Practical Variational Inference of Bayesian Neural Networks

Andrei-Ioan Bleahu

Matriculation Number: **230031308**

Supervisor: Dr. Lei Fang

School of Computer Science

University of
St Andrews

This thesis is submitted in partial fulfilment for the degree of
MSc Data Science
at the University of St Andrews, School of Computer Science.

July 10, 2025

Abstract

Bayesian Neural Networks (BNNs) offer a principled and mathematically grounded approach to producing calibrated, uncertainty-informed predictions, in contrast to traditional neural networks trained using Frequentist methods. However, this increased interpretability often comes at the cost of significantly greater computational expense. In this work, I explore the practical implementation of several variational inference techniques applied to multilayer perceptrons (MLPs). Specifically, I examine variational dropout and variational dense layers across three architectures: the standard LeNet-300-100 model for classification, a shallow MLP with one hidden layer of 64 units for univariate regression, and the feedforward sublayer within the Transformer Encoder block. Experimental results demonstrate that variational dropout and variational dense layers can induce useful sparsity, allowing weights to be pruned in various proportions. An interesting finding is that Bayesian methods can be used to induce sparsity in already trained models, thus opening the possibility of more compact and deployable models. Experiments also reveal that variational dense layers lead to sparser models for MNIST and Fashion MNIST, compared to variational dropout layers. However, variational dropout performs well when data training is scarce, particularly because of its ability to adapt its learned uncertainty intervals to missing training data. The application of variational inference within the Transformer Encoder yields inconclusive results, raising new questions for further research.

Declaration

I hereby certify that this dissertation, which is approximately 14,700 words long, has been composed by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a degree. This project was conducted by me from March 2025 to July 2025 towards fulfilment of the requirements of the University of St Andrews for the degree of Data Science MSc under the supervision of Dr. Lei Fang.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Acknowledgements

I would like to sincerely thank my supervisor, Dr. Lei Fang, for his constant support throughout the course of this project. His guidance has been a steady compass for this work. Without his explanations and mentorship, my understanding of Bayesian methods would have remained rudimentary at best.

Completing this dissertation would not have been possible without the love and support of my family. Thank you to my wife, Kirke, for giving me the much-needed time and space to work on this project, and for being so understanding and supportive. She believed in me when I doubted myself the most. To my little Ada, who has not yet taken her first steps but gazed with such wonder at my diagrams that I was inspired beyond words. To my mother, for her constant encouragement.

Lastly, I dedicate this acknowledgement to the memory of my great-grandparents and grandparents, who faced insurmountable challenges in very different times, yet ensured that values like honour, dignity, and chivalric courage were passed down.

Prin noi înșine

Contents

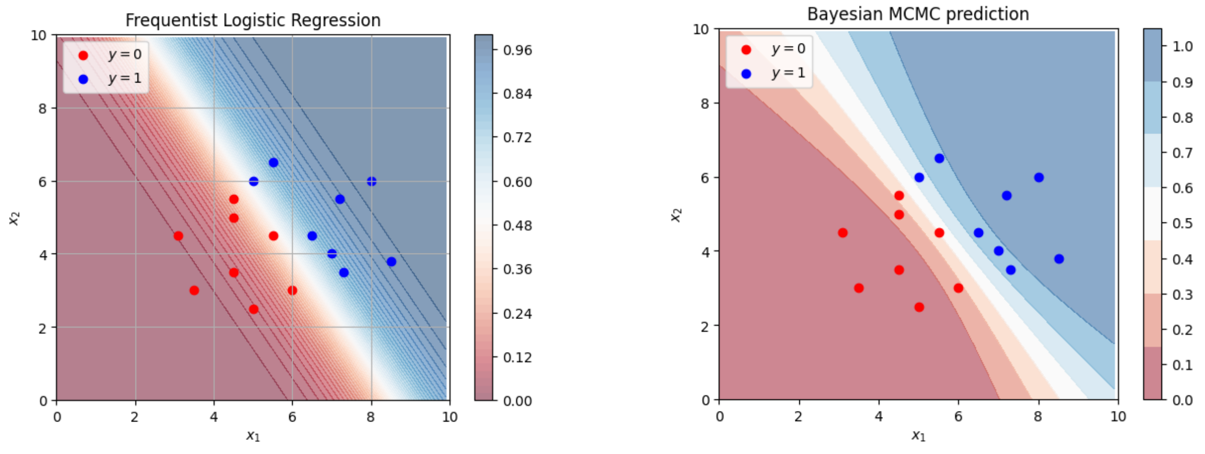
1 Introduction

Neural networks are a particular type of machine learning technology that can be found in many state-of-the-art AI applications. One of the very first neural networks, the perceptron, was modeled based on biological nervous systems(Rosenblatt, 1958). Today, neural networks are far more diverse, and can be found in various technologies. A famous example is the Transformer architecture(Vaswani, 2017), which is a core technology in Large Language Models (LLMs).

According to Srivastava et al. (2014), deep neural networks (DNNs) are very expressive models that can learn very complicated relationships between inputs and outputs. The aim of neural networks (NNs) is to capture nonlinear, usually subtle or otherwise hard to detect, underlying patterns in the training data that are persistent to new data (Samek et al., 2021). In the context of training machine learning models, two main statistical paradigms are applied: Frequentist and Bayesian. These paradigms differ not only in how NNs are trained, but also in how predictions are interpreted. When trained in the Frequentist approach, NNs can suffer from over-confident predictions, perform poorly when training data are scarce, and they lack a principled way to quantify uncertainty, especially for individual predictions (Arbel et al., 2023).

The Frequentist paradigm allows learning a set of parameters – in supervised learning these fixed parameters are also known as weights – without learning any associated confidence measure per parameter. While it is possible to estimate confidence intervals around model parameters using methods such as bootstrapping (Bruce et al., 2020, pp. 161–163), these intervals are not learned during training, unlike the Bayesian paradigm. Learning uncertainties alongside weights is a key feature of the Bayesian paradigm, which can prove advantageous. For example, stochastic variational inference, a type of Bayesian learning, has been demonstrated to be more robust to data set shift, despite yielding lower accuracy compared to ensembles of Frequentist networks (Ovadia et al., 2019, p.5).

To build an intuition about the differences between these paradigms, consider a toy example of binary classification with just 16 training instances. This example, adapted from Assignment 2 of CS5914, illustrates the conceptual distinction between the two approaches. While the Frequentist method yields a hard decision boundary between classes, a Bayesian model using MCMC sampling provides soft probabilistic boundaries, indicating the confidence of the model in its decisions.



(a) Frequentist training

(b) Bayesian MCMC training

Figure 1.1: Comparison between Frequentist and Bayesian logistic regression on a toy binary classification task with 16 training instances. Frequentist training produces a sharp decision boundary. In contrast, Bayesian MCMC training incorporates uncertainty into the predictions, resulting in several probabilistic classification zones.

This dissertation focuses on the implementation of Bayesian Neural Networks (BNNs). Both training and inference will be explored from a Bayesian perspective and compared to the Frequentist approach. The remainder of this section introduces the specific research problem, presents the guiding research questions, and defines the aims and objectives of the study.

1.1 Motivation and problem statement

Bayesian neural networks (BNN) have been studied extensively(Goan & Fookes, 2020; Jospin et al., 2022; Blundell et al., 2015; Molchanov et al., 2017). A commonly agreed definition of a BNN is a stochastic artificial neural network (ANN) trained using Bayesian inference (Jospin et al., 2022, p.2). There are several advantages of BNNs that make them attractive candidates: (i) they provide a natural way to quantify uncertainty, (ii) they can learn from small datasets without overfitting, (iii) they allow integrating prior knowledge into ANNs and (iv) they provide a principled approach to the learning method (Jospin et al., 2022, p.4). Despite these advantages, BNNs are more computationally expensive than their Frequentist counterparts – this is implied in the literature, rather than explicitly stated. In addition, due to their stochastic nature, predictions made by BNNs are inherently probabilistic rather than deterministic. Consequently, averaging over multiple predictions may be necessary for relevant results- BNNs can be envisaged as a particularly interesting case of ensemble methods (Zhou, 2012).

A type of ANN architecture is the Multi-Layer Perceptron(MLP) (Pinkus, 1999). Despite being older than other types of neural networks, MLPs are clearly very relevant today, both as standalone architectures, and as integrated parts of more sophisticated architectures. An interesting example is the incorporation of fully-connected feed-forward networks in the Transformer architecture(Vaswani, 2017) – the two-layered feed-forward components in the encoder and decoder are MLPs. MLPs have also found recent applications in time series forecasting: an interesting application is adapting MLP architectures to make predictions in the frequency domain (Yi et al. 2023). In computer vision, the MLP-Mixer, an architecture based solely on MLPs, has demonstrated results that are as good as Convolutional Neural Networks (CNNs) and Transformer architectures (Tolstikhin et al., 2021), the latter being the go-to, state-of-the-art vision models. Thus, there is potential in investigation Bayesian MLPs.

Bayesian MLPs have been explored, showing promising results in solving the inverse problem in electrical impedance and in image recognition (Vehtari & Lampinen, 2000). However, more recent work on BNNs tends to focus more sophisticated networks. For example, in their work on Variational Dropout(VD), Molchanov et al. (2017) insist more on applying the Bayesian approach to convolutional neural networks (CNN), although they briefly experiment with the Bayesian MLPs LeNet-300-100. MLPs contain layers that implement linear operations followed by non-linear activation functions – see chapter 2 for a detailed summary – which makes them computationally cheaper than many more sophisticated architectures and therefore good candidates for Bayesian inference. This work aims at exploring the benefits of Bayesian MLPs both as standalone architectures and as part of Transformer architectures, through the following research questions.

- Q1 : What advantages and practical challenges come with implementing Bayesian MLPs for supervised learning?
- Q2 : How do different methods of implementing Bayesian MLPs compare?
- Q3 : What are the advantages of replacing feedforward layers (MLPs) with Bayesian Variational Layers in transformer building blocks(Vaswani, 2017)?

1.2 Aims and Objectives

The aims and objectives of this work target the practical implementation of Bayesian (variational) layers in neural networks. In the following, I present the original primary objectives and I briefly explain what has been achieved in this project.

1.2.1 Primary Objectives

- *Objective 1*: Implement several variational inference (VI) methods for Multi-Layer Perceptron networks to approximate posterior distributions of the weights given the training data.

- *Status*: Completed. Implemented three VI methods for MLP layers in the language of Julia, benchmarked them and compared them across experiments.

- *Objective 2*: Evaluate the performance of each of the algorithms on benchmark datasets and simulated data to compare them.

- *Status*: Completed. I made a comparison between three VI methods using different MLP architectures on 5 i.i.d datasets, both real and simulated.

- *Objective 3*: Analyze time/space complexity to evaluate all VI algorithms that are implemented, to understand how they would scale.

- *Status*: Completed. I used `BenchmarkTools`, a Julia package to analyze the performance of the layers when processing data and as part of a training loop.

- *Objective 4*: Where possible, compare the VI implementations with non-Bayesian methods such as MLE (Maximum Likelihood Estimation).

- *Status*: Completed. All networks were pre-trained in the Frequentist fashion until convergence, so comparing VI with non-Bayesian methods, as well as combining the two, has been investigated.

1.2.2 Secondary Objectives

- *Objective 5*: Compare the implemented VI algorithms with the MCMC-based methods, the Laplace approximation, and MAP estimation.

- *Status*: Partially completed. The VI approaches alone require extensive study, so MCMC and MAP were considered for general background. However, MC sampling was used to make predictions with BNNs and MCMCM was used to demonstrate the advantage of the Bayesian paradigm.

- *Objective 6*: Provide a higher-level tool for end-user implementation of the model written in Julia that gives the user a high degree of flexibility and compare with existing implementations (for example, in PyTorch, TensorFlow).

- *Status*: Completed. The tool is a Julia custom module `VariationalMLP.jl` that can be imported into a notebook. This tool can be used to create Variational Layers with three distinct methods, with a great degree of faithfulness to the mathematics.

- *Objective 7*: Compare and test the performance of existing optimizers (SGD, Adam, AdaGrad) in optimizing the lower evidence bound (ELBO).

- *Status*: Completed. This was done during benchmarking, for Adam, RMSProp, Descent (SGD), and AdaGrad.
- *Objective 8*: Extend the Bayesian MLP to probabilistic auto-encoders.
- *Status*: Partially Completed/Refined. As fully Bayesian auto-encoders require a different treatment than Bayesian MLP, progression from Bayesian MLP to auto-encoders may not be straightforward. However, replacing the Feed-Forward layers with Variational Layers in Transformer architectures – which are a type of autoencoders – has been tested.

This investigation builds upon the work of Molchanov et al. (2017), Kingma et al. (2015), and Graves (2011). The practical outcome is represented by experimental results realized via a Julia module, `VariationalMLP.jl`, which enables users to define MLP models incorporating variational dropout and variational dense layers with a high degree of flexibility:

- The number of layers can be specified.
- The sampling procedure via reparameterisation trick and variational objective can be specified.
- Different initialization strategies can be selected. This means that the model can be initialized based on weights and biases obtained from pre-training MLP in the Frequentist approach.
- Activation functions can be specified for each layer.
- The number of neurons can be specified for each layer.

1.3 Structure

The remainder of this dissertation is structured as follows. Chapter 2 introduces most of the mathematical and theoretical background that is necessary for understanding Bayesian inference in general and Variational Dropout in particular. Chapter 3 explains the design decisions behind the Julia implementation and benchmarks the proposed Julia module. Chapter 4 presents experiments, for both classification and regression tasks, for datasets with different properties, and analyzes the results. Chapter 5 draws conclusions, and proposes some future directions for this work.

2 Context Survey

This section presents a review of some seminal papers to provide a theoretical background for this work. It starts by introducing the Multilayer Perceptron(MLP) architecture. Then, it defines the process of "learning", also known as training a neural network, in the Frequentist approach. Following this, it continues with a discussion of the concept of dropout as a method of regularization. Then an explanation of Bayes' rule is provided, followed by a general formulation of the Bayesian approach to training. Then, it proceeds with explaining the mathematical formulation of a particular Bayesian methodology named Variational Inference (VI) – Variational Dropout is a special case of VI. Finally, a discussion of Bayesian inference for training and its connection to Monte Carlo (MC) sampling is provided.

2.1 The Multilayer Perceptron

An extensive overview of neural networks is beyond the scope of this dissertation. Here, I focus on the Multilayer Perceptron(MLP) architecture, a simple but foundational NN architecture. A discussion of MLP, from first principles, is provided by Géron(2023, pp. 309 - 316). Figure 2.1 below is adapted from Figure 10-7 (Géron, 2023, p. 309), and it shows a very shallow MLP architecture with just one hidden layer.

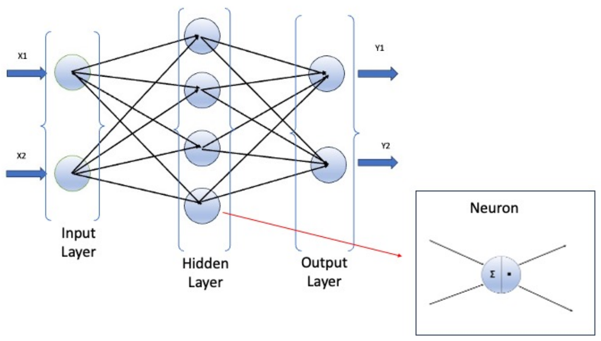


Figure 2.1: An MLP with just one hidden layer. The input layer is designed to receive features X_1 and X_2. The hidden layers consist of four neurons that process the input, before passing it to the output layer with two outputs Y_1 and Y_2. Each neuron performs a weighted sum on incoming inputs, and then applies a layer-specific activation function before propagating the signal further.

Consider the context of "supervised learning", a subset of machine learning tasks. In supervised learning, a dataset $\mathcal{D}$ consists of features and a response. In Figure 2.1, for simplicity, the data consists of just two features X_1 and X_2, and one binary response variable $\mathbf{y}$ – this is a supervised classification problem.

Training a network involves a forward and a backward pass. In the forward pass, the MLP in Figure 2.1 receives two features via its input layer. The number of features must be equal to the number of neurons in the input layer : in the example presented above, there are two neurons for two features. The neurons in the hidden layer first apply a linear transformation on all the incoming input (denoted Σ), before applying a non-linear activation $\square$. Non-linear activations are typically defined per layer, so each of the four neurons in the hidden layers applies the same activation function. It is important to have these non-linear activation functions between layers, otherwise the MLP cannot model non-linear patterns. The outputs from the four neurons pass forward to the next layer of neurons, where two operations are applied: linear weighted sum followed by non-linear activation. When a network contains a deep stack of hidden layers, it is called a Deep Neural Network (DNN) (Géron, 2023, 0. 309).

A detailed explanation of deep learning inference and training can be found in Hoeffler et al. (2021, pp. 5-6). The deep learning model is essentially a graph of parameterizable layers that implements a complex non-linear function (Hoeffler et al, 2021, p.5). Based on Hoeffler et al. (2021, p.5), I developed the following definition for supervised learning:

Definition 2.1.1. *For a training set, composed of pairs of features* $\mathbf{X_{train}} \in \mathcal{X}$ *and labels* $\mathbf{y_{train}} \in \mathcal{Y}$*, supervised learning is to learn the function* $f : \mathcal{X} \to \mathcal{Y}$ *parameterized by a set of weights* $\mathbf{w} \in \mathcal{R}^d$*, so that given an input* $\mathbf{X} \in \mathcal{X}$*, the prediction* $f(\mathbf{X}; \mathbf{w})$ *is close to* $\mathbf{y} \in \mathcal{Y}$*.*

Training also means computing gradients "backwards"; this is called a backward pass. The backward pass allows propagating gradients backwards, a feedback mechanism. Training a network is essentially an optimization problem and the objective of this optimization is, in the Frequentist paradigm, a loss function that describes how far removed are the learned model's parameters from the "true" signal. Deep neural networks, consisting of multiple layers, are usually trained using a penalized maximum likelihood objective to find the set of parameters (Murphy, 2023, p.647). The learned weights $\mathbf{w}$ can be later applied to data consisting of new features $\mathbf{X_{new}}$, resulting in new responses $\mathbf{y_{new}}$ – these new responses are called predictions. The network is usually evaluated on a separate set of examples that were not used to train the model so that the generalization capability of the trained model to unseen data can be evaluated (Hoeffler, 2021, p.5). The trained model will output fixed parameters without uncertainties (although these can be calculated via various methods), so it outputs definite answers; the ability to produce uncertainty estimates is a critical feat in many applications (Jospin et al., 2022, p.3).

2.1.1 Dropout

With limited training data, NNs can lead to overfitting – they can learn the noise from the data rather than the signal (Srivastava et al., 2014). "Dropout" means dropping out units in a neural network during the training phase. In practice, applying dropout means deactivating neurons, with a certain probability p, resulting in a "thinned" network consisting of the units that survived dropout (Srivastava et al., 2014, p. 1930). This selection of pruned neurons occurs probabilistically for each iteration, which means that the resulting network becomes an ensemble of sub-networks (Srivastava et al., 2014, p. 1934), each trained on different configurations of the original networks. Figure 2.2 aims at an intuitive understanding of dropout, and it captures one "snapshot" during one of the forward passes : certain neurons go "offline", and the resulting network that is trained is much simpler.

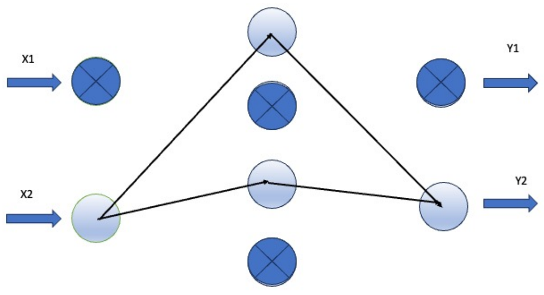


Figure 2.2: An MLP with two inputs, features X_1 and X_2, on hidden layers consisting of four neurons, and one output layer with two outputs Y_1 and Y_2 with Dropout. During a training pass, half the neurons are "switched" off, which for such a small network leads to quite dramatic simplification.

Dropout is like an ensemble method. For a neural network with n units, applying dropout means training a collection of 2^n thinned networks with extensive weight sharing, where each thinned network gets trained very rarely, if at all (Srivastava et al., 2014, p.1930). Applying dropout can also be seen as injecting noise into the networks nodes during training (Li & Liu, 2020).

To understand dropout from the Bayesian perspective, consider the example of an intermediate layer of the BNN where dropout is applied. As discussed in Molchanov et al. (2017), the operation of a layer can be reduced to matrix multiplication as follows:

- $M \times I$ input matrix $\mathbf{A}$
- $I \times N$ Weights matrix $\mathbf{W}$
- $M \times N$ Resulting matrix $\mathbf{B}$

- Before any non-linearity is applied: $\mathbf{B} = \mathbf{AW}$

Dropout is seen as **adding multiplicative noise** to the input layer $\mathbf{A}$, so that $\mathbf{B} = (\mathbf{A} \odot \Xi)\mathbf{W}, \quad \xi_{mi} \sim p(\xi)$. and ξ can be sampled from different probability distributions. For example, if $p(\xi) - Bernoulli(1-p)$, then each element of the input matrix $\mathbf{A}$ is set to 0 with a probability p. Molchanov et al. (2017) equate putting multiplicative Gaussian noise ξ_{ij} on weight w_{ij} to sampling w_{ij} from a Gaussian parametrized by θ_{ij}.

Dropout can be understood as either shutting off neurons, or as inserting Gaussian noise. The perspective of injecting noise is important to the Bayesian paradigm, which considers dropout as a variational parameter that is learned during training, rather than as a fixed hyper-parameter to be set before training commences.

2.1.2 Pruning

Pruning weights has emerged as an effective approach to model compression, especially when deploying large, accurate deep learning models would pose problems in energy efficiency (Zhu & Gupta, 2017; Anwar et al., 2015). Model compression techniques based on pruning remain widely explored, especially as they provide several advantages to model deployment; In their review paper, Li et al.(2023) provides a synthesis of the advantages of pruning weights based on previous works:

- Conserves storage space, especially on edge devices.
- Reduces computational demand and speeds up model inference.
- Reduces the complexity of the model and prevents overfitting.
- Reduces training time and computational resource consumption, thus reducing training costs.
- Improves the deployability of the model, as smaller models are easier to deploy on edge devices

This is especially interesting due to the integration of deep learning networks in mobile devices, which are energy constrained and cannot handle the amount of processing required by DNNs – such computation is usually hosted on high-end cloud servers (Kang et al., 2017, p. 615). Partitioned DNNs between Edge and Cloud have been widely researched (Varghese et al., 2016). The important idea for this work is that more compact models on the Edge improve deployability.

A structured way to prune trained models is via a sparsity function (Zhu & Gupta, 2017, pp. 2-3). Sparsity can be understood as the percentage of weights that do not contribute anything to inference, so they can be safely removed (pruned), resulting in a more compact model, at negligible loss of predictive performance. Considering the example in the previous section, it is the weight matrices **W** that are pruned. This means pruning can be associated to given layers, but it is not applied to the layers themselves, but to the parameters linking the layers of a NN.

The work by Molchanov et al.(2017) shows how variational dropout can induce useful sparsity. Sparse models are also discussed in Graves(2011) and Kingma et al. (2015). Sparsity proves useful as an informed way to prune weights without much loss of accuracy, potentially facilitating the deployment of significantly smaller models at only a small cost in predictive performance.

2.2 Bayes' Rule

This section summarizes the Bayesian approach applied to supervised learning. The crux of the Bayesian approach lies in the following equation (Barber, 2012, p.18):

$$p(w \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid w)p(w)}{p(\mathcal{D})} \tag{2.1}$$

Equation 2.1 above is Bayes' rule. I will give an interpretation of each term, based on Barber (2012, p.18) :

- $p(w \mid \mathcal{D})$ is the posterior distribution. This term can be interpreted as the probability of the model parameters w given the observed data $\mathcal{D}$. This is a very natural interpretation of a model : model parameters $\mathbf{w}$ have an associated probability distribution based on observed data $\mathcal{D}$. In the Bayesian paradigm, learning such a distribution is the goal of training, unlike learning the set of point-like weights in the Frequentist approach.

- $p(\mathcal{D} \mid w)$ is the likelihood, a generative model of the data given the parameters. The likelihood can be chosen based on the type of observed data. For example, for a regression model the likelihood can be a Gaussian distribution, while for a binary classification problem the likelihood can be categorical distribution(Blundell et al., 2015).

- There is also a prior belief about the parameters, $p(w)$. This is actually a significant advantage of the Bayesian approach, as it allows to control the distribution from which the parameters are drawn – it can be interpreted as a form of regularization.

- Finally, the denominator $p(\mathcal{D})$ is the evidence term, and it is simply the probability of observing the data.

The evidence term is a constant that can be obtained by integrating over the conditional probability, which means it is an integral over the prior multiplied by the likelihood:

$$p(\mathcal{D}) = \int p(w)p(\mathcal{D} \mid w)dw \tag{2.2}$$

However, in practice, $p(\mathcal{D})$ is often intractable (Blei, 2012, p.4), which means it cannot be computed in closed form. Hence, a direct calculation of the posterior distribution in Eq. (2.1) is often not possible. In practice, resorting to an approximation is desirable – this will be discussed in detail in the next subsections.

Bayes' rule provides a principled scientific method for inference – it is a very natural approach to modeling. This is because given some assumptions of the type of generative model(likelihood), coupled with assumptions of the parameters w (the prior), the conditional probability of the parameters given the data, that is, the model, can be inferred. The Bayesian approach does not exist in a vacuum : it can be linked to well-tested methods such as Maximum Likelihood Estimation (MLE). In fact, it can be shown that Bayes's rule applied to learning is a more general case of Maximum Likelihood Estimation(MLE), or alternatively, that MLE is a specific case of Bayesian inference.

2.3 MAP and MLE

The term 'likelihood' is used for the conditional probability that a model generates new data (Barber, 2012, p.184). There is an unspoken assumption for the terms in Eq. (2.1) : the data is generated by some model $\mathcal{M}$ such that the likelihood $p(\mathcal{D} \mid w)$ is a conditional probability on the model $p(\mathcal{D} \mid w, \mathcal{M})$ (Barber, 2012, p.184). However, since the weights are assumed to be produced by the model, the weights imply the existence of the model, so the likelihood can be simply expressed as $p(\mathcal{D} \mid w)$ without loss of rigor.

The Maximum Likelihood (ML) methodology is a staple of statistical signal processing. As in Tzikas et al. (2008, p.132), for some observed data $\mathcal{D}$ and parameters w of the unknown model $\mathcal{M}$, the ML estimate is:

$$\hat{w}_{ML} = \arg\max_{w} p(\mathcal{D}; w) \tag{2.3}$$

The notation $p(\mathcal{D}; w)$ implies that w are parameters of the model that generated the data (Tzikas et al., 2008, p.132). Maximum likelihood estimation is closely related to Maximum a Posteriori (MAP), the latter being known as "Poor Man's Bayesian Inference" (Tzikas et al., 2008, p. 133). In MAP the posterior is maximized, not the likelihood, as follows:

$$\hat{w}_{MAP} = \arg\max_{w} p(w \mid \mathcal{D}) \tag{2.4}$$

Tzikas et al.(2008), use the Expectation-Maximization (EM) algorithm for MAP estimation, but the principle behind MAP remains very simple : instead of maximizing the likelihood, it is better to maximize the posterior because of the incorporation of extra information in the form of a prior $p(w)$. This is actually the important link between MAP and MLE. Clearly, setting $p(w) = C$, where C is a constant, means MAP is reduced to MLE. This is because constants under maximization/minimization operations do not influence the optimized quantity. Since $p(\mathcal{D})$ is already a constant w.r.t w, $w_{\hat{MAP}} = w_{\hat{MLE}}$. This is an important result, because it shows that the Bayesian approach, is actually a more general form of MLE through the incorporation the prior $p(w)$.

2.4 Bayesian SVM

A general Bayesian framework for obtaining sparse solutions for linear regression and classification models has been proposed by Tipping (2001). The Bayesian SVM (Tipping, 2001) remains highly relevant as it demonstrates many of the advantages of working within the Bayesian paradigm. In the following, a summary of Tipping's(2001) approach will be provided. First, consider that the data to be predicted, the signal $\mathbf{y}$, can be expressed as a linear combination of fixed basis expansions applied to the features (Tipping, 2001, pp. 211 - 212):

$$\mathbf{y} = \sum_i w_i \phi_i(\mathbf{x}_i) = \mathbf{\Phi w} \tag{2.5}$$

In the above, $\mathbf{w} = (w_1, w_2, \cdots, w_i, \cdots, w_m)^T$ are the weights of the linear combination. The design matrix can be defined as $\mathbf{\Phi} = (\phi_1, \cdots, \phi_m)$, where the basis expansion functions $\phi_m = (\phi_m(\mathbf{x_1}), \cdots, \phi_m(\mathbf{x_N}))^T$ are applied to the features (Tipping, 2001, pp. 211 - 212). Thus Eq. (2.5) provides a formulation that is not at all dissimilar to classical neural networks : the response $\mathbf{y}$ is a linear combination, the weighted sum over the transformed features $\phi_i(\mathbf{x}_i)$. Essentially, the design matrix is a form of feature engineering. Comparing this with the MLP in section 2.1, it becomes clear that the order of transformations somewhat changes: the features are first transformed according to the non-linear basis expansion functions, and then the parameters of a linear combination of the transformed features are learned. The important similarity is that the the non-linearity introduced by the design matrix serves the same role as the non-linear activation functions in MLPs.

In choosing the likelihood of this model, Tipping makes two assumptions: (i) that the observations are independent and (ii) that the data are drawn from a Gaussian distribution. The equation below is the log-likelihood function, based on Tipping (2001):

$$\ln p(\mathbf{y}|\mathbf{w}, \sigma^2) = -\frac{n}{2} \ln (2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y^{(i)} - (\mathbf{\Phi w})^{(i)})^2 \tag{2.6}$$

In Eq. (2.6) there are as many parameters $\mathbf{w}$ as labels $\mathbf{y}$, which means the model is prone to severe overfitting without regularization (Tipping, 2001, p. 214). This is another similarity to MLPs, as these networks are also vulnerable to overfitting without dropout. The Bayesian approach offers an elegant way of applying regularization: choosing a prior that constrains the weights.

$$p(\mathbf{w}|\mathbf{0}, \alpha) = \mathcal{N}(\mathbf{w}; 0, \mathbf{A}^{-1}) \tag{2.7}$$

Eq. (2.7) shows how the weights are regularized, implicitly showing the assumptions that come with this particular form. The first assumption is, clearly, that the weights are drawn from a Gaussian distribution $\mathcal{N}$ - so most weights are expected to be close to the mean. Since the mean of this Gaussian is 0, this type of regularization reflects the belief that all weights are small. The weights are effectively controlled by individual variances α_m^{-1} that are diagonal elements of the matrix $\mathbf{A} = diag(\alpha_1, \cdots, \alpha_m) = diag(\alpha)$. Hence, this model assumes that the effect of each feature is small, so that the signal is constructed from many equally important features.

Beyond regularization, two characteristics of this particular Bayesian approach can be noted :

- Each weight w_m is drawn from a Gaussian with mean 0 and variance α_m^{-1}. This means that the confidence in parameters α_m^{-1} are also learned, which is very different from the Frequentist cases where only the weights are learned.
- The aim is learning a probabilistic model akin to the posterior $p(\mathbf{w}|\mathbf{y}, \mathbf{A}^{-1})$, from which the weights can be sampled.

A Bayesian model will be stochastic, as each prediction presupposes sampling a new set of weights from the learned distribution, ideally from $p(\mathbf{w}|\mathbf{y}, \mathbf{A}^{-1})$, before making a prediction. In this case, the learned parameters are the elements of the matrix $\mathbf{A}$. In terms of computational complexity, it becomes apparent that the Bayesian approach is not cheap, but it provides advantages as it is interpretable. However, learning the exact distribution in many cases is not feasible. There are two big classes of methods for learning an approximate distribution to the true posterior : MCMC and Variational Inference. Variational inference is the focus of the next section.

2.5 Variational Inference

In this section, I aim to explain the key mathematical equations behind variational inference (VI) . In general terms, VI is a Bayesian methodology : given data $\mathcal{D}$, the aim is to learn an approximate distribution $q(w)$ of the true posterior $p(w \mid \mathcal{D})$. To measure how close the approximated distribution is to the true distribution, a similarity metric named the Kullback-Leibler (KL) divergence is used.

2.5.1 Information theory

The KL divergence is a key metric for Variational Inference that measures the quality of an approximated distribution q with respect to a target distribution p (Molchanov et al., 2017; Kingma et al., 2015, Blei et al. 2018; Graves, 2011). Eq. (2.8) gives the mathematical expression of the KL divergence.

$$\mathrm{KL}(q \,||\, p) = \int q(w) \ln \left[\frac{q(w)}{p(w \mid \mathcal{D})} \right] dw \tag{2.8}$$

In Eq.(2.8) above, $q(w)$ is the approximate distribution of the real posterior distribution $p(w \mid \mathcal{D})$ and the use of an integral instead of a sum means that these probability distributions are continuous rather than discrete. A discussion of the KL Divergence, through the lens of Information Theory, is provided by Bishop (2006) in *Pattern Recognition and Machine Learning.* If q is used to construct a coding scheme for transmitting values of w to a receiver, then the average additional amount of information required to specify w as a result of using q instead of p is given by the KL divergence (Bishop, 2006, p.55). Bishop simply follows Eq.(2.8), which is simply the expectation value over the q distribution of the logarithm of the two differences:

$$\mathrm{KL}(q \,||\, p) = \langle \ln \left[\frac{q(w)}{p(w \mid \mathcal{D})} \right] \rangle_{q(w} \tag{2.9}$$

Clearly, the logarithm fraction presented above can be expressed as a difference:

$$\mathrm{KL}(q \,||\, p) = \langle [\ln q(w) - \ln p(w \mid \mathcal{D})] \rangle_{q(w} \tag{2.10}$$

From an information theory perspective, the information content $h(w)$ of a probability distribution $p(w)$ can be expressed as $h(w) = -\ln p(w)$, with the choice of the basis of the logarithm being arbitrary (Bishop, 2006, pp. 48 - 49). Thus, the KL divergence is simply the expected value of the difference in information content between the two distributions $q(w)$ and $p(w)$. Now, given that the expectation value of a distribution converges to the mean for large samples, it follows that the description given by Bishop(2006, p.55) is very intuitive : since the KL divergence is the expected amount of information loss when choosing $q(w)$ instead of $p(w)$, minimizing the KL divergence between $q(w)$ and $p(w)$ means choosing a distribution $q(w)$ that would result in minimal information loss when sending values of w to a receiver. The problem of finding the approximated distribution $q(w)$ to the real intractable posterior $p(w)$ is reduced to optimizing the KL divergence:

$$q^* \leftarrow \arg\min_{q(w)} \mathrm{KL}(q \,||\, p) \tag{2.11}$$

The KL divergence has some interesting properties. First, it is not symmetric, which implies it is not a distance. This can be proven by simply swapping q and p in Eq. (2.8).

It is trivial to show :

$$\mathrm{KL}(q\,||\,p) \neq \mathrm{KL}(p\,||\,q) \tag{2.12}$$

Second, the KL divergence satisfies $KL(q\,||\,p) \geq 0$ with equality if and only if $p(x) = q(x)$ (Bishop, 2006, p. 55). If $q(w) = p(w \mid D)$, the logarithm in Eq. (2.8) is $\ln 1 = 0$, which means $KL(q\,||\,p = q) = 0$. It can be shown using Jensen's inequality that KL divergence is strictly positive for all other cases (Bishop, 2006, p.56). To understand the KL divergence better, I will review another information theory key concept, namely entropy. For a discrete signal, Bishop(2006, p.49) gives the following definition of entropy:

$$H[x] = -\sum_x p(x) \log_2 p(x) \tag{2.13}$$

The above definition is the entropy of a random variable x, and it represents the average amount of information that a sender transmits to a receiver (Bishop, 2006, p.49). Since the above is simply the negative expectation value over $p(x)$, I will adapt this to the continuous case and choose e as the logarithm's basis. Hence, for a continuous probability $q(w)$, the entropy becomes:

$$H(q(w)) = -\int q(w) \ln q(w)\, dw \tag{2.14}$$

Given that the amount of information transmitted is simply $-lnq(w)$, the entropy of the random variable w is the expected amount of information w transmitted between the sender and the receiver. KL divergence is practically a difference of the entropy of the two distributions, the true posterior $p(w \mid D)$ and the approximate distribution $q(w)$. But how does this apply, practically, to the "learning" of the approximate distribution $q(w)$? In the next section, I will show that finding the approximate distribution is actually an optimization problem involving both the entropy and the KL divergence.

2.5.2 Lower Bound(Variational Energy)

In practice, the KL divergence cannot be directly computed due to the presence of the constant evidence term $\ln[p(D)]$. However, when optimizing, constants do not matter, so instead of optimizing the full KL divergence, one can optimize the KL divergence minus the constant $\ln[p(D)]$; this is known as the lower bound (Blei et al., 2018, p.6). To demonstrate this, I will return to Eq.(2.8). Since the logarithm of a fraction is the difference between the logarithm of the numerator and the logarithm of the denominator:

$$\mathrm{KL}(q\,||\,p) = \int q(w) \ln[q(w)]\, dw - \int q(w) \ln[p(w \mid \mathcal{D})]\, dw \tag{2.15}$$

The first term of the equation above is just the negative entropy $H(q)$:

$$\mathrm{KL}(q\,||\,p) = -H(q) - \int q(w) \ln[p(w \mid D)]\, dw \tag{2.16}$$

For the second term, one can use the definition of the posterior $p(w \mid D) = \frac{p(w,D)}{p(D)}$, to express everything in terms of the joint distribution $p(w, D)$. This leads us to the following expression:

$$\mathrm{KL}(q\,||\,p) = -H(q) - \int q(w) \ln[p(w, D)]\, dw + \int q(w) \ln[p(D)]\, dw \tag{2.17}$$

The expectation value of a constant, is simply that constant. So the third integral on the right hand side is simply the logarithm of the evidence term. This leads to the following equation:

$$\mathrm{KL}(q\,||\,p) = -H(q) - \int q(w) \ln[p(w, D)]\, dw + \ln[p(D)] \tag{2.18}$$

When optimizing the KL divergence with respect to q, the term $\ln[p(D)]$ is just a constant, so it can be ignored under optimization. Plucking the form above into the Eq. (2.11) yields:

$$q^* \leftarrow \arg\min_{q(w)} \left(-H(q) - \int q(w) \ln[p(w, D)]\, dw \right) \tag{2.19}$$

So the optimization problem is now reduced to Eq. (2.19) above. Previously, I discussed that the KL divergence is always positive or equal to 0. This means:

$$\ln(p(D)) \geq H(q) + \int q(w) \ln[p(w, D)]\, dw \equiv ELBO(q) \tag{2.20}$$

The term on the RHS , which is the negative of term under minimization in Eq.(2.20), is lower bounded by the natural logarithm of the evidence term. This is naturally called the **E**vidence **L**ower **BO**und, or ELBO. In Eq. (2.20), it is the negative of the lower bound that is being minimized, which is the same as maximizing the lower bound.

$$q^* \leftarrow \arg\max_{q(w)} ELBO(q) \tag{2.21}$$

This result is quite interesting, for at least two reasons:

- The problem of finding the approximate distribution q is reduced to maximizing the lower bound (ELBO).
- The ELBO is just the expectation value of the full joint distribution $p(w, D)$ regularized by the entropy term $H(q)$

The Lower Bound is explained extensively in literature (Tran et al, 2021; Blei et al., 2018, p.6; Kingma et al., 2015, p.2; Molchanov et al., 2017), as the optimization objective for Variational Inference. There is also an alternative formulation approach to maximizing the lower bound, which is minizing the variational free energy (Graves, 2011). Naturally, going from one method to another is straightforward, because minimizing a function is the same as maximizing its negative. Thus, the variational free energy is simply the negative of the lower bound $\mathcal{F}(q) = -ELBO(q)$. This means that finding the best approximate distribution can be framed as minimizing the variational free energy:

$$q^* \leftarrow \arg\min_{q(w)} F(q) \tag{2.22}$$

For completeness, the variational free energy can be simply written in the following form:

$$\mathcal{F}(q) = -H(q) - \int q(w) \ln[p(w, D)] \, dw \tag{2.23}$$

The variational free energy $\mathcal{F}$, can practically be minimized with Gradient Descent. The particular form of $q(w)$ is important, as it allows an analytical form of the variational free energy. Thus, a "fixed" form approach to the distribution $q(w)$ can be taken, which means learning the parameters of the fixed distribution – this is the focus of the next section.

2.5.3 Fixed-form Variational Inference

Tran et al. (2021) give a very detailed practical tutorial on Variational Bayes. The tutorial addresses two forms of Variational Inference : Mean Field Variational Bayes and Fixed-form Variational Bayes (Tran et al., 2021). For BNNs, the Mean Field approach would fail because the joint distribution $p(w \mid \mathcal{D})$, presented in Eq. (2.23), does not have a clean form for neural networks. However, Fixed Form Variational Bayes is suitable for BNN. Fixed form means the approximating distribution is fixed (known), for example it belongs to some class of distributions (Tran et al., 2021, p.11). A special case of fixed form variational inference is when approximate distribution is a Gaussian distribution. Tran et al. (2021, p.23) discuss several variants of Gaussian Variational Bayes, but in the following I will only introduce the main ideas behind Cholesky decomposition as the other examples are strikingly similar. I will start with a Gaussian fixed form:

$$q(\mathbf{w}) = \mathcal{N}(\mathbf{w}; \mathbf{m}, \mathbf{V}) \tag{2.24}$$

The above is a Gaussian with mean $\mathbf{m}$ and covariance $\mathbf{V}$. This means that to "learn" the distribution, it is sufficient to "learn" its mean and variance, which are variational parameters. Expressing the above analytically and taking the logarithm gives the following:

$$\ln q(\mathbf{w}) = -\frac{D}{2}\ln(2\pi) - \frac{1}{2}\ln(|\mathbf{V}|) - \frac{1}{2}(\mathbf{w}-\mathbf{m})^{\mathbf{T}}\mathbf{V}^{-\mathbf{1}}(\mathbf{w}-\mathbf{m}) \tag{2.25}$$

The problem now becomes taking the gradients of the expectation of the full joint, an expectation which depends on $q(\mathbf{w})$. Here, one can apply reparameterisation trick that can reduce the variance (Tran et al., 2021, p.23). Instead of taking the expectation over the approximate distribution $q(\mathbf{w}) = \mathcal{N}(\mathbf{w}; \mathbf{m}, \mathbf{V})$, which is hard to compute, the expectation can be taken over a different parameter ν such that the following linear transformation exists:

$$\mathbf{w} = \mathbf{m} + \mathbf{L}\nu \tag{2.26}$$

In a way, the linear transformation above means translating the original unknown Gaussian $q(\mathbf{w})$ to a known Gaussian $\mathcal{N}(\mathbf{0}, \mathbf{I})$ from which it is easy to sample. In the above, $\mathbf{L}$ is a lower triangular matrix, where $\mathbf{V}$ uses Cholesky decomposition such that $\mathbf{V} = \mathbf{L}\mathbf{L}^{\mathbf{T}}$ (Tran et al., 2021, p.23). The following identity is useful: $\langle f(\mathbf{w})\rangle_{\mathbf{w}\sim\mathcal{N}(\mathbf{m},\mathbf{V})} = \langle f(\nu)\rangle_{\nu\sim\mathcal{N}(\mathbf{0},\mathbf{I})}$. It can be shown that the gradient with respect to $\mathbf{m}$ on both sides of the identity yields :

$$\nabla_{\mathbf{L}}\langle f(\mathbf{w})\rangle_{\mathbf{w}\sim\mathcal{N}(\mathbf{m},\mathbf{V})} = \nabla_{\mathbf{m}+\mathbf{L}\nu}\langle f(\mathbf{m}+\mathbf{L}\nu)\nu^{T} \circ \mathbf{S}_{\mathbf{L}}\rangle_{\nu\sim\mathcal{N}(\mathbf{0},\mathbf{I})} \tag{2.27}$$

In the above added $\circ$. This is the Hadamard product, a binary operation taking two matrices as input and returning their entry-wise product as output. Here, $\mathbf{S}_{\mathbf{L}}$ is a lower triangular matrix, with 0 above the diagonal and entries of 1 beneath it that ensure the expectation respects the form of the lower triangular matrix $\mathbf{L}$. A different factorization of $\mathbf{V}$ can be applied, but the main idea of using a reparameterisation trick for a fixed form VI remains. This is particularly useful when trying to minimize the variational free energy $F(q)$. As an example, consider the case of fixed basis expansion Tipping(2001), where the signal is modeled according to the design matrix Φ. It can easily be shown that applying the tricks above renders the following form of the variational free energy :

$$\nabla_{\mathbf{m}}F = -\langle\nabla_{\mathbf{m}+\mathbf{L}\nu}\ln p(\mathbf{m}+\mathbf{L}\nu, \alpha, \mathbf{y}, \beta|\mathbf{\Phi})\rangle_{\nu\sim\mathcal{N}(\mathbf{0},\mathbf{I})} \tag{2.28}$$

As seen above, reparameterizing $\mathbf{w}$ can actually be useful because now the sampling process is taken with respect to a known Gaussian $\mathcal{N}(\mathbf{0}, \mathbf{I})$, which greatly simplifies the problem. This type of reparameterisation can be integrated into a forward pass of a neural network such that two steps are followed:

- Step 1 : Sample some noise from a known Gaussian : $\nu \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
- Step 2: Use the reparameterisation trick to obtain the weights: $\mathbf{w} = \mathbf{m} + \mathbf{L}\nu$

In effect, the weights $\mathbf{w}$ are not learned directly, but by sampling ν and then computing them through parameters $\mathbf{m}$ and $\mathbf{L}$ in an iterative way. The fixed form above shows that variational parameters can be effectively learned, from which the weights can be reconstructed. As discussed by Kingma et al. (2015) and Molchanov et al.(2017), variational dropout is simply a case of variational inference where the dropout parameter is treated as a variational parameter, which means it is learned as the network is trained, rather than fixed a priori – Chapter 3 discusses exact methods of implementing Variational Dropout. Making predictions with Bayesian models, as well as approximating expectation values, can be practically achieved via Monte Carlo methods.

2.6 Monte Carlo Methods

In this section, I give brief overview of Monte Carlo methods. MCMC techniques have a long history of application to solve integration and optimization problems, and they are directly applicable to Bayesian Inference (Andrieu, 2003, p. 7; Andrieu & Thoms, 2008). Monte Carlo estimators can be used in the implementation of algorithms, such as the SGVB estimator (Molchanov et al, 2017; Kingma et al, 2015), or simpler Monte Carlo integration such as in Graves(2011, p.4). There is an important distinction to be made between Monte Carlo(sampling) and MCMC(Markov Chain Monte Carlo) methods, which I aim to clarify below.

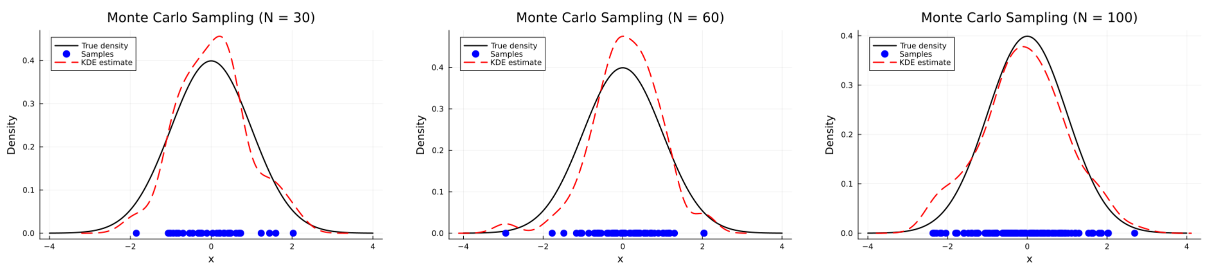
2.6.1 The Monte Carlo Principle

First, I will introduce the Monte Carlo principle, as illustrated by Andrieu et al. (2003, p.8) : given a set of identically distributed set of samples (i.i.d samples) $\{x^{(i)}\}_{i=1}^{N}$ drawn from a target probability density $\rho(x)$, one can construct an empirical point-mass function $\rho_N(x)$ that approximates the target density :

$$\rho_N(x) = \frac{1}{N}\sum_{i=1}^{N} \delta_{x^{(i)}}(x) \tag{2.29}$$

Eq. (2.29) shows that the approximated target is an average over Dirac-delta functions centered at x_i. To visualize it, I will use a Kernel Density Estimation (KDE), which is a useful tool in visualizing how samples can be translated into a distribution. The plot below shows the KDE becomes closer to the original distribution as more samples are drawn.

This type of sampling is directly applicable to trained networks. As in the preceding section, consider a trained Gaussian through VI : $q(w) = \mathcal{N}(\mu, \sigma^2)$. Sampling from this Gaussian means drawing values of the mean μ^k, which represents the expected weight, and the associated uncertainty σ^k. This means that each set of parameters w is actually



(a) $N = 30$ (b) $N = 60$ (c) $N = 100$

Figure 2.3: Monte Carlo approximation improves with increasing number of i.i.d. samples. Each plot shows samples (dots) and the true target distribution (black curve). The KDE distribution is just a post-processing tool, to better visualise the reconstruction.

the mean of the Gaussian with the added standard deviation. Making a prediction with such a network can be reduced to the following three steps:

- Draw a set of weights (sample) from the trained Gaussian : $u^{(k)}, \sigma^{(k)} \sim \mathcal{N}(\mu, \sigma^2)$
- Compute the corresponding weight, different each time based on the drawn parameters : $w^{(k)} = u^{(k)} + \epsilon\sigma^{(k)}$
- Make T predictions on the data, and calculate the inherent standard deviation from these samples.

Applying the model once gives only one probable prediction – one of many possible predictions. This is where the model's probabilistic nature comes into play – each prediction is stochastic, so drawing multiple values allows to derive a statistically significant expectation value. Hence, the Monte Carlo principle can prove to be a very effective tool for Bayesian Inference. Under the assumptions that the samples are i.i.d, the following can be done:

- Sample T samples $\{u^{(k)}, \sigma^{(k)}\}_{i=k}^{T}$, in the same way as before, from $\mathcal{N}(\mu, \sigma^2)$
- Make T predictions, where each prediction has an intrinsic uncertainty.
- Take the mean over the predictions to construct an empirical value and associated uncertainty.

Monte Carlo sampling does not need to occur only in the context of Bayesian Inference. For example, Srivastava et al. (2014, pp. 1946 - 1947) use Monte Carlo sampling to obtain improved predictions for neural networks with dropout trained in the Frequentist approach: they sample k neural networks using dropout for each test case and average their predictions. Given the probabilistic nature of the Bayesian approach, using MCMC seems like a natural extension. However, this type of principled averaging is very different from constructing a Markov Chain, which is an alternative method to VI.

2.6.2 MCMC

MCMC methods are a strategy to draw samples using a Markov chain mechanism(Andrieu et al., 2003, p.13). A precondition of using MCMC is to calculate the approximate distribution up to a constant, and then MCMC mimics drawing from the true distribution (Andrieu et al., 2003, p.13). For example, an ideal situation for using MCMC is when the posterior in Eq. (2.1) can be calculated up to the evidence term $p(D)$. It is sufficient to have an approximate form of the posterior given by $p(\mathcal{D} \mid w)p(w)$, that can be sampled from. This means that an intractable evidence term does not matter, as it is not included in the approximate distribution. Clearly, this means that both the likelihood $p(\mathcal{D} \mid w)$ and the posterior $p(w)$ must have a known analytical form, so that the approximate posterior can already be built and sampled.

A very famous algorithm is the *Metropolis -Hastings* algorithm (Metropolis et al., 1953) which consists of only two steps : sampling a current candidate from a proposal distribution and accepting or rejecting the candidate based on an acceptance ratio. More sophisticated algorithms, such as adaptive MCMC have also been proposed (Andrieu & Thoms, 2008). MCMC differs from VI since it constructs a Markov chain to approximate the true distribution. The advantage of Metropolis is that it can be used effectively to draw nuanced decision boundaries, since it involves sampling to make predictions. As an example, consider the following decision boundary zones for a simple binary classification problem – figure 2.4 is the same problem presented in the introduction, but contextualized here.

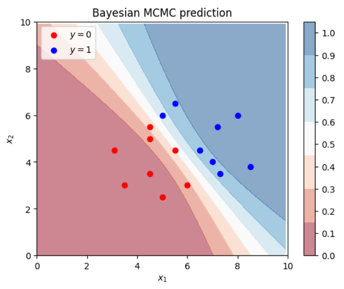


Figure 2.4: Binary classification problem and Bayesian boundary. Diagram created as part of the submission of Assignment 2 of CS5914.

In my approach, I will use MC sampling for inference, as this keeps everything closer to the Bayesian paradigm. In the next section, I will conclude this chapter by presenting a comparison between Bayesian and Frequentist training and inference.

2.7 Bayesian vs. Frequentist

As previously discussed, Bayesian approaches are diverse, but I will take consider only a specific subset of Bayesian methods. For the training phase I will use (i) Variational

Inference, while (ii) assuming a Gaussian fixed form of the approximate posterior q, which will practically be found by minimizing the (iii) Variational Free energy. For inference, I will draw MC samples from the learned approximate Gaussian q and compute expectation values together with uncertainty estimates. In light of the previous discussion, the following table summarizes the main differences between Frequentist and Bayesian paradigms.

Table 2.1: Comparison between Bayesian and Frequentist Neural Networks

Aspect	**Frequentist Neural Network**	**Bayesian Neural Network**
Objective	Loss : Cross-Entropy, MSE	Variational Free Energy (negative ELBO)
Learnt parameters	Point estimates $\mathbf{w}$	Approximate posterior $q(\mathbf{w}) = \mathcal{N}(\theta, \sigma^2)$ over weights $\mathbf{w}$
Uncertainty	No inherent uncertainty estimation.	Uncertainty derived from posterior samples.
Inference	Forward pass with fixed weights	Monte Carlo sampling: multiple forward passes using samples from $q(\mathbf{w})$

For each objective function, a form of Gradient Descent can be used to learn the parameters. For BNNs, during either training or inference, a sample trick can be used as in Eq. (2.26), which allows the calculation of backward gradients, as per the example in Eq. (2.28). In the next chapter, I will introduce several methodological considerations for implementing variational layers.

2.8 VI in Transformers

Bayesian variational transformers have been investigated in literature: Bayesformer (Xiao et al., 2024), VTNs (Arroyo et al., 2021), Variational Transformers for anomaly detection (Wang et al., 2022), to name only a few. However, the aforementioned works seem to focus on either applying VI to the attention layer, or to model the Transformer based on a fully Variational Auto-Encoder(VAE). A seemingly unexplored direction is to investigate the performance of a hybrid Variational Transformer block where the attention layer is implemented in the Frequentist approach and a Variational MLP replaces the FFN layer. To better understand this, consider the following diagram taken from the original Transformer paper(Vaswani, 2017).

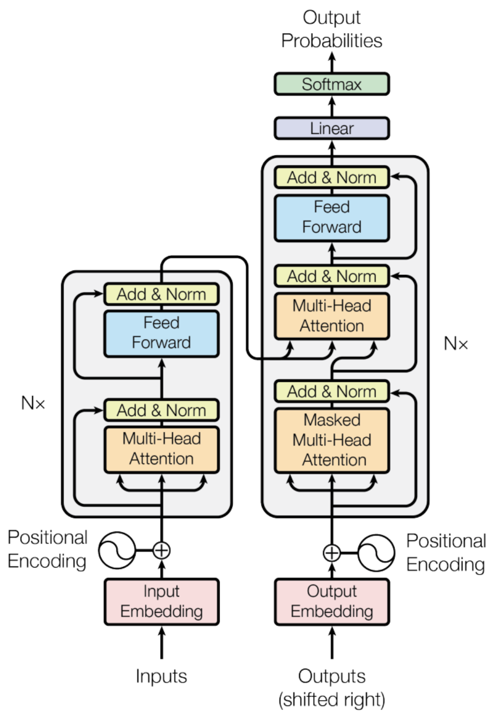


Figure 2.5: Transformer architecture (Vaswani, 2017) – diagram taken from the original paper.

As seen in the figure above, the Transformer is essentially an Auto-encoder based on Multi-Head Attention and Feed Forward layers. The encoder comprises $N = 6$ blocks, each such block consisting of one Multi-Head attention and a Feed Forward MLP, together with Add & Norm layers that normalize the outputs – the latter are known as Residual connections (Vaswani, 2017, p.2).

In this dissertation the focus is on Variational MLPs and Transformers provide an interesting investigation of replacing the Feed-Forward layer with a Variational MLP. For simplicity, I will consider a simple sentiment binary classifier based on one Encoder block. Hence, the classifier will contain a Multi-Head attention component and a Feed Forward component. The Feed Forward component is a two-layered MLP. Replacing the two-layered MLP with variational layers will be the focus of experiments in section 4.

3 Methodology and Design

The aim of this chapter is to explain how several algorithms discussed in Graves(2011), Molchanov et al.(2017) and Kingma et al. (2015) will be practically implemented in the `VariationalMLP.jl` module. A user guide for installing and using `VariationalMLP.jl` is provided in Appendix A.2. The algorithms will be discussed as standalone implementations and as part of Transformer Encoder Block. In addition, I will discuss how to quantify sparsity and prune weights.

3.1 Algorithms

The general logic for the forward pass with VI is given in Algorithm 1. Different methods can be implemented by using a different sampling trick (Step 3), and a different form for the KL divergence (step 7).

Algorithm 1: Forward Pass : Bayesian MLP

```
  Input: $x$, labels $y$, model
  Output: Variational Energy
  ;                                                     /* Initialization */
1 Initialize parameters: $\mu$, $\sigma^2$, $b$, $f$
  ;                                                     /* Weight sampling */
2 Sample noise: $\epsilon \sim \mathcal{N}(0, I)$;
3 Calculate weights $W(\epsilon, \mu, \sigma^2)$ ;
4 Apply linear operation and activate: $\hat{y} = f(Wx + b)$;
  ;                                                     /* Variational Energy */
5 Compute prediction loss per batch: $\frac{1}{M} L^N(\mathbf{w}, \mathcal{D}) = \text{MSE}(\hat{y}, y)$ or $\text{CE}(\hat{y}, y)$ ;
6 Scale up to training data size : $\frac{N}{M} L^N(\mathbf{w}, \mathcal{D})$ ;
7 Compute total KL divergence: $\text{KL} = \sum_{layer} \sum_{ij} \text{KL}_{ij}$ ;
8 Return variational loss(epoch): $\mathcal{F} = \frac{N}{M} L^N(\mathbf{w}, \mathcal{D}) + \lambda \cdot \text{KL}/N$;
9 return $\mathcal{F}$ ;
```

The above implementation uses the `Flux` package in Julia. The type of initialization can be chosen by the users. I use a `struct` to define the parameters of each layer, but each layer is a custom abstract type defined as follows:

```
abstract type AbstractVariationalLayer end
```

The `AbstractVariationalLayer` serves as a parent type for all variational layers. Each layer is made callable via the @functor macro from Flux. This ensures automatic handling of parameters for optimization.

In Step 8, the KL divergence is scaled relative to the training dataset size. This is necessary because, especially for large datasets, an unscaled KL term can dominate the loss function. When this happens, the model can prematurely collapse its weights to random values, preventing meaningful learning.

In Step 5, the loss functions `logitcrossentropy` and `mse` are imported from Flux. By default, these compute the mean loss per batch, which accounts for the $\frac{1}{M}$ factor in front of the loss function. The λ term introduced in Step 8 functions as an annealing parameter, as it controls the gradual incorporation of the KL divergence during training. When $\lambda = 0$, the variational energy reduces to the standard loss function, making the optimization equivalent to the Frequentist case. However, even in this scenario, fresh noise ϵ is sampled in each forward pass (as per Step 2), so the training still retains a Bayesian character.

Algorithm 1 is can be adapted for each type of variationa layer : molchanov, graves or kingma. This is because these methods only differ in their weights sampling step and the form of their KL divergence. In the next section, I will show how steps 3 and 8 translate to mathematical equations.

3.1.1 Variational Dense

Eq. (1) of Graves(2011, p.2) introduces the network loss:

$$L^N(\mathbf{w}, \mathcal{D}) = -\ln p(\mathcal{D} \mid \mathbf{w}) = -\sum_{\mathbf{x},\mathbf{y} \in \mathcal{D}} \ln p(\mathbf{y} \mid \mathbf{x}, \mathbf{w}) \tag{3.1}$$

Graves(2011) defines in Eq. (3) (Graves, 2011, p.2), a form of variational free energy that combines the expectation loss with the KL divergence:

$$F = KL(q(\mathbf{w}|\beta)||p(\mathbf{w}|\alpha)) + \langle L^N(\mathbf{w}, \mathcal{D})\rangle_{\mathbf{w}\sim q(\mathbf{w}|\beta)} \tag{3.2}$$

This is equivalent to Eq.(2.23). For a a complete proof from first principles, please see Appendix A.1. An interesting case, treated by Graves(2011, pp. 3-4), assumes that the approximate distribution is a diagonal Gaussian, which means that $\beta = \{\mu, \sigma^{\mathbf{2}}\}$ (Graves, 2011, p.3). The expectation of the loss can be approximated by Monte Carlo sampling :

$$\langle L^N(\mathbf{w}, \mathcal{D})\rangle_{\mathbf{w}\sim q(\mathbf{w}|\beta)} \approx \frac{1}{S}\sum_{k=1}^{S} L^N(\mathbf{w}, \mathcal{D}) \tag{3.3}$$

Eq. (3.3) above is also in Graves(2011, p.4) – Equation (9). Comparing (3.3) with (2.29), it is clear that this is the Monte Carlo point-mass function for identically distributed samples – the i.i.d assumption holds. Hence, the samples are drawn independently from the approximate distribution q, for example a Gaussian, and the error loss L^N is effectively averaged. It also can be shown – please see Appendix A.1 for the full proof – that this result is differentiable, which means that gradients are easy to compute. If the prior is also Gaussian, then $\alpha = \{\mu, \sigma^2\}$ calculating the KL divergence term is straightforward – equation (13) of Graves(2011, p.4) – and the full proof is given in Appendix A:

$$KL(q(\mathbf{w}|\beta)||p(\mathbf{w}|\alpha)) = \sum_{i=1}^{N} \ln \frac{\sigma}{\sigma_i} - \frac{1}{2}(1 - \frac{\sigma_i^2}{\sigma^2} + \frac{(\mu_i - \mu)^2}{\sigma^2}) \tag{3.4}$$

Notice that in Eq. (3.4) μ, σ^2 are the mean and variance of the prior, and these parameters are fixed and assumed a priori, reflecting the belief in the weights, as explained in section 2. However, μ_i, σ_i are the parameters of the approximate distribution q, which in fixed form is a Gaussian, and these are learnable. Indeed, by learning μ_i, σ_i, one learns the approximate posterior, which is the goal of Bayesian learning. Combining Eq. (3.3) and (3.4) yield the final form of the variational energy:

$$F(q) \approx \frac{1}{S} \sum_{k=1}^{S} L^N(\mathbf{w}, \mathcal{D}) + \sum_{i=1}^{N} \ln \frac{\sigma}{\sigma_i} - \frac{1}{2}(1 - \frac{\sigma_i^2}{\sigma^2} + \frac{(\mu_i - \mu)^2}{\sigma^2}) \tag{3.5}$$

Eq. (3.5) is valid when both prior and posterior are Gaussian – Graves(2011) calls this approach "adaptive weight noise". Equation (18) from Graves(2011, p.5) introduces a practical consideration: the KL divergence term, which Graves(2011) identifies with the compression loss, is scaled based on number of batches. This ensures the KL divergence matches the loss which is already averaged per batch. Recalling the fixed-form variational inference treatment in Section 2, sampling from an unknown Gaussian such as q is difficult. Hence, the reparameterisation trick described in Eq. (2.26) can be adapted as follows:

$$w = \mu_i + \sigma_i \epsilon \tag{3.6}$$

Clearly, sampling means just two steps:

- Sample some noise from a known Gaussian: $\epsilon \sim \mathcal{N}(0, 1)$
- Then update the weights based on the current mean and variance of the Gaussian : $w_{ij} = \mu_{ij} + \sigma_{ij}\epsilon$

The Julia implementation of the algorithm follows directly from Eq (3.5) and Eq (3.6). In Step 3 of Algorithm 1 the weights are sampled according to Eq. (3.6), and the variational energy is a direct implementation of Eq. (3.5).

Adaptation of Algorithm 1 (Dense)

Modified Step 2 (Weight Sampling):
Sample weights using reparameterization:

$$W = \mu + \exp\left(0.5 \cdot \log \sigma^2\right) \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

Modified Step 7 (KL divergence):
Calculate the KL divergence as per Eq (3.4) :

$$KL_{graves} = \sum_{ij} \ln \frac{\sigma}{\sigma_{ij}} - \frac{1}{2}(1 - \frac{\sigma_{ij}^2}{\sigma^2} + \frac{(\mu_{ij} - \mu)^2}{\sigma^2})$$

The algorithm uses $\ln \sigma^2$ for stability. Also, $\ln \sigma^2$ ensures this is not a constrained optimization problem, as the logarithm of a strictly positive quantity can be either positive or negative. For this implementation, the user can choose the parameters of the prior μ, σ to reflect assumptions about the weights.

3.1.2 Variational Dropout

Variational Dropout is a particular case of Variational Inference where the dropout rate is considered a variational parameter. My implementation of variational dropout is largely based on the work of Molchanov et al. (2017), with some additions from Graves (2011) and Kingma et al. (2015).

The well-known result of VI, that the optimal value of the variational parameters ϕ can be found by maximization of the lower bound is reproduced by Molchanov et al.(2017) in their equations (1) and (2). The expected log-likelihood $L_D(\phi)$ is regularized by the KL divergence, and together they give the lower bound (ϕ). The ELBO formulation found in (1) and (2) of Molchanov et al.(2017) is equivalent with Eq. (2.20) – Appendix A.1 shows the full treatment for VI. Computing the variational lower bound is intractable, hence Molchanov et al.(2017) use the SGVB estimator from Kingma et al. (2015) to arrive at equations (3) - (5). The reparameterization trick to represent the 'parametric noise' $q_\phi(\mathbf{w})$ can be viewed as a deterministic differentiable function f of non-parametric noise (Molchanov et al., 2017) $\mathbf{w} = f(\epsilon, \phi)$, where ϵ is a random noise variable $\epsilon \sim p(\epsilon)$. In the following, I will introduce two ways to sample the weights, found in Molchanov et al.(2017) and Kingma et al.(2015).

The specific trick introduced by Molchanov et al. (2017) to reduce the variance of the dropout parameter α considers a new independent variable σ_{ij}. As per Eq. (11) of Molchanov et al.(2017), σ is defined as :

$$\sigma_{ij}^2 = \alpha_{ij}\theta_{ij}^2 \tag{3.7}$$

This simple reparameterisation leads to the following update rule, which is **additive**, since now the noise term is added to the mean.

$$w_{ij} = \theta_{ij} + \sigma_{ij}\epsilon_{ij} \tag{3.8}$$

In contrast, the noise can be multiplied to the mean, which leads to the multiplicative reparameterisation trick as per equation (7) in Molchanov et al.(2017)

$$w_{ij} = \theta_{ij}\xi_{ij} \tag{3.9}$$

The noise can be re-written in term of α, the dropout rate. This means that dropout is actually injecting noise into the networks layers.

$$w_{ij} = \theta_{ij}(1 + \sqrt{\alpha}\epsilon_{ij}) \tag{3.10}$$

Kingma et al.(2015) and Molchanov et al.(2017) use different approximations of the KL divergence, which means the final form of the variational free energy will be different for the two methods. Variational dropout uses $q(W|\theta, \alpha)$ explicitly as an approximate distribution for a model with a special prior on the weights. According to Kingma et al (2015), θ captures the mean of this Gaussian, and α is a multiplicative noise term. If the multiplicative term α enters the noise term, this becomes the dropout posterior. During dropout training, θ is adapted to maximize the expected log-likelihood $\mathbb{E}_{q_\phi(\mathbf{w})}[L_D(\theta)]$. But for this to be consistent with optimizing $L_D(\theta)$, the prior on the weights must be chosen to be improper log-scale uniform (Molchanov et al., 2017), which leads to a very specific approximation of the KL divergence found in equation (14) of Molchanov et al.(2017). The problem remains the practical implementation of the variational free energy, as this should be able to address both regression and classification. With this in mind, I will focus on interpreting Eq. (2) from Molchanov et al. (2017), in a more practical way:

$$L_D(\theta) = \sum_{n=1}^{N} \mathbb{E}_{q_\alpha(\mathbf{w})}\left[\log p(y_n \mid x_n, \mathbf{w})\right]$$

Recalling that in Graves(2011), the loss of the network is the negative log probability of the data given the weights:

$$L^N(\mathbf{w}, \mathcal{D}) = -\sum_{m=1}^{M} \log p(\tilde{y}_m \mid \tilde{x}_m, f(\phi, \varepsilon_m)) \tag{3.11}$$

This means the SGVB estimator, given by Molchanov et al (2017) in their Eq. (4), can be written as a function of the loss:

$$L_{\mathcal{D}}^{\mathrm{SGVB}}(\phi) \approx -\frac{N}{M} L^N(\mathbf{w}, \mathcal{D}) \tag{3.12}$$

Molchanov et al.(2017) use the ELBO maximization ,which is equivalent to minimizing the variational free energy. The variational energy is simply the negative of the ELBO, so I simply change the sign of the ELBO to obtain the variational energy:

$$\mathcal{F}(\phi) = -L_D(\phi) + D_{\mathrm{KL}}(q_\phi(\mathbf{w}) \,\|\, p(\mathbf{w})) \tag{3.13}$$

Replacing the L_D term with the SGVB estimator yields: behcbenchamrking

$$\mathcal{F}(\phi) = \frac{N}{M} L^N(\mathbf{w}, \mathcal{D}) + D_{\mathrm{KL}}(q_\phi(\mathbf{w}) \,\|\, p(\mathbf{w})) \tag{3.14}$$

$$KL_{molchanov} = \sum_{ij} -k_1 \sigma(k_2 + k_3 \alpha_{ij}) + 0.5 \ln(1 + \alpha_{ij}^{-1}) + k_1 \tag{3.15}$$

The values of the constants k_1, k_2, k_3 are reported in Molchanov et al (2017). For the multiplicative reparameterization trick proposed by Kingma et al.(2015), the KL divergence will be approximated by the respective term, with the values of the constants reported in the paper (Kingma et al, 2015, p.6), :

$$KL_{Kingma} = \sum_{ij} -0.5 log(\alpha_{ij}) - c_1 \alpha_{ij} - c_2 \alpha_{ij}^2 - c_3 \alpha_{ij}^3 \tag{3.16}$$

For additive reparameterisation, the exact steps for the forward pass are:

Adaptation of Algorithm 1 (Additive Reparameterisation)

Modified Step 2 (Weight Sampling):
Sample weights using reparameterization in (2.39) and (2.40):

$$W = \mu + \exp\left(0.5 \cdot \log \sigma^2\right) \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

Modified Step 7 (KL divergence):
Calculate the KL divergence as per Eq (3.15) :

$$KL_{Molchanov} = \sum_{ij} -k_1 \sigma(k_2 + k_3 \alpha_{ij}) + 0.5 \ln(1 + \alpha_{ij}^{-1}) + k_1$$

The implementation of the Kingma et al.(2015) method, uses the reparameterization in Eq. (3.10) and the KL divergence is given by Eq. (3.16). $ln\alpha$ is used to avoid solving a constrained optimization problem.

Adaptation of Algorithm 1 (Multiplicative Reparameterisation)

Modified Step 2 (Weight Sampling):
Sample weights using reparameterization in (2.37):

$$W = \mu + \mu\sqrt{\exp(\ln \alpha)}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

Modified Step 7 (KL divergence):
Calculate the KL divergence as per Eq (3.16) :

$$KL_{Kingma} = \sum_{ij} -0.5log(\alpha_{ij}) - c_1\alpha_{ij} - c_2\alpha_{ij}^2 - c_3\alpha_{ij}^3$$

In sum, the practical implementation of Molchanov's algorithm will be minimizing the variational inference, while using the dropout rates α_{ij} to sample the weights. This is how dropout is actually learned, and it becomes clear that Variational Dropout is simply Variational Inference with the introduction of the variational parameter α_{ij} via the sampling trick. The method proposed by Molchanov et al. (2017) for training BNNs with Variational Dropout Layers is claimed to be suitable for large values of the variational dropout parameters α – this seems to be their initial motivation as Kingma et al.(2015) consider the case of $\alpha < 1$. Clearly, large α value imply large sparsity. There are a couple of unverified claims made by Molchanov et al. (2017):

- First, Molchanov et al. (2017) claim that training networks from random initialization causes many weights to get pruned away early, a common problem for Bayesian Neural Networks. This result can be easily replicated using the approach outlined by Molchanov et al. (2017). However, Molchanov et al. (2017) claim, without showing concretely how, that the problem of weights being pruned away due to random initialization can be resolved by either starting from a pre-trained network, or by using a "warm-up" strategy that gradually introduces the KL divergence term. I believe that different pre-training strategies deserve an investigation.

- Second, Molchanov et al. (2017) do not extend their method to regression, which is an interesting problem.

3.2 Transformer Classifier

In this section, I will discuss the model architecture that I will use for sentiment classification, which incorporates a Transformer Encoder block consisting of a Multi-head attention layer and a FNN. My aim is to incorporate the previously discussed variational layers for MLPs within a Transformer Encoder block. Hence, rather than developing a fully-Variational Transformer, I aim to investigate the advantages of replacing the FFN inside the Encoder with variational MLPs

I will use an architecture that is publicly available on Github [32] (Sinai, 2022). Diagram 3.1 highlights how the Transformer block is incorporated in the binary classifier model. In the subsequent experiments, only the FNN within the Transformer will be changed, the rest of the model remaining exactly as depicted. Figure 3.2 clarifies the structure of the Transformer Block (highlighted in blue in Figure 3.1). Hence, Figure 3.1 presents the classifier architecture, that contains the Transformer block and Figure 3.2 presents the Transformer block in details. Both of these architectures are adapted from a Transformer-based classifier published online(Sinai, 2022). These figures are presented on the next page.

The forward pass of this Transformer block is presented in **Algorithm 2**, which is on the page following the diagrams. The KL divergence is calculated only for the two variational dropout layers, and added to the scaled batch loss to obtain the variational free energy.

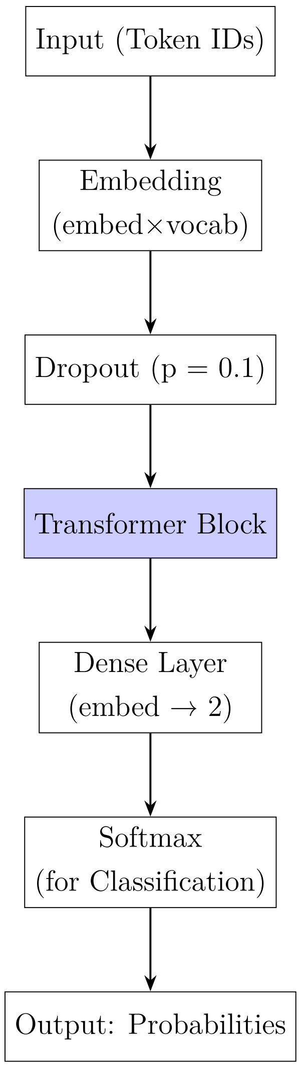


Figure 3.1: Model Architecture: Embedding + Transformer + Dense Classification

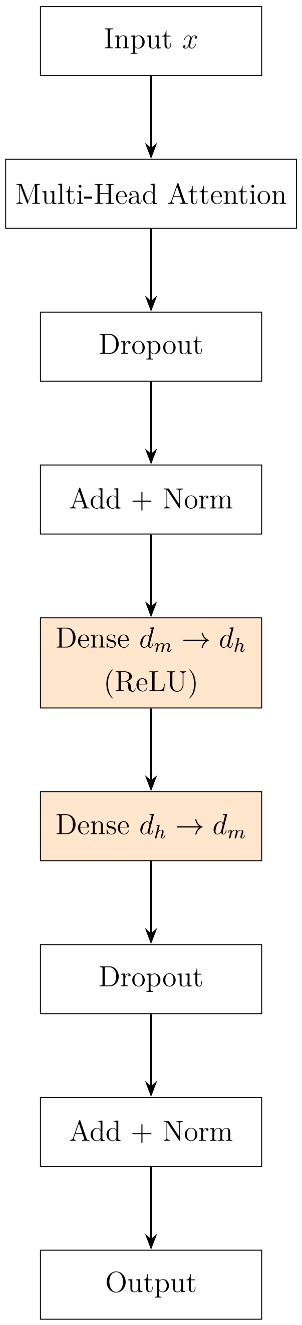


Figure 3.2: Transformer Block: Multi-Head Attention + Feed-Forward Sublayer

Algorithm 2: Forward Pass : Transformer Encoder Block with Bayesian MLP Layers

```
   Input: Input features $x$, labels $y$, model parameters
   Output: Variational Free Energy $\mathcal{F}$
   /* Initialization                                                         */
 1 Initialize parameters with Glorot: $\boldsymbol{\theta}, \log \boldsymbol{\sigma}^2 = -10$, $\mathbf{b} = 0$ ;
 2 Set activation function $f = \mathrm{ReLU}$ ;
   /* Forward Pass                                                           */
 3 Run Multi-Head Attention (MHA): $\mathbf{A} \leftarrow \mathrm{MHA}(x)$ ;
 4 ... ;
 5 Bayesian Feed-Forward Layer 1:
 6 Sample noise: $\boldsymbol{\epsilon}_1 \sim \mathcal{N}(0, \mathbf{I})$ ;
 7 Compute weights: $\mathbf{W}_1 = \boldsymbol{\theta}_1 + \boldsymbol{\epsilon}_1 \boldsymbol{\sigma}_1$ ;
 8 Compute activations: $\mathbf{h}_1 = f(\mathbf{W}_1 \mathbf{A} + \mathbf{b}_1)$ ;
 9 Bayesian Feed-Forward Layer 2:
10 Sample noise: $\boldsymbol{\epsilon}_2 \sim \mathcal{N}(0, \mathbf{I})$ ;
11 Compute weights: $\mathbf{W}_2 = \boldsymbol{\theta}_2 + \boldsymbol{\epsilon}_2 \boldsymbol{\sigma}_2$ ;
12 Final prediction: $\hat{\mathbf{y}} = f(\mathbf{W}_2 \mathbf{h}_1 + \mathbf{b}_2)$ ;
13 ...;
   /* Variational Objective                                                  */
14 Compute batch loss: $\frac{1}{M} \mathcal{L}^N(\mathbf{w}, \mathcal{D}) = \mathrm{CE}(\hat{\mathbf{y}}, \mathbf{y})$ ;
15 Scale to full dataset: $\frac{N}{M} \mathcal{L}^N(\mathbf{w}, \mathcal{D})$ ;
16 Compute total KL divergence: $\mathrm{KL} = \sum_{\text{layer}} \sum_{i,j} \mathrm{KL}(\theta_{ij}, \sigma_{ij})$ ;
17 Compute variational free energy: $\mathcal{F} = \frac{N}{M} \mathcal{L}^N(\mathbf{w}, \mathcal{D}) + \lambda \cdot \mathrm{KL}/N$ ;
18 return $\mathcal{F}$
```

3.3 Sparsity Calculation and Pruning

In both Graves(2011) and Molchanov et al. (2017), sparsity is quantified based on learned parameters, which provides an informed approach to pruning weights. In this section, I will briefly discuss how sparsity will be calculated and used to prune models.

Pruning is the process of removing weights –efectively setting them at 0 – and it can lead to reduced complexity and improved generalization (Graves, 2011, p.5). Pruning can be done in a principled way, according to learned parameters, or in a random way, where a certain percentage of weights is simply set to 0. Bayesian training provides a clear advantage in selecting heuristics for a principled approach to pruning.

A pruning heuristic for Graves (2011) involves removing weights which have a sufficiently high probability density under probability density q. For a diagonal Gaussian, the pruning heuristic can be defined as removing all weights for which the probability at 0 exceeds some threshold, as per Eq. (19) from (Graves, 2011, p.5) :

$$\left|\frac{\mu_i}{\sigma_i}\right| < \lambda \tag{3.17}$$

This pruning heuristic works for a diagonal Gaussian posterior, which is the case of my implementation of Variational Dense layers. Graves(2011) provides the following threshold:

$$\lambda_{thresh} = 0.83 \tag{3.18}$$

If the ratio between the mean and variance of a particular weight is below this threshold, the weight can be pruned(set to 0). This follows directly from the logic outline by Graves(2011), according to which, at threshold 0, no weights are pruned (all weights above are accepted), so pruning means setting to zero all weights for which ratio of mean and variance is under the threshold set in Eq. (3.18).

The logic for pruning weights is made much simpler, and perhaps more intuitive in Molchanov et al. (2017) : if the corresponding dropout rate is high enough, the weight can be set to zero. Since dropout can be interpreted as the probability of a neuron being shut, a high value of dropout will practically translate to the weight always being shut. The specific threshold for Molchanov et al. (2017) is the following :

$$\ln \alpha_{thresh} = 3.0 \tag{3.19}$$

Such a high dropout rate, which also fits the scenario of Molchanov et al.(2017), would correspond to a binary dropout rate $p = 0.95$, which means weights for which α is above the threshold can be safely pruned. To better understand how sparsity is practically calculated, consider the case of implementing Variational Dropout for a random MLP layer in a given architecture, as per the figure below.

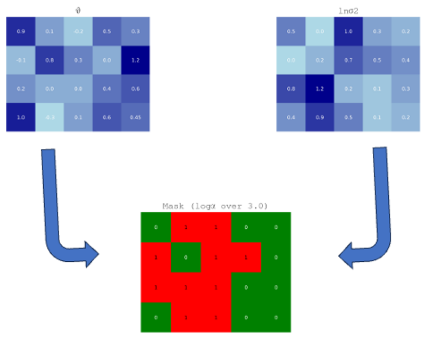


Figure 3.3: An example set of means ν and log-variances $\ln \sigma^2$ learnt for an MLP layer with 4 input neurons and 5 output neurons. The $\ln \alpha$ mask is calculated and then transformed into a binary mask. Sparsity is percentage of entries over the threshold (marked red)

The two matrices depicted above allow for effectively calculating the dropout rates, from Eq. (3.7). Applying the logarithm yields:

$$\alpha_{ij} = \ln \sigma_{ij}^2 - 2 \ln \theta_{ij} \tag{3.20}$$

Once the values of α_{ij} are computed, a binary mask is constructed: entries where α_{ij} exceeds the threshold are marked as 1 (red), indicating high uncertainty and low importance. The remaining entries are marked as 0 (green), which means the weights are retained. Pruning is applying the mask: weights at red positions are explicitly removed by setting the corresponding mean to zero, $\theta_{ij} = 0$, and the log-variance to a large negative value, $\log \sigma_{ij}^2 = -10^{10}$, while the green positions are left alone. It is easy to see how the sampled weights are effectively pruned:

$$w_{ij} = \theta_{ij} + \exp \ln \sigma^2 \epsilon \tag{3.21}$$

For $\theta_{ij} = 0$ and $\exp -10^{10} = 0$:

$$w_{ij} = 0 + 0\epsilon \tag{3.22}$$

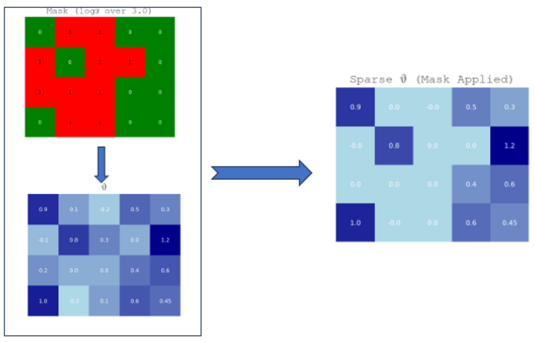


Figure 3.4: The red positions (where $ln\alpha$ is above the threshold) are identified and used to prune (set to 0) specific means ν

3.4 Model Constructor

`VariationalMLP.jl` provides a model constructor that allows the user to flexibly build Bayesian multilayer perceptrons (MLPs) by selecting one of three previously discussed variational inference algorithm –Molchanov et al. (2017), Kingma et al. (2015), or Graves(2011). The initialization strategy – custom or random – and the exact architecture of the network can also be specified. Two functions handle the construction of the MLP and determine which variational approach to use at each layer.: `make_model` and `make_layer`.

This module is compatible with Flux thanks to the explicit declaration of trainable parameters and the use of Flux's @functor macro, which allows automatic differentiation to identify and track layer parameters, namely means, variances and biases.

To train the model, the user defines a variational energy function (Algorithm 1), which includes a negative log-likelihood (e.g. cross-entropy or MSE) via the KL divergence term. The KL divergence is weighted by an annealing schedule. The variational energy is passed to `Flux.withgradient` as the loss function would be in a typical Frequentist scenario. `Flux.withgradient` calculated and updates gradients for mean, variance, and bias via Adam (Algorithm 2). Thus, the training loop implements gradient descent over the variational objective, and the full model remains fully differentiable and trainable in Flux.

Throughout training, the total variational energy per epoch is recorded, and layer-wise sparsity is calculated based on thresholds defined in the previous sections, as in the algorithm below.

Algorithm 3: Training Loop for Bayesian MLP

```
Input: $model, data\_loader, N, epochs, \lambda_{max}, pretrain\_epochs, task\_type, \eta, enable\_warmup$
Output: F, S, trained_model
;                                                  /* Initialization */
1  Initialize model via make_model ;
2  Initialize optimizer and loop parameters ;
3  Create empty logs: $F \leftarrow [,]$, $S \leftarrow [,]$ ;
4  Define annealing schedule: $\lambda = f(\text{epoch})$ ;
;                                                  /* Training over batches */
5  for each epoch do
6      Set KL scale $\lambda$ according to schedule ;
7      for each batch $(x, y)$ in data_loader do
8          Compute energy loss $F_{\text{batch}}$ using energy_loss ;
9          Compute gradients with Flux.withgradient ;
10         Update model parameters $\theta = (\mu, \log \sigma^2, b)$ ;
11     end
12     ;
13     Record total energy loss $F_{\text{epoch}}$ ;
14     Compute sparsities $S_{\text{epoch}}$ for each layer ;
15 end
16 ;
;                                                  /* Logging and return */
17 Append $F_{\text{epoch}}$ and $S_{\text{epoch}}$ to logs  return $F$, $S$, trained model
```

3.5 Benchmarking

In this section, I present the benchmarking procedure for all three types of layers, and provide a comparison in terms of performance. The custom variational layers will be compared to the highly optimized `Dense` layer in Flux. I will use the publicly available `BenchmarkTools` [16] (JuliaCI, 2025), a Julia package that can be installed directly into the virtual environment using Pkg. As per docs, the package has the macro `@benchmark` which can be used to obtain summary statistics over many repetitions, and `@btime`, which measures time. The following will be benchmarked:

- Time and memory for each layer to process an input of synthetic data - the performance for one layer applied to one input.

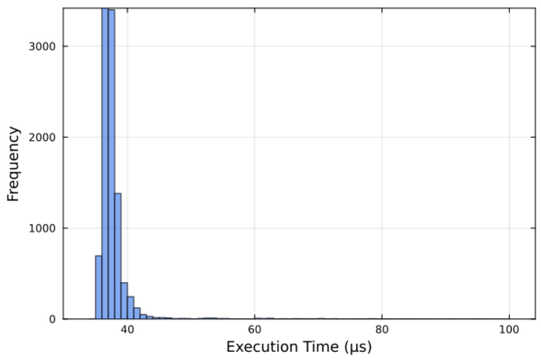
- Time and memory to run one epoch and return the variational energy.

For the following, the weights are applied to transform 300 input neurons into 100 output neurons, and the `relu` activation is used. The table below presents the time and memory estimates, as well as the number of allocations for all layers when `@benchmark` is applied for 10000 samples with one evaluation per sample.

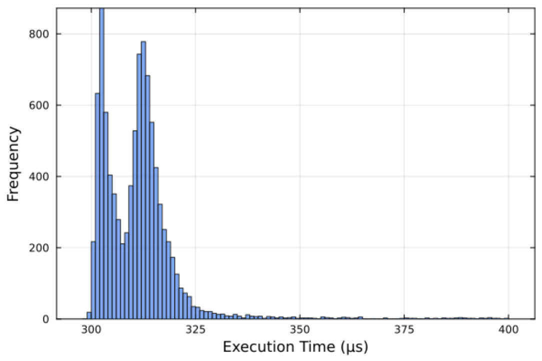
Table 3.1: Benchmarking Forward Pass for Different Layer Types (Input: 300×32)

Layer Type	**Median Time (μs)**	**Memory (KiB)**	**Allocations**
Dense (Flux)	37.791	25.16	6
Variational Dropout (Molchanov)	313.292	259.81	12
Variational Dense (Graves)	322.667	259.81	12
Variational Dropout (Kingma)	364.375	259.81	12

All customs layers perform significantly worse than the highly optimized `Dense` layer from `Flux`. Molchanov et al. (2017) and Graves (2011) perform the same, since the forward pass is identical for them. Kingma et al.(2015) performs the worst. Overall, the custom variational layers take about 10 times as long, and occupy 10 times as much memory compared to the `Dense` layer. For a more detailed overview of time, it is interesting to compare the Molchanov et al. (2017) approach to the Dense layers.



(a) Dense(Flux)



(b) Bayesian

Figure 3.5: Execution times for simple foward pass for (a) Dense and (b) Variational Dropout with Molchanov et al. (2017) method.

Both histograms reveal right-skewed distributions, which seem to indicate that execution times tend to be rather predictable. In scenario (a), for a `Flux` Dense layer, it is clear that expected execution times are more deterministic than for (b).

It is interesting to benchmark a model for one training epoch. I will use a model with two hidden layers, such that the connecting matrices have the forms (300, 100), (100, 10), for a classification task (10 classes) run on some dummy data fed to a `data_loader` object. The Dense model is made via `Chain`, with two `Dense` layers, and the variational models are built via the `make_model` constructor.

Layer Type	Median Time (ms)	Median GC (%)	Memory (MiB)	Allocs
Dense	1.417	0.00%	2.64	1584
Molchanov	24.614	10.13%	62.67	19838
Graves	16.183	9.51%	46.16	19508
Kingma	20.519	9.05%	47.25	19168

Table 3.2: Benchmarking forward pass for Dense and Variational layers. All values are for one training epoch, consisting of 10 batches of randomly generated synthethic data.

Based on the results above, it seems the Molchanov et al.(2017) method is the most expensive. Comparing to Graves(2011), since their forward pass is identical, the implementation of the KL divergence is likely very expensive. In sum, the methods in `VariationalMLP.jl` are naively implemented and expensive from a computational point of view, but they follow closely the mathematics ensuring interpretability.

I also tested several optimizers from Julia: Adam, RMSProp, Descent (SGD) and AdaGrad. All these were tested for one training epoch, for median time, estimated memory, number of allocation, and median Garbage Collection (GC) usage. I chose a fixed learning rate of 0.01 for the subsequent tests.

Table 3.3: Benchmark results of optimizers with fixed learning rate for Molchanov et al(2017)

optimizer	time(ms)	Memory(bytes)	Allocations	GC(median time)
DataType	Float64	Int64	Int64	Float64
Adam	22.2943	63220720	19083	0.0
RMSProp	21.1658	62974320	19223	0.0
Descent	19.5266	62700464	18751	0.0
AdaGrad	21.0122	62953968	18767	0.0

Table 3.4: Benchmark results of optimizers with fixed learning rate for Graves(2011)

optimizer	time(ms)	Memory(bytes)	Allocations	GC(median time)
DataType	Float64	Int64	Int64	Float64
Adam	15.6079	45918800	18593	0.0
RMSProp	14.2171	45672400	18733	0.0
Descent	14.5077	45398544	18261	0.0
AdaGrad	14.25	45652048	18277	0.0

Table 3.5: Benchmark results of optimizers with fixed learning rate for Kingma(2015)

optimizer	**time(ms)**	**Memory(bytes)**	**Allocations**	**GC(median time)**
`DataType`	`Float64`	`Int64`	`Int64`	`Float64`
Adam	22.2943	63220720	19083	0.0
RMSProp	21.1658	62974320	19223	0.0
Descent	19.5266	62700464	18751	0.0
AdaGrad	21.0122	62953968	18767	0.0

In all the scenarios presented above, Descent and AdaGrad prove to be superior optimizers to Adam, which is used in Molchanov et al.(2017). Despite this, in the following experiments I will use Adam, to ensure a consistent comparison with the research papers.

4 Experimental Results

This chapter contains the experimental results and their analysis. Section 4.1 introduces the datasets used for the subsequent experiments, and explains any pre-processing steps. Section 4.2 presents the first experiment: comparing training strategies for the LeNet-300-100 architecture. For each case, the experiments compare two scenarios. In the first scenario, the network is pre-trained in a purely Frequentist approach until full convergence, and then further trained in the Bayesian approach until the variational energy converges. In the second scenario, pre-training is done by simply cutting off the KL divergence term for a number of "warm-up" epochs, followed by an "annealing" phase where the KL divergence is gradualy introduced. Section 4.3 compares a simple custom MLP with one hidden layer on a regression signal(Blundell et al., 2015), to explore advantages of variational layers when data is scarce. Section 4.4 investigates how variational dropout layers perform when incorporated into a transformer architecture.

4.1 Datasets

In this section, I will introduce three types of datasets : (i) for classification, (ii) for regression and (iii) for sentiment analysis. All datasets considered here are independent and identically distributed (i.i.d) (Bishop, 2006, p.26). This is an essential aspect for evaluating the subsequent BNNs, because the models assume that (i) each feature can be drawn independently from the same distribution and (ii) there is no particular feature that dominates all the others.

4.1.1 Pre-processing

The following datasets have all been slightly pre-processed before learning. For MNIST and Fashion MNIST, 28 by 28 images are flattened to 784 features, where each feature represents a pixel value. The responses are also one-hot-encoded. All data was pre-processed using a `dataloader` in Julia, allowing batching. The batch size is a hyperparameter that can be changed, but it is not the focus of the following experiments.

4.1.2 Classification

Two Moons

The "Two Moons" Dataset is a simple toy dataset with only two features. The figure below presents 1000 samples of moon data where 50 are reserved for training and the remaining 950 for testing.

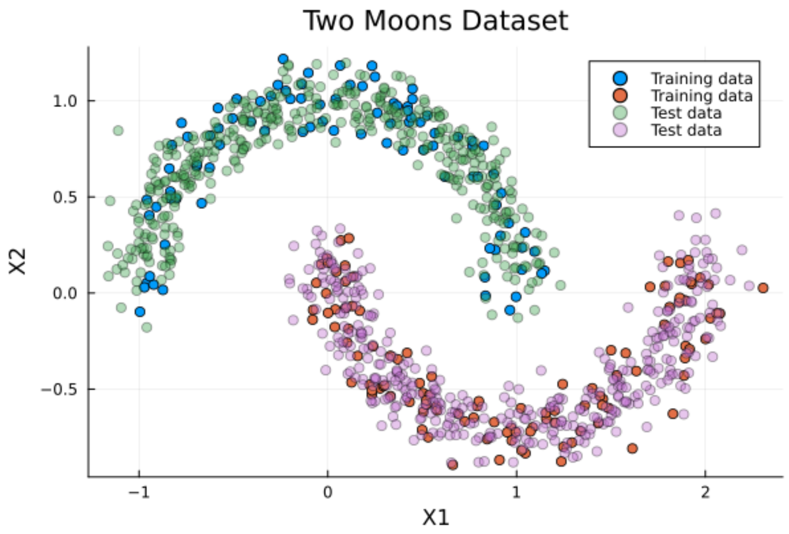


Figure 4.1: Two moons dataset

The dataset was retrieved directly from Julia `MJLBase` module, using the `make_moons` function. As seen, the data are very clean and i.i.d. – there are no outliers, the classes are perfectly balanced and easily separable, which means the decision boundary is easy to learn. The batchsize is set a 1 due to the small size of the training data.

MNIST

The MNIST dataset is a collection of handwritten digits (LeCun et al., 2010), which consists of 70000 images of 28 by 28 pixels : 60000 instances for training and 10000 for testing. There are 10 classes, one for every digit from 0 to 9. In Figure 4.2 below, an example of few digits, randomly sampled from the training data, and their corresponding labels is shown. The dataset is very clean, with no class-imbalance. For the subsequent experiments the batchsize is 256.

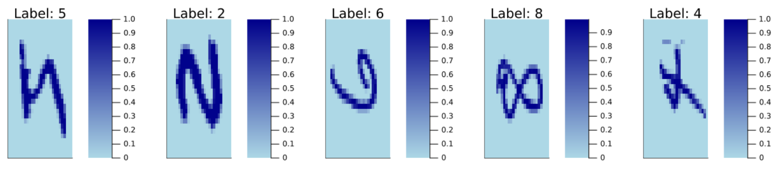


Figure 4.2: Five digits from MNIST and their corresponding labels

FashionMNIST

The Fashion MNIST dataset is remarkably similar to MNIST: it consists of 28 by 28 grayscale images of 70,000 fashion products from 10 categories, with 7,000 images per category(Xiao et al., 2017). Hence, the classes are perfectly balanced and the dataset strongly mathces the i.i.d assumption. Due to it's similarity to MNIST, I applied the exact same pre-processing steps as for MNIST.

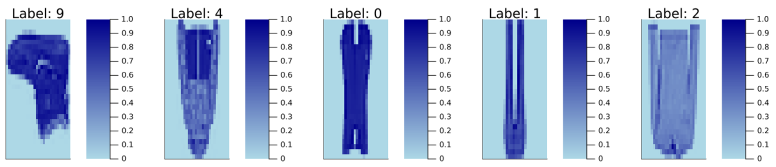


Figure 4.3: Five articles from Fashion MNIST and their corresponding labels

As seen above, the fashion articles have the same labels 0 : 9. This is because each label encodes the name of an article – a requirement as all algorithms must process numeric data.

4.1.3 Regression

I use the following regression curve, inspired by (Blundel et al., 2015) :

$$y(x) = x + 0.3\sin(2\pi(x+\epsilon)) + 0.3\sin(4\pi(x+\epsilon)) + \epsilon \tag{4.1}$$

This is a univariate signal, with just one feature X, and a response y. I generated 500 samples based on Eq.(4.1). $\epsilon \sim \mathcal{N}(0, 0.02)$ is some small random noise. As seen in the figure below, the size of the training set is only 40 samples, such that training data is lacking for entire regions of the signal. This makes it an interesting problem to study, specifically to better understand how Bayesian methods deal with learning from small data and how they deal with sparse training data.

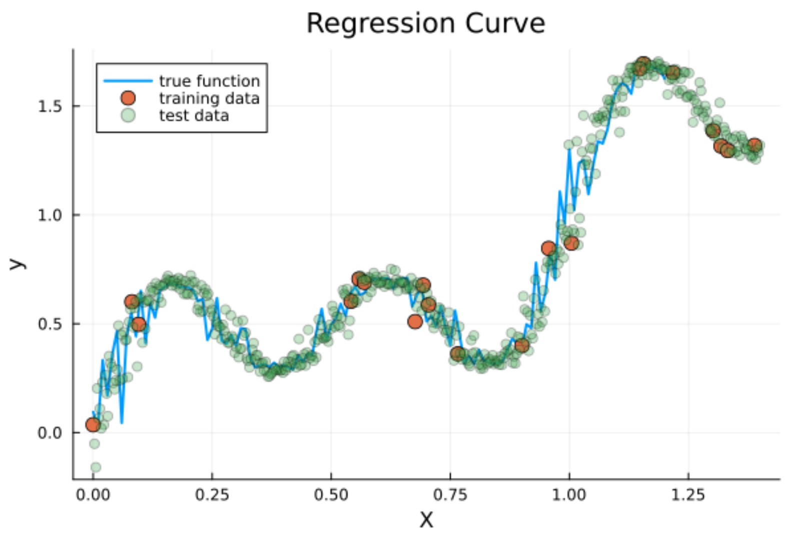
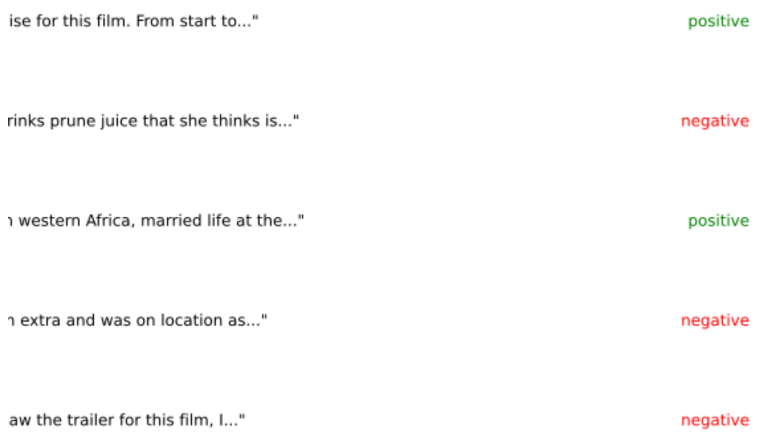


Figure 4.4: Sinusoidal regression signal. The training data is severely limited to only 40 samples.

4.1.4 NLP

The IMDB Review Dataset (Maas et al., 2011), comprising 50000 film reviews is equally split into two perfectly balanced classes based on sentiment : positive and negative. Hence, 25000 of reviews are positive and 25000 reviews are negative. 50000 is abundant data, so I will also sample smaller datasets of sizes 5000 and 1000 for the transformer experiment.



ise for this film. From start to..."	positive
rinks prune juice that she thinks is..."	negative
ı western Africa, married life at the..."	positive
ı extra and was on location as..."	negative
aw the trailer for this film, I..."	negative

Figure 4.5: Truncated reviews and sentiment for 5 random data of the IMDB Review Dataset.

The number of words per review tends to vary. The median number of words per review is 165, and the mean 220. However, the distribution of reviews based on number of words is right-skewed, as shown below.

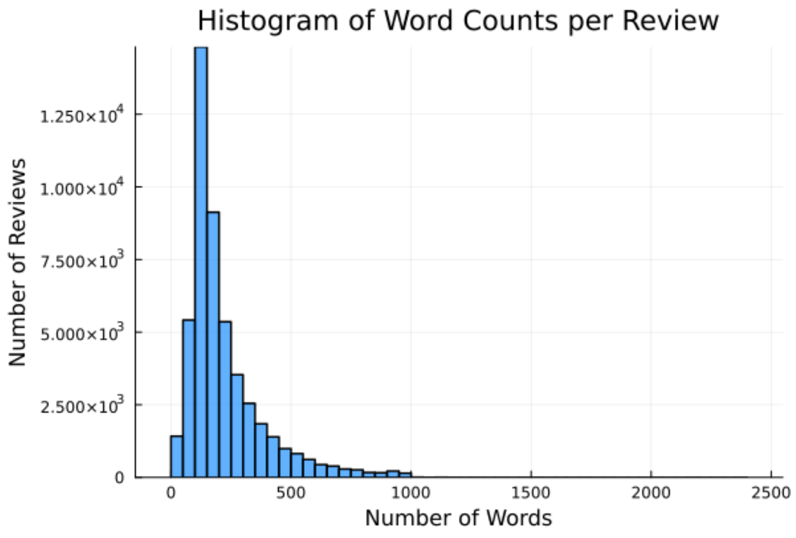


Figure 4.6: Distribution of word count per review.

Figure 4.6 shows the distribution of review lengths based on word count. The sharp peak around 120 words indicates this is the most common length for a review. When choosing the maximum sequence length for sentiment analysis, there is a trade-off: all sequences are padded to this maximum length, so a longer maximum increases computational cost, while a shorter length risks truncating important sentiment-bearing words.

Practical pre-processing steps for this dataset include: (i) removing special characters, (ii) selecting a vocabulary based on word frequency, and (iii) filtering words using a regular expression. After simplification into word vectors, a tokenizer is used to assign indices to words, with unknown or out-of-vocabulary words assigned the first available index.

4.2 Training strategies

4.2.1 Experimental Design

This section presents an experiment comparing two training scenarios for variational inference, both using KL divergence annealing as proposed by Sønderby et al. (2016). All experiments are conducted using the LeNet-300-100 architecture.

In the first scenario, which I call "Frequentist pre-training", the network is fully trained in the Frequentist approach until full convergence. This means the negative log-likelihood is minimized to near zero, effectively training the model using standard deterministic optimization. The weights from pre-training are used to initialize the mean and variance of a Gaussian : the mean is initialized with the pre-trained weights, the variance with 0. Then, Bayesian annealing begins : the KL divergence is linearly increased by the number of training epochs. For a current epoch C and a total number of training epochs T, the KL divergence at epoch C :

$$KL(C) = \frac{C}{T} * KL_{max} \tag{4.2}$$

The optimization objective during Bayesian pre-training consists of minimizing a scaled negative log-likelihood, while applying the reparameterization trick – meaning noise is injected into the weights during training. As before, pre-training is followed by an "annealing phase", where the KL divergence is gradually introduced. Table 4.1 aims to summarize the difference between the two pre-training scenarios.

	Objective function	**Parameter**
Frequentist	$\frac{1}{M}L^N(\mathbf{w}, \mathcal{D})$	W
Bayesian	$\frac{N}{M}L^N(\mathbf{w}, \mathcal{D})$	$W = \mu + \sigma\epsilon$

Table 4.1: Optimization objective and learned paramters under each pre-training regime.

The linear introduction of the KL divergence during the annealing phase is done to avoid dominating the negative log-likelihood too early.

For each case, I investigate the sparsity of the trained model – calculated based on the previously reported parameters – to understand if it is useful. To determine if the induced sparsity is useful, during inference, I make a comparison between the following strategies:

- No pruning : Using the model as-is, without setting any weights to zero.
- Principled pruning: Setting weights to 0 based on thresholds λ or $\ln \alpha$. This means weights automatically pruned based on the sparsity percentage for each layer.
- Random pruning: To match the sparsity level, a comparable number of weights is pruned at random

These strategies allow testing whether the learned sparsity is meaningful (principled), or whether it could be matched by random chance (random pruning). For all three approaches, the final network samples weights from Gaussian distribution with learned parameters, so this allows for deriving epistemic uncertainties not only for predictions, but also for accuracies, as follows:

- Step 1 : Average T drawn samples (predictions) from the learned Gaussian. Each individual prediction is different due to the probabilistic nature of the model. Hence, taking the mean of these predictions will result in mean prediction representing the expected values of the model and a standard deviation representing the confidence in each of the predictions.

- Step 2 : Compute the accuracy of the model, against the test set, for different numbers of drawn samples T. To calculate an epistemic uncertainty in the accuracies themselves, for each T, the procedure can be repeated n times. Hence the mean accuracy and the standard deviation can be calculated.

The purpose of this experiment is to elucidate the first two research questions. Given the significant theoretical background I discussed in Chapter 2, the aims can be made less general:

- To understand which method of VI produces useful sparsity in MLPs.
- To investigate if using either of the two pre-training strategies, Frequentist vs Bayesian, is equivalent to good initialization.

In the next two subsections, I will perform experiments for Variational Dense layers with a fixed Gaussian prior and for the two Variational Dropout implementations.

4.2.2 Frequentist Pre-training

The results presented below can serve as a useful comparison to the potential drop in accuracy when further tuning is performed in the Bayesian paradigm.

Dataset	**Epochs**	**Learning Rate**	**Accuracy (%)**
MNIST	100	0.001	99.97
Fashion MNIST	100	0.001	93.19
Two Moons	50	0.001	100

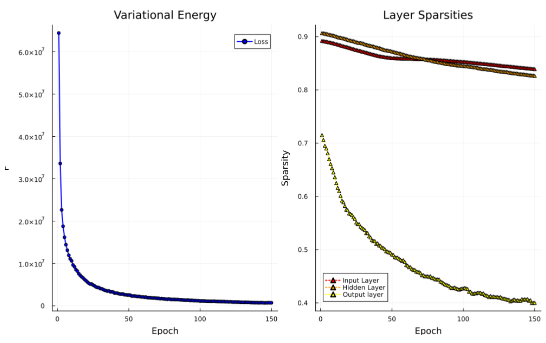
Table 4.2: Training results of LeNet-300-100 in the Frequentist paradigm.

The pre-training is done until the cross-entropy loss function has converged. Combining this Frequentist pre-training with Bayesian annealing will be done by converting the learned weights to mean and variance that serve as initialization parameters for the a Gaussian such that :

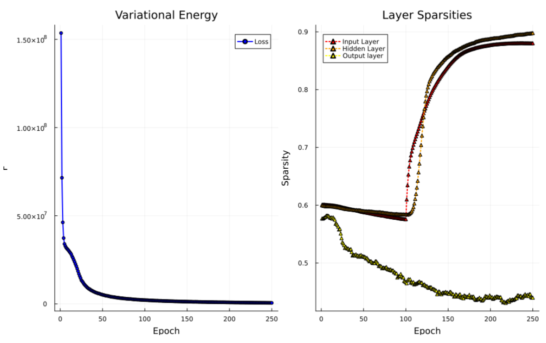
- The mean parameters are initialized with the weights from Frequentist pre-training and the standard deviation parameters are initialized with 0, which ensures that the initial weights are simply the pre-trained weights.
- The bias for the forward bias is initialized with the bias from Frequentist pre-training.

In the next section I will present a comparison of the two types of pre-training for Variational Dense layers.

4.2.3 Variational Dense Layers



(a) Frequentist



(b) Bayesian

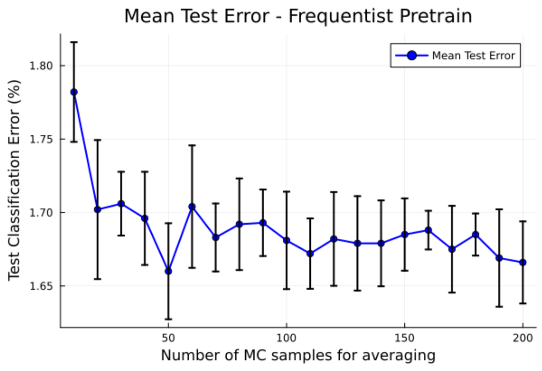
Figure 4.7: Comparison between (a) Frequentist and (b) Bayesian pre-training of LeNet-300-100 with fully Variational Dense layers implemented with Graves's approach. Training on the MNIST dataset.

Figure 4.7 shows a simple comparison between the two scenarios for Variational Dense Layers on MNIST. For scenario (a) in Fig. 4.7, the network was pre-trained in the Frequentist approach for 100 epochs. As shown, sparsity decreases monotonically across all layers during the annealing phase.

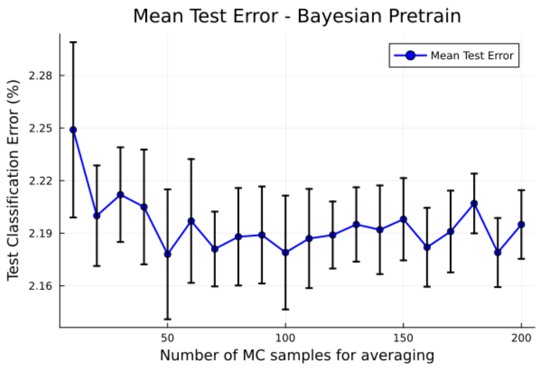
In this context, sparsity per layer refers specifically to the sparsity of the weight matrix connecting adjacent layers – that is, the proportion of weights set to zero between layers, as discussed in section 3. For example, in the LeNet-300-100 architecture, sparsity in the first layer corresponds to the $(784, 300)$ weight matrix between the input and first hidden layer. This distinction is important, as sparsity is not measured in the layer outputs or activations, but in the parameter space of the model.

Scenario (b) involves Bayesian training from the start, but with the KL divergence term disabled during the initial 100 epochs (*warm-up phase*), and then gradually introduced during the *annealing phase*. Here, the output layer becomes less sparse over time, while the hidden layers become increasingly sparse.

Despite the different optimization paths, both scenarios converge to similar final sparsity levels across layers. This suggests that both training strategies are capable of inducing similar sparse structures in the model.



(a) Frequentist



(b) Bayesian

Figure 4.8: Mean test error for (a) Frequentist and (b) Bayesian pre-training of LeNet-300-100 with fully Variational Dense layers implemented with Graves's approach. For each number of samples, the test error of the final trained model is repeated 10 times. The error bars represent the standard deviations in each test error.

Figure 4.8 presents the mean test error (averaged over 10 runs) as a function of the number of Monte Carlo (MC) samples, T. Across all values of T, the Frequentist pre-training strategy consistently results in lower test errors compared to Bayesian pre-training. In particular, at $T = 200$, the Bayesian pre-training scenario exhibits slightly lower standard deviation in test error (higher confidence), even if its mean error remains higher.

In terms of efficiency, Frequentist pre-training offers a clear advantage: as shown in the benchmarking results from Section 3, Variational Dense layers are approximately 10 times slower to train than standard layers. By pre-training the model in the Frequentist regime and only introducing variational inference during annealing, optimization becomes significantly faster, while maintaining performance. Overall, this makes Frequentist pre-training followed by KL annealing a superior strategy, since it achieves lower test errors with reduced total training time.

Table 4.3: Pruning strategies and final accuracies for each dataset using Variational Dense layers on training the LeNet-300-100. Each accuracy value is calculated per 200 MC samples.

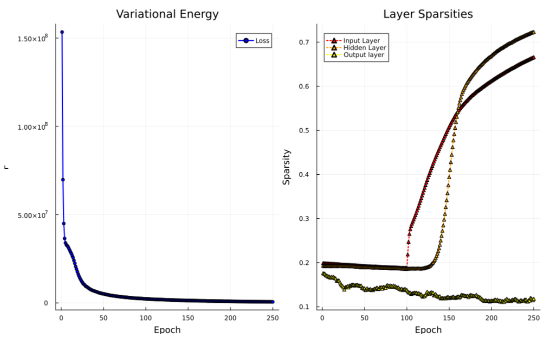
Dataset	**Pre-training**	**Sparsity**	**Pruning**	**Accuracy**
MNIST	Frequentist	83.9% - 82.62% - 40.0%	Not Pruned	98.28%
			Principled	97.94%
			Random	10.01%
	Bayesian	88.03% - 89.79% - 44.0%	Not Pruned	97.71%
			Principled	97.65%
			Random	10.09%
Fashion MNIST	Frequentist	85.14% - 86.34% - 47.8%	Not Pruned	87.93%
			Principled	84.83%
			Random	13.70%
	Bayesian	90.26% - 94.38% - 49.8%	Not Pruned	87.34%
			Principled	87.35%
			Random	10.0%
Two Moons	Frequentist	13.5% - 37.84% - 6.0%	Not Pruned	99.5%
			Principled	98.12%
			Random	99.75%
	Bayesian	14.5% - 33.68% - 4.0%	Not Pruned	99.75%
			Principled	99.37%
			Random	95.62%

For MNIST and Fashion MNIST, results seem to confirm that principled pruning (based on sparsity thresholds like λ or $\ln \alpha$) leads to insignificant accuracy loss, while random pruning, at similar sparsity levels, significantly degrades performance. The final classification accuracy is comparable across both pre-training strategies, suggesting that useful sparsity can be induced with either approach, provided the KL divergence term is annealed.

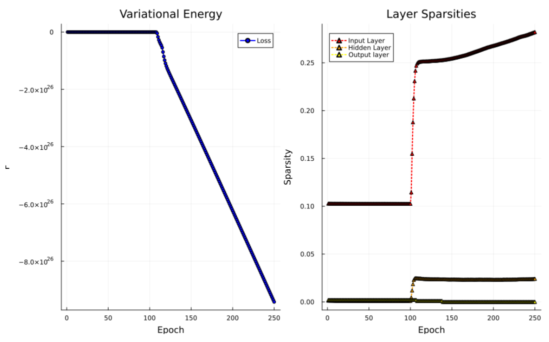
Interestingly, Frequentist pre-training tends to produce slightly lower sparsity overall, yet this does not seem to yield a meaningful improvement in accuracy. Due to its significantly lower computational cost, it remains the preferred choice for training variational dense networks.

For the **Two Moons** dataset, sparsity levels remain low, and pruning has negligible effect on performance. This is likely because LeNet-300-100 is overparameterized for such a simple task – pruning a small proportion of weights does not meaningfully reduce its expressive capacity. Across all pruning methods, the accuracy remains high.

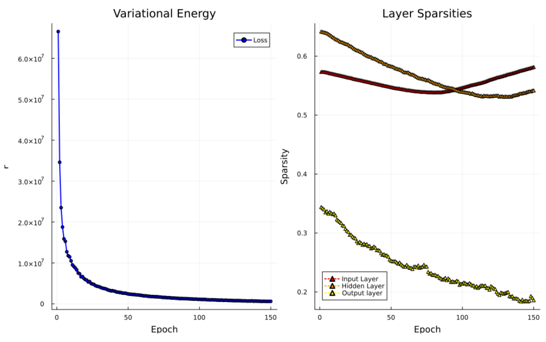
4.2.4 Variational Dropout Layers



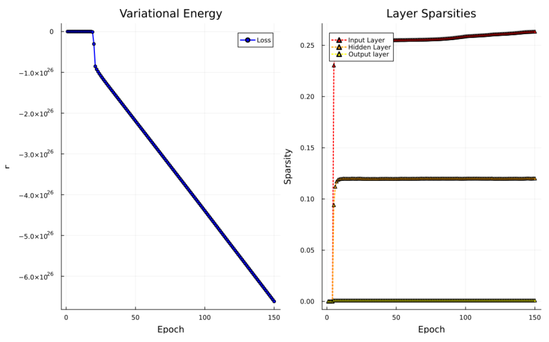
(a) Bayesian : Additive Reparameterisation



(b) Bayesian: Multiplicative Reparameterisation



(c) Frequentist : Additive Reparameterisation



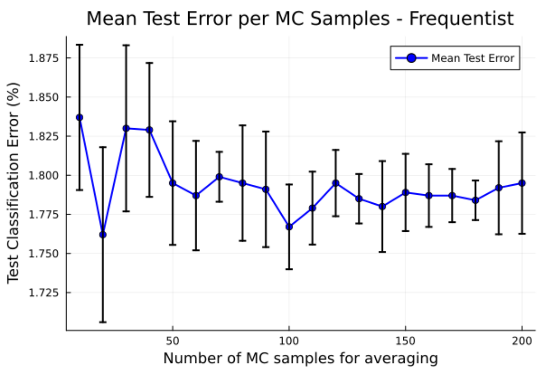
(d) Frequentist : Multiplicative Reparameterisation

Figure 4.9: Comparison of additive vs. multiplicative reparameterisation trick in Variational Dropout layers under both Frequentist and Bayesian pre-training regimes for LeNet-300-100 on the MNIST dataset.

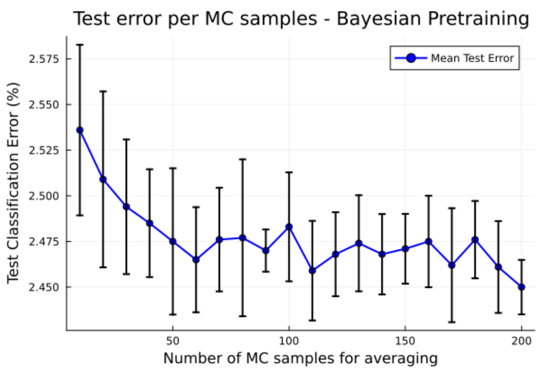
The results above compare two methods for training and evaluating the LeNet-300-100 architecture on the MNIST dataset, using different reparameterisation tricks. In scenarios (c) and (d), a Frequentist pre-training phase was performed with a learning rate of 10^{-3} for 100 epochs to ensure full convergence.

The results suggest that the additive reparameterisation performs better than the multiplicative reparameterisation. A key issue with the multiplicative reparameterisation is that the variational energy either fails to converge or does so much more slowly than with the additive approach. As a result, the method proposed by Kingma et al. (2015) proves impractical for MNIST and Fashion-MNIST tasks.

Below, I present the mean test error calculated over $n = 10$ runs for variational dropout layers , where the number of samples T is varied.



(a) Frequentist



(b) Bayesian

Figure 4.10: Mean test error for (a) Frequentist and (b) Bayesian pre-training of LeNet-300-100 with fully Variational Dropout layers implemented with Molchanov et al.(2017) additive reparameterization trick. For each number of MC samples there are 10 repetitions. The error bars represent the standard deviations in each test error across the 10 repetitions.

The uncertainty associated with each test error slightly decreases as the number of Monte Carlo (MC) samples increases, while the mean test error also decrease. In other words, with more samples, the model becomes both more accurate and more confident in its predictions.

As in the previous case, Frequentist pre-training results in faster training and achieves lower final test error. However, when comparing test errors at $T = 200$ the Bayesian pre-training approach yields a slightly narrower uncertainty interval, suggesting a more confident and potentially more robust estimate of the model's performance.

Table 4.4 presents the results of using Variational Dropout on MNIST, Fashion MNIST and Two Moons. Results are reported for $T = 200$ samples.

Final accuracies for each dataset using the additive reparameterisation (Variational Dropout) trick on training the LeNet-300-100. Each accuracy is calculated per 200 MC samples.

Dataset	**Pre-training**	**Sparsity**	**Pruning**	**Accuracy**
MNIST	Frequentist	57.36% - 53.96% - 23.9%	Not Pruned	98.44%
			Principled	98.44%
			Random	25.73%
	Bayesian	65.56% - 72.42% - 12.0%	Not Pruned	97.7%
			Principled	97.69%
			Random	9.98%
Fashion MNIST	Frequentist	57.83% - 66.75% - 24.9%	Not Pruned	87.82%
			Principled	87.81%
			Random	25.91%
	Bayesian	48.03% - 82.89% - 15.1%	Not Pruned	87.17%
			Principled	87.3%
			Random	13.7%
Two Moons	Frequentist	26.67% - 34.56% - 30.5%	Not Pruned	94.87%
			Principled	94.87%
			Random	95.375%
	Bayesian	17.0% - 19.65% - 21.5%	Not Pruned	92.875%
			Principled	98.875%
			Random	98.125%

It is interesting to observe that for the **Two Moons** dataset, the results show low induced sparsity and no clear benefit from principled pruning. This outcome aligns with expectations for variational dropout layers: in small or simple datasets, there is little motivation for the model to suppress parameters, leading to lower sparsity levels. Also, LeNet-300-100 is too complex for this task, so even random pruning does not lead to a drop in performance. Applying the Bayesian approach for **Two Moons** does not degrade performance, indicating that it remains a viable strategy even when sparsity is not beneficial.

For **MNIST** and **Fashion MNIST**, the method is less effective in inducing sparsity, compared to variational dense layers. This is because the levels of sparsity achieved with variational dropout are significantly lower than those observed with variational dense layers. Overall, while both methods can induce useful sparsity, variational dense is the stronger candidate for larger datasets for two reasons :

- As seen during benchmarking, variational dense is computationally cheaper than variational dropout.
- It achieves significantly higher sparsity levels.

However, setting the parameters of the prior is required for variational dense layers – for these experiments $\mu_{prior} = 0$ and $\sigma_{prior} = 1.0$. So in practice, it may require more manual tuning than variational dropout, which has a KL divergence term that depends only on the learned α.

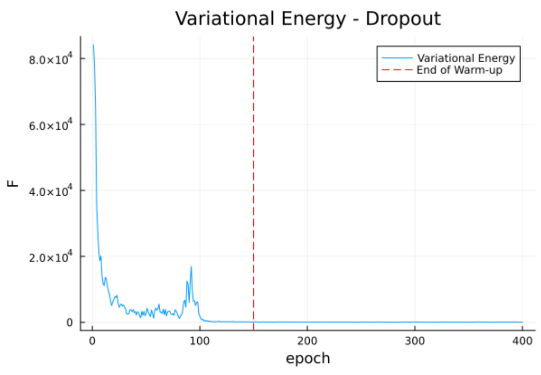
4.3 MLP: Regression

As seen in the previous section, variational inference can induce useful sparsity, particularly when the training the data is sufficiently large and complex. It is therefore interesting to explore what advantages these variational methods bring when training data is limited. In this section, I will compare Variational Dropout (he additive reparameterization trick) with Variational Dense with a Gaussian prior and posterior on the regression dataset discussed in 4.1.3.

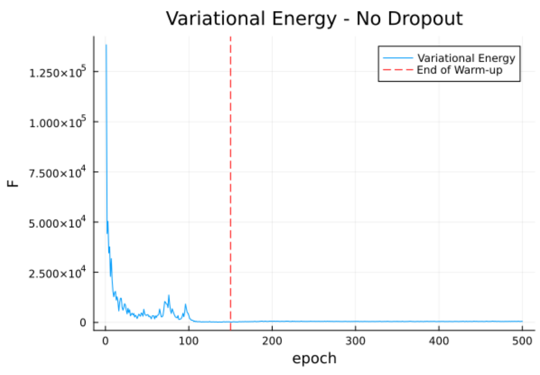
The KL divergence for variational dense layers will depend on the mean and variance prior Gaussian, and these can be controlled by the user. In this experiment, the prior is set to a Gaussian with t $\mu_{prior} = 0.0$ and $\sigma_{prior} = 2.0$, reflecting that the weights are loosely regularized.

For both cases, I implemented a learning schedule, in addition to the annealing schedule. An MLP with just one single hidden layer was used for both scenarios – the hidden layer has 64 neurons. For both variational dense and variational dropout, during warm-up, which is set to 150 epochs, the KL divergence term is shut. Then, for the remainder of 250 epochs, the KL divergence is gradually annealed. As explained previously, the KL divergence for variational dropout depends only on the dropout rate α. The learning rate was varied as follows: for the first 100 epochs (during part of the "warm-up" phase) the learning rate was set to 0.1 and then, for the remainder of the 300 epochs, it was decreased to 0.01 (for both warm-up and annealing phases).

The variational energy during training is shown below.



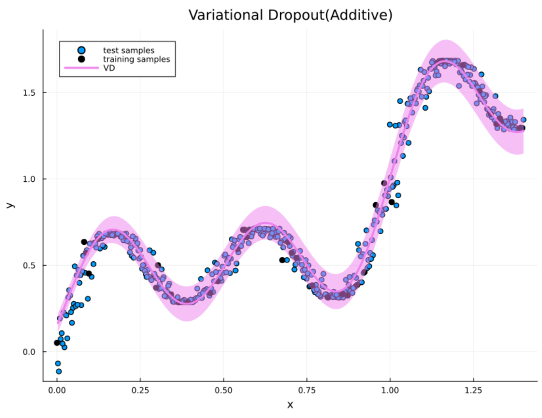
(a) Variational Dropout(Additive)



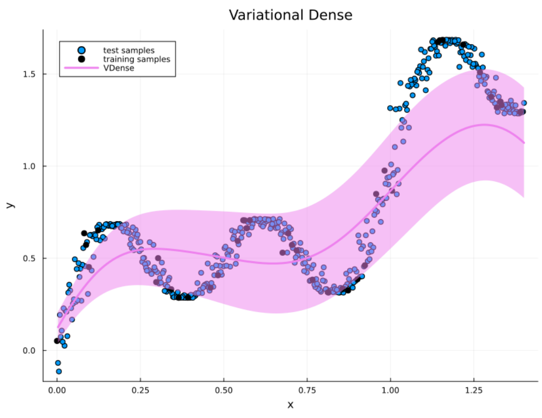
(b) Variational Dense

Figure 4.11: Variational Energy for the two scenarios (a) Variational Dropout using Additive Reparameterisation and (b) Variational Dense for a custom MLP with one hidden layer. A learning rate schedule was implemented in addition to the KL annealing schedule.

The final value for the variational energy is 60 for Variational Dropout, and 500 for Variational Dense. I use 200 samples from the trained model to make predictions and calculate the uncertainty.



(a) Variational Dropout(Additive)



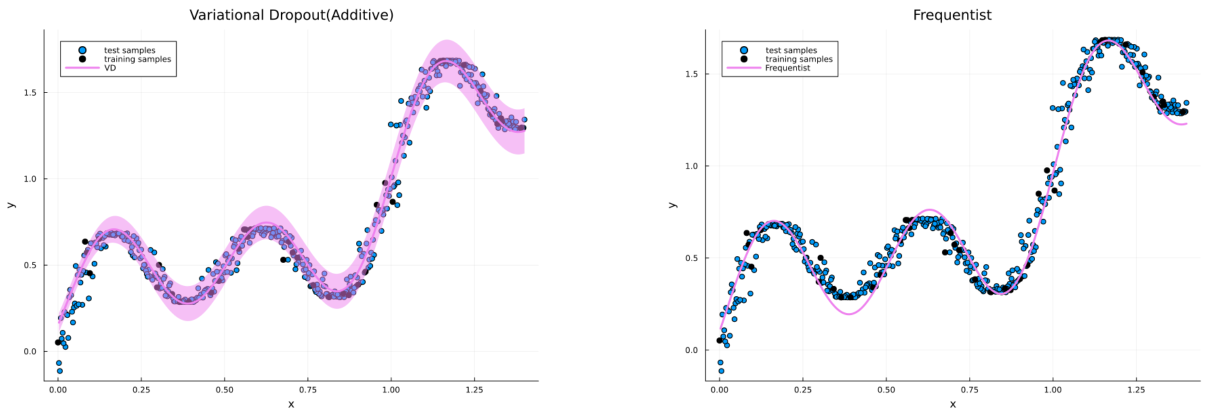
(b) Variational Dense

Figure 4.12: Predictions on test set (a) Variational Dropout using Additive Reparameterisation and (b) Variational Dense for a custom MLP with one hidden layer

The results indicate that Variational Dropout significantly outperforms Variational Dense layers in this regression task. In Scenario (b), one possible explanation for the poor performance of the Variational Dense model is the choice of prior, which must be manually selected and effectively treated as a hyperparameter. In contrast, Variational Dropout does not require a manually specified prior, as the KL divergence is computed directly from the dropout rate α.

In Scenario (a), Variational Dropout not only produces predictions that closely follow the test set, but also demonstrates meaningful uncertainty estimates in regions with sparse or no training data. Interestingly, in areas with little or no training data, the model tends to be more confident, which suggests that it has learned a useful inductive bias from the available data. In contrast, the Variational Dense model in Scenario (b) appears to be both highly uncertain and poorly fitted, indicating that its posterior distribution is not well calibrated to the data.

It is also useful to compare Variational Dropout with a standard Frequentist regression model trained using the same architecture, learning rate, and number of epochs. As shown in the figure below, the Frequentist model tends to overfit and fails to capture certain features of the target function – missing some of the peaks in the signal. Moreover, the Frequentist model lacks uncertainty estimates, whereas the Variational Dropout model provides both accurate predictions and calibrated uncertaint, making its outputs more nuanced and interpretable.



(a) Variational Dropout(Additive)   (b) Frequentist

Figure 4.13: Predictions on test set (a) Variational Dropout using Additive Reparameterisation and (b) Frequentist for a custom MLP with one hidden layer

Even when disregarding uncertainty quantification and considering only the point-like predictions, the Variational Dropout model more accurately captures the sinusoidal peaks of the target function. A direct comparison of the mean squared error (MSE) for both models confirms that Variational Dropout outperforms standard Frequentist training in terms of predictive accuracy.

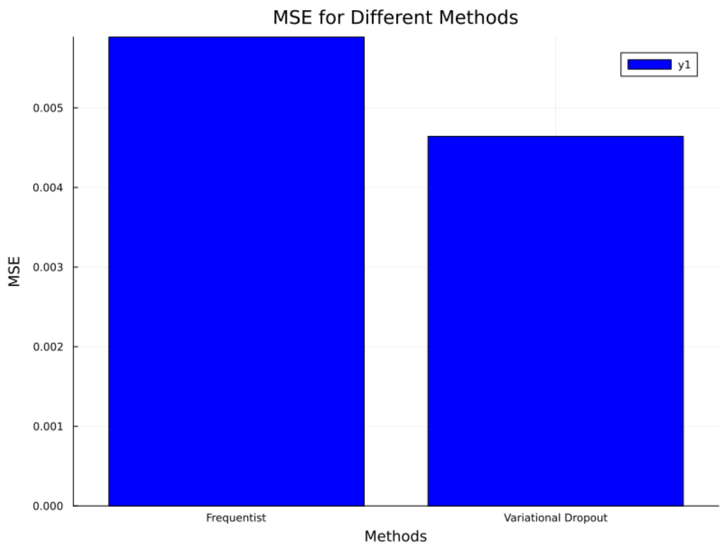


Figure 4.14: Mean Squared Error

4.4 MLP in Transformer: Binary Sentiment Classification

For this experiment, I will use the classifier discussed in section 3.2. I will investigate if replacing the FNN in the original classifier with variational dropout layers using Molchanov et al. (2017) method can induce useful sparsity. The experiment is conducted on the IMDB Review Dataset introduced earlier, across three different scenarios:

- The entire dataset is used (50000 instances).
- A randomly sampled subset of 5000 instances is used.
- An even smaller randomly subset of 1000 instances.

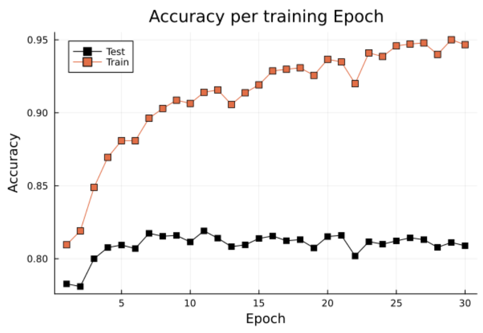
In each of the three scenarios the train - test split is 50% - 50%/ . I first present the results on the full dataset, such that 25000 instances are used for training, and 25000 for testing.

Both variants of the classifier (Frequentist and part-Bayesian) are trained under identical conditions. A vocabulary of size 18457 is constructed, the maximum length of tokens is set at 120 and the embedding dimension is 32. Training is performed using the Adam optimizer with a fixed learning rate of 0.01, batch size of 32, and for 30 epochs.. For the Transformer Classifier with a Variational MLP, the KL divergence term is linearly annealed. The KL divergence term is annealed but the models are not pre-trained.

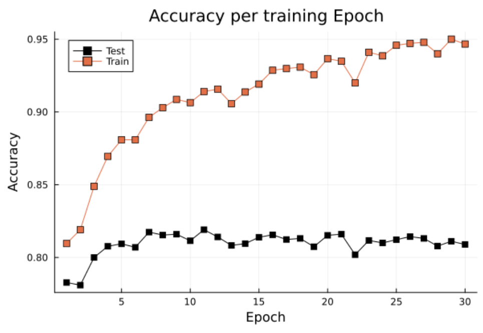
The figure below presents the training and test accuracies for the two scenario:

- The classifier is trained with Frequentist FFN.

- The classifier is trainec with Variational Dropout FFN.



(a) Transformer with Dense FFN.



(b) Transformer with Variational Dropout FFN.

Figure 4.15: Test and training accuracy for (a) a Dense layer FFN and (b) VD FFN inserted inside the Transformer Encoder block.

The final accuracy on the test set is 80.9% for both cases. Looking at the evolution of training and test accuracies, it is clear the model overfits. Also, training the model for longer does not seem to help, as the accuracy on the test set peaks after a few epochs. Principled pruning seems to offer a small increase in accuracy, which is not the same for randomly pruning.

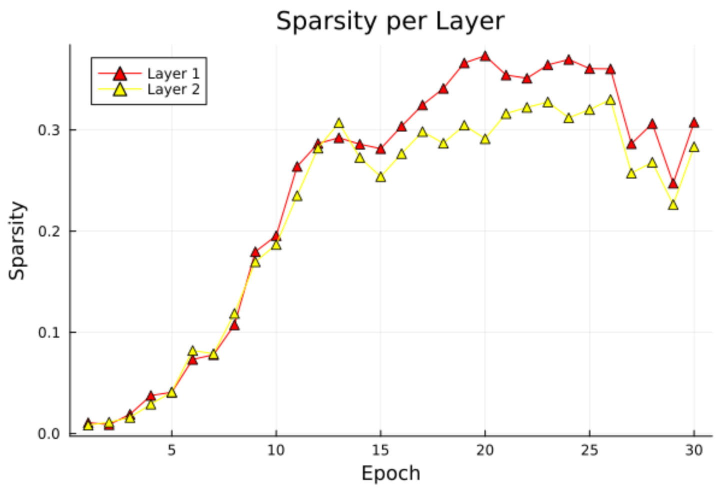


Figure 4.16: Sparisities for the Variational FFN

The steps above are repeated for smaller samples. The size of the dataset represents the number of total samples from which 50% will be used for training and the remaining 50% for testing.

Size	**Vocab Size**	**Dense Acc.**	**VD Acc.**	**Sparsity**	**Principled**	**Random**
50000	18510	80.9%	80.9%	30.76% - 28.32%	80.9%	80.7%
5000	4019	74.3%	74.9%	57.47% - 61.25%	74.9%	74.6%
1000	119	66.4%	66.8%	93.51% - 94.24%	66.8%	67%

Table 4.4: Comparison of model accuracy and FFN sparsity across different dataset sizes. "VD" refers to Variational Dropout, "Principled" to the accuracy when the weights are pruned based on the learned dropout α, and "Random" to the accuracy where a number of weights are pruned at random based only on the calculated sparsities.

The results in Table 4.5 indicate that replacing the feedforward network (FFN) with Variational Dropout layers does not result in clear improvements for this particular task. Interestingly, as the total dataset size decreases, the sparsity induced between variational dropout layers increases, yet pruning these weights do not improve accuracy. One possible explanation is that the Transformer classifier may be overparameterized for this dataset, so even random pruning has little to no effect on accuracy.

Overall, the current results do not seem to demonstrate a clear advantage to using Variational layers within Transformer architectures for sentiment classification. However, I believe this topic warrants further investigation. A more comprehensive exploration of Bayesian methods in Transformer architectures could form the basis of different research project.

5 Conclusions

The overall goals of this project were to investigate the Bayesian neural network and implement a practical inference algorithm for BNNs, that is, a practical building block to do Bayesian BNN in the language of Julia.

I created a Julia module, `VariationalMLP.jl`, that allows users to create a practical Bayesian MLP based on several VI implementations found in literature, namely in Molchanov et al.(2017), Kingma et al.(2015) and Graves(2011). I investigated the performance of Bayesian MLPs both as standalone architectures, or as part of an Encoder block of a Transformer, using the `VariationalMLP.jl` module.

In the following, I will circle back to the research questions, and try to answer them based on the results.

- Q1 : What advantages and practical challenges come with implementing Bayesian MLPs for supervised learning?

The main challenge is balancing implementation efficiency with adherence to the mathematical formulation. Benchmarking and experiments reveal that BNN layers consume roughly ten times more time and memory compared to standard dense layers in \texttt{Flux}. This computational cost is justified by the closer fidely to the Bayesian mathematical framework, which enables the model to learn useful sparsity, particularly when the KL divergence term is annealed.

An interesting finding is that BNNs with annealed KL divergence can serve as effective tools for inducing sparsity in pre-trained networks, facilitating weight pruning without accuracy loss. Practically, pre-training BNNs in a Frequentist fashion yields better initialization, improved accuracy (lower test error), and reduced training time overall.

A summary of the previous results :

- Variational dense layers can both induce up to 80-90% sparsity, with negligible drops in accuracy for large datasets like MNIST and Fashion MNIST.
- Variational dropout layers induce significantly less sparsity for MNIST and Fashion MNIST.

- For small datasets, variational dropout layers shine when it comes to interpretability as they prove robust to error quantification, but sparsity induced in these scenarios is not necessarily useful. As seen for the regression experiment, variational dense layers require more tuning, as the correct prior must be used, but variational dropout layers outperform all methods and they present good uncertainty quantification.

- MLPs with variational layers offer uncertainty quantification for accuracy but also for individual predictions.

Thus, Bayesian training offers a principled approach for pruning and uncertainty quantification at the cost of increased computational complexity. This cost can be mitigated by combining BNN training with Frequentist pre-training, as demonstrated in Section 4.1.

- Q2 : How do different methods of implementing Bayesian MLPs compare?

Variational dropout with additive reparameterization and variational dense layers converge faster than the multiplicative reparameterization approach. This difference appears to be related to parameterization: both additive methods use σ to absorb the dropout parameter α. Performance differences largely stem from variations in KL divergence implementations.

- Q3 : What are the advantages of replacing feedforward layers (MLPs) with Bayesian Variational Layers in transformer building blocks(Vaswani, 2017)?

No clear advantages were observed in this context, but replacing feedforward layers with variational ones does not harm accuracy. Random pruning of weights in Transformer feedforward networks similarly has little impact on performance, so the benefits of variational dropout remain unclear here. Also, randomly pruning the FFN layers seems to suggest that the Transformer classifier already contains to many parameters, which make it even more difficult to understand the advantages of variational dropout in this context. Further experiments should be conducted, but they would merit a separate, dedicated study.

There are several future directions that can be envisaged:

- Develop highly optimized variational layers, both with and without dropout.

- Extend variational inference techniques to the Transformer's attention layers and study variational Transformers more thoroughly.

- Consider the implementation of these reparameterisation tricks for convolutional layers.

Bibliography

[1] Andrieu, C., de Freitas, N., Doucet, A., & Jordan, M. I. (2003). An introduction to MCMC for machine learning. *Machine Learning*, *50*(1), 5–43. doi:10.1023/A:1020281327116

[2] Andrieu, C., & Thoms, J. (2008). A tutorial on adaptive MCMC. *Statistics and Computing*, 18, 343–373. https://doi.org/10.1007/s11222-008-9110-y

[3] Anwar, S., Hwang, K., & Sung, W. (2015). Structured pruning of deep convolutional neural networks. *arXiv preprint arXiv:1512.08571*. Retrieved from https://arxiv.org/abs/1512.08571

[4] Arbel, J., Pitas, K., Vladimirova, M., & Fortuin, V. (2023). A primer on Bayesian neural networks: Review and debates. *arXiv preprint arXiv:2309.16314*. https://arxiv.org/abs/2309.16314

[5] Arroyo, D. M., Postels, J., & Tombari, F. (2021). Variational Transformer Networks for Layout Generation. *arXiv preprint arXiv:2104.02416*. https://arxiv.org/abs/2104.02416

[6] Barber, D. (2012). *Bayesian Reasoning and Machine Learning*. Cambridge University Press.

[7] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer. Publisher Address: Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA. ISBN-10: 0-387-31073-8, ISBN-13: 978-0387-31073-2.

[8] Blei, D. M., Kucukelbir, A., & McAuliffe, J. D. (2018). Variational inference: A review for statisticians. *arXiv*. https://arxiv.org/abs/1601.00670

[9] Blundell, C., Cornebise, J., Kavukcuoglu, K., & Wierstra, D. (2015). Weight uncertainty in neural networks. *arXiv*. https://arxiv.org/abs/1505.05424

[10] Bruce, P., Bruce, A., & Gedeck, P. (2020). *Practical statistics for data scientists: 50 essential concepts* (2nd ed.). O'Reilly Media.

[11] Géron, A. (2023). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems* (3rd ed.). O'Reilly Media.

[12] Goan, E., & Fookes, C. (2020). Bayesian Neural Networks: An Introduction and Survey. In Mengersen, K., Pudlo, P., & Robert, C. (Eds.), *Case Studies in Applied Bayesian Data Science* (Lecture Notes in Mathematics, vol. 2259). Springer, Cham. https://doi.org/10.1007/978-3-030-42553-1_3

[13] Graves, A. (2011). Practical Variational Inference for Neural Networks. Retrieved March 18, 2025 from https://proceedings.neurips.cc/paper_files/paper/2011/file/7eb3c8be3d411e8ebfab08eba5f49632-Paper.pdf

[14] Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., & Peste, A. (2021). Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *arXiv.* https://arxiv.org/abs/2102.00554

[15] Jospin, L. V., Laga, H., Boussaid, F., Buntine, W., and Bennamoun, M. (2022). *Hands-On Bayesian Neural Networks—A Tutorial for Deep Learning Users. IEEE Computational Intelligence Magazine*, 17(2), 29–48. https://doi.org/10.1109/MCI.2022.3155327

[16] JuliaCI. (n.d.). *BenchmarkTools.jl documentation.* Retrieved May 20, 2025, from https://juliaci.github.io/BenchmarkTools.jl/stable/reference/

[17] Kang, Y., Hauswald, J., Gao, C., Rovinski, A., Mudge, T., Mars, J., & Tang, L. (2017). Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)* (pp. 615–629). ACM.

[18] Kingma, D. P., & Welling, M. (2022). Auto-encoding variational Bayes. *arXiv.* https://arxiv.org/abs/1312.6114

[19] Kingma, D. P., Salimans, T., & Welling, M. (2015). Variational Dropout and the Local Reparameterization Trick. *arXiv.* https://arxiv.org/abs/1506.02557

[20] LeCun, Y., Cortes, C., & Burges, C. J. (2010). *The MNIST database of handwritten digits.* Retrieved from http://yann.lecun.com/exdb/mnist/

[21] Li, Y., & Liu, F. (2020). Adaptive Gaussian Noise Injection Regularization for Neural Networks. In M. Han, S. Qin, & N. Zhang (Eds.), *Advances in Neural Networks – ISNN 2020*, Lecture Notes in Computer Science (Vol. 12557). Springer, Cham. https://doi.org/10.1007/978-3-030-64221-1_16

[22] Li, Z., Li, H., & Meng, L. (2023). Model compression for deep neural networks: A survey. *Computers, 12*(3), 60. https://doi.org/10.3390/computers12030060

[23] Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies* (pp. 142–150). Portland, Oregon, USA: Association for Computational Linguistics. Retrieved from http://www.aclweb.org/anthology/P11-1015

[24] Melville, J. (2017). Datasets. *Smallvis*. Retrieved February 20, 2025, from https://jlmelville.github.io/smallvis/datasets.html#frey_faces

[25] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., & Teller, E. (1953). Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, *21*(6), 1087–1091.

[26] Molchanov, D., Ashukha, A., & Vetrov, D. (2017). Variational Dropout Sparsifies Deep Neural Networks. *arXiv*. https://arxiv.org/abs/1701.05369

[27] Murphy, K. P. (2023). *Probabilistic Machine Learning: Advanced Topics*. MIT Press. http://probml.github.io/book2

[28] Ovadia, Y., Fertig, E., Ren, J., Nado, Z., Sculley, D., Nowozin, S., Dillon, J. V., Lakshminarayanan, B., & Snoek, J. (2019). *Can you trust your model's uncertainty? Evaluating predictive uncertainty under dataset shift* [Preprint]. arXiv. https://arxiv.org/abs/1906.02530

[29] Pinkus, A. (1999). *Approximation theory of the MLP model in neural networks*. Acta Numerica, 8, 143–195.10.1017/S0962492900002919

[30] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386–408. https://doi.org/10.1037/h0042519

[31] Samek, W., Montavon, G., Lapuschkin, S., Anders, C. J., & Müller, K.-R. (2021). Explaining deep neural networks and beyond: A review of methods and applications. *Proceedings of the IEEE*, *109*(3), 247–278. doi:10.1109/JPROC.2021.3060483

[32] Sinai, L. (2022, May 18). *Transformers from scratch* [Blog post]. Retrieved from https://liorsinai.github.io/machine-learning/2022/05/18/transformers.html

[33] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of*

Machine Learning Research, *15*(56), 1929–1958. Retrieved from http://jmlr.org/papers/v15/srivastava14a.html

[34] Tipping, M. E. (2001). Sparse Bayesian learning and the relevance vector machine. *Journal of Machine Learning Research*, *1*(Jun), 211–244.

[35] Tolstikhin, I., Houlsby, N., Kolesnikov, A., Beyer, L., Zhai, X., Unterthiner, T., Yung, J., Steiner, A., Keysers, D., Uszkoreit, J., Lucic, M., & Dosovitskiy, A. (2021). *MLP-Mixer: An all-MLP architecture for vision*. arXiv preprint arXiv:2105.01601. https://arxiv.org/abs/2105.01601

[36] Tran, M.-N., Nguyen, T.-N., & Dao, V.-H. (2021). A practical tutorial on Variational Bayes. *arXiv*. https://arxiv.org/abs/2103.01327

[37] Tzikas, D. G., Likas, A. C., & Galatsanos, N. P. (2008). The variational approximation for Bayesian inference. *IEEE Signal Processing Magazine*, *25*(6), 131–146. doi:10.1109/MSP.2008.929620

[38] Varghese, B., Wang, N., Barbhuiya, S., Kilpatrick, P., & Nikolopoulos, D. S. (2016). Challenges and opportunities in edge computing. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)* (pp. 20–26). IEEE. https://doi.org/10.1109/SmartCloud.2016.18

[39] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., . . . & Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems* (Vol. 30). Curran Associates, Inc.

[40] Vehtari, A., & Lampinen, J. (2000). Bayesian MLP neural networks for image analysis. *Pattern Recognition Letters*, 21(13), 1183–1191. *Selected Papers from The 11th Scandinavian Conference on Image*. https://doi.org/10.1016/S0167-8655(00)00080-5

[41] Wang, X., Pi, D., Zhang, X., Liu, H., & Guo, C. (2022). Variational transformer-based anomaly detection approach for multivariate time series. *Measurement, 191*, 110791. https://doi.org/10.1016/j.measurement.2022.110791

[42] Wenzel, F., Roth, K., Veeling, B. S., Świątkowski, J., Tran, L., Mandt, S., Snoek, J., Salimans, T., Jenatton, R., and Nowozin, S. (2020). *How Good is the Bayes Posterior in Deep Neural Networks Really?* arXiv preprint. https://arxiv.org/abs/2002.02405

[43] Xiao, H., Rasul, K., & Vollgraf, R. (2017). *Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms*. arXiv preprint arXiv:1708.07747. Retrieved from https://arxiv.org/abs/1708.07747

[44] Xiao, Y., Shao, H., Wang, J., Yan, S., & Liu, B. (2024). Bayesian variational transformer: A generalizable model for rotating machinery fault diagnosis. *Mechanical Systems and Signal Processing*, *207*, 110936. https://doi.org/10.1016/j.ymssp.2023.110936

[45] Yi, K., Zhang, Q., Fan, W., Wang, S., Wang, P., He, H., An, N., Lian, D., Cao, L., & Niu, Z. (2023). *Frequency-domain MLPs are More Effective Learners in Time Series Forecasting*. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, & S. Levine (Eds.), *Advances in Neural Information Processing Systems*, 36, 76656–76679. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2023/file/f1d16af76939f476b5f040fd1398c0a3-Paper-Conference.pdf

[46] Zhou, Z.-H. (2012). *Ensemble Methods: Foundations and Algorithms* (1st ed.). Chapman and Hall/CRC.

[47] Zhu, M., & Gupta, S. (2017). To prune, or not to prune: Exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*. Retrieved from https://arxiv.org/abs/1710.01878

A Appendices

A.1 VI: Mathematical Proofs

In this section, I want to derive the formula for variational energy based on Graves(2011). While the following derivation follows Graves(2011) closely, there are many details that I consider here.

I define a Neural Network as a parametric model that assigns a conditional probability to the data given its weights. This is the likelihood:

$$P(\mathcal{D} \mid \mathbf{w}) \tag{1}$$

The dataset $\mathcal{D}$ consists of features $\mathbf{x}$ and targets $\mathbf{y}$:

$$\mathcal{D} = \{\mathbf{x}, \mathbf{y}\} \tag{2}$$

The weights $\mathbf{w}$ are a set of real-valued parameters:

$$\mathbf{w} = \{w_i\}_{i=1}^{W} \tag{3}$$

A prior over weights, ensures the weights are regularized via $\boldsymbol{\alpha}$:

$$P(\mathbf{w} \mid \boldsymbol{\alpha}) \tag{4}$$

The objective is to compute the posterior over weights given the data:

$$P(\mathbf{w} \mid \mathcal{D}, \boldsymbol{\alpha}) = \frac{P(\mathbf{w} \mid \boldsymbol{\alpha}) P(\mathcal{D} \mid \mathbf{w})}{P(\mathcal{D})} \tag{5}$$

The loss as the negative log-likelihood:

$$\mathcal{L}^N(\mathbf{w}, \mathcal{D}) = -\ln P(\mathcal{D} \mid \mathbf{w}) \tag{6}$$

In variational inference, the aim is to find an approximate posterior Q:

$$Q(\mathbf{w} \mid \boldsymbol{\beta}) \tag{7}$$

This is done via minimizing the KL divergence between the variational distribution and the true posterior:

$$\mathrm{KL}(Q \parallel P) = \int Q(\mathbf{w} \mid \boldsymbol{\beta}) \ln \left[\frac{Q(\mathbf{w} \mid \boldsymbol{\beta})}{P(\mathbf{w} \mid \mathcal{D}, \boldsymbol{\alpha})} \right] d\mathbf{w} \tag{8}$$

Using $\ln(a/b) = \ln a - \ln b$:

$$\mathrm{KL}(Q \parallel P) = \int Q(\mathbf{w} \mid \boldsymbol{\beta}) \ln Q(\mathbf{w} \mid \boldsymbol{\beta}) \, d\mathbf{w} \tag{A.1}$$

$$- \int Q(\mathbf{w} \mid \boldsymbol{\beta}) \ln P(\mathbf{w} \mid \mathcal{D}, \boldsymbol{\alpha}) \, d\mathbf{w} \tag{9}$$

Substituting Eq. (5) into the second term:

$$\mathrm{KL}(Q \parallel P) = \int Q(\mathbf{w} \mid \boldsymbol{\beta}) \ln Q(\mathbf{w} \mid \boldsymbol{\beta}) \, d\mathbf{w} \tag{A.2}$$

$$- \int Q(\mathbf{w} \mid \boldsymbol{\beta}) \ln \left[P(\mathbf{w} \mid \boldsymbol{\alpha}) P(\mathcal{D} \mid \mathbf{w}) \right] d\mathbf{w} \tag{A.3}$$

$$+ \ln P(\mathcal{D}) \tag{10}$$

Using properties of logs:

$$\mathrm{KL}(Q \parallel P) = \int Q(\mathbf{w} \mid \boldsymbol{\beta}) \ln \left[\frac{Q(\mathbf{w} \mid \boldsymbol{\beta})}{P(\mathbf{w} \mid \boldsymbol{\alpha}) P(\mathcal{D} \mid \mathbf{w})} \right] d\mathbf{w} \tag{A.4}$$

$$+ \ln P(\mathcal{D}) \tag{12}$$

This becomes:

$$\mathrm{KL}(Q \parallel P) = -\mathbb{E}_Q \left[\ln \frac{P(\mathbf{w} \mid \boldsymbol{\alpha}) P(\mathcal{D} \mid \mathbf{w})}{Q(\mathbf{w} \mid \boldsymbol{\beta})} \right] + \ln P(\mathcal{D}) \tag{14}$$

Since $\mathrm{KL}(Q \parallel P) \geq 0$, it follows:

$$\ln P(\mathcal{D}) \geq \mathbb{E}_Q \left[\ln \frac{P(\mathbf{w} \mid \boldsymbol{\alpha}) P(\mathcal{D} \mid \mathbf{w})}{Q(\mathbf{w} \mid \boldsymbol{\beta})} \right] \tag{17}$$

The right-hand side is the Evidence Lower Bound (ELBO), and its negative is the variational free energy:

$$F(Q) = -\mathbb{E}_Q \left[\ln \frac{P(\mathbf{w} \mid \boldsymbol{\alpha}) P(\mathcal{D} \mid \mathbf{w})}{Q(\mathbf{w} \mid \boldsymbol{\beta})} \right] \tag{18}$$

Separating likelihood and prior:

$$F(Q) = \mathrm{KL}(Q(\mathbf{w} \mid \boldsymbol{\beta}) \parallel P(\mathbf{w} \mid \boldsymbol{\alpha})) + \mathbb{E}_Q \left[\mathcal{L}^N(\mathbf{w}, \mathcal{D})\right] \tag{20}$$

Now define:

$$L^E(\boldsymbol{\beta}, \mathcal{D}) = \mathbb{E}_Q \left[\mathcal{L}^N(\mathbf{w}, \mathcal{D})\right] \tag{21}$$

$$L^C(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \mathrm{KL}(Q(\mathbf{w} \mid \boldsymbol{\beta}) \parallel P(\mathbf{w} \mid \boldsymbol{\alpha})) \tag{22}$$

So the variational loss becomes:

$$L(\boldsymbol{\alpha}, \boldsymbol{\beta}, \mathcal{D}) = L^E(\boldsymbol{\beta}, \mathcal{D}) + L^C(\boldsymbol{\alpha}, \boldsymbol{\beta}) \tag{23}$$

A.1.1 Gaussian Approximation and Monte Carlo Estimation

Let $Q(\mathbf{w} \mid \boldsymbol{\beta})$ be a diagonal Gaussian:

$$Q(\mathbf{w} \mid \boldsymbol{\beta}) = \prod_{i=1}^{N} \mathcal{N}(w_i \mid \mu_i, \sigma_i^2) \tag{25}$$

And let the prior also be Gaussian:

$$P(\mathbf{w} \mid \boldsymbol{\alpha}) = \prod_{i=1}^{N} \mathcal{N}(w_i \mid \mu, \sigma^2) \tag{27}$$

The expected negative log-likelihood can be estimated via Monte Carlo:

$$L^E(\boldsymbol{\beta}, \mathcal{D}) \approx \frac{1}{S} \sum_{k=1}^{S} \mathcal{L}^N(\mathbf{w}^{(k)}, \mathcal{D}) \quad \text{where } \mathbf{w}^{(k)} \sim Q \tag{28}$$

A.1.2 Gradients

Using reparameterization and standard identities it is easy to show:

$$\frac{\partial L^E}{\partial \mu_i} \approx \frac{1}{S} \sum_{k=1}^{S} \frac{\partial \mathcal{L}^N(\mathbf{w}^{(k)}, \mathcal{D})}{\partial w_i} \tag{30}$$

$$\frac{\partial L^E}{\partial \sigma_i^2} \approx \frac{1}{2S} \sum_{k=1}^{S} \frac{\partial^2 \mathcal{L}^N(\mathbf{w}^{(k)}, \mathcal{D})}{\partial w_i^2} \tag{32}$$

KL Divergence Between Gaussians

Practically, the KL divergence is the expectation of the logarithm of the ratio:

$$L^C(\alpha,\beta) = \left\langle \log \frac{Q(\mathbf{w} \mid \beta)}{P(\mathbf{w} \mid \alpha)} \right\rangle_{\mathbf{w}\sim Q(\mathbf{w}|\beta)} \tag{33}$$

The logarithm of a ratio is the difference of logarithms:

$$L^C(\alpha,\beta) = \langle \log Q(\mathbf{w} \mid \beta) - \log P(\mathbf{w} \mid \alpha) \rangle_{\mathbf{w}\sim Q(\mathbf{w}|\beta)} \tag{34}$$

Recalling Eqs. (25) and (27), we expand the log-densities of Gaussians (dropping constants):

$$\log Q(\mathbf{w} \mid \beta) = \sum_{i=1}^{N} -\frac{1}{2}\log(2\pi\sigma_i^2) - \frac{1}{2\sigma_i^2}(w_i - \mu_i)^2 \tag{35}$$

$$\log P(\mathbf{w} \mid \alpha) = \sum_{i=1}^{N} -\frac{1}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}(w_i - \mu)^2 \tag{36}$$

Therefore, the difference becomes:

$$\log Q(\mathbf{w} \mid \beta) - \log P(\mathbf{w} \mid \alpha) = \sum_{i=1}^{N} \log\frac{\sigma}{\sigma_i} - \frac{1}{2}\left[\frac{1}{\sigma_i^2}(w_i - \mu_i)^2 - \frac{1}{\sigma^2}(w_i - \mu)^2\right] \tag{38}$$

Taking the expectation over Q:

$$\left\langle \sum_{i=1}^{N} \log\frac{\sigma}{\sigma_i} \right\rangle = \sum_{i=1}^{N} \log\frac{\sigma}{\sigma_i} \tag{39}$$

$$\left\langle \sum_{i=1}^{N} \frac{1}{\sigma_i^2}(w_i - \mu_i)^2 \right\rangle = \sum_{i=1}^{N} 1 \tag{40}$$

To compute the third term, we rewrite the squared term:

$$\left\langle \sum_{i=1}^{N} \frac{1}{\sigma^2}(w_i - \mu)^2 \right\rangle = \left\langle \sum_{i=1}^{N} \frac{1}{\sigma^2}\left[(w_i - \mu_i) + (\mu_i - \mu)\right]^2 \right\rangle \tag{42}$$

Expanding and using linearity of expectation:

$$\begin{aligned}\left\langle \sum_{i=1}^{N} \frac{1}{\sigma^2}(w_i - \mu)^2 \right\rangle &= \sum_{i=1}^{N} \frac{1}{\sigma^2}\left(\mathbb{E}_Q[(w_i - \mu_i)^2] + (\mu_i - \mu)^2 + 2(\mu_i - \mu)\mathbb{E}_Q[w_i - \mu_i]\right) \\ &= \sum_{i=1}^{N} \frac{\sigma_i^2}{\sigma^2} + \frac{(\mu_i - \mu)^2}{\sigma^2} \end{aligned} \tag{44}$$

Note: The cross-term vanishes because $\mathbb{E}_Q[w_i - \mu_i] = 0$.
Putting everything together:

$$L^C(\alpha,\beta) = \sum_{i=1}^{N} \log \frac{\sigma}{\sigma_i} + \frac{1}{2\sigma^2}\left(\sigma_i^2 + (\mu_i - \mu)^2 - \sigma^2\right) \tag{46}$$

The KL divergence can be expanded analytically:

$$L^C = \sum_{i=1}^{N} \ln \frac{\sigma}{\sigma_i} + \frac{1}{2\sigma^2}\left[\sigma_i^2 + (\mu_i - \mu)^2 - \sigma^2\right] \tag{46}$$

And its gradients:

$$\frac{\partial L^C}{\partial \mu_i} = \frac{\mu_i - \mu}{\sigma^2} \tag{47}$$

$$\frac{\partial L^C}{\partial \sigma_i^2} = \frac{1}{2}\left(\frac{1}{\sigma^2} - \frac{1}{\sigma_i^2}\right) \tag{48}$$

A.1.3 Gradient of the Total Loss

Combining the above:

$$\frac{\partial L}{\partial \mu_i} \approx \frac{\mu_i - \mu}{\sigma^2} + \frac{1}{S}\sum_{k=1}^{S} \frac{\partial \mathcal{L}^N(\mathbf{w}^{(k)}, \mathcal{D})}{\partial w_i} \tag{49}$$

$$\frac{\partial L}{\partial \sigma_i^2} \approx \frac{1}{2}\left(\frac{1}{\sigma^2} - \frac{1}{\sigma_i^2}\right) + \frac{1}{2S}\sum_{k=1}^{S} \frac{\partial^2 \mathcal{L}^N(\mathbf{w}^{(k)}, \mathcal{D})}{\partial w_i^2} \tag{50}$$

A.2 User Manual for VariationalMLP.jl

This is a quick guide to using the `VariationalMLP.jl` module, including model creation, data preparation, training, and pruning. All code was developed and tested in a Julia environment using version 1.11.3 and the Jupyter interface. The full list of package dependencies is recorded in the file `Project.toml`.

A.2.1 Installation and Requirements

Ensure you have the following dependecies installed:

- `Flux` (v0.14.25)
- `Random`

To use the custom `VariationalMLP.jl` module, the user must ensure that the file is correctly included and loaded from its local path

```
julia
cd /your/project/path
include("VariationalMLP.jl")
using .VariationalMLP
```

If already in a Jupyter notebook, then simply include the path:

```
julia
path = "your_path/VariationalMLP.jl"
include(path)
using .VariationalMLP
```

A.2.2 Creating a Model

Use the `make_model` function to build a variational MLP. The user can specify the architecture, the type of layers, the type of initialization. For a randomly initialized LeNet-300-100 network, with variational dropout layers done as per Molchanov et al.(2017) do:

Listing A.1: Creating a 3-layer variational MLP

```
using VariationalMLP, Flux

model = make_model([784, 300, 100, 10];
                   activations = [relu, relu],
                   final_activation = identity,
                   variant = :molchanov,
                   init = :random)
```

A.2.3 Training with Flux

The model is compatible with standard `Flux` training workflows via `withgradient`. The model can be trained for either a regression or classification task, and, with or without warming up, and up to the full variational objective. To set up training the model for classification, with warm up and up to the full variational objective do

Listing A.2: Training the variational model

```
opt = Flux.setup(Adam(1e-3), model)

 for (x, y) in data_loader

    loss, grads = Flux.withgradient(vd_model)do m
            energy_loss(m, x, y, N; kl_scale=1.0f0,
                                    enable_warmup=:true,
                                    task_type=:classification)
        end

    Flux.update!(opt, model, grads[1])
end
```

A.2.4 Monitoring Sparsity

You can inspect layer-wise sparsity after training:

Listing A.3: Checking sparsity

```
sparsity_logs = model_sparsity(model)
println("Layer-wise sparsity: ", sparsity_logs[1], sparsity_logs[2], sparsi
```

A.2.5 Pruning the Model

After training, prune based on thresholds on $\log\alpha$:

Listing A.4: Example pruning logic

```
for layer in model.layers
    if isa(layer, VariationalDropoutMolchanov)
        log   = @. layer.log 2 - 2f0 * log(abs(layer.  ) + 1f-8)
        mask = log   .> 3.0f0
        layer.   [mask] .= 0.0f0
        layer.log 2[mask] .= −1e10f0
    end
end
```

Different pruning functions exist for other layers.

A.2.6 Notes

- KL divergence is automatically selected based on the chosen variant.
- Supports both regression and classification via the `task_type` flag.
- Variational energy is summed per batches and logged per epoch, and can be plotted over time, same as sparisity.

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.
Tick one box

☐ **Staff Project**
☒ **Postgraduate Project**
☐ **Undergraduate Project**

Title of project

Practical variational inference of Bayesian neural networks

Name of researcher(s)

Andrei-Ioan Bleahu

Name of supervisor (for student research)

Lei Fang

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted **YES** ☒ **NO** ☐

There are no ethical issues raised by this project

Signature Student or Researcher

A.I.Bleahu

Print Name

Andrei-Ioan Bleahu

Date

Signature Lead Researcher or Supervisor

[illegible]

Print Name

Date

This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms. The School Ethics Committee will be responsible for monitoring assessments.

Computer Science Preliminary Ethics Self-Assessment Form

Research with secondary datasets

Please check UTREC guidance on secondary datasets (https://www.st-andrews.ac.uk/research/integrity-ethics/humans/ethical-guidance/secondary-data/ and https://www.st-andrews.ac.uk/research/integrity-ethics/humans/ethical-guidance/confidentiality-data-protection/). Based on the guidance, does your project need ethics approval?

YES ☐ **NO** ☒

** If your research involves secondary datasets, please list them with links in DOER.*

Research with human subjects

Does your research involve collecting personal data on human subjects?

YES ☐ **NO** ☒

If YES, full ethics review required

Does your research involve human subjects or have potential adverse consequences for human welfare and wellbeing?

YES ☐ **NO** ☒

If YES, full ethics review required
For example:
Will you be surveying, observing or interviewing human subjects?
Does your research have the potential to have a significant negative effect on people in the study area?

Potential physical or psychological harm, discomfort or stress

Are there any foreseeable risks to the researcher, or to any participants in this research?

YES ☐ **NO** ☒

If YES, full ethics review required
For example:
Is there any potential that there could be physical harm for anyone involved in the research?
Is there any potential for psychological harm, discomfort or stress for anyone involved in the research?

Conflicts of interest

Do any conflicts of interest arise?

YES ☐ **NO** ☒

If YES, full ethics review required
For example:
Might research objectivity be compromised by sponsorship?
Might any issues of intellectual property or roles in research be raised?

Funding

Is your research funded externally?

YES ☐ **NO** ☒

If YES, does the funder appear on the 'currently automatically approved' list on the UTREC website?

YES ☐ **NO** ☐

If NO, you will need to submit a Funding Approval Application as per instructions on the UTREC website.

Research with animals

Does your research involve the use of living animals?

YES ☐ [✓ **NO** ☒]

If YES, your proposal must be referred to the University's Animal Welfare and Ethics Committee (AWEC)

University Teaching and Research Ethics Committee (UTREC) pages
http://www.st-andrews.ac.uk/utrec/