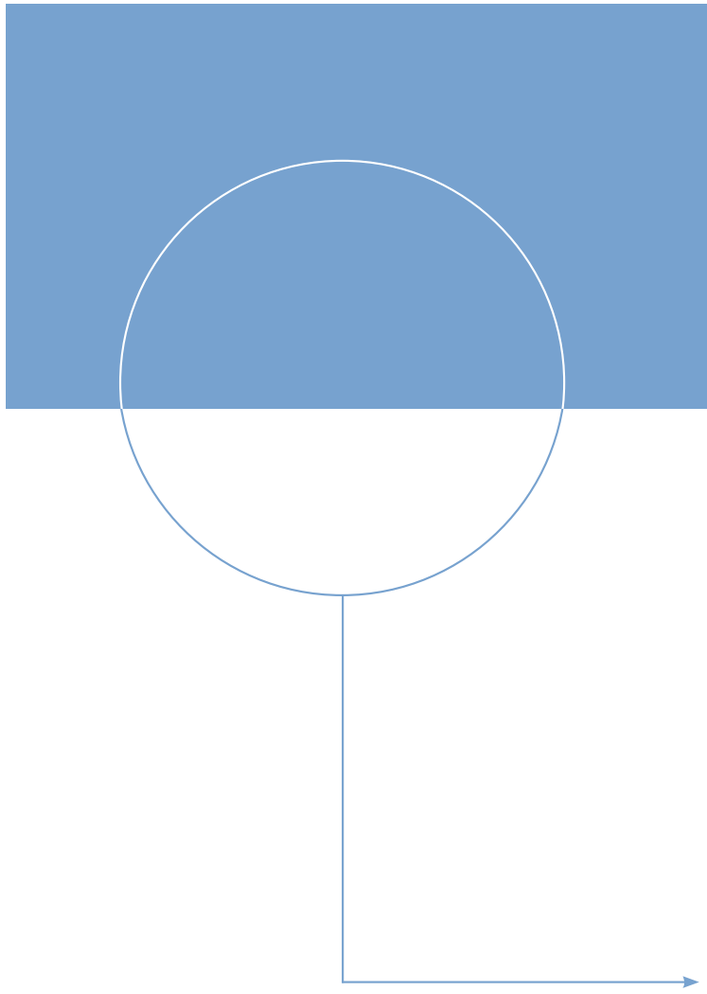


Master's thesis

2021

Ludvig Killingberg

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

Ludvig Killingberg

Generative Adversarial Networks for Flexible Variational Posteriors in Bayesian Neural Networks

April 2021

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Generative Adversarial Networks for Flexible Variational Posteriors in Bayesian Neural Networks

Ludvig Killingberg

Master of Science in Computer Science
Submission date: April 2021
Supervisor: Helge Langseth

Norwegian University of Science and Technology
Department of Computer Science

Abstract

Bayesian neural networks refers to an extension of neural networks made by treating each parameter as a stochastic variable rather than a point estimate. From a probabilistic perspective, neural networks limit themselves to a maximum likelihood estimation for its parameters. This method is prone to overfitting, as we often see happen. Typically regularization techniques are applied to avoid this, which, if we look at it from a probabilistic perspective, is equivalent to maximum a posteriori estimation. Although this works well in practice, neural networks are still restricting themselves to point estimated parameters. By instead predicting the full posterior on each weight, we can model both the epistemic and aleatory uncertainty in the data. Current implementations for Bayesian neural networks rely on approximating the posterior distribution with variational distributions. Some of these methods can severely limit the flexibility of the posterior and thus the performance of the model. Joining ongoing work in the field, this thesis aims to improve the flexibility of the approximate posterior.

In our work, we use both theoretical and experimental approaches to develop a novel method for Bayesian inference in neural networks. We provide the mathematical foundation for a new method of approximating posterior distributions, and support it with experimental results. We present a new method for generating posterior distributions in Bayesian neural networks through generative adversarial networks. Initially, we show that neural networks are able to approximate the KL-divergence between two distributions, and go on to use this with a generative network to learn the posterior distribution of the weights in a Bayesian neural network. In experiments we show that this method is able to compete with the state-of-the-art methods in the field with respect to both accuracy and predictive uncertainty.

Preface

This report is submitted to Norwegian University of Science and Technology as my final work towards a MSc degree in Computer Science. It has been worked on part time while simultaneously commencing work on a PhD in the year 2020/21.

The report targets computer science and statistics students with a specialization in artificial intelligence. I recommend a good grasp of linear algebra, calculus, statistics, and deep learning to thoroughly follow the ideas presented in this report.

I would like to thank my supervisor Helge Langseth for invaluable support, guidance and feedback.

Thanks to Schyler Bennett for many hours of proofreading.

Ludvig Killingberg
Trondheim, April 30, 2021

Contents

List of Figures

List of Tables

Notation

This section is a modified version of the notation section found in Goodfellow et al. [2016]. It provides a concise reference describing notation used throughout the thesis. If you are unfamiliar with any of the corresponding mathematical concepts, Goodfellow et al. [2016] describe most of these ideas in Chapters 2–4.

Numbers and Arrays

a	A scalar (integer or real)
$\boldsymbol{a}$	A vector
$\boldsymbol{A}$	A matrix
$\boldsymbol{I}$	Identity matrix with dimensionality implied by context
$\text{diag}(\boldsymbol{a})$	A square, diagonal matrix with diagonal entries given by $\boldsymbol{a}$
$\text{tr}(\boldsymbol{A})$	The trace of the matrix $\boldsymbol{A}$
a	A scalar random variable
$\mathbf{a}$	A vector-valued random variable
$\mathbf{A}$	A matrix-valued random variable

Sets and Graphs

$\mathbb{A}$	A set
$\mathbb{R}$	The set of real numbers
$\{0, 1\}$	The set containing 0 and 1
$\{0, 1, \dots, n\}$	The set of all integers between 0 and n

Indexing

a_i	Element i of vector $\boldsymbol{a}$, with indexing starting at 1
$\boldsymbol{a}_{-i}$	All elements of vector $\boldsymbol{a}$ except for element i
$\boldsymbol{a}_{n:m}$	Elements n through m of vector $\boldsymbol{a}$
$A_{i,j}$	Element i, j of matrix $\boldsymbol{A}$

Linear Algebra Operations

$\boldsymbol{A}^\top$	Transpose of matrix $\boldsymbol{A}$
$\boldsymbol{A} \odot \boldsymbol{B}$	Element-wise (Hadamard) product of $\boldsymbol{A}$ and $\boldsymbol{B}$
$\det(\boldsymbol{A})$ or $\lvert\boldsymbol{A}\rvert$	Determinant of $\boldsymbol{A}$

Calculus

$\frac{dy}{dx}$	Derivative of y with respect to x
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$\nabla_{\boldsymbol{x}} y$	Gradient of y with respect to $\boldsymbol{x}$
$\frac{\partial f}{\partial \boldsymbol{x}}$	Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ of $f : \mathbb{R}^n \to \mathbb{R}^m$
$\int f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral over the entire domain of $\boldsymbol{x}$
$\int_{\mathbb{S}} f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral with respect to $\boldsymbol{x}$ over the set $\mathbb{S}$

Probability and Information Theory

$P(\mathrm{a})$	A probability distribution over a discrete variable
$p(\mathrm{a})$	A probability distribution over a continuous variable, or over a variable whose type has not been specified
$P(a)$	Probability of event a occuring
$p(a)$	Likelihood function for the probability distribution p evaluated at a
$\mathrm{a} \sim p$	Random variable a has distribution p
$\mathbb{E}_{\mathrm{x}\sim p}[f(x)]$ or $\mathbb{E}[f(x)]$	Expectation of $f(x)$ with respect to $p(\mathrm{x})$
$\mathrm{Var}(f(x))$	Variance of $f(x)$ under $P(\mathrm{x})$
$\mathrm{Cov}(f(x), g(x))$	Covariance of $f(x)$ and $g(x)$ under $P(\mathrm{x})$
$D_{\mathrm{KL}}[Q\|P]$ or $D_{\mathrm{KL}}[q\|p]$	Kullback-Leibler divergence from Q to p (or q to p)
$\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$
$\mathcal{U}(\boldsymbol{l}, \boldsymbol{u})$	Uniform distribution with boundaries l and u
$p(\mathcal{D}\|\boldsymbol{\theta})$	The probability distribution $p(\boldsymbol{y}\|\boldsymbol{x}, \boldsymbol{\theta})$ for $\boldsymbol{x}$ and $\boldsymbol{y}$ in $\mathcal{D}$

Functions

$f : \mathbb{A} \to \mathbb{B}$	The function f with domain $\mathbb{A}$ and range $\mathbb{B}$
$f \circ g$	Composition of the functions f and g
$f(\boldsymbol{x}; \boldsymbol{\theta})$ or $f_{\boldsymbol{\theta}}(\boldsymbol{x})$	A function of $\boldsymbol{x}$ parametrized by $\boldsymbol{\theta}$. (Sometimes we write $f(\boldsymbol{x})$ and omit the argument $\boldsymbol{\theta}$ to lighten notation)
$\log x$	Natural logarithm of x
$\sigma(x)$	Logistic sigmoid, $\dfrac{1}{1+\exp(-x)}$

Sometimes we use a function f whose argument is a scalar but apply it to a vector, matrix, or tensor: $f(\boldsymbol{x})$ or $f(\boldsymbol{X})$. This denotes the application of f to the array element-wise. For example, if $\boldsymbol{C} = \sigma(\boldsymbol{X})$, then $C_{i,j} = \sigma(X_{i,j})$ for all valid values of i and j.

Datasets and Distributions

$\mathcal{D}$	A dataset containing training data $\boldsymbol{x}$ and target data $\boldsymbol{y}$
$\boldsymbol{x}^{(i)}$	The i-th example (input) from a dataset
$y^{(i)}$ or $\boldsymbol{y}^{(i)}$	The target associated with $\boldsymbol{x}^{(i)}$ for supervised learning

Chapter 1

Introduction

This chapter presents the motivation behind the research conducted for this report. Our research questions and goals will be stated, and we will give a description of how we aim to achieve those goals. We then briefly discuss the contributions we have made to the field. Finally we outline the structure and content of the report.

1.1 Background and Motivation

Bayesian deep learning is a sub-field of deep learning that aims to implement Bayesian statistics into traditional deep learning methods. Within this field there is ongoing research on Bayesian neural networks, an extension of neural networks which treats the weights as stochastic variables, rather than point estimates. The motivation behind this is to give the neural network the ability to represent uncertainty. Uncertainty comes in two forms:

Aleatory uncertainty: This is the uncertainty in the data we are using. It could be from inaccurate measurements, human error, or because the method that generates the data is a random process. This uncertainty is unavoidable.

Epistemic uncertainty: This uncertainty stems from a lack of data. With a finite dataset, there will always be some uncertainty about what the correct prediction should be at any point that is not included in the dataset. This uncertainty is called the epistemic uncertainty.

Bayesian neural networks aim to capture both of these uncertainties by introducing un-

certainty to every parameter of the model. Successfully capturing this uncertainty has great benefits to many applications of deep learning. Deep learning models are known to make predictions with unprecedented accuracy, but sometimes even when mistakes are far in-between, they can be devastating. The instances where it makes mistakes are often due to lack of data similar to the ones it is making predictions for. In these cases Bayesian deep learning models would be able to communicate this uncertainty, essentially preventing the model from being overconfident in its predictions.

1.2 Goals and Research Questions

Goal Increase the flexibility of the posterior distribution in Bayesian neural network by avoiding strong assumptions about the distribution.

Research question 1 What is state-of-the-art on Bayesian neural network?

Research question 1.1 How do we maximize ELBO without making assumption on the distribution?

Research question 2 Can generative adversarial networks be used for variational inference of more flexible distributions in Bayesian neural networks?

1.3 Thesis Structure

Chapter 2 - Background Theory: We give necessary background theory in deep learning and statistics. We start by iterating on fundamental theory that we expect the reader to be relatively familiar with, and go on to present more advanced details necessary to understand the rest of the report.

Chapter 3 - State of the Art: We introduce some state-of-the-art methods in Bayesian neural networks.

Chapter 4 - Method: We present our novel work and necessary mathematical proofs for a theoretical foundation of our work. We also give details related to the implementation of our method and how to achieve desirable results with it.

Chapter 5 - Evaluation and Results: We show the performance of our model on classic problems in the field, and discuss how these compare to current state-of-the-art methods.

Chapter 6 - Discussion and Conclusion: Based on both the theoretical and experimental work, we discuss how our method fits into current advancements in the field. We present ways that our method can be extended upon, and other opportunities for future work.

Chapter 2

Background Theory

This chapter introduces some fundamental theory in machine learning and statistics. It should provide the reader with enough information to understand the rest of the thesis.

2.1 Deep Learning

This section gives a brief introduction to deep learning. We first present the core idea and inspiration behind deep learning, followed by a theoretical background on the fundamentals of deep learning. The background skips many important contributions to deep learning, instead focusing on the key concepts necessary for this thesis. For a more complete introduction to the field of deep learning we refer the reader to Goodfellow et al. [2016].

2.1.1 Objective

The objective of any machine learning algorithm can be described as an optimization problem. In an optimization problem, we have some function $f : \mathbb{A} \to \mathbb{R}$, and wish to find an element $a^* \in \mathbb{A}$, such that $f(a^*) \leq f(a)\ \forall a \in \mathbb{A}$. The function we want to optimize, f, is called the *objective function*, while $\mathbb{A}$ is called the *search space*. In machine learning, the search space is typically a subset of the function space $\mathbb{A} : \mathbb{R}^n \to \mathbb{R}^m$. This means that we want to find the some function a from the inputs $\boldsymbol{x} \in \mathbb{R}^n$, to the outputs $\boldsymbol{y} \in \mathbb{R}^m$, that minimizes the objective function $f(a)$.

To see why this interpretation is helpful in deep learning, we need to consider what we actually want to achieve. We typically have some data $\mathcal{D} = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) | 1 \leq i \leq d\}$, and want to learn to predict $\boldsymbol{y}$ from $\boldsymbol{x}$. This is equivalent to finding a function a such

that $a(\boldsymbol{x}^{(i)}) = \boldsymbol{y}^{(i)}$. To be able to search for such a function, we let a be an element in a function-space $\mathbb{A}$, where each function in $\mathbb{A}$ is parameterized by vw. Our objective then becomes to find a function $a_{\boldsymbol{w}}$, so that $a_{\boldsymbol{w}}(\boldsymbol{x}^{(i)}) \approx \boldsymbol{y}^{(i)}$.

2.1.2 Representation

Neural networks are a way of defining $a_{\boldsymbol{w}}$, inspired by the neurons in our brain. We start by defining an artificial neuron. Each neuron is a function parameterized by $\boldsymbol{w}$ and b, and is defined as

$$\varphi\left(b + \sum_{i=1}^{n} x_i \cdot w_n\right), \tag{2.1}$$

where φ is some function $\varphi : \mathbb{R} \to \mathbb{R}$. This function is referred to as the *activation function*. An illustration of a neuron can be seen in Figure 2.1a.

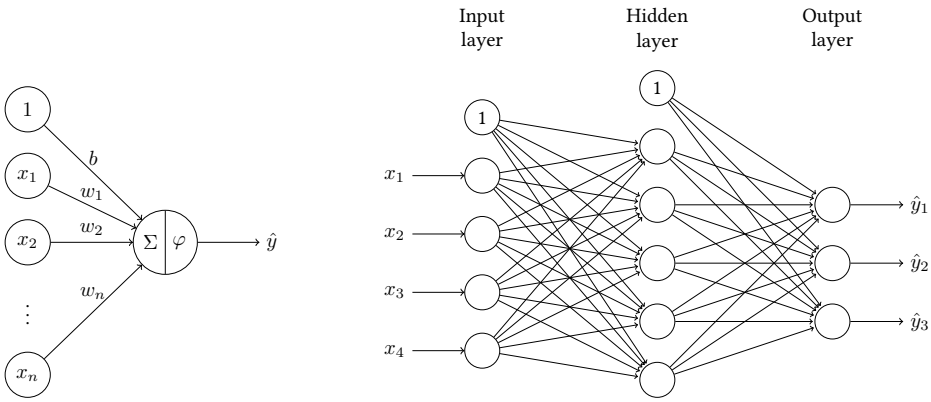
Individual neurons are then combined to form a neural network, as seen in Figure 2.1b. This specific composition of neurons is a feedforward neural network with one hidden layer. Composing neurons like this creates a much more complex function that, assuming some conditions on φ, is able to represent the relationship between $\boldsymbol{x}$ and $\boldsymbol{y}$ very well. In fact, Leshno et al. [1993] showed that if φ is a nonpolynomial locally bounded piecewise continuous function, then a multilayered feedforward neural network with sufficiently many hidden nodes can approximate any function to any degree of accuracy. This is known as the *universal approximation theorem*.

Different activation functions can be used in the same neural network, but each layer generally have the same activation function. Figure 2.2 shows some of the most common activation functions.

2.1.3 Loss Function

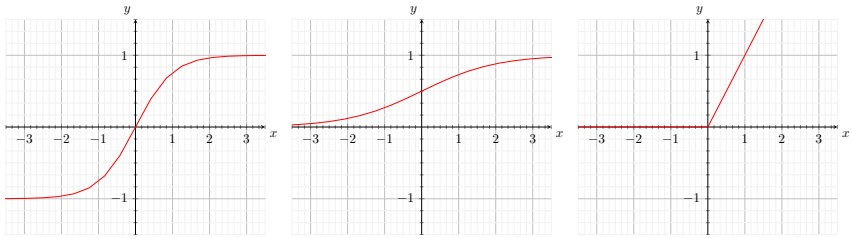
We have looked at how a neural network is able to represent a functional relationship in data. We will now look at how we can search for the optimal parameters of the network, the ones that makes the network represent a good approximation of the function from $\boldsymbol{x}$ to $\boldsymbol{y}$. This is where the *learning* part comes in; we say that we learn the functional relationship between $\boldsymbol{x}$ and $\boldsymbol{y}$.

To be able to find the optimal parameters, we first have to define what properties they hold. We start by looking at the data-generative process $\mathbf{X} \to \mathbf{Y}$ as a stochastic process. This means that the output of our network is $p_{\boldsymbol{w}}(\boldsymbol{y}|\boldsymbol{x})$. We now want to find the parameters $\boldsymbol{w}$ that makes our observations most probable, which means maximizing



(a) (b)

Figure 2.1: (a) Illustration of an artificial neuron with n inputs and activation function φ. (b) Illustration of a fully connected feedforward neural network with one hidden layer.



(a) tanh (b) sigmoid (c) relu

Figure 2.2: Some of the most common activation functions.

$$p_{\boldsymbol{w}}(\boldsymbol{y}_1 \ldots, \boldsymbol{y}_1 | \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) = \prod_{i=1}^{n} p_{\boldsymbol{w}}(\boldsymbol{y}_i | \boldsymbol{x}_i).$$

Equivanlent would be maximizing the log-likelihood, which has the benefit of being better numerically,

$$\log \prod_{i=1}^{n} p_{\boldsymbol{w}}(\boldsymbol{y}_i | \boldsymbol{x}_i) = \sum_{i=1}^{n} \log p_{\boldsymbol{w}}(\boldsymbol{y}_i | \boldsymbol{x}_i).$$

We will be use $\mathcal{L}(\boldsymbol{w}; \boldsymbol{x}, \boldsymbol{y})$, $\mathcal{L}(\boldsymbol{w})$ (data implied) or $\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$ to symbolize the loss function[1] throughout the thesis.

In machine learning it is common to formulate the optimization problem as a minimization problem. Usually we call the objective function to be minimized the *loss function*. To maximize the log-likelihood we therefore use the *negative log-likelihood* (*nll*) as a loss function. The most common loss functions for neural networks are negative log-likelihood functions for different distributions. The mean squared error loss function equates to minimizing the nll for a Gaussian distribution, while mean absolute error does the same for a Laplace distribution.

2.1.4 Optimization

Now we have to find out how to change the parameters in the network so that we minimize the loss. For this we will use a method called gradient descent. We start by randomly initializing the parameters of the network, and then compute the gradients of the loss function with respect to each of the parameters. We will then change each parameter in the direction that minimizes the loss function.

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w})$$

η is the *learning rate* parameter that decides how much to move in the direction of the negative gradient. This is typically set to a small number e.g. 10^{-3}. Using the chain rule we can formulate the gradient as

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial n_j} \frac{\partial n_j}{\partial w_{ij}},$$

where n_j is the input to the activation function of neuron j. Because $n_j = \sum_{i=1}^{n} w_{ij} o_i$, the partial derivative $\frac{\partial n_j}{\partial w_{ij}}$ is simply the output of the previous neuron o_i. We are then

[1] Not just likelihood-based loss function

left with having to calculate $\frac{\partial L}{\partial n_j}$ for each neuron. For the output layer of neurons this is simply $\frac{\partial \mathcal{L}}{\partial \hat{y}_i}$, the derivative of the loss function with respect to the output. For an arbitrary neuron in the network, however, it is less obvious. For this we use a method called *backwards propagation of error*, or *backpropagation* for short. Figure 2.3 shows how we can create a recursive formulation for the partial derivative $\frac{\partial \mathcal{L}}{\partial n_j}$ by utilizing the partial derivatives for all the succeeding neurons (all neurons that depend on the value of this neuron).

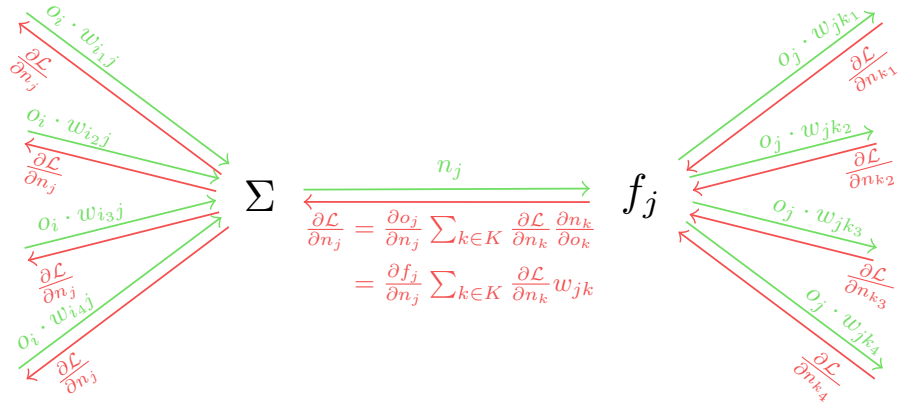


Figure 2.3: An illustration of backpropagation through one neuron.

First notice that

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial n_j} &= \frac{\partial o_j}{\partial n_j} \sum_{k \in K} \frac{\partial \mathcal{L}}{\partial n_k} \frac{\partial n_k}{\partial o_k} \\ &= \frac{\partial f_j}{\partial n_j} \sum_{k \in K} \frac{\partial \mathcal{L}}{\partial n_k} w_{jk},\end{aligned}$$

where K is the set of neurons that directly depend on the neuron's output o_j, and f_j is the neuron's activation function. This requires that the activation function is continuous everywhere and differentiable almost everywhere[2]. By iteratively applying this calculation for every neuron in the network, we are able to find the gradient of the loss function with respect to all the parameters.

When updating the weights of the network with this gradient we will move towards a minimum, where any small change of the parameters will increase the loss value.

[2] *Almost everywhere* is a term from measure theory. We use it to simply say that there is a countable amount of non-differentiable points.

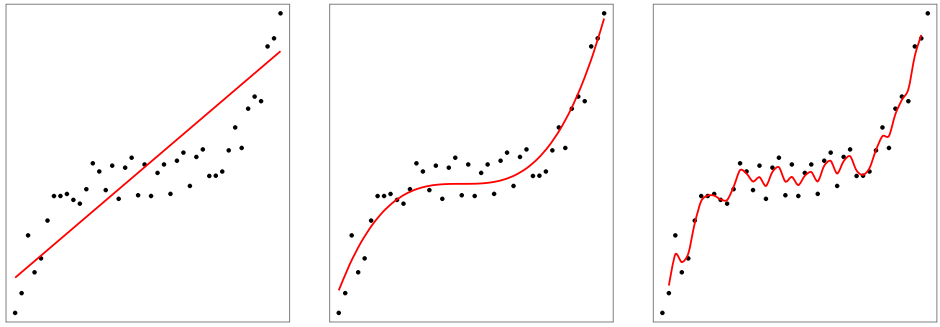


Figure 2.4: An illustration of three models fitted on a noisy 3rd degree polynomial with varying degrees of success.

Although this is a minimum, there might exist other minima too, so it is not necessarily the global minimum. This is a limitation to the gradient descent algorithm, but in neural networks it is generally agreed upon that most local minima are very close in loss-value to the global minimum [Choromanska et al., 2015], hence it is usually not necessary to reach a global minimum for good performance.

We now know how to find parameters of the network that will make it approximate a functional relationship between $\boldsymbol{x}$ and $\boldsymbol{y}$. Hopefully this function also generalizes to data that it has not seen before. This is the goal. Next we will look at how we can help the function generalize better to unseen data.

2.1.5 Regularization

We are now able to learn a functional relationship between inputs and outputs, but for this to be useful, we also need it to generalize to data that we have not seen during training. We want to make sure that our model is complex enough to be able to learn the relationship between the data points, but not fit the noise in the data. This is illustrated in Figure 2.4, where we can see that the *underfitted* model is unable to learn the complexity of the data, while the *overfitted* model is fitting the noise in the data. This will prevent the model from generalizing well to new data.

Underfitting is generally easy to solve in deep learning. We simply increase the number or size of the hidden layers in the model. This will make the model able to fit more complex functions, hence preventing underfitting. A greater challenge is to prevent the model from overfitting, while still remaining complex enough to be able to model the data. Here we employ what is called *regularization*. Regularization involves adding information to the learning process in order to restrict the function space of the

model.

One of the simplest regularization techniques in deep learning is to include an extra term in the loss function that penalizes parameters that are far away from zero. L1 and L2 regularization are two such methods illustrated below.

$$R_{L1}(\boldsymbol{\theta}) = \lambda \sum_{i=1}^{n} |\theta_i| \tag{2.2}$$

$$R_{L2}(\boldsymbol{\theta}) = \lambda \sum_{i=1}^{n} \theta_i^2 \tag{2.3}$$

The regularization term $R(\boldsymbol{\theta})$ is added to the existing loss function. This penalizes models with large parameters, hence imposing a trade-off between complexity and accuracy. $\lambda \in \mathbb{R}_+$ is a regularization parameter that controls the magnitude of the regularization. A larger value for λ will penalize large weights more,

Another common regularization technique is called *dropout*. Dropout involves selecting a random set of neurons every iteration dring training, and setting their output to zero. This ensures that the function cannot become overly reliant on certain neurons, since if that neuron is dropped then the model will completely fall apart. It therefore needs to model the trend in the data multiple times, so that they are not dropped, causing the model to focus less on fitting noise in the data. There exist many more regularization methods, but it is beyond the scope of this thesis to introduce them all.

2.2 Probabilistic AI

This section introduces the foundations of probabilistic AI. We introduce the statistical theory required to understand the state of the art methods introduced in Chapter 3 and the theoretical work in Chapter 4. We assume familiarity with probability theory and statistics.

2.2.1 Bayesian Inference

Bayesian statistics is one approach to statistics that views probability as a *degree of belief* in a certain event. This contrasts with the frequentist interpretation of probability, which sees probability is the relative frequency of events.

The process of deducing properties of a model underlying a distribution of data is called statistical inference. Bayesian inference views the parameters of the underlying model as random variables, rather than fixed values, as is the case in frequentist

inference. The distribution of the parameters is calculated using Bayes' theorem:

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathcal{D})}, \tag{2.4}$$

where $\boldsymbol{\theta}$ are the parameters of the chosen model, and $\mathcal{D}$ is the data.

$p(\boldsymbol{\theta}|\mathcal{D})$: The distribution of the parameter after we have observed data. The **posterior** (distribution).

$p(\mathcal{D}|\boldsymbol{\theta})$: The likelihood of the data being observed given the parameter. The **likelihood**.

$p(\boldsymbol{\theta})$: The probability over the parameters before observing any data. The **prior** (distribution).

$p(\mathcal{D})$: The probability of the data under any parameters. The **normalizing constant**, or **evidence**.

We consider both $p(\boldsymbol{\theta}|\mathcal{D})$ and $p(\mathcal{D}|\boldsymbol{\theta})$ to be be functions of $\boldsymbol{\theta}$. In that case the likelihood is a property of the model we choose to explain the data. It is not a probability, and does therefore not need to sum to one. The likelihood $p(\mathcal{D}|\boldsymbol{\theta})$ is however proportional to the probability $p(\boldsymbol{\theta}|\mathcal{D})$ divided the prior $p(\boldsymbol{\theta})$.

To use Bayesian inference in order to get a posterior distribution we must choose a prior distribution. The prior can be chosen based on domain knowledge, intuition, or as we will see later, as a regularization parameter. The choice of prior will affect the posterior distribution, though with more data the effect the prior has on the posterior will diminish. In the case where the likelihood comes from a very simple distribution, we can sometimes choose a prior that will cause the posterior and prior to have the same distribution family. The prior is then called a *conjugate prior* to that likelihood function. Conjugate priors are known for all exponential family distributions, but for deep learning, where the likelihood is defined by a neural network, we do not have conjugate priors. In Bayesian deep learning, we do not have a good idea of what the parameter values should be before we start training, but by using a prior distribution that pushes them closer to zero we restrict the freedom of the model, making it less likely to fit noise in the data.

The normalizing constant $p(\mathcal{D}) = \int p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})\,\mathrm{d}\boldsymbol{\theta}$, is generally intractable to compute. In the case where we have a conjugate prior, we can bypass the calculation of the normalizing constant, but in other cases we will we have to resort to approximate methods.

2.2.2 Rejection Sampling

Rejection sampling is a technique that can be used to sample from a posterior distribution, without knowing the normalizing constant. Here is an outline of the method:

Given a probability density function $f(\boldsymbol{x})$ with finite support, find its maximum value. Now create a finite bounding box encompassing all non-zero values of $f(\boldsymbol{x})$. Sample uniformly from within this bounding box, and accept a candidate x^* with probability $\frac{f(x^*)}{\max f}$. The non-rejected samples will be from the distribution $f(\boldsymbol{x})$. Intuitively this makes sense, as any less than $f(\boldsymbol{x}_1)$ will be half as likely to be accepted as a point less than $f(\boldsymbol{x}_2)$ if $f(\boldsymbol{x}_1) = 2f(\boldsymbol{x}_2)$. Now if $f(\boldsymbol{x})$ has infinite support, we cannot sample from a bounding box encompassing all non-zero values $f(\boldsymbol{x})$ as we just did. We instead find a function $cg(\boldsymbol{x})$, such that $f(\boldsymbol{x}) \leq cg(\boldsymbol{x})$ for all $\boldsymbol{x}$, and $g(\boldsymbol{x})$ is a pdf, and $1 < c < \infty$. $g(\boldsymbol{x})$ is called a *proposal distribution.* We can now sample points under $g(\boldsymbol{x})$, and accept with probability $\frac{f(\boldsymbol{x})}{cg(\boldsymbol{x})}$.

This method can sample points under the curve of any positive real function $f(\boldsymbol{x})$, making it a useful tool for sampling from a posterior distribution where the normalizing constant is unknown. We can see from the acceptance probability that a larger value for c will cause a lot of samples to be rejected, hence as we want c to be as close to 1 as possible. Finding suitable proposal distributions can be very difficult, thus this method is not widely used in practice.

2.2.3 Markov Chain Monte Carlo Methods

Markov Chain Monte Carlo (MCMC) methods are a family of algorithms used to sample from intractable posterior distributions. These algorithms all construct Markov chains with an equilibrium distribution equal to the target distribution. There are many methods for constructing Markov chains with this property. As our number of samples goes to infinity, the samples from these methods will perfectly fit the distribution we are sampling from. As opposed to rejection sampling, however, we don't have to find an encompassing function. Because of this it is usually used as a benchmark for other methods. Although the method is exact in the long run, the Markov chain introduces auto correlation between samples, meaning that a smaller number of samples will not be representative samples from the distribution. Most newer improvements to MCMC are attempts to reduce the auto correlation between samples, thus needing fewer samples for a good approximation.

Metropolis Hastings is an MCMC algorithm based on rejection sampling. Given a posterior distribution $p(\boldsymbol{\theta}|\mathcal{D})$, we use a conditional proposal distribution $g(\theta_t|\mathcal{D}; \theta_{t-1})$. This means that a sample proposal $\theta_t^{'}$ is conditioned on the previous sample θ_{t-1}.

The acceptance ratio is a product of the relative probability of the next sample compared to the previous sample, and the relative probability of getting that sample given the

Algorithm 1: Metropolis Hastings

Select an initial sample θ_0.
Define a proposal density $g(\theta_t|\mathcal{D};\theta_{t-1})$.

For n iterations:

1. Draw a proposal $\theta_t^{'} \sim g(\theta_t|\mathcal{D};\theta_{t-1})$.
2. Calculate the acceptance probability $A = \min\left\{\frac{p(\theta_t^{'}|\mathcal{D})g(\theta_t^{'}|\mathcal{D};\theta_{t-1})}{p(\theta_{t-1}|\mathcal{D})g(\theta_{t-1}|\mathcal{D};\theta_t^{'})}, 1\right\}$
3. Accept $\theta_t = \theta_t^{'}$ with probability A, otherwise $\theta_t = \theta_{t-1}$.

current sample and vice versa. We can simplify computation by choosing a symmetric proposal distribution, which would eliminate the latter ratio. Because the acceptance probability relies on the likelihood ratio we can approximate non-normalized posterior density functions. The downside of this method is that it introduces autocorrelation between samples. A wider proposal distribution will decrease autocorrelation, but will also decrease acceptance rate, hence there is a trade-off between the two. For a Gaussian proposal distribution Roberts et al. [1997] proved that an acceptance ratio of about ~ 0.23 is optimal. A lot of newer MCMC algorithms are relying on the gradient of the posterior to move further away from the current sample without sacrificing acceptance rate, and have generally proved much more efficient.

One of the most important points to take away from MCMC is that we can achieve an approximation to any degree of accuracy, as long as we have enough samples. Because of these two factors, it is often used as a gold standard to evaluate the performance of other methods. However, because MCMC only lets you sample from the posterior distribution, and not evaluate probabilities, it is not suitable for all tasks, and can be very slow at estimating properties such as mean and variance of a distribution.

2.2.4 Variational Inference

Variational inference is a faster inference method than MCMC, but as opposed to MCMC, does in general not have the property that it converges to the exact distribution. It is based on the asssumption that the posterior $p(\boldsymbol{\theta}|\mathcal{D})$ can be approximated by a *variational distribution* $q(\boldsymbol{\theta})$. By defining a dissimilarity measure between the posterior and the variational distribution $D[p(\boldsymbol{\theta}|\mathcal{D}); q(\boldsymbol{\theta})]$, we effectively have an optimization problem that we need to solve. By minimizing the dissimilarity measure we find the $q(\boldsymbol{\theta})$ that best approximates $p(\boldsymbol{\theta}|\mathcal{D})$. What 'best approximates' means depends on the dissimilarity measure.

A common choice of dissimilarity measure is the Kullback-Leibler (KL) divergence

[Kullback and Leibler, 1951], and is defined as

$$D_{KL}[q(\boldsymbol{\theta})\|p(\boldsymbol{\theta}|\mathcal{D}))] = \mathbb{E}_{\boldsymbol{\theta}\sim q}[-\log p(\boldsymbol{\theta}|\mathcal{D})] - \mathbb{E}_{\boldsymbol{\theta}\sim q}[-\log q(\boldsymbol{\theta})]. \tag{2.5}$$

You might notice that the KL-divergence relies on $p(\boldsymbol{\theta}|\mathcal{D})$, which is intractable; the whole reason we want to use variational inference to begin with. We will manipulate the form to get rid of it:

$$\begin{aligned} D_{KL}[q(\boldsymbol{\theta})\|p(\boldsymbol{\theta}|\mathcal{D})] &= \mathbb{E}_{\boldsymbol{\theta}\sim q}[-\log p(\boldsymbol{\theta}|\mathcal{D})] - \mathbb{E}_{\boldsymbol{\theta}\sim q}[-\log q(\boldsymbol{\theta})] \\ &= \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log q(\boldsymbol{\theta})] - \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log p(\boldsymbol{\theta}|\mathcal{D})] \\ &= \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log q(\boldsymbol{\theta})] - \mathbb{E}_{\boldsymbol{\theta}\sim q}\left[\log \frac{p(\boldsymbol{\theta},\mathcal{D})}{p(\mathcal{D})}\right] \\ &= \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log q(\boldsymbol{\theta})] - \mathbb{E}_{\boldsymbol{\theta}\sim q}\left[\log p(\boldsymbol{\theta},\mathcal{D})\right] + \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log p(\mathcal{D})] \\ &= \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log q(\boldsymbol{\theta})] - \mathbb{E}_{\boldsymbol{\theta}\sim q}\left[\log p(\boldsymbol{\theta},\mathcal{D})\right] + \log p(\mathcal{D}) \end{aligned} \tag{2.6}$$

We are still left with the intractable normalizing constant $p(\mathcal{D})$, but since this is now an optimization problem we can ignore it. Hence

$$D_{KL}[q(\boldsymbol{\theta})\|p(\boldsymbol{\theta}|\mathcal{D})] = \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log q(\boldsymbol{\theta})] - \mathbb{E}_{\boldsymbol{\theta}\sim q}\left[\log p(\boldsymbol{\theta},\mathcal{D})\right] + const. \tag{2.7}$$

We use Equation 2.7 to find the variational distribution $q^* \in \mathcal{Q}$ that minimizes the KL-divergence to the posterior, where $\mathcal{Q}$ is a family of distributions. How to choose $\mathcal{Q}$ is further discussed below.

$$\begin{aligned} q^* &= \arg\min_{q\in\mathcal{Q}} \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log q(\boldsymbol{\theta})] - \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log p(\boldsymbol{\theta},\mathcal{D})] \\ &= \arg\max_{q\in\mathcal{Q}} \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log p(\boldsymbol{\theta},\mathcal{D})] - \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log q(\boldsymbol{\theta})] \\ &= \arg\max_{q\in\mathcal{Q}} \textsc{elbo}(q) \end{aligned}$$

Here ELBO stands for *evidence lower bound*, from the fact that it create a lower bound for the logarithmic evidence $\log p(\mathcal{D})$. From Equation 2.6 we can see that

$$\begin{aligned} \log p(\mathcal{D}) - D_{KL}[q(\boldsymbol{\theta})\|p(\boldsymbol{\theta}|\mathcal{D})] &= \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log p(\boldsymbol{\theta},\mathcal{D})] - \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log q(\boldsymbol{\theta})] \\ &= \textsc{elbo}(q). \end{aligned} \tag{2.8}$$

Since the KL-divergence is always positive, this indeed shows that ELBO provides a lower bound for the log evidence. Additionally, since $\log p(\mathcal{D})$ does not depend on q, maximizing ELBO will minimize the KL-divergence.

Mean Field

We now have a way of approximating the posterior via maximizing the evidence lower bound for a variational distribution. Now we need to choose a variational distribution that will make this optimization problem computationally feasible. A mean field variational family refers to a class of multivariate distributions that can be expressed as a product of independent partitions of the latent space. Specifically for a mean field family $\mathcal{Q}$ that factorizes into m partitions, any $q \in \mathcal{Q}$ is such that

$$q(\boldsymbol{\theta}) = \prod_{i=1}^{m} q_i(\theta_i).$$

Typically $m = dim(\boldsymbol{\theta})$, but we sometimes have multidimensional partitions. This assumption means that the optimal $q^*(\boldsymbol{\theta})$, with respect to maximizing the evidence lower bound, is identical to the product of each optimal $q_i^*(\boldsymbol{\theta}|\mathcal{D})$. To see how this helps us, we start by noting that

$$\mathbb{E}_{\boldsymbol{\theta}\sim q}\left[\log q(\boldsymbol{\theta})\right] = \sum_{i=1}^{m} \mathbb{E}_{\boldsymbol{\theta}\sim q}\left[\log q_i(\theta_i)\right],$$

and through the chain rule we get that

$$\begin{aligned}\mathbb{E}_{\boldsymbol{\theta}\sim q}\left[p(\boldsymbol{\theta}, \mathcal{D})\right] &= \mathbb{E}_{\boldsymbol{\theta}\sim q}\left[\log\left(p(\mathcal{D})\prod_{i=1}^{m} p(\theta_i|\boldsymbol{\theta}_{1:(i-1)}, \mathcal{D})\right)\right] \\ &= \log p(\mathcal{D}) + \sum_{i=1}^{m} \mathbb{E}_{\boldsymbol{\theta}\sim q}\Big[\log p(\theta_i|\boldsymbol{\theta}_{1:(i-1)}, \mathcal{D})\Big].\end{aligned}$$

We can now substitute these into the definition of ELBO given in Equation 2.8 to get

$$\begin{aligned}
\textsc{elbo}(q) &= \mathbb{E}_{\boldsymbol{\theta}\sim q}\left[\log p(\boldsymbol{\theta}, \mathcal{D})\right] - \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log q(\boldsymbol{\theta})] \\
&= \log p(\mathcal{D}) + \sum_{i=1}^{m} \mathbb{E}_{\boldsymbol{\theta}\sim q}\Big[\log p(\theta_i|\boldsymbol{\theta}_{1:(i-1)}, \mathcal{D}) - \log q_i(\theta_i)\Big] \\
&= \sum_{i=1}^{m} \mathbb{E}_{\boldsymbol{\theta}\sim q}\Big[\log p(\theta_i|\boldsymbol{\theta}_{1:(i-1)}, \mathcal{D}) - \log q_i(\theta_i)\Big] + const \\
&= -\sum_{i=1}^{m} D_{KL}[q_i(\theta_i) \| p(\theta_i|\boldsymbol{\theta}_{1:(i-1)}, \mathcal{D})] + const.
\end{aligned}$$

This shows us that maximizing $\textsc{elbo}(q)$ over mean field is equivalent to maximizing $\textsc{elbo}(q_i)$ over $q_i(\theta_i)$, for each $i = 1, \ldots, m$:

$$\begin{aligned}
\arg\max_{q\in\mathcal{Q}} \textsc{elbo}(q) &= \sum_{i=1}^{m} \arg\max_{q_i\in\mathcal{Q}} \textsc{elbo}(q_i) \\
&= \sum_{i=1}^{m} \arg\max_{q_i\in\mathcal{Q}} \mathbb{E}_{\boldsymbol{\theta}\sim q}\Big[\log p(\theta_i|\boldsymbol{\theta}_{1:(i-1)}, \mathcal{D}) - \log q_i(\theta_i)\Big]. \qquad (2.9)
\end{aligned}$$

For each q_i, we will now consider $\textsc{elbo}$ as a function of that q_i instead of q. Each time we rearrange Equation 2.9 so that q_i is the last variable in the sum. Doing this lets us see that

$$\arg\max_{q_i\in\mathcal{Q}} \textsc{elbo}(q_i) = \arg\max_{q_i\in\mathcal{Q}} \mathbb{E}_{\boldsymbol{\theta}\sim q}\big[\log p(\boldsymbol{\theta}_i|\boldsymbol{\theta}_{-i}, \mathcal{D}) - \log q_i(\boldsymbol{\theta}_i)\big],$$

where $\boldsymbol{\theta}_{-i}$ means a vector containing all elements of $\boldsymbol{\theta}$ except element i. We can further rearrange this to get

$$\begin{aligned}
\arg\max_{q_i \in \mathcal{Q}} \textsc{elbo}(q_i) &= \arg\max_{q_i \in \mathcal{Q}} \mathbb{E}_{\boldsymbol{\theta} \sim q}\left[\log p(\theta_i | \boldsymbol{\theta}_{-i}, \mathcal{D})\right] - \mathbb{E}_{\theta_i \sim q_i}\left[\log q_i(\theta_i)\right] \\
&= \arg\max_{q_i \in \mathcal{Q}} \int_{\boldsymbol{\theta}} q(\boldsymbol{\theta}) \log p(\theta_i | \boldsymbol{\theta}_{-i}, \mathcal{D}) \, \mathrm{d}\boldsymbol{\theta} \\
&\qquad - \int_{\theta_i} q_i(\theta_i) \log q_i(\theta_i) \, \mathrm{d}\theta_i \\
&= \arg\max_{q_i \in \mathcal{Q}} \int_{\theta_i} q(\theta_i) \mathbb{E}_{\boldsymbol{\theta}_{-i} \sim q_{-i}}\left[\log p(\theta_i | \boldsymbol{\theta}_{-i}, \mathcal{D})\right] \mathrm{d}\theta_i \\
&\qquad - \int_{\theta_i} q_i(\theta_i) \log q_i(\theta_i | \mathcal{D}) \, \mathrm{d}\theta_i,
\end{aligned}$$

We now want to set the derivative of ELBO equal to zero, so that we can find its maximum. Note that $q_i s(\theta_i)$ is a PDF, hence $\int_{\theta_i} q_i(\theta_i) = 1$. We will use a Lagrange multiplier to include this constraint when finding the derivative.

$$\begin{aligned}
\textsc{elbo}(q_i) = &\int_{\theta_i} q(\theta_i) \mathbb{E}_{\boldsymbol{\theta}_{-i} \sim q_{-i}}\left[\log p(\theta_i | \boldsymbol{\theta}_{-i}, \mathcal{D})\right] \mathrm{d}\theta_i \\
&- \int_{\theta_i} q_i(\theta_i) \log q_i(\theta_i) \, \mathrm{d}\theta_i \\
&+ \lambda \left(\int_{\theta_i} q_i(\theta_i) \, \mathrm{d}\theta_i - 1 \right)
\end{aligned} \tag{2.10}$$

We defer the calculation of the derivative to the appendix, but note that it gives us the following expression for the optimal q_i^*

$$\begin{aligned}
q_i^*(\theta_i) &= \frac{\exp\left(\mathbb{E}_{\boldsymbol{\theta}_{-i} \sim q_{-i}}\left[\log p(\theta_i | \boldsymbol{\theta}_{-i}, \mathcal{D})\right]\right)}{\int_{\theta_i} \exp\left(\mathbb{E}_{\theta_i \sim q_{-i}}\left[\log p(\theta_i | \boldsymbol{\theta}_{-i}, \mathcal{D})\right]\right) \mathrm{d}\theta_i} \\
&\propto \exp\left(\mathbb{E}_{\boldsymbol{\theta}_{-i} \sim q_{-i}}\left[\log p(\theta_i, \boldsymbol{\theta}_{-i}, \mathcal{D})\right]\right).
\end{aligned} \tag{2.11}$$

We are now left with an optimization problem for each q_i that only uses the joint probability distribution $p(\boldsymbol{\theta}, \mathcal{D})$, which we know we can calculate. One way to solve this optimization problem is with the optimization algorithm coordinate ascent. A high level algorithm for mean field variational inference with coordinate ascent mean field variational inference [Ghahramani and Beal, 2001] is detailed in Algorithm 2. This algorithm updates each q_i separately, utilizing Equation 2.11. For exponential family distributions this update has a closed form, but cannot be used for an arbitrary variational family. When using coordinate ascent we need to make sure that when updating

q_i according to Equation 2.11 q_i is still in its distributional family. Usually this means that we have a distribution in the exponential family, such as a Gaussian distribution. Another downside of this algorithm is that we have to construct a new updating scheme for the parameters of q whenever we change its variational family.

Algorithm 2: Coordinate Ascent Mean Field Variational Inference

input: Data $\mathcal{D}$
Varitional distribution $q(\boldsymbol{\theta})$
Joint probability $p(\boldsymbol{\theta}, \mathcal{D})$

Initialize all variational distributions $q_1, \ldots, q_m$ with random parameters;
do
  for $i \leftarrow 0$ **to** m **do**
    Update q_i according to Equation 2.11;
  Calculate $\textsc{elbo}(q) = \mathbb{E}_{\boldsymbol{\theta}\sim q}\left[\log p(\boldsymbol{\theta}, \mathcal{D})\right] - \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log q(\boldsymbol{\theta})]$;
while *ELBO(q) has not converged*;

Black Box Variational Inference

To tackle the limitations of coordinate ascent Ranganath et al. [2013] employs stochastic optimization in an algorithm they call Black Box Variational Inference (BBVI). With BBVI we only have to make the assumption that we can evaluate $p(\boldsymbol{\theta}, \mathcal{D})$ almost everywhere, as well as sampling from q and evaluating its gradient with respect to its parameters. These assumptions are much weaker than those for coordinate ascent variational inference. We would like to calculate $\nabla\textsc{elbo}(q) = \nabla\mathbb{E}_{\boldsymbol{\theta}\sim q}\left[\log p(\boldsymbol{\theta}, \mathcal{D})\right] - \nabla\mathbb{E}_{\boldsymbol{\theta}\sim q}[\log q(\boldsymbol{\theta})]$, so that we can use gradient descent as an optimization scheme, but even though we can evaluate $p(\boldsymbol{\theta}, \mathcal{D})$, we have made no assumption that we can evaluate $\mathbb{E}_{\boldsymbol{\theta}\sim q}\left[\log p(\boldsymbol{\theta}, \mathcal{D})\right]$, let alone its gradient. This is where we resort to stochastic optimization.

Stochastic optimization generalizes gradient descent to noisy gradients. Let $f(\boldsymbol{x})$ be the function that we wish to optimize, and $G(\boldsymbol{x})$ be a random variable with expectation $\mathbb{E}[G(\boldsymbol{x})] = \nabla f(\boldsymbol{x})$. To update $\boldsymbol{x}$ at step t would then look as follows

$$
\begin{aligned}
g(\boldsymbol{x}_t) &\sim G(\boldsymbol{x}_t) \\
\boldsymbol{x}_t &\leftarrow \boldsymbol{x}_t + \rho_t g(\boldsymbol{x}_t),
\end{aligned}
$$

where ρ_t is the learning rate at step t. Robbins and Monro [1951] showed that this converges to a local maximum of f if ρ_t satisfies the following conditions

$$\sum_{t=1}^{\infty} \rho_t = \infty$$
$$\sum_{t=1}^{\infty} \rho_t^2 < \infty.$$

One learning rate ρ_t satisfying these condition suggested by Robbins and Monro [1951] is $\rho_t = \frac{1}{t}$.

To use stochastic optimization to maximize $\text{ELBO}(q)$, BBVI needs to define an unbiased estimator for the gradient $\nabla\text{ELBO}(q)$. They derive the following expression

$$\nabla\text{ELBO}(q) = \mathbb{E}_{\boldsymbol{\theta}\sim q}\left[\nabla \log q(\boldsymbol{\theta})(\log p(\boldsymbol{\theta}, \mathcal{D}) - \log q(\boldsymbol{\theta}))\right].$$

With this estimator we can compute an unbiased estimate for the gradient with Monte Carlo samples.

$$\nabla\text{ELBO}(q) \approx \frac{1}{S}\sum_{s=1}^{S}\left[\nabla \log q(\theta_s)(\log p(\theta_s, \mathcal{D}) - \log q(\theta_s))\right],$$
$$\text{where } \theta_s \sim q(\boldsymbol{\theta}).$$

By combining this stochastic optimization scheme with the mean field assumption we get Algorithm 3. Ranganath et al. [2013] goes on to further improve this algorithm with methods to reduce the variance of the estimator for the gradient, but the details for the improved algorithm is outside the scope of this thesis.

2.2.5 Normalizing Flows

This section will show how we can increase the flexibility of a simple variational distribution. We present Normalizing Flows [Rezende and Mohamed, 2016] as background for a state of the art Bayesian neural network model presented in Section 3.2.

Change of Variables in Probability Density Functions

The change of variables formula tells us that if we have two random variables, $\boldsymbol{\theta}$ and $\mathbf{z} \sim q_{\mathbf{z}}$, where $\boldsymbol{\theta} = f(\mathbf{z})$ with $f : \mathbb{R}^n \to \mathbb{R}^n$ being a bijective, differentiable function, then

Algorithm 3: Black Box Variational Inference

input: Data $\mathcal{D}$
Varitional distribution $q(\boldsymbol{\theta})$
Joint probability $p(\boldsymbol{\theta}, \mathcal{D})$
Initialize all variational distributions $q_1, \ldots, q_m$ with random parameters;
$t \leftarrow 0$;
do
  for $s \leftarrow 0$ **to** S **do**
    $\boldsymbol{\theta}[s] \sim q$
  for $i \leftarrow 0$ **to** m **do**
    // Let λ_i denote the parameters of distribution q_i
    $\lambda_i \leftarrow \lambda_i + \rho_t \nabla_{\lambda_i} \sum_{s=1}^{S} \log q_i(\boldsymbol{\theta}[s]) \left(\log p(\mathcal{D}, \boldsymbol{\theta}[s]) - \log q_i(\boldsymbol{\theta}[s])\right)$
  $t \leftarrow t + 1$;
  Calculate $\textsc{elbo}(q) = \mathbb{E}_{\boldsymbol{\theta} \sim q}[\log q(\boldsymbol{\theta})] - \mathbb{E}_{\boldsymbol{\theta} \sim q}\left[\log p(\boldsymbol{\theta}, \mathcal{D})\right]$;
while *ELBO(q) has not converged*;

$$q_{\boldsymbol{\theta}}(\boldsymbol{\theta}) = q_{\mathbf{z}}\left(f^{-1}(\boldsymbol{\theta})\right) \left| \det \left(\frac{\partial f^{-1}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right) \right| \tag{2.12}$$

$$\log q_{\boldsymbol{\theta}}(\boldsymbol{\theta}) = \log q_{\mathbf{z}}\left(f^{-1}(\boldsymbol{\theta})\right) + \log \left| \det \left(\frac{\partial f^{-1}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right) \right|.$$

Using $\mathbf{z} = f^{-1}(\boldsymbol{\theta})$, and $\det\left(A^{-1}\right) = \det(A)^{-1}$, we can alternatively formulate Equation 2.12 as

$$q_{\boldsymbol{\theta}}(\boldsymbol{\theta}) = q_{\mathbf{z}}(\mathbf{z}) \left| \det \left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \right) \right|^{-1} \tag{2.13}$$

$$\log q_{\boldsymbol{\theta}}(\boldsymbol{\theta}) = \log q_{\mathbf{z}}(\mathbf{z}) - \log \left| \det \left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \right) \right|.$$

Given that we know f^{-1}, Equation 2.12 makes it is easy to evaluate the likelihood of a sample $\boldsymbol{\theta}$; you simply insert the value into the equation. If we only have the inverse transformation, however, it is very difficult to sample from $\boldsymbol{\theta}$ since we cannot simply

calculate $\theta = f(\mathbf{z})$. This is where Equation 2.13 comes in handy. Here $\mathbf{z}$ is typically a random variable that is easy to sample from, such as from a Gaussian distribution. We can then sample $\boldsymbol{z} \sim \mathbf{z}$ and calculate $\theta = f(\boldsymbol{z})$, while at at the same time evaluating its likelihood with Equation 2.13. This means that even when we do not have access to f^{-1}, we can still evaluate the likelihood of samples. We need Equation 2.12 if we want to calculate probabilities of events.

Normalizing Flow Models

Normalizing flow models [Rezende and Mohamed, 2016] takes advantage of change of variables to create very complex distributions. If we consider the mapping between θ and $\mathbf{z}$ to be a function $f_{\boldsymbol{\lambda}}$ parameterized by $\boldsymbol{\lambda}$, then we get

$$\log q_{\theta}(\theta) = \log q_{\mathbf{z}}(\mathbf{z}) - \log \left| \det \left(\frac{\partial f_{\boldsymbol{\lambda}}(\mathbf{z})}{\partial \mathbf{z}} \right) \right|.$$

We can now see that q_{θ} can be a very complex variational distribution, given the right parameterized function $f_{\boldsymbol{\lambda}}$. We can imagine this being very useful in black box variational inference to better approximate multi-modal or otherwise complex posterior distributions. To better understand normalizing flows we will look at what the words mean.

1. *Normalizing* comes from the fact that after the transformation we end up with a normalized distribution.

2. *Flow* comes from how we can chain transformations together to create arbitrarily complex distributions. The transformations create a "flow" of random variables.

We will look at why we want to chain simple transformations together. Consider a function

$$f_{\boldsymbol{\lambda}}(\mathbf{z}) = f_{\boldsymbol{\lambda}_n} \circ f_{\boldsymbol{\lambda}_{n-1}} \circ \cdots \circ f_{\boldsymbol{\lambda}_1}(\mathbf{z}).$$

We can now have a complex transformation composed of multiple parameterized transformations. Typically the same transformation is repeated multiple times with different parameters, but we can also compose completely different transformations. Considering the formulation of Equation 2.13, all we need is an analytical form for $\det \left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \right)$, and then we can sample a variable $\mathbf{z}$, and iteratively apply the transformation. To calculate the likelihood of that sample we can iteratively apply Equation 2.13 to obtain

$$\log q_{\boldsymbol{\theta}}(\boldsymbol{\theta}) = \log q_{\mathbf{z}}(\mathbf{z}_0) - \sum_{i=1}^{n} \log \left| \det \left(\frac{\partial f_{\boldsymbol{\lambda}_i}(\mathbf{z}_{i-1})}{\partial \mathbf{z}_{i-1}} \right) \right|.$$

Here $\mathbf{z}_i$ refers to the result after an initial sample $\mathbf{z}_0$ has passed through i transformations. That means that $\boldsymbol{\theta} = \mathbf{z}_n$.

Variational inference finds a $q \in \mathcal{Q}$ that maximizes ELBO and normalizing flows gives us a way to extend the variational family $\mathcal{Q}$. With a sufficiently complex transformation we $\min_{q \in \mathcal{Q}} D_{\mathrm{KL}}[q \| p] \to 0$. In other words, our variational approximation could in theory converge to the true posterior.

Planar Flow

One transformation suggested by Rezende and Mohamed [2016] is planar flow:

$$f(\mathbf{z}) = \mathbf{z} + \boldsymbol{u} h(\boldsymbol{w}^T \mathbf{z} + b), \tag{2.14}$$

where $\boldsymbol{u}, \boldsymbol{w} \in \mathbb{R}^d$, and $b \in \mathbb{R}$ are the parameters; h is a non-linear continuously differentiable function. For this transformation we have

$$\left| \det \left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \right) \right| = |1 + h'(\boldsymbol{w}^T \mathbf{z} + b) \boldsymbol{u}^T \boldsymbol{w}|.$$

Not all transformations of the form Equation 2.14, are invertible. Rezende and Mohamed [2016] shows that for $h(x) = \tanh(x)$, $f(\mathbf{z})$ is invertible when $w^T u \geq -1$. They further go into detail about how to enforce this constraint, but we consider the details for that as outside the scope of this thesis. Still, even when the inverse exists it is usually not easy to compute analytically, meaning we cannot evaluate integrals over $\boldsymbol{\theta}$. Figure 2.5 shows how a single planar transformation can transform a normal distribution.

Real-Valued Non-Volume Preserving Flow

Dinh et al. [2017] presents a class of invertible transformations with tractable Jacobians which they call Real-Valued Non-Volume Preserving (RealNVP) flow. We will first dissect the name. If $\left| \det \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \right| = 1$, then f is called a *volume preserving* flow. This means that $q_{\boldsymbol{\theta}}(\boldsymbol{\theta}) = q_{\mathbf{z}}(\mathbf{z})$ with $\boldsymbol{\theta} = f(\mathbf{z})$, hence the transformed random variable still integrates to 1 and is thus volume preserving. RealNVP is based on a volume preserving transformation by Dinh et al. [2015], which they call an additive coupling layer. Consider the following transformation

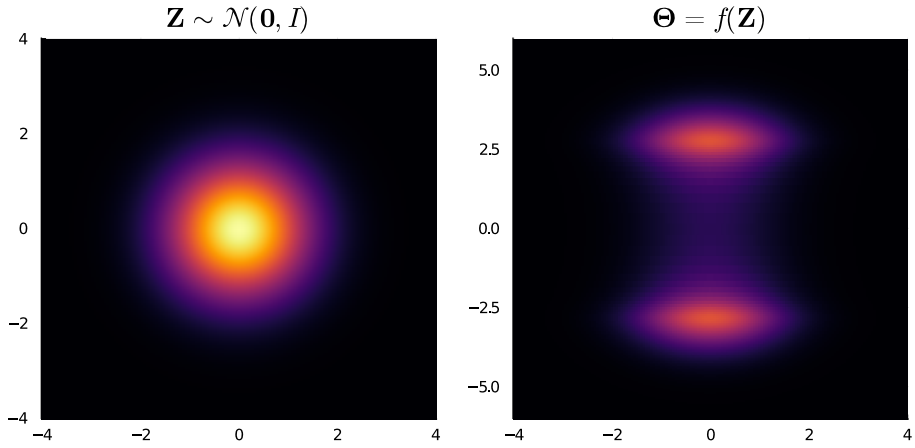


Figure 2.5: Shows $q_{\mathbf{z}}$ for a standard normal distribution on the left together with $q_{\theta}\left(f(\mathbf{z})\right)$, the result of a planar flow transformation, with parameters $\boldsymbol{u} = \begin{bmatrix}2 & 0\end{bmatrix}^T$, $\boldsymbol{w} = \begin{bmatrix}3 & 0\end{bmatrix}^T$, $b = 0$, $h = \tanh$.

$$\begin{aligned}\theta_{1:k} &= \mathbf{z}_{1:k}\\ \theta_{k+1:d} &= \mathbf{z}_{k+1:d} + m_{\boldsymbol{\lambda}}(\mathbf{z}_{1:k}).\end{aligned}$$

Here we have split θ and $\mathbf{z}$ into two disjoint subsets of size k and $d-k$. $m_{\boldsymbol{\lambda}}$ is called the coupling function. We will use a neural network as this function. This transformation has a very simple Jacobian determinant. Because $\frac{\partial f_i}{z_j} = 0$ for $i < j$ we have a lower triangular Jacobian. This means that the Jacobian determinant is just the product of the diagonal entries. Furthermore, we have $\frac{\partial f_i}{\partial \mathrm{z}_i} = 1$, so the Jacobian determinant $\det\left(\frac{\partial f}{\partial \mathbf{z}}\right) = \prod_{i=1}^{d} \frac{\partial f_i}{\partial \mathrm{z}_i} = 1$. We also have a tractable inverse for this transformation, namely

$$\begin{aligned}\mathbf{z}_{1:k} &= \theta_{1:k}\\ \mathbf{z}_{k+1:d} &= \theta_{k+1:d} - m_{\boldsymbol{\lambda}}(\theta_{1:k}).\end{aligned}$$

We can see that the inverse, or reverse mapping, is no more difficult to compute than the forward mapping. Because the Jacobian determinant of this transformation equals 1 this is volume-preserving. RealNVP combines this mapping with a scaling function in the following fashion

$$\begin{aligned}\boldsymbol{\theta}_{1:k} &= \mathbf{z}_{1:k} \\ \boldsymbol{\theta}_{k+1:d} &= \mathbf{z}_{k+1:d} \odot \exp\left(s_{\boldsymbol{\lambda}_1}(\mathbf{z}_{1:k})\right) + m_{\boldsymbol{\lambda}_2}(\mathbf{z}_{1:k}),\end{aligned}$$

where $\odot$ denotes element-wise multiplication. The Jacobian determinant for this transformation is simply $\exp\left(\sum_{i=1}^{d-k} s_{\boldsymbol{\lambda}_1}(\mathbf{z}_{1:k})_i\right)$. It still does not depend on a derivative of s or m, so we can let those functions be arbitrarily complex. The reverse mapping is still just as simple

$$\begin{aligned}\mathbf{z}_{1:k} &= \boldsymbol{\theta}_{1:k} \\ \mathbf{z}_{k+1:d} &= \left(\boldsymbol{\theta}_{k+1:d} - m_{\boldsymbol{\lambda}_2}(\boldsymbol{\theta}_{1:k})\right) \odot \exp\left(-s_{\boldsymbol{\lambda}_1}(\boldsymbol{\theta}_{1:k})\right).\end{aligned}$$

What is important about these coupling transformations is that they only alter some dimensions at a time. This means that we *have* to compose multiple transformations where $\mathbf{z}$ is split into different subsets in order to transform all the dimensions. Dinh et al. [2015] found that you have to compose at least 3 transformations in order for all dimensions to be able to affect one another. By chaining together these transformations we are able to create arbitrarily complex variational distributions that we can both sample from as well as evaluate. Normalizing Flows can give us complex variational distributions to use for the Black Box Variational Inference algorithm.

2.3 Generative Adverserial Networks

In 2014, Ian Goodfellow and his colleagues [Goodfellow et al., 2014] invented a machine learning algorithm capable of generating photo-realistic images through what they called a generative adversarial network (GAN). Although most commonly used for generating and manipulating photos and videos, it is a method that can be used as a generative model for any sort of data. The method was originally proposed as an unsupervised algorithm, but has since been extended to be used for both supervised and reinforcement learning. In this section we will present the original GAN by Goodfellow et al. [2014], along with some of the main improvements that has has been proposed since.

2.3.1 Method

A generative adversarial network is comprised of two parts, a generator (G) and a discriminator (D). Each part is a neural network. The generator takes as input random

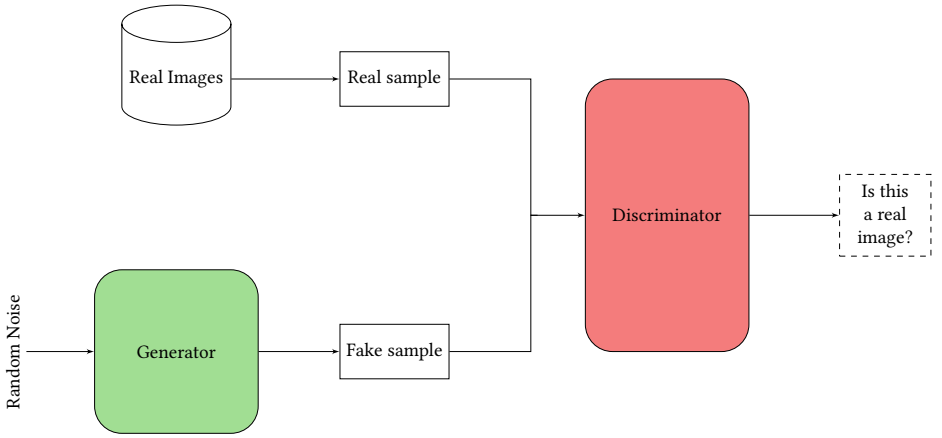


Figure 2.6: An illustration of the structure of a generative adversarial network.

noise and outputs a candidate. In the context of image generation, the candidate will be an image. The discriminators job is to evaluate whether this image is real or generated. Figure 2.6 illustrates the structure of a generative adversarial network model. The discriminator will output a number between 0 and 1, indicating whether it believes it is seeing a fake or a real image. A number closer to 0 means it is more sure that it is seeing a fake image, and a number close to 1 means that it predicts more strongly that it is a real image.

Loss function

Because we have two neural networks to train, we also need two loss functions. Those loss functions are as follows

$$\text{Discriminator loss: } -\mathbb{E}_z[\log(1 - D(G(\mathbf{z})))] - \mathbb{E}_{\mathbf{x}}[\log(D(\mathbf{x}))] \tag{2.15}$$

$$\text{Generator loss: } \quad \mathbb{E}_{\mathbf{z}}[\log(1 - D(G(\mathbf{z})))], \tag{2.16}$$

where G is the generator function, D is the discriminator function, $\mathbf{z}$ is the random noise input to the generator, and $\boldsymbol{x}$ is the real image. The loss functions reflect that the discriminator tries to learn to distinguish between real and fake images, while the generator tries to fool the discriminator. In the original paper [Goodfellow et al., 2014] they found that in the early phases of training, when the discriminators job is very easy, it can get stuck and stop learning. This happens because the gradient of the generators loss function is too small when the discriminator is very good. To tackle this issue they

proposed an alternative loss function for the generator:

$$\text{Alternative generator loss: } -\mathbb{E}_{\mathbf{z}}[\log(D(G(\mathbf{z})))]. \tag{2.17}$$

Because $\log$ has a much larger gradient near 0 than 1, when the discriminator becomes very good, this alternative generator loss will have a larger gradient, hence combating the vanishing gradient problem.

Training

Generally when training GANs we alternate between training the discriminator and the generator. Because both loss functions need $D(G(\mathbf{z}))$, this value can be shared if the networks are trained alternatingly. If either network starts dominating however, it can be more efficient to train one network more often than the other. Arjovsky et al. [2017] found that this was particularily effective when using an alternative loss function called the Wasserstein distance. GANs are notorious for how difficult they are to train. One problem is that it is difficult to know when the training has converged. In a standard neural network, the loss function will stabilize at a low level, and you know that you have reached a local minimum. Because GANs have two competing loss functions, the value will not always converge, but can keep oscillating. Another thing that makes GANs hard to train is that the model can be subject to mode collapse. This means that instead of learning a distribution over the entire dataset, it learns to generate very realistic datapoints in a small portion of the dataset. The discriminator will then learn that it is seeing too many samples from this portion of the sample-space, and start predicting anything from that subset as fake. This will punish the generator for only replicating a small portion of the dataset, but what often happens is that it then starts generating samples from a different portion of the distribution.

2.4 Bayesian Neural Networks

In this section we will define Bayesian neural networks, and discuss the differences between these and standard neural networks. We reserve the discussion of specific implementations of Bayesian neural networks for Chapter 3.

The term *Bayesian neural networks* refers to an extension of standard neural networks, that treat each weight as a random variable. We will get to what this means, but we first need to look at standard neural networks from a probabilistic standpoint. A standard neural network can be seen as a probabilistic model $p(\hat{\boldsymbol{y}}|\boldsymbol{\theta}, \boldsymbol{x})$, that given a set of weights $\boldsymbol{\theta}$ and an input vector $\boldsymbol{x}$, outputs a value $\hat{\boldsymbol{y}}$. When training a standard neural network, what we do is to try to find $\arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{n} p(\boldsymbol{y}_i|\boldsymbol{\theta}, \boldsymbol{x}_i)$; the parameters $\boldsymbol{\theta}$, that makes the network best predict the training data. That means that we want $\hat{\boldsymbol{y}}$ to

be the value $\boldsymbol{y}$ that we expect to see given $\boldsymbol{x}$. This is also called maximum likelihood estimation.

In Bayesian neural networks, the probabilistic interpretation of the model is $p(\boldsymbol{\theta}|\boldsymbol{x}, \boldsymbol{y})$. Instead of finding the parameters that maximizes the likelihood that the model's output is the true output value, we try to find out for every set of parameters how likely it is that a model with these parameters generated the data that we are looking at. The important difference is that Bayesian neural networks give us a random variable as an output instead of a fixed point. This means that when we $\boldsymbol{\theta}$ to make our prediction, $p(\hat{\boldsymbol{y}}|\boldsymbol{\theta}, \boldsymbol{x})$ will also have a distribution. This gives us the very valuable ability to see how certain the network is with its predictions. The standard network will give an output $\hat{\boldsymbol{y}}$, but we do not know if this is something close to the data it has trained on, or if it is making unjustified extrapolations. In this case a Bayesian neural network will give a high uncertainty if it has not seen any training data that is similar to what it is predicting. This makes Bayesian neural networks particularly good for small data sets.

For standard neural networks one can usually find a simple loss function, that will steer the gradient descent towards a local maximum likelihood. Usually a negative log likelihood is employed as a loss function, and for many distributions this is a simple expression. In Bayesian neural networks, the natural loss function, the posterior probability $-\log p(\boldsymbol{\theta}|\mathcal{D})$ is not computationally tractable. This means that we have to resort to approximate methods. These methods generally build on variational inference, which we discussed in Section 2.2.4. We discuss the specifics of these methods in Chapter 3, but we will present the general structure behind those methods here.

First, we start by defining some variational family of distributions. We denote this and element in this family by $q(\boldsymbol{\theta}|\boldsymbol{\lambda}) \in \mathcal{Q}_{\boldsymbol{\lambda}}$, where $\boldsymbol{\lambda} \in \Omega$ is the set of indices in the variational family. For simplicity we usually omit $\boldsymbol{\lambda}$, and write $q(\boldsymbol{\theta})$. Now that we have a variational family of distributions we will try to find the distribution in this family that has the minimum KL-divergence to the true posterior. To do this we employ the evidence lower bound introduced in Section 2.2.4. We replace our definition of $q^*(\mathbf{z})$ with $q(\mathbf{z}|\boldsymbol{\lambda}^*)$, where $\boldsymbol{\lambda}^*$ is

$$\boldsymbol{\lambda}^* = \arg\max_{\boldsymbol{\lambda} \in \Omega} \text{ELBO}(q(\mathbf{z}|\boldsymbol{\lambda}))$$

The terms in the evidence lower bound can be easily approximated through Monte Carlo simulations, by simply running the network multiple times over the same data and averaging the loss function. All the methods discussed in Chapter 3 follows this structure. The difference is how they represent the variational distribution, and how they approximate the evidence lower bound. In chapter 4 we will present a new method for approximating the posterior in Bayesian neural networks.

Chapter 3

State of the Art

In this chapter we will introduce some state of the art Bayesian deep learning algorithms, along with their respective pros and cons. These methods build on the background presented in Section 2.2, particularly on variational inference discussed in Subsection 2.2.4, as well as the introduction to Bayesian neural networks in Section 2.4.

3.1 Bayes by Backprop

Bayes by Backprop is a method by Blundell et al. [2015] for approximating the posterior distribution in a neural network. The method assumes that independent Gaussian distributions on each weight is sufficient to approximate the posterior distribution. By making this assumption, variational inference can be used to fit the marginal Gaussian distributions to the true posterior. Our aim is then to minimize the KL-divergence between the posterior and the variational distribution. We can formulate that in the following way:

$$\begin{aligned}
\mathcal{F}(\mathcal{D}, \boldsymbol{\lambda}) &= D_{\mathrm{KL}}(q(\boldsymbol{\theta}|\boldsymbol{\lambda}) \| p(\boldsymbol{\theta}|\mathcal{D})) \\
&\propto \int q(\boldsymbol{\theta}|\boldsymbol{\lambda}) \log \frac{q(\boldsymbol{\theta}|\boldsymbol{\lambda})}{p(\boldsymbol{\theta}) p(\mathcal{D}|\boldsymbol{\theta})} \, \mathrm{d}\boldsymbol{\theta} \\
&\propto D_{\mathrm{KL}}[q(\boldsymbol{\theta}|\boldsymbol{\lambda}) \| p(\boldsymbol{\theta})] - \mathbb{E}_q[\log p(\mathcal{D}|\boldsymbol{\theta})] \\
&\propto \mathbb{E}_q \left[\log q(\boldsymbol{\theta}|\boldsymbol{\lambda}) - \log p(\boldsymbol{\theta}) - \log p(\mathcal{D}|\boldsymbol{\theta}) \right]
\end{aligned} \tag{3.1}$$

where $\boldsymbol{\theta}$ are the (sampled) weights of the network, and $\boldsymbol{\lambda}$ are the parameters ($\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$) of the variational distribution. This optimization function can be approximated by

Monte Carlo sampling by the following sampling procedure

$$\mathcal{F}(\mathcal{D}, \boldsymbol{\lambda}) \approx \frac{1}{n} \sum_{i=1}^{n} \log q(\boldsymbol{\theta}^{(i)}|\boldsymbol{\lambda}) - \log p(\boldsymbol{\theta}^{(i)}) - \log p(\mathcal{D}|\boldsymbol{\theta}^{(i)}),$$

where $\boldsymbol{\theta}^{(i)}$ is the i-th sample from the variational posterior $q(\boldsymbol{\theta}^{(i)}|\boldsymbol{\lambda})$. To be able to differentiate $\mathcal{F}(\mathcal{D}, \boldsymbol{\lambda})$ with respect to $\boldsymbol{\lambda}$, a few tricks has to be employed. First we use a reparameterization trick, that will allow us to differentiate a normal distribution with respect to its mean and standard deviation, as shown in Equation 3.2. The random variable is now independent on the parameters we differentiate on, so straightforward differentiation will work for both parameters.

$$\mathcal{N}(\mu, \sigma^2) = \mu + \epsilon\sigma, \quad \text{where } \epsilon \sim \mathcal{N}(0, 1). \tag{3.2}$$

Second, we do not actually differentiate with respect to the standard deviation σ. Instead we try to optimize a parameter ρ such that $\sigma = \log(1 + \exp(\rho))$. This transformation ensures that we can only fit a valid range for σ, i.e. the interval $(0, \infty)$. It also increases precision for representing small values of σ.

Blundell et al. [2015] found that a sample size of one was sufficient for the method to converge to a good local minimum. Algorithm 4 shows the Bayes by Backprop procedure for training a Bayesian neural network with one sample for each Monte Carlo approximation.

One of the strengths of this algorithm compared to the other algorithms we will discuss is that it is very fast. It only requires twice as many parameters as compared to a regular non-Bayesian neural network for the same model structure. The number of forward and backward operations is also not significantly increased. Its limitation is that it is only able to express a posterior that consists of independent Gaussian distributions on each weight. Subsequent methods have generally been trying to deal with this limitation and increase the posterior space beyond independent Gaussian distributions. This is also the goal of the algorithm we propose in Chapter 4.

3.2 Multiplicative Normalizing Flow

Multiplicative normalizing flow [Louizos and Welling, 2017] can be seen as an extension of Bayes by Backprop [Blundell et al., 2015]. It starts with the same posterior approximation of independent Gaussian distributions for the posterior, but then applies normalizing flows to augment the posterior to form a more accurate approximation. Normalizing flows are discussed in Section 2.2.5.

Algorithm 4: Bayes by Backprop (taken from Blundell et al. [2015])

For each iteration:

1. Sample $\boldsymbol{\epsilon} \sim \mathcal{N}(0, I)$
2. Let $\theta = \boldsymbol{\mu} + \log(1 + \exp(\boldsymbol{\rho})) \circ \boldsymbol{\epsilon}$
3. Let $\boldsymbol{\lambda} = (\boldsymbol{\mu}, \boldsymbol{\rho})$
4. Let $f(\theta, \boldsymbol{\lambda}) = \log q_{\boldsymbol{\lambda}}(\theta) - \log p(\theta) - \log p(\mathcal{D}|\theta)$
5. Calculate the gradient with respect to the mean
$$\boldsymbol{\Delta}_{\boldsymbol{\mu}} = \frac{\partial f(\theta, \boldsymbol{\lambda})}{\partial \theta} + \frac{\partial f(\theta, \boldsymbol{\lambda})}{\partial \boldsymbol{\mu}}$$
6. Calculate the gradient with respect to the standard deviation parameter $\boldsymbol{\rho}$
$$\boldsymbol{\Delta}_{\boldsymbol{\rho}} = \frac{\partial f(\theta, \boldsymbol{\lambda})}{\partial \theta} \frac{\boldsymbol{\epsilon}}{1 + \exp(-\boldsymbol{\rho})} + \frac{\partial f(\theta, \boldsymbol{\lambda})}{\partial \boldsymbol{\rho}}$$
7. Update the variational parameters
$$\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} - \eta \boldsymbol{\Delta}_{\boldsymbol{\mu}}$$
$$\boldsymbol{\rho} \leftarrow \boldsymbol{\rho} - \eta \boldsymbol{\Delta}_{\boldsymbol{\rho}}$$

Normalizing flows need the Jacobian of the transformation to calculate the log-likelihood. If we use normalizing flows directly on the weights in the neural network, the Jacobian will have dimension $|\boldsymbol{w}|^2$. This matrix would not be tractable to compute or even store. To keep the computations tractable, they needed to limit the dimensions of the normalizing flow. For this they chose to parameterize the posterior in the following way

$$\mathbf{z} \sim q_{\boldsymbol{\lambda}}(\mathbf{z})$$
$$\theta \sim q_{\boldsymbol{\phi}}(\theta|\mathbf{z}),$$

where $q_{\boldsymbol{\lambda}}(\mathbf{z})$ is a density that has been modified by the normalizing flow. $\boldsymbol{\lambda}$ are the parameters of the normalizing flow, and $\boldsymbol{\phi}$ are the mean and standard deviation parameters of $q_{\boldsymbol{\phi}}(\theta|\mathbf{z})$ defined as

$$q_{\boldsymbol{\phi}}(\theta|\mathbf{z}) = \prod_{i=1}^{D_{in}} \prod_{j=1}^{D_{out}} \mathcal{N}(z_i \mu_{ij}, \boldsymbol{\Sigma}_{ij}^2), \tag{3.3}$$

where D_{in} and D_{out} are the number of input and output nodes for the layer, respectively. This parameterization scheme is illustrated in Figure 3.1. Applying normalizing flow to transform $q(\mathbf{z})$ is possible because $\mathbf{z}$ is much lower dimension than θ. The

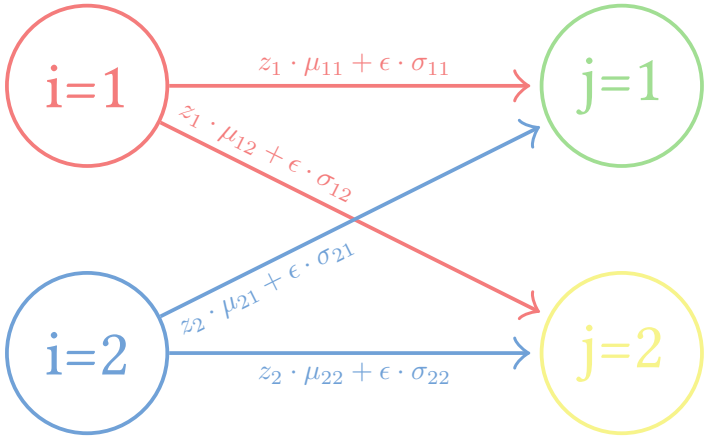


Figure 3.1: Illustration of weight parameterization in Multiplicative Normalizing Flows

distribution we are interested in is the variational posterior $q(\boldsymbol{\theta})$. To get this they approximate

$$q(\boldsymbol{\theta}) = \int q_{\boldsymbol{\phi}}(\boldsymbol{\theta}|\mathbf{z}) q_{\boldsymbol{\lambda}}(\mathbf{z}) \, \mathrm{d}\mathbf{z}. \tag{3.4}$$

through Monte Carlo sampling. They can now reformulate the KL-divergence between the variational posterior and the true posterior as follows

$$\begin{aligned}
\mathcal{F}(\mathcal{D}, \mathbf{z}, \boldsymbol{\theta}) &= D_{\mathrm{KL}}(q(\boldsymbol{\theta}) \| p(\boldsymbol{\theta}|\mathcal{D})) \\
&\propto \int q(\boldsymbol{\theta}) \log \frac{q(\boldsymbol{\theta})}{p(\boldsymbol{\theta}) p(\mathcal{D}|\boldsymbol{\theta})} \\
&\propto D_{\mathrm{KL}}(q(\boldsymbol{\theta}) \| p(\boldsymbol{\theta})) - \mathbb{E}_q[\log p(\mathcal{D}|\boldsymbol{\theta})] \\
&\propto \mathbb{E}_q\big[\log q(\boldsymbol{\theta}|\mathbf{z}) + \log q(\mathbf{z}) - \log p(\mathcal{D}|\boldsymbol{\theta}, \mathbf{z}) - \log p(\boldsymbol{\theta}) - \log q(\mathbf{z}|\boldsymbol{\theta})\big]
\end{aligned}$$

We would like to use this function as the loss function, but the posterior $q(\mathbf{z}|\boldsymbol{\theta}) = \frac{q(\boldsymbol{\theta}|\mathbf{z}) q(\mathbf{z})}{q(\boldsymbol{\theta})}$ is intractable. To handle this, Louizos and Welling [2017] introduced another variational distribution $r(\mathbf{z}|\boldsymbol{\theta})$, that is also parameterized by a normalizing flow. This results in the following loss function

$$f(\boldsymbol{\theta}, \boldsymbol{\lambda}) = \log q(\boldsymbol{\theta}|\mathbf{z}) + \log q(\mathbf{z}) - \log p(\mathcal{D}|\boldsymbol{\theta}, \mathbf{z}) - \log p(\boldsymbol{\theta}) - \log r(\mathbf{z}|\boldsymbol{\theta}), \tag{3.5}$$

where $\boldsymbol{\lambda}$ is now used to describe all parameters of q, including the mean and standard deviation, and all the parameters in the normalizing flow.

The optimization procedure for this algorithm is much more involved than for Bayes by Backprop [Blundell et al., 2015], but we give a rough outline of the procedure in Algorithm 5.

Algorithm 5: An outline of the multiplicative normalizing flow algorithm for a fully connected layer.

Requires:

- $\boldsymbol{x}$, the input value to the current layer.
- $\boldsymbol{\lambda}$, the parameters of the layer (includes mean, standard deviation and parameters of the normalizing flow).

For each iteration, for each layer:

1. Sample $\mathbf{z}_0 \sim q(\mathbf{z})$
2. Transform with normalizing flow $\mathbf{z}_{T_f} = NF(\mathbf{z}_0)$
3. Calculate output mean $\boldsymbol{\mu}_O = (\boldsymbol{x} \odot \mathbf{z}_{T_f})\boldsymbol{\mu}_N$
4. Calculate output variance $\boldsymbol{\Sigma}_O = \boldsymbol{x}^2\boldsymbol{\Sigma}_N$
5. Sample $\boldsymbol{\epsilon} \sim \mathcal{N}(0, I)$
6. Let $\theta = \boldsymbol{\mu}_O + \sqrt{\boldsymbol{\Sigma}_O} \odot \boldsymbol{\epsilon}$
7. Update $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} - \alpha\nabla_{\boldsymbol{\lambda}} f(\theta, \boldsymbol{\lambda})$

Louizos and Welling [2017] show that it is possible to parameterize the posterior through multiplicative normalizing flows, and that this gives a more accurate posterior approximation. Results show predictive ability on par with other state-of-the-art methods, while providing significantly better uncertainty estimates than Bayes by Backprop. Although this method increases the flexibility of the posterior, the multiplicative nature of the parameterization limits its ability to model arbitrary relations between weights.

3.3 Bayes by Hypernet

Bayes by Hypernet [Pawlowski et al., 2017] is a different approach to approximating the posterior distribution. Instead of making strict assumptions about the shape of the posterior and minimizing the KL-divergence analytically, they let the variational posterior be almost arbitrarily flexible, and use an approximation to the KL-divergence. The complex variational posterior is generated from a neural network, which they choose to call a hypernet. The hypernet is fed a set of random samples from some simple distribution, and then outputs a new distribution. This functionality is equivalent to that of a generator in a generative adversarial network (see Section 2.3). To calculate the KL-divergence between this distribution and the prior, a kernel function is used. This

requires multiple samples from both the prior and the posterior. Equation 3.6 shows the kernel they used to estimate the KL-divergence, and is taken from Jiang [2018].

$$D_{\mathrm{KL}}(q(\boldsymbol{\theta})\|p(\boldsymbol{\theta})) \approx \frac{d}{n}\sum_{i=1}^{n}\log\frac{\min_j \|\boldsymbol{\theta}_q^i - \boldsymbol{\theta}_p^j\|}{\min_{j\neq i}\|\boldsymbol{\theta}_q^i - \boldsymbol{\theta}_q^j\|} - \log\frac{m}{n-1} \tag{3.6}$$

Pawlowski et al. [2017] use five samples from the prior and five from the posterior for each KL-approximation. Increasing the number of samples used will naturally lead to a more accurate approximation, but will also be more computationally demanding since each posterior sample is generated by a neural network. Algorithm 6 shows the procedure for updating the parameters of the hypernet.

Algorithm 6: Bayes by Hypernet

We denote the function described in Equation 3.6 as $f(\boldsymbol{\theta}_q, \boldsymbol{\lambda}|\boldsymbol{\theta}_p)$, where $\boldsymbol{\lambda}$ is the set of weights of the neural network q.

For each iteration:

1. Sample $\boldsymbol{\theta}_q^i$, for $i = 1, \ldots, n$ from the hypernet.
2. Sample $\boldsymbol{\theta}_p^j$, for $j = 1, \ldots, m$ from the prior.
3. Update $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} - \alpha\left(\nabla f(\boldsymbol{\theta}_q, \boldsymbol{\lambda}|\boldsymbol{\theta}_p) + \frac{\partial \log p(\mathcal{D}|\boldsymbol{\theta}_q)}{\partial \boldsymbol{\theta_q}}\right)$.

Notice that only the hypernet is being trained. The network used for making predictions doesn't actually have any parameters itself. It depends on the hypernet to supply it with parameters every time it makes a prediction.

The algorithm is fairly straightforward, but there are a few varieties in how the hypernet generate the weights that can have an impact on performance. Pawlowski et al. [2017] tried a few different methods. The first method they tried was to use a single hypernet to generate all the weights of the network in slices. The network takes as an input a random vector $\mathbf{z}$, together with a one-hot encoded vector $\boldsymbol{c}$ describing which subset of the weights it is generating. This is to avoid having a hypernet with too many outputs. Secondly they tried using a single hypernet for each layer. Thirdly, they tried using a combination of these methods, where each layer has a hypernet, which splits the weights by its output dimension. They found that the layer-wise approach was the most computationally demanding, but also gave the best result in terms of accuracy and uncertainty.

Chapter 4

Method

In this chapter we will present a novel method for approximate Bayesian inference in neural networks. This method builds upon material previously discussed in this thesis. In particular generative adversarial networks [Goodfellow et al., 2014] and Bayes by Hypernet [Pawlowski et al., 2017].

4.1 Concept and Motivation

The general concept of the method we propose is to use a generative adversarial network to get the posterior of the weights in a Bayesian neural network. The generator will generate samples from the posterior distribution of the weights given the training data, while the discriminator approximates the KL-divergence between this sampling distribution and the prior. The discriminator acts only as a penalty while training the model, and is not necessary for inference. This means that once the model is fully trained, we no longer need the discriminator. Figure 4.1 illustrates the concept. The yellow diamonds represents the values necessary to assemble a loss function.

Recall from Section 2.4 that a Bayesian neural network can use variational inference and minimize the evidence lower bound to approximate the posterior distribution of the weights. Also recall that the evidence lower bound can be represented as the sum of the KL-divergence from $q(\boldsymbol{\theta})$ to $p(\boldsymbol{\theta})$ and the negative log likelihood, $-\mathbb{E}_{\boldsymbol{\theta}\sim q}[\log p(\mathcal{D}|\boldsymbol{\theta})]$.

$$\begin{aligned}\textsc{elbo}(q) &= \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log p(\boldsymbol{\theta}, \mathcal{D})] - \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log q(\boldsymbol{\theta})] \\ &= D_{KL}[q(\boldsymbol{\theta})\|p(\boldsymbol{\theta})] - \mathbb{E}_{\boldsymbol{\theta}\sim q}[\log p(\mathcal{D}|\boldsymbol{\theta})]\end{aligned}$$

We know that the negative log likelihood, $-\mathbb{E}_{\boldsymbol{\theta}\sim q}[\log p(\mathcal{D}|\boldsymbol{\theta})]$, can be easily approximated by assuming that $p(\mathcal{D}|\boldsymbol{\theta})$ is part of a known family of distribution that we can

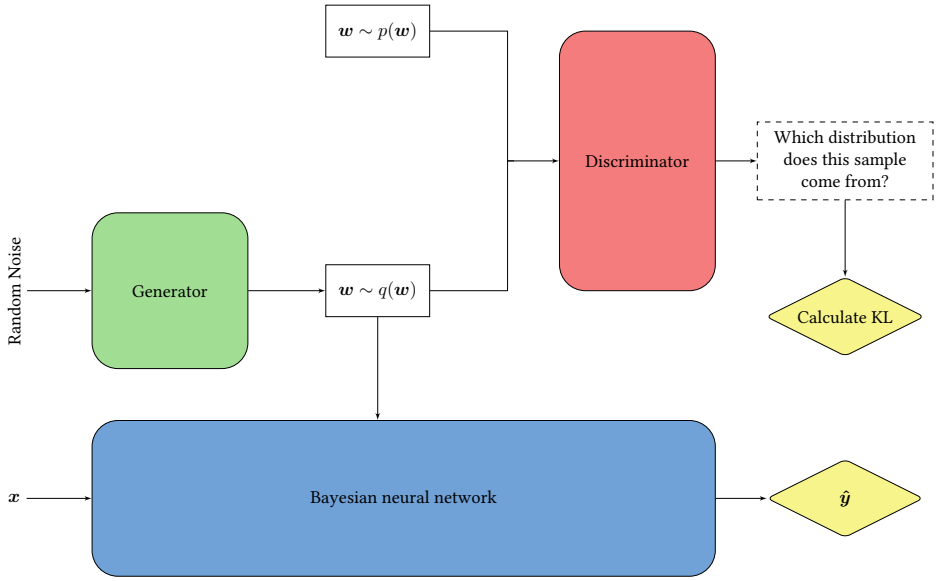


Figure 4.1: An illustration of the primary concept.

evaluate, such as a Gaussian for regression or a Bernoulli for binary classification. So for ELBO, the difficult part is calculating $D_{KL}[q(\boldsymbol{\theta})\|p(\boldsymbol{\theta})]$. We want $\mathcal{Q}$ to be as flexible as possible to more accurately model the true posterior, but this makes an analytical solution for the KL-divergence difficult to obtain. Our $\mathcal{Q}$ is a complex warping of a multivariate normal distribution modeled by a generator network, meaning $\mathcal{Q}$ can be highly flexible. We then want to use a discriminator to calculate $D_{KL}[q(\boldsymbol{\theta})\|p(\boldsymbol{\theta})]$.

4.2 Details

This section goes into more detail on the method we have developed.

4.2.1 Theoretical Foundation

First we have to show that it is possible to use a discriminator to approximate the KL-divergence between two distributions. In Proposition 1 we prove that an optimal discriminator[1] with respect to the loss $\ell(D) = -\mathbb{E}_{\boldsymbol{x}\sim p}[\log(D(\boldsymbol{x}))] - \mathbb{E}_{\boldsymbol{x}\sim q}[\log(1 - D(\boldsymbol{x}))]$, approximates the KL-divergence between two distributions to

[1] An optimal discriminator is one that yields the minimum mean loss over an infinite number of samples.

arbitrary precision. This means that we can get an arbitrarily good approximation of $D_{KL}[q\|p]$ by increasing the complexity of the discriminator and the number of samples for the discriminator to evaluate.

Proposition 1
Let $p(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}_+$ and $q(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}_+$ be two pdfs.
Furthermore, let $D(\boldsymbol{x}) : \mathbb{R}^n \to (0, 1)$, and let ℓ be an operator on D, such that $\ell(D) = -\mathbb{E}_{\boldsymbol{x}\sim q}[\log(D(\boldsymbol{x}))] - \mathbb{E}_{\boldsymbol{x}\sim p}[\log(1 - D(\boldsymbol{x}))]$. Finally, denote $D^* = \arg\min_D \ell(D)$.

Then,

$$D_{KL}[q\|p] = \mathbb{E}_{\boldsymbol{x}\sim q}\left[\log(D^*(\boldsymbol{x})) - \log(1 - D^*(\boldsymbol{x}))\right]$$

Proof.

$$\begin{aligned}
\ell(D) &= -\mathbb{E}_{\boldsymbol{x}\sim q}[\log(D(\boldsymbol{x}))] - \mathbb{E}_{\boldsymbol{x}\sim p}[\log(1 - D(\boldsymbol{x}))] \\
&= \int_{\boldsymbol{x}} \left[-q(\boldsymbol{x})\log(D(\boldsymbol{x})) - p(\boldsymbol{x})\log(1 - D(\boldsymbol{x}))\right] \mathrm{d}\boldsymbol{x} \\
&= -\int_{\boldsymbol{x}} \left[q(\boldsymbol{x})\log(D(\boldsymbol{x})) + p(\boldsymbol{x})\log(1 - D(\boldsymbol{x}))\right] \mathrm{d}\boldsymbol{x}
\end{aligned}$$

Using the fact that

$$\arg\max_y a\log(y) + b\log(1 - y) = \frac{a}{a + b} \quad ,$$

we can pointwise optimize D so that we get

$$D^* = \arg\min_D \ell(D) = \frac{q(\boldsymbol{x})}{q(\boldsymbol{x}) + p(\boldsymbol{x})} \quad .$$

Because there are no constraints on D (as opposed to a PDF, it does not have to integrate to 1), a pointwise mimunum function of $\ell(D)$ is valid, and must be at least as small as any other function.

We now have that

$$
\begin{aligned}
&\mathbb{E}_{\boldsymbol{x}\sim q}\big[\log\left(D^*(\boldsymbol{x})\right) - \log\left(1 - D^*(\boldsymbol{x})\right)\big] \\
&\qquad = \mathbb{E}_{\boldsymbol{x}\sim q}\left[\log\left(\frac{q(\boldsymbol{x})}{p(\boldsymbol{x}) + q(\boldsymbol{x})}\right) - \log\left(1 - \frac{q(\boldsymbol{x})}{p(\boldsymbol{x}) + q(\boldsymbol{x})}\right)\right] \\
&\qquad = \mathbb{E}_{\boldsymbol{x}\sim q}\left[\log\left(\frac{q(\boldsymbol{x})}{p(\boldsymbol{x}) + q(\boldsymbol{x})}\right) - \log\left(\frac{p(\boldsymbol{x})}{p(\boldsymbol{x}) + q(\boldsymbol{x})}\right)\right] \\
&\qquad = \mathbb{E}_{\boldsymbol{x}\sim q}\left[\log\left(\frac{q(\boldsymbol{x})}{p(\boldsymbol{x}) + q(\boldsymbol{x})}\frac{p(\boldsymbol{x}) + q(\boldsymbol{x})}{p(\boldsymbol{x})}\right)\right] \\
&\qquad = \mathbb{E}_{\boldsymbol{x}\sim q}\left[\log\left(\frac{q(\boldsymbol{x})}{p(\boldsymbol{x})}\right)\right] \\
&\qquad = D_{KL}[q\|p].
\end{aligned}
$$

□

Now that we have shown that a discriminator can be used to approximate the KL-divergence between two arbitrary PDFs, we will use it to approximate the KL-divergence between the prior and posterior in a Bayesian neural network. This has a few key advantages compared to other state of the art methods. Bayes by Backprop does not make any assumptions about the prior distribution, but is only able to make independent Gaussian approximations of the posterior. Bayes by Hypernet does not make any assumptions about the shape of the prior or posterior distribution, but uses a KL-approximations that is not well understood, and that we will show in Chapter 5 does not approximate the KL-divergence well with few samples. My method does not make any assumptions about the shape of the prior or posterior distribution either, but has a theoretical guarantee of an arbitrarily precise approximation.

Figure 4.1 illustrates the general structure of our proposed method. The generator takes in random noise, and outputs a set of weights for the Bayesian neural network. The Bayesian neural network then acts as a regular neural network, using the generated weights to predict $\boldsymbol{y}$ from $\boldsymbol{x}$. At the same time, the discriminator approximates the probability that these sets of weights comes from the posterior distribution. We use this probability to estimate the KL-divergence between the prior and the posterior. Now we can sum the estimated KL-divergence with the negative log likelihood from the prediction that our Bayesian neural net made to get the loss function for the generator. We now end up with the following loss functions:

$$\text{Discriminator loss: } -\mathbb{E}_{\boldsymbol{\theta}\sim G(Z)}[\log(D(\boldsymbol{\theta}))] - \mathbb{E}_{\boldsymbol{\theta}\sim p(\boldsymbol{\theta})}[\log(1-D(\boldsymbol{\theta}))] \tag{4.1}$$

$$\text{Generator loss: } \mathbb{E}_{\boldsymbol{\theta}\sim G(Z)}\big[\underbrace{-\log(p(\mathcal{D}|\boldsymbol{\theta}))}_{\text{prediciton loss}} + \underbrace{\log(D(\boldsymbol{\theta})) - \log(1-D(\boldsymbol{\theta}))}_{\text{Normal generator loss}}\big] \tag{4.2}$$

Comparing the loss functions with those of a standard GAN, we can see that the discriminator loss in Equation 4.1 is the same as the one for a regular GAN shown in Equation 2.15. The generator loss, however, is slightly different. It is a combination of the regular generator loss from Equation 2.16, the modified loss from Equation 2.17, and the negative log likelihood from regular neural networks.

4.2.2 Tackling dimensionality

For this method to work, there are a few requirements. Firstly, the discriminator must be able to learn to differentiate between two distributions. In small dimensions, this will not be a problem, but all neural networks tend to struggle with very large unstructured input spaces. The size of the input space for the discriminator equals the number of weights in the Bayesian neural network. Neural networks can have millions of weights, so we know that this will pose a problem. We need a way of reducing the input space for the discriminator.

Layer independence

In Section 2.2.4 we introduced mean field, a family of distributions characterized by how they factorize in such a way that we can easily perform our desired operations. If we apply the mean field assumption that $q(\boldsymbol{\theta}|\mathcal{D})$ factorizes into $\prod_{l=1}^{N} q_l(\boldsymbol{\theta_l})$, where $\boldsymbol{\theta_l}$ are the weights in layer l of the Bayesian neural network, we can ensure that this assumption holds by giving each layer its own generator, and giving each generator independent noise samples when generating weights. Now that we have ensured independence between each layer, we can actually give each layer its own discriminator too. We can show that the KL-divergence between the weights and priors for the whole network is the sum of KL-divergences for each layer:

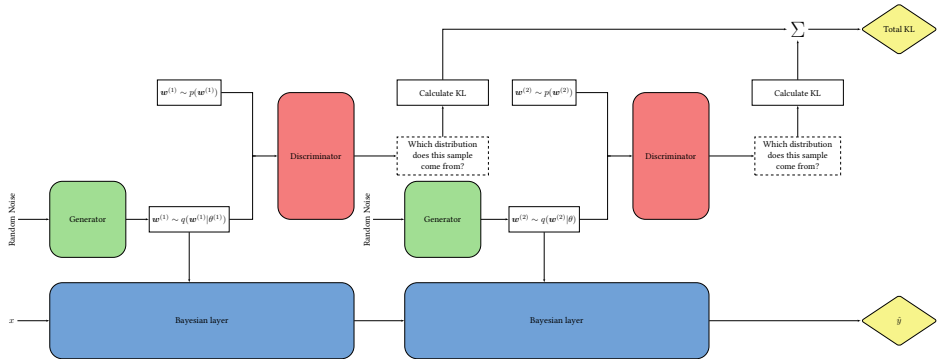


Figure 4.2: Shows how two layers with independent generators and discriminators can be chained together.

$$
\begin{aligned}
D_{KL}[q(\boldsymbol{x},\boldsymbol{y})\|p(\boldsymbol{x},\boldsymbol{y})] &= \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim q}\left[\log\left(\frac{q(\boldsymbol{x},\boldsymbol{y})}{p(\boldsymbol{x},\boldsymbol{y})}\right)\right] \\
&= \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim q}\left[\log\left(\frac{q_1(\boldsymbol{x})q_2(\boldsymbol{y})}{p_1(\boldsymbol{x})p_2(\boldsymbol{y})}\right)\right] \\
&= \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim q}\left[\log\left(\frac{q_1(\boldsymbol{x})}{p_1(\boldsymbol{x})}\right) + \log\left(\frac{q_2(\boldsymbol{y})}{p_2(\boldsymbol{y})}\right)\right] \\
&= \mathbb{E}_{\boldsymbol{x}\sim q}\left[\log\left(\frac{q_1(\boldsymbol{x})}{p_1(\boldsymbol{x})}\right)\right] + \mathbb{E}_{\boldsymbol{y}\sim q}\left[\log\left(\frac{q_2(\boldsymbol{y})}{p_2(\boldsymbol{y})}\right)\right] \\
&= D_{KL}[q_1(\boldsymbol{x})\|p_1(\boldsymbol{x})] + D_{KL}[q_2(\boldsymbol{y})\|p_2(\boldsymbol{y})]. \qquad (4.3)
\end{aligned}
$$

Now that we have shown that by sacrificing association between weights in different layers, each layer can have its own generator and discriminator. This means that we can turn our Bayesian neural network into Bayesian layers and chain them together. Figure 4.2 shows how we can chain multiple layers together.

Partial weight independence

Separating the network into independent layers reduces the dimension of the discriminator's input somewhat, but each layer still has $N_{in} \times N_{out}$, where N_{in} and N_{out} are the number of input and output nodes for the layer respectively. This means that we can still have millions of weights per layer. Just as for Louizos and Welling [2017], the

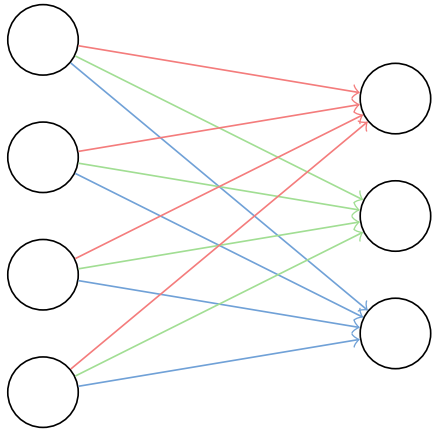


Figure 4.3: Illustration of how weights are separated into subspaces, each color representing its own subspace.

complexity required to model an arbitrary transformation of the entire weight space is computationally very complex even for medium sized layers.

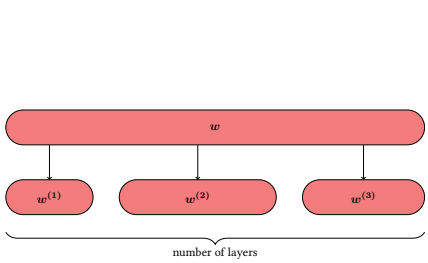
We will now propose a method to reduce this number to $N_{in} + N_{out}$. We can extend the argument for splitting the network into layers to giving each weight its own generator and discriminator, but this is not computationally feasible. It would also take away the goal of the method, to improve the capacity of variational posterior to model association between weights, since this would make the posterior fully factorized. What we can do instead is to divide the weights into independent subspaces where we can model dependencies between weights in each subspace. Figure 4.3 shows a natural way of dividing the weights into such subspaces. If we otherwise follow the same method as for separating layers we end up with N_{out} generators and discriminators, each generator generating N_{in} weights, and each discriminator handling N_{in} weights. With this method we have reduced the dimensionality of the discriminators input space considerably, while still being able to model association between weights connected to the same node. This dimensionality reduction is similar to the one made in MNF [Louizos and Welling, 2017], in that we are both able to model arbitrary dependencies between weights going into the same output node.

Having N_{out} generators and discriminators for each layer is still very demanding computationally. We will now borrow a trick from Pawlowski et al. [2017], where they append a one-hot vector to the noise input vector for the generator so that they can use a single generator to generate all the weights of a layer slice-wise. For our situation, this means that the input vector to the generator will be a vector $[\mathbf{z}, \boldsymbol{c}]$, where $\mathbf{z}$ is the noise vector, and $\boldsymbol{c}$ is a one-hot encoded vector denoting which subspace of weights

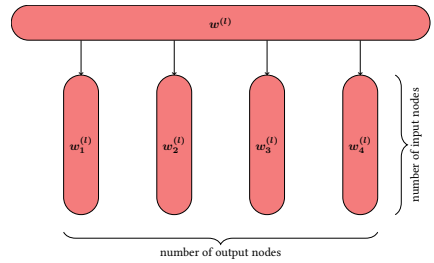
we are generating. $\boldsymbol{c}$ will have dimensionality N_{out}. We can extend this method to the discriminator, so that we also only use one discriminator for each layer. Figure 4.4 shows a complete picture of how we group the weights for the whole Bayesian network into vectors that the discriminators can handle. There are a few reasons that this method works. First, splitting the weights up into subspaces based on their output node connection means that subspace of weights will have the same dimensionality. This is a necessary requirement in order to be able to use a single generator and discriminator. Second, as long as we use independent noise input to the generator for each weight batch we generate, we can assure independence between the weight subspaces. Finally, concatenating a one-hot encoded vector to the input allows the generator and discriminator to learn the function that all the multiple generators/discriminators would learn. We can see that this is the case, by creating a function $G(\mathbf{z}, \boldsymbol{c})$ such that

$$G(\mathbf{z}, \boldsymbol{c}) = \sum_{i=1}^{N_{out}} c_i G_i(\mathbf{z}),$$

where $\boldsymbol{c}_i$ is the ith entry in the one-hot encoded vector $\boldsymbol{c}$, and $G_i(\mathbf{z})$ is the generator that generates the weights connected to the ith output node. Because G is a neural network, by the universal approximation theorem it should be able to approximate this function. Algorithm 7 is a rough pseudo-code implementation of the algorithm.



(a) Splitting weights into separate vectors for each layer.



(b) Splitting weights for each layer into separate vectors for each output node.

Figure 4.4: An illustration of how the weight space in the network is decomposed into subspaces.

4.2.3 Implementation details

We have so far described our idea on a conceptual level. Now we will discuss details necessary to successfully implement our algorithm.

Algorithm 7: Bayesian neural network with discriminator KL-approximation

Parameters for the discriminator: θ
Parameters for the generator: $\boldsymbol{\lambda}$
$P_{\boldsymbol{w}}$ predicted probability that a sample $\boldsymbol{w}$ from q comes from q.
$P_{\boldsymbol{w}}$ predicted probability that a sample $\boldsymbol{p}$ from p comes from q.
$\ell(P_{\boldsymbol{w}}, P_{\boldsymbol{p}}) = -\log(P_{\boldsymbol{w}}) - \log(1 - P_{\boldsymbol{p}})$
$f(P_{\boldsymbol{w}}) = \log(P_{\boldsymbol{w}}) - \log(1 - P_{\boldsymbol{w}})$

For each iteration, for each layer:

1. Generate a random sample $\boldsymbol{z}_i \sim i.i.d\, \mathcal{U}(-0.5, 0.5)$ for $i = 1, \ldots, N_{out}$.
2. Append a one-hot encoded vector to the random samples, $\boldsymbol{\zeta_i} = \begin{bmatrix} \boldsymbol{z}_i & \boldsymbol{c_i} \end{bmatrix}$.
3. Generate weights $\boldsymbol{w} = \begin{bmatrix} G_{\boldsymbol{\lambda}}(\boldsymbol{\zeta_1}) & \ldots & G_{\boldsymbol{\lambda}}(\boldsymbol{\zeta_N}) \end{bmatrix}$.
4. Sample priors $\boldsymbol{p} \sim i.i.d\, \mathcal{N}(0, 1)$.
5. Calculate discriminator predictions for weights $P_{\boldsymbol{w}} = D_{\theta}(\boldsymbol{w})$.
6. Calculate discriminator predictions for priors $P_{\boldsymbol{p}} = D_{\theta}(\boldsymbol{p})$.
7. Update $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} - \alpha \nabla_{\boldsymbol{\lambda}} \left[\log p(\mathcal{D}|\boldsymbol{w}) + f(P_{\boldsymbol{w}}) \right]$
8. Update $\theta \leftarrow \theta - \alpha \nabla_{\theta} \ell(P_{\boldsymbol{w}}, P_{\boldsymbol{p}})$

Batching and stochastic gradient descent

Regular neural networks, as well as GANs use batching to reduce the variance in the gradient. Averaging the gradient over multiple examples means a more steady convergence to a local minimum. Because of the reduced variance, it also allows for a higher learning rate, leading to faster convergence. In our algorithm there are a few different ways of doing batching

- Sample weights $\mathbf{w} \sim G(\mathbf{z})$. Use the weights to make prediction on a mini-batch of the training data $\mathcal{D}_i \subset \mathcal{D}$. Calculate the average loss across the mini-batch.
- Sample multiple weights $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_n \sim G(\mathbf{z})$. For one datapoint $d_i \in \mathcal{D}$ make n predictions, one for each set of weights. Calculate the average loss across all predictions.
- Combine the above and run a mini-batch of data through multiple sampled weights and calculate the average loss.

Each of these methods have their own advantages and disadvantages. Running a batch through the same sampled weights works like regular batching, benefits from hardware acceleration in the form of fast matrix multiplication, thus it is computationally cheap to apply. The benefits, however, are lesser than for the other methods. This batching scheme only reduces the variance in the gradient with respect to the likelihood. Unfortunately, most of the variance in the gradient comes from the KL term. To reduce this

variance we can use the second batching scheme. By sampling the weights multiple times and running them through the discriminator, and taking the mean predicted KL-divergence, we will have reduced the variance of the gradient with respect to the KL term. This process is unfortunately much more difficult to hardware accelerate, which means it also costs more. We found the third scheme to work best. We can calculate the full loss and its gradient multiple times, but only update the weights for every N data point with the average gradient. This method cannot be hardware accelerated, hence has similar computational costs as the previous method, but reduces the variance of the gradient further than method 2 with minimal performance penalty.

Whether we are using batching or not, we are still only seeing a subset of the data at a time. This means that the negative log-likelihood part of our loss is only calculated on this subset of data we are seeing, while in Proposition 1, we are assuming that the nll is calculated across the whole dataset. Usually this is not a problem, because we assume that the gradient of that subset of datapoints gives us an approximate gradient. For Bayesian neural networks, it is different. For the loss, we have one component, the negative log-likelihood, that depends on the data and the sampled weights, and one component, the KL-divergence, that depends only on the sampled weights.

$$\mathcal{L}(\mathcal{D}, \theta) = -\sum_{i=1}^{|\mathcal{D}|} \log p(\boldsymbol{y}^{(i)}|\boldsymbol{x}^{(i)}, \theta) + D_{KL}[q\|p] \tag{4.4}$$

The negative log-likelihood for the whole dataset is the sum of the nll for each data point, so when we are using a subset of all the data points, we are underestimating the total nll for the whole dataset. This means that the KL-divergence will contribute too much to the loss, and cause it to overestimate the uncertainty. To counteract this, we first make sure to average the nll loss for each batch across all the data points. This means that we would be underestimating the nll by a factor of $|\mathcal{D}|$. We can then divide the estimated KL-divergence by this same factor.

Noise Vector

We can drastically alter the behavior of the generator by changing the dimensionality of the noise vector that we give it as input. A vector with few dimensions will force dependency between weights, as we cannot transform a set of independent random variables into more independent random variables[2]. This means that if we want to be able to model independence between all weights we will need the noise vector to contain at least as many elements as the output vector, i.e. the number of input vectors for

[2]Assuming our generated set of random variables does not contain almost surely constant random variables. In our case, our generator will generate these distributions with probability 0, so we do not have to consider this case.

its associated layer. Having a noise vector this size will keep the input size to the generator $\mathcal{O}(n)$ with respect to the layer size, which is important for training to be feasible. Increasing the dimensionality of the noise vector beyond the minimum requirement to model independent random variables for each weight will make the task for the generator more difficult and subsequently increase training time. We hypothesize that providing a smaller noise vector will, despite its limitations in modelling capacity, provide better results due to simplifying the generators task. Experimental results shows that this is in fact the case.

Another factor is the distribution of the noise vector. Typically in GANs, each element is sampled from either a Gaussian or uniform distribution. We found that our model performed better with uniform samples. We also experienced some convergence issues with Gaussian noise on certain tasks, hence we recommend using uniformly sampled noise for the generator.

Network Parameters

Our algorithm consists of three neural networks, the Bayesian neural network whose weights are generated by the generator, said generator, and the discriminator. The Bayesian neural network is used to predict $p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{w})$, and is hence comparable to a regular neural network. We can therefore choose to use the same number of layers, the same activation functions, and the same node count as in a regular neural network; and hopefully expect similar predictive performance. A more difficult task is to determine the best network for the generator and discriminator. We find that increasing the size of the generator and discriminator will quickly slow down training. Another consideration is the relative performance between the generator and discriminator. The correctness of the method relies on having a perfect discriminator. While this is impossible in practice, it is crucial that it is much stronger than the generator. In a regular GAN, a very strong discriminator can be detrimental to performance [Arjovsky and Bottou, 2017], but because we are also using the log-likelihood $p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{w})$ to train the generator, our situation is different. To get an accurate estimate for ELBO for each iteration we need the discriminator to not only predict the correct distribution that the sample comes from, it needs to do it with correct certainty, a much more difficult task. Finding the best parameters for a specific task is a difficult problem, but Table 4.1 shows some general guidelines we have found to work well.

Annealing

Louizos and Welling [2017] and Pawlowski et al. [2017] use annealing during their training phase to ensure convergence. This means that we alter the loss to

Generator learning rate	10^{-3}
Discriminator learning rate	10^{-3}
Generator size	2 hidden layer with 0.25x and 0.5x as many nodes compared to the maximum of input and output nodes in the Bayesian layer.
Discriminator size	3 hidden layers with 0.5x as many nodes compared to the maximum of input and output nodes in the Bayesian layer.
Noise vector length	≤ 10.

Table 4.1: Suggested network parameters.

$$\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{x}, \boldsymbol{\theta}) = \log p(\hat{\boldsymbol{y}}|\boldsymbol{x}, \boldsymbol{\theta}) + \eta D_{KL}\left[q, p\right],$$

where η is a variable depicting the importance of the KL-divergence in the loss function. By starting with a small value for η and slowly increasing it to 1 we can make the network learn accurate predictions before needing to focus on uncertainties. We can think of this as speeding up convergence by first moving to a good point estimate, and from there develop appropriate uncertainties for each weight. This makes sense if a good local minimum in our search space is close to a maximum likelihood solution. We believe this is a fair assumption; we imagine going from an MLE solution to an approximate posterior by increasing the variance of each weight according to its importance in making an accurate prediction. This way we still make good predictions, but move our weight distribution closer to the prior, hence reducing the KL-divergence.

Another advantage of annealing for our method is that we do not have to worry about training the discriminator while the weights are far from the optimum. Because we assume that a good local minimum for the generator(s) is close to the MLE solution, we can move there quicker if we don't have to also consider the noisy loss from the discriminator. Once we are close to a local minimum in the MLE search space we can train the discriminator on this solution until we are satisfied with its performance, and then proceed to train them together.

Batch Normalization

Radford et al. [2016] presents a few architectural guidelines for training deep convolutional generative adversarial networks (DCGAN), and while our use of GANs does not fit their description, we still found some of their tips useful. They suggest using batch normalization in the generator and discriminator to avoid training problems arising due to poor initialization, and to help gradient flow in deep models. Our generative

networks are rather shallow, but we still found batch normalization layers in the generator achieves significantly better results. We suspect that batch normalization makes it much easier to approximate the prior distribution by limiting large activations. We can only use batch normalization when we have multiple batches that we feed into the generator. This means that we cannot use it for layers that have a 1-dimensional output. This is typically only the case for output layers, so we expect to be able to use batch normalization for most layers. For the discriminator network, we found that it performed significantly worse with batch normalization. Limiting large activation values for the discriminator likely causes it to underestimate the KL-divergence between the distributions, hence the poor results. We suspect that for regular GANs limiting the predicted KL-divergence results in more stable gradients, hence the improved performance.

Multiple samples

Reducing noise in the gradient of KL-approximation can significantly improve performance as Ranganath et al. [2013] showed when reducing the noise in the gradient for Black Box Variational Inference. One of the ways we thought we could do this was to give the discriminator multiple samples from the generator and the discriminator for each prediction, as described above. By increasing the input dimensions to the discriminator and giving it multiple samples each time we can expect the discriminators task to be easier. From Equation 4.3 we can see that giving the discriminator n samples, and then dividing the resulting predicted KL-divergence by n we get the same KL-divergence we expect from a single sample. The problem with this approach is that as we are giving the discriminator more samples, because the KL-divergence is calculated from the $\log$ of the output from the discriminator, it must become exponentially more certain about each prediction. Consequently, implementing this idea actually significantly hinders training performance.

Chapter 5

Evaluation and Results

This chapter presents and discusses the performance of our method introduced in Chapter 4. We compare our method to the methods presented in Chapter 3, and discuss possible future work in the use of discriminator networks for Bayesian neural networks.

5.1 Evaluation

There are no standard performance metrics for Bayesian neural network. For predictive performance we can use the same measures as for regular neural networks (accuracy, precision, recall, F1-score, etc), but evaluating uncertainty is much more difficult. We can generate data with a known noise level, but we cannot set the epistemic uncertainty (uncertainty about the model generating the data) in the experiment. The papers presented in Chapter 3 all partially rely on qualitative analysis of their method. Still, there are quantitative ways of measuring how well our method models uncertainty. One such method used in Louizos and Welling [2017] is to train a model on one dataset (for instance MNIST), then run the model on a different dataset (for instance notMNIST [Bulatov, 2011]), recording the distribution of entropy in the output predictions. Because we are running the model on a dataset it has no training data on, we want the model to exhibit a high entropy, meaning it exhibits no confidence about any prediction.

In order to be able to reference our method in comparison to existing methods we have coined the term Bayes by GAN (BbG) for our method. Throughout this chapter we will refer to our method as BbG.

5.1.1 KL-approximation

First we start by showing that a discriminator can approximate the KL-divergence between two distributions. To do this, start by defining p and q as two distributions with known KL-divergence. For two Gaussian distributions, $\mathcal{N}(\boldsymbol{\mu}_q, \Sigma_q)$ and $\mathcal{N}(\boldsymbol{\mu}_p, \Sigma_p)$, we have

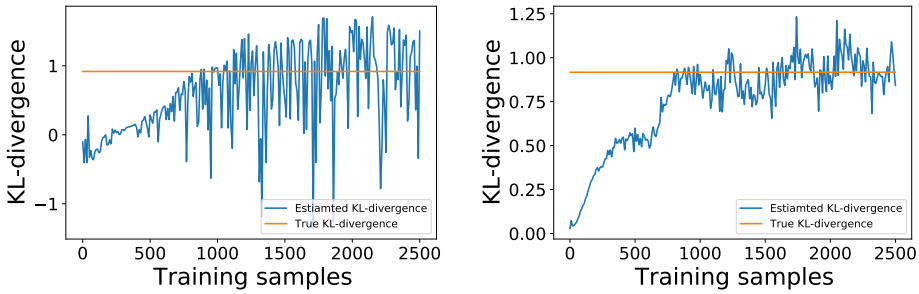
$$D_{KL}[q\|p] = \frac{1}{2}\left(\log\frac{\left|\boldsymbol{\Sigma}_p\right|}{\left|\boldsymbol{\Sigma}_q\right|} - k + \left(\boldsymbol{\mu}_q - \boldsymbol{\mu}_p\right)^T \boldsymbol{\Sigma}_p^{-1}\left(\boldsymbol{\mu}_q - \boldsymbol{\mu}_p\right) + \operatorname{tr}\left(\boldsymbol{\Sigma}_p^{-1}\boldsymbol{\Sigma}_q\right)\right). \tag{5.1}$$

If we choose p and q to be two Gaussian distributions we can use Equation 5.1 to calculate the true KL-divergence between the distributions, and compare this to the estimates from the discriminator. Let $X \sim \mathcal{N}(0,1)$ with PDF $q(x)$, and $Y \sim \mathcal{N}(0,4^2)$ with PDF $p(y)$, we then get

$$\begin{aligned} D_{KL}[q\|p] &= \frac{1}{2}\left(\log\frac{4^2}{1} - 1 + \frac{(0-0)^2}{4^2} + \frac{1}{4^2}\right) \\ &= 2\log 2 - \frac{15}{32} \\ &\approx 0.92. \end{aligned}$$

Let us now define a discriminator D with input and output size 1, and one hidden layer of size 5. We will then use D to approximate the KL-divergence between p and q and compare to our analytical KL. Figure 5.1 shows the estimated KL-divergence between p and q for samples drawn from each training iteration. Notice that while the analytical KL-divergence must always be positive, the estimated KL-divergence can sometimes be negative. This is because the KL-divergence is the *expected* value $\mathbb{E}_{\boldsymbol{\theta}\sim q}\left[\log\frac{q(\boldsymbol{\theta})}{p(\boldsymbol{\theta})}\right]$, and while this is positive, the Monte Carlo estimate $\frac{1}{S}\sum_{s=1}^{S}\log\frac{q(\boldsymbol{\theta}_s)}{p(\boldsymbol{\theta}_s)}$ can be negative. This is particularly likely to happen when we have a very few number of samples, as can be seen in Figure 5.1a. Intuitively, for our case this means that the discriminator sees the sample from q as more likely to come from p. Since this can in fact be the case, we expect to sometimes get a negative estimate for the KL-divergence, even with a perfect discriminator, but with enough samples S the mean should approach the true KL-divergence. We now have empirical evidence that a neural network can be used to approximate the KL-divergence between two distributions by using Proposition 1. This is the foundation that BbG builds on, so both the convergence rate and the accuracy of this result is critical to the performance of BbG.

Having shown that a discriminator will converge to the correct KL-divergence for



(a) KL from single samples

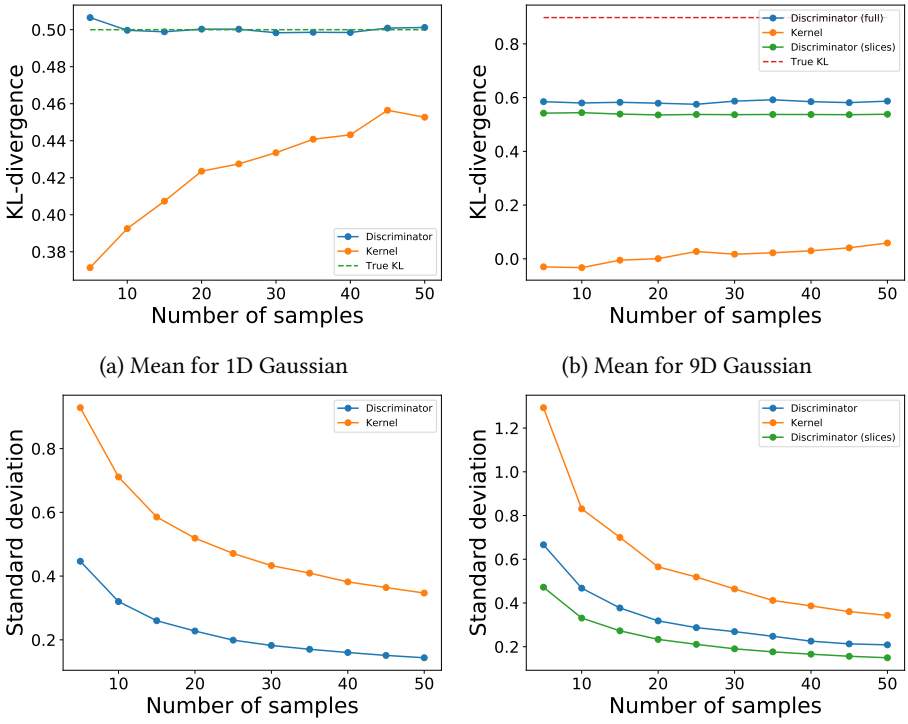
(b) Average KL over 100 samples

Figure 5.1: Shows predicted and true KL-divergence between two univariate Gaussian distributions.

two univariate Gaussian distribution, we will compare its performance to the kernel method used in BbH [Pawlowski et al., 2017]. Figure 5.2 shows the mean and standard deviation of the predicted KL-divergence for both methods between two univariate Gaussians, and between two multi-variate Gaussian distributions. In Figure 5.2a we can see that the mean of the discriminator's estimate for the KL-divergence between two distribution is almost exactly correct, regardless of the number of samples. It makes sense that the mean prediction for the discriminator will be constant with respect to the number of samples it gets, as it does not take multiple samples into account, but rather averages the prediction over each sample individually. The kernel method on the other hand performs poorly with few samples. We can see that it severely underestimates the KL-divergence, while at the same time having a much larger variance in its predictions. This is especially noticeable in the multi-variate case (Figure 5.2b), where it even predicts a negative KL-divergence with few samples. Pawlowski et al. [2017] use 5 samples from the prior and posterior distribution of the weights to estimate the KL-divergence. Our graph shows that this is not at all sufficient to accurately estimate the KL-divergence. This means that the BbH network will underestimate uncertainty, and become overconfident in its predictions.

5.1.2 Fitting priors

After showing that our discriminator is able to approximate the KL-divergence between two distributions, a logical next step is to see if the generator is able to fit a prior distribution. To test this we train our Bayesian neural network without a log-likelihood loss. Because our model only outputs samples from the posterior distribution, to quantitatively compare the posterior and prior distributions, we will assume that the generated



(a) Mean for 1D Gaussian

(b) Mean for 9D Gaussian

(c) Std for 1D Gaussian

(d) Std for 9D Gaussian

Figure 5.2: Mean and standard deviation of predicted KL-divergence between two distributions. In (a) and (c) the KL-divergence is predicted from $\mathcal{N}(0, 1^2)$ to $\mathcal{N}(0, 2^2)$. In (b) and (d), the KL-divergence is approximated between two 9D Gaussian distributions.

posterior is Gaussian. This might seem like a strong assumption, and it is, but it will allow us to look at the KL-divergence between the prior and posterior distributions of the weights as the network sees more and more samples. Once this measure has converged, we can compare samples of a weight to the prior distribution for that weight. Figure 5.3 shows that the KL-divergence between the prior and posterior distribution for weights in a Bayesian neural network layer trained without a negative log-likelihood reaches close to zero with both a kernel and a discriminator approximation. We can also see that we are able to fit the prior distribution in a neural network with a discriminator in fewer steps than with a kernel approximation, despite having to train both the generator and the discriminator.

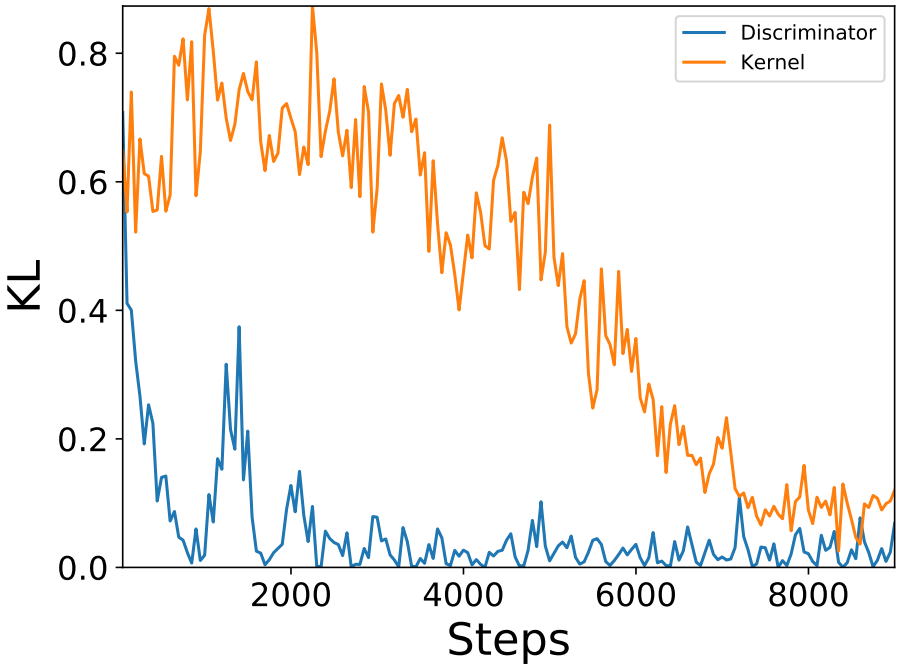
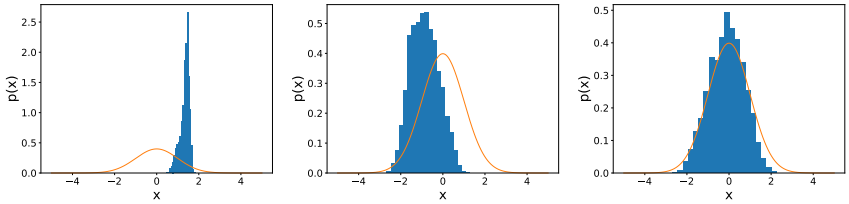


Figure 5.3: KL-divergence between prior and posterior for weights in a Bayesian layer trained solely to minimize KL-divergence.

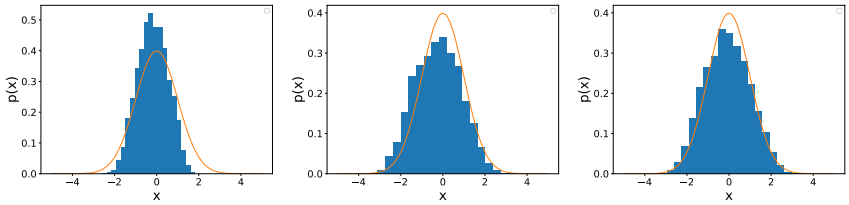
Knowing that we are able to fit a prior distribution with the discriminator we will compare the fitted distributions with those made with a kernel approximation. Figure 5.4 shows the approximate distribution fitted by the kernel approximation after 3 000, 6 000, and 9 000 iterations. We can see that both methods are able to fit a univariate Gaussian distribution, but we can also see that BbG, using a discriminator, converges

much quicker than BbH, using the kernel approximation. This confirms the results in Figure 5.3.



(a) 3 000 (b) 6 000 (c) 9 000

Figure 5.4: Generator fitting a standard normal distribution with a kernel KL-approximation. Numbers denote number of training steps.



(a) 3 000 (b) 6 000 (c) 9 000

Figure 5.5: Generator fitting a standard normal distribution with a discriminator KL-approximation. Numbers denote number of training steps.

5.1.3 Regression on toy dataset

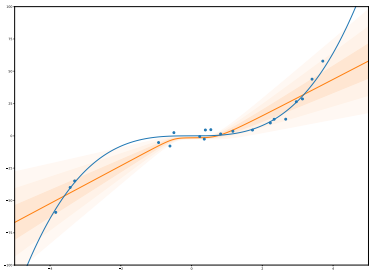
Regression is a simple qualitative test of Bayesian neural networks. With a one-dimensional regression problem with a known generative model we can plot the mean and standard deviation against the true generative model. This will show us the predictive performance of BbG while also being able to inspect the uncertainty estimates by the model. We would expect a large variance in prediction in regions with little to no data. To have comparable results to other methods we will use the same regression problem used for evaluation in [Louizos and Welling, 2017; Pawlowski et al., 2017]. We let $\boldsymbol{x}$ be 20 values sampled from $\mathcal{U}(-4, 4)$, and $y_i = x_i^3 + \epsilon_i$, where $\epsilon_i \sim \mathcal{N}(0, 3^2)$. Fitting this cubic function should be a simple task for most methods, but finding the uncertainty around each prediction is a more difficult task. Figure 5.6 shows BbG compared to some other state-of-the-art methods. All methods were used to train a neural network with two hidden layers of 100 nodes. The implementation of all other state-of-the art methods were borrowed from Pawlowski et al. [2017]'s official GitHub repository at

`https://github.com/pawni/BayesByHypernet`. We will note that our results do not entirely match up with the ones presented by Pawlowski et al. [2017]'s for BbB, MNF, and BbH. In their paper none of these methods seem to have converged, and does not fit the polynomial as well as the non-Bayesian methods. They do, however, present have a larger predicted accuracy, but we do not see it as a fair result to stop training before the model has fitted the data in order to preserve predicted uncertainty. For this reason we have opted to run the experiment for longer. As we trained their model for longer, however, we saw that the variance in their prediction started going to zero. We therefore had to stop the training once the model seemed to have fitted the dataset and started reducing the variance in its prediction. This phenomenon was not present when training BbG, which meant that we could train the model for longer. We can see in Figure 5.6 that BbG outperforms all other methods with respect to both predictive accuracy and uncertainty. We have made further details on how we obtained these results available in Appendix A.

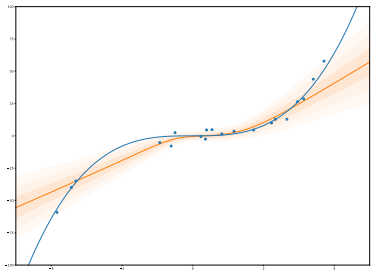
5.1.4 MNIST

MNIST [LeCun et al., 2010] is a dataset containing 70 000 handwritten digits. Each datapoint is a 28×28 pixel grey-scale image with a corresponding label indicating the digit in the picture. The dataset is split into a training set of 60 000 images and a test set of 10 000 images.

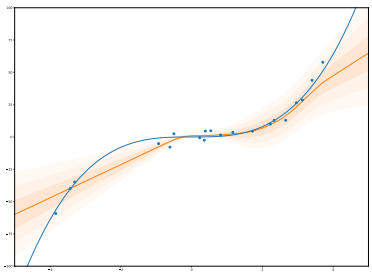
We will begin by training the network described in Table 5.1 using a variety of methods. Table 5.2 shows the classification error results for this tasks. We ran experiments with dropout and BbG. The rest of the results are taken from the Bayes by Hypernet paper [Pawlowski et al., 2017]. We were not able to replicate the performance of Dropout that they claim in their paper, which might indicate that the changes required to get dropout to perform as advertised would also increase the performance of BbG. We therefore can't definitely say that our method is significantly worse than BbB, MNF, and BbH with respect to predictive performance. We can see that just like in their results dropout outperforms the Bayesian neural network methods with respect to predictive performance. We have a few hypotheses as to why it seems to perform worse than the other Bayesian neural network methods when it outperforms them on the regression task. First of all, the increased complexity in the model makes it much more difficult to find the optimal network size for the generator and discriminator. It might be that the method would perform much better with much larger networks for the generator and discriminator, but this would also mean a much longer training time. We also saw on the regression problem that the model only started outperforming other state of the art methods after very many epochs, and while the model seemed to have converged, this is notoriously difficult to confirm for GANs since there are two competing loss functions.



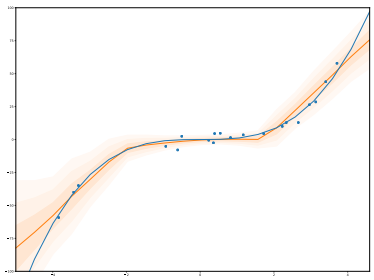
(a) BbB



(b) BbH

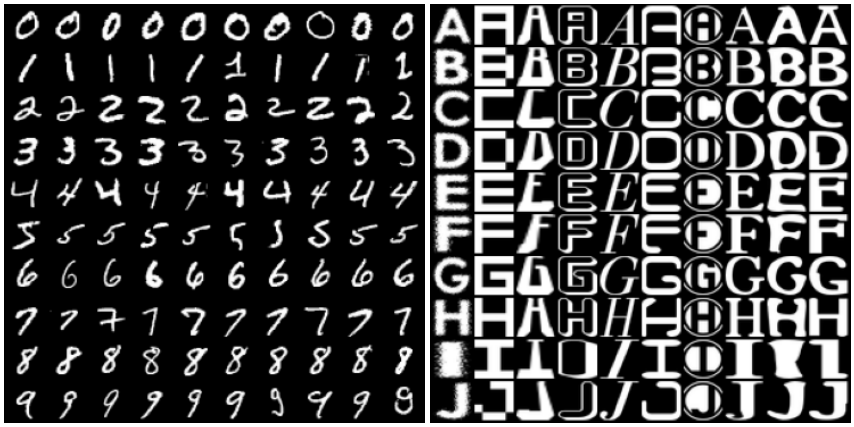


(c) MNF



(d) BbG

Figure 5.6: Predictive distribution on a 1d regression task for different methods. The datapoints were sampled from $y_i \sim x_i^3 + \epsilon_i$, where $x_i \sim \mathcal{U}(-4, 4)$, and $\epsilon_i \sim \mathcal{N}(0, 3^2)$. Orange curve is the mean prediction. Orange shaded area corresponds to 1, 2, and 3 standard deviations away from the mean. Blue curve is the third degree polynomial that the datapoints were sampled from. Blue points are the datapoints used to train the model.



(a) MNIST. (b) notMNIST.

Figure 5.7: Sample of images from two different datasets.

Layer		Size	Channels	Kernel Size	Stride	Activation
1	Input	28×28	1	-	-	-
2	Conv	28×28	6	5×5	1	relu
3	MaxPool	24×24	6	2×2	2	-
4	Conv	12×12	16	5×5	1	relu
5	MaxPool	8×8	16	2×2	2	-
6	Flatten	4×4	16	-	-	-
7	Dense	256	-	-	-	relu
8	Output	10	-	-	-	softmax

Table 5.1: Neural network structured used to classify MNIST images.

Method	Dropout	Dropout*	BbB*	MNF*	BbH*	BbG
Error (%)	0.89	0.47	0.72	0.63	0.56	1.12

Table 5.2: Error on the MNIST test set. * Results as reported by Pawlowski et al. [2017].

To quantify uncertainty for this task we made predictions on a different dataset and measured the cumulative distribution of the entropy for each prediction This method was also used to measure the quality of the predicted uncertainty in Louizos and Welling [2017]. The entropy for a prediction is defined as:

$$H(\hat{y}) = -\sum_{i=1}^{n} p(\hat{y}_i) \log p(\hat{y}_i).$$

The minimum entropy is always 0. For MNIST, which has 10 classes, the maximum entropy is $\log 10 \approx 2.3$. We predict the entropy of the prediction on all the elements in the test set for both MNIST and notMNIST. Figure 5.8 shows the cumulative distribution of the entropy on the whole test set. We can see that most predictions on MNIST have a very low entropy. This means that it is very sure in its prediction. We can explain that the curve for the dropout model is higher in this case because of its higher accuracy. It might, however, not be better to have a lower entropy in its predictions, considering the model actually does make wrong predictions, and a low entropy in those cases would mean that it predicts the wrong class with high certainty. For this reason, we can't necessarily say that it performs better than BbG in terms of uncertainty on the MNIST dataset. On the notMNIST dataset we want the model to have a very low entropy in its predictions, considering none of the classes are correct. We can see that BbG has a much lower entropy in its predictions, meaning it significantly outperforms the dropout model with respect to uncertainty. We do not have a similar comparison to other Bayesian neural network models, but we hypothesize that it would be difficult to stop training at the right time to preserve a good predictive uncertainty, just as we saw with the regression problem.

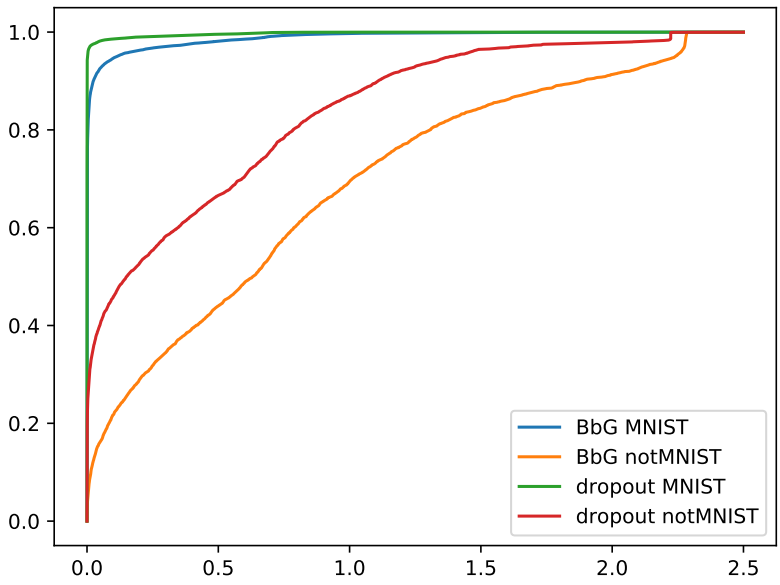


Figure 5.8: Cumulative distribution of entropy on the MNIST and notMNIST dataset for dropout and BbG trained on the MNIST dataset.

Chapter 6

Discussion and Conclusion

In this final chapter we discuss our contribution to the field of Bayesian deep learning. We discuss how our work might be extended upon, and what new methods could spring from our work.

6.1 Discussion

When we started our research into a new method for Bayesian inference in neural networks we were wondering if generative adversarial networks could be used to generate a flexible posterior distribution over the parameter space in a Bayesian neural network. We hoped that the use of generative adversarial networks for generating posterior distributions could achieve better predictive uncertainty than existing methods.

Proposition 1 gives a theoretical proof that a neural network can indeed be used to approximate the KL-divergence between two distributions. In Figure 5.1 we back this up with empirical evidence. This provides an essential foundation for the viability of our method. We go on to to show that using the KL-divergence estimated by the discriminator we can train a generative network to approximate a distribution. We see in Figure 5.3 that the KL-divergence between the approximated distribution and the true distribution is almost zero. Finally, in Figure 5.6 we show the performance of our method relative to other known methods on a regression problem. All these results show that it is possible to use generative adversarial networks to approximate the posterior distribution in neural networks. Because quantitative evaluation of posterior approximation is difficult, we opted to use the same regression problems found in multiple other papers [Pawlowski et al., 2017; Louizos and Welling, 2017]. These results show that the mean prediction of our model makes a good fit compared to

other Bayesian neural network methods. Yao et al. [2019] show that Bayesian inference methods in general underestimates uncertainty, which we also see in Figure 5.6, particularly in areas with little data. This is attributed to the fact that we are minimizing the KL-divergence between the variational posterior and the posterior. Other divergences between distributions might perform better with regard to uncertainty estimation [Steinbrener et al., 2020]. Our method seems to perform at least on par with existing methods with regards to uncertainty and accuracy. We should note, however, that we were not able to reproduce the results presented in [Louizos and Welling, 2017] and [Pawlowski et al., 2017].

Our method most closely resembles that of Bayes by Hypernet, as we are both using generators as our parameterized distribution. Our methods differentiate in how we approximate the KL-divergence. While their method uses a kernel approximation, we are training a model to approximate the KL-divergence. We hypothesized that a discriminator is more efficient at approximating the KL-divergence, e.g. needs fewer samples to achieve a better approximation of the KL-divergence. We base this on the fact that the kernel approximation has no information about the shape of the prior distribution, while the discriminator will learn the shape of the posterior distribution in its effort to differentiate between the two distributions. This also means that the discriminator will not be able to generally calculate the KL-divergence between two distributions, only between that specific distribution and a prior.

6.2 Contributions

We show that generative adversarial networks can be used to approximate posterior distributions in Bayesian neural networks. We find that it is possible to use discriminators to approximate the KL-divergence between distributions, and that this approximation can be used to train generators on fitting a prior distribution. We have developed a method using generative adversarial networks for posterior approximation in Bayesian neural networks. We have shown through experimental results that our method outperforms other state-of-the-art implementations of Bayesian neural networks on a standard regression problem both with respect to predictive accuracy and uncertainty. We have further results that shows that our method scales to larger networks and achieves high predictive accuracy on the MNIST [LeCun et al., 2010] dataset.

6.3 Future Work

There were a few ideas that presented themselves during development and evaluation of our method, but that we found to be outside the scope of this report. The following

ideas could be interesting to explore

1. We believe that our method should generalize to recurrent neural networks, and it would be interesting to see the performance of our method on those architectures. There seems to be particularly little work on predicting Bayesian posteriors in recurrent neural networks.

2. Optimize the network structure of the generator and discriminator, along with other hyper-parameters. It is not clear what is a sufficient generator or discriminator size for good convergence. Because our method does not behave like regular neural networks or GANs, it is also interesting to see how other hyperparameters can affect convergence and performance. It is also interesting to see whether the right hyperparameters could make our model competitive with respect to predictive accuracy on the MNIST dataset.

3. Look at different loss functions for both the generator and the discriminator. It would be interesting to see if it is possible to use GANs to minimize other divergence measures besides KL-divergence. This could possibly reduce the underestimation of variance.

4. Look at the performance of BbG on reinforcement learning problems compared to existing models, and whether or not it has a favorable exploration/exploitation trade-off. Blundell et al. [2015] looked at the performance of Bayes by Backprop on reinforcement learning problems in their paper.

Bibliography

Arjovsky, M. and Bottou, L. (2017). Towards Principled Methods for Training Generative Adversarial Networks. *ICLR*.

Arjovsky, M., Chintala, S., and Bottou, L. (2017). Wasserstein Generative Adversarial Networks. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 214–223, International Convention Centre, Sydney, Australia. PMLR.

Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight Uncertainty in Neural Network. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1613–1622, Lille, France. PMLR.

Bulatov, Y. (2011). notMNIST.

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2015). The Loss Surfaces of Multilayer Networks. *arXiv:1412.0233 [cs]*. arXiv: 1412.0233.

Dinh, L., Krueger, D., and Bengio, Y. (2015). NICE: Non-linear Independent Components Estimation. *arXiv:1410.8516 [cs]*. arXiv: 1410.8516.

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

Ghahramani, Z. and Beal, M. J. (2001). Propagation Algorithms for Variational Bayesian Learning. In Leen, T. K., Dietterich, T. G., and Tresp, V., editors, *Advances in Neural Information Processing Systems 13*, pages 507–513. MIT Press.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. Adaptive computation and machine learning. The MIT Press, Cambridge, Massachusetts.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative Adversarial Networks. *Advances in Neural Information Processing Systems*, 3.

Jiang, B. (2018). Approximate Bayesian Computation with Kullback-Leibler Divergence as Data Discrepancy. In Storkey, A. and Perez-Cruz, F., editors, *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pages 1711–1721, Playa Blanca, Lanzarote, Canary Islands. PMLR.

Kullback, S. and Leibler, R. A. (1951). On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86. Publisher: Institute of Mathematical Statistics.

LeCun, Y., Cortes, C., and Burges, C. (2010). MNIST handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2.

Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867.

Louizos, C. and Welling, M. (2017). Multiplicative Normalizing Flows for Variational Bayesian Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML17, pages 2218–2227, Sydney, NSW, Australia. JMLR.org.

Pawlowski, N., Rajchl, M., and Glocker, B. (2017). Implicit Weight Uncertainty in Neural Networks. *ArXiv*, abs/1711.01297.

Radford, A., Metz, L., and Chintala, S. (2016). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *arXiv:1511.06434 [cs]*. arXiv: 1511.06434.

Ranganath, R., Gerrish, S., and Blei, D. M. (2013). Black Box Variational Inference. *arXiv:1401.0118 [cs, stat]*. arXiv: 1401.0118.

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

Robbins, H. and Monro, S. (1951). A Stochastic Approximation Method. *Annals of Mathematical Statistics*, 22(3):400–407. Publisher: Institute of Mathematical Statistics.

Roberts, G. O., Gelman, A., and Gilks, W. R. (1997). Weak Convergence and Optimal Scaling of Random Walk Metropolis Algorithms. *The Annals of Applied Probability*, 7(1):110–120. Publisher: Institute of Mathematical Statistics.

Steinbrener, J., Posch, K., and Pilz, J. (2020). Measuring the Uncertainty of Predictions in Deep Neural Networks with Variational Inference. *Sensors*, 20(21):6011. Number: 21 Publisher: Multidisciplinary Digital Publishing Institute.

Yao, J., Pan, W., Ghosh, S., and Doshi-Velez, F. (2019). Quality of Uncertainty Quantification for Bayesian Neural Network Inference. *arXiv:1906.09686 [cs, stat]*. arXiv: 1906.09686.

Appendices

Appendix A

Implementation of all other Bayesian neural network methods are from `https://github.com/pawni/BayesByHypernet`. Each model has a single hidden layer of width 100. BbB, MNF, and BbH was trained for 3 000 epochs. BbG was trained for 70 000 epochs. The prior was an independent Gaussian with mean 0 and standard deviation 1 for all weights for all models. Other model specific parameters are listed below.

Bayes by Backprop

Learning rate is 0.001.

Bayes by Hypernet

The hypernet has two hidden layers of width 32 and 64, respectively. We are using 5 samples from the prior distribution and 5 samples from the posterior distribution in the kernel approximation of the KL-divergence. Learning rate is 0.001.

Multiplicative Normalizing Flow

We use a composition of 2 planar flows (Equation 2.14) for the first layer, and 2 NVP flows with a single layer network of width 50 for the second layer. The auxiliary distributions are identical, except with a width of 100 instead of 50. Learning rate is 0.001.

Bayes by GAN

For our method we use a neural network with a two hidden layers of width 32 and 64, respectively, as the generator. The discriminator is a three-layer network, all of width 64. This applies to both layers of the network. The noise vector is of size 10. Learning rate is 0.0001.

Appendix B

Proof for Section 2.2.4 (Variational Inference)

Lemma 1

$$\arg\max_{q_i \in \mathcal{Q}} \textit{ELBO}(q_i) = \frac{\exp\left(\mathbb{E}_{\boldsymbol{\theta}_{-i} \sim q_{-i}}\left[\log p(\theta_i|\boldsymbol{\theta}_{-i}, \mathcal{D})\right]\right)}{\int_{\Theta_i} \exp\left(\mathbb{E}_{\theta_i \sim q_{-i}}\left[\log p(\theta_i|\boldsymbol{\theta}_{-i}, \mathcal{D})\right]\right) \mathrm{d}\theta_i}$$

Proof.

$$\begin{aligned}\textsc{elbo}(q_i) = \int_{\Theta_i} q(\theta_i) \mathbb{E}_{\boldsymbol{\theta}_{-i} \sim q_{-i}}\left[\log p(\theta_i|\boldsymbol{\theta}_{-i}, \mathcal{D})\right] \mathrm{d}\theta_i \\ - \int_{\Theta_i} q_i(\theta_i) \log q_i(\theta_i|\mathcal{D}) \, \mathrm{d}\theta_i \\ + \lambda \left(\int_{\Theta_i} q_i(\theta_i) \, \mathrm{d}\theta_i - 1 \right)\end{aligned}$$

Now to find the maximum we will take the functional derivative with respect to q_i and set it equal to zero, in the same fashion we would optimize any other function.

$$\begin{aligned}\frac{\delta \textsc{elbo}(q_i)}{\delta q_i} &= \mathbb{E}_{\boldsymbol{\theta}_{-i} \sim q_{-i}}\left[\log p(\theta_i|\boldsymbol{\theta}_{-i}, \mathcal{D})\right] - \log q_i(\theta_i|\mathcal{D}) - 1 + \lambda = 0 \\ \frac{\mathrm{d}\textsc{elbo}(q_i)}{\mathrm{d}\lambda} &= \int_{\Theta_i} q_i(\theta_i) \, \mathrm{d}\theta_i - 1 = 0\end{aligned}$$

This gives us the following relation for the optimal function

$$\begin{aligned}q_i^*(\theta_i) &= \exp \mathbb{E}_{\boldsymbol{\theta}_{-i} \sim q_{-i}}\left[\log p(\theta_i|\boldsymbol{\theta}_{-i}, \mathcal{D})\right] - 1 + \lambda \\ \lambda &= 1 - \log \int_{\Theta_i} \exp\left(\mathbb{E}_{\theta_i \sim q_{-i}}\left[\log p(\theta_i|\boldsymbol{\theta}_{-i}, \mathcal{D})\right]\right) \mathrm{d}\theta_i,\end{aligned}$$

which altogether gives us

$$q_i^*(\theta_i) = \frac{\exp\left(\mathbb{E}_{\boldsymbol{\theta}_{-i} \sim q_{-i}}\left[\log p(\theta_i|\boldsymbol{\theta}_{-i}, \mathcal{D})\right]\right)}{\int_{\Theta_i} \exp\left(\mathbb{E}_{\theta_i \sim q_{-i}}\left[\log p(\theta_i|\boldsymbol{\theta}_{-i}, \mathcal{D})\right]\right) \mathrm{d}\theta_i}.$$

□