UiO : Department of Mathematics
University of Oslo

Improving latent binary Bayesian neural networks using the local reparametrization trick and normalizing flows

Lars Skaaret-Lund

Master's Thesis, Spring 2022

This master's thesis is submitted under the master's programme *Data Science*, with programme option *Statistics and Machine Learning*, at the Department of Mathematics, University of Oslo. The scope of the thesis is 60 credits.

The front page depicts a section of the root system of the exceptional Lie group E_8, projected into the plane. Lie groups were invented by the Norwegian mathematician Sophus Lie (1842–1899) to express symmetries in differential equations and today they play a central role in various parts of mathematics.

Abstract

An artificial neural network (ANN) is a powerful machine learning method that is used in many modern big data applications such as facial recognition, machine translation and cancer diagnostics, to name a few. A common issue with ANNs is that they usually have millions of trainable parameters, and therefore tend to overfit to the training data. This is especially problematic in applications where it is important to have reliable uncertainty estimates. Bayesian neural networks (BNN) can improve on this, since they include parameter uncertainty in the model. In addition, latent binary Bayesian neural networks (LBBNN) are able to sparsify the networks to a large degree, without losing predictive power. In this thesis, we will build on the LBBNN model in two ways. Firstly, by using the local reparametrization trick (LRT) to sample the hidden units directly, and secondly by using normalizing flows on the variational posterior distribution of the LBBNN parameters. Experimental results show that using the LRT significantly improves predictive accuracy, in addition to being more computationally efficient. Using normalizing flows further improves accuracy. In both cases, these results are obtained with a similar or higher degree of sparsity than the LBBNN model.

Acknowledgements

To my wife Ariana, for all your patience, encouragement and support over the years. I could not have done this without you by my side. Our dog Ozzy, all those early morning walks helped me clear my head and provided a much needed distraction. I would also like to thank my supervisors Aliaksandr Hubin and Geir Storvik. It feels like a privilege to have been introduced to your work, and I have taken a strong interest in this field. Thank you for providing guidance, challenging me to become better and for the many interesting discussions (even if mostly on zoom)!

Contents

Contents 3

1 Introduction 4

2 Background 6
2.1 Bayesian neural networks (BNN) 6
2.2 Markov chain Monte Carlo (MCMC) 8
2.3 Variational Inference (VI) 10
2.4 Sparsity in neural networks 11

3 Normalizing flows 13
3.1 Background 13
3.2 Using normalizing flows on the parameters of a BNN 14
3.3 Types of transformations 14

4 The latent binary Bayesian neural network (LBBNN) 18
4.1 LBBNN-GP-MF with the local reparametrization trick (LRT) . 19
4.2 Multiplicative normalizing flows (MNF) 22
4.3 Combining LBBNN-GP-MF with MNF 23

5 Experiments 24
5.1 Background 24
5.2 Logistic regression simulation study 25
5.3 Classification experiments 28
5.4 Uncertainty estimation 32

6 Discussion 36
6.1 Contributions 37
6.2 Limitations 39
6.3 Future research 40

Bibliography 41

1 Introduction

The idea of using a mathematical model to imitate how the brain works was first introduced in McCulloch and Pitts (1943). However it was not until more recent years that the true power of these models could be harnessed with the idea of using backpropagation (Rumelhart et al. 1986) to train the model with gradient descent. With the advent of modern GPU architectures, deep neural networks can be scaled to big data, and have shown to be very successful on a variety of tasks including computer vision (Voulodimos et al. 2018), natural language processing (Young et al. 2018) and reinforcement learning (Li 2018).

Modern deep learning architectures can have millions of trainable parameters (Khan et al. 2020). Due to the large number of parameters in the model, the network has the capacity to overfit, and therefore may not generalize well to unseen data. Various regularization methods are used to try to deal with this, such as early stopping (Prechelt 1998), dropout (Srivastava et al. 2014) or data augmentation (Shorten and Khoshgoftaar 2019). These techniques are empirical in nature and therefore it is not always clear how to use them and how well they work in practice. Another issue with deep learning models is that they often make overconfident predictions. In Szegedy et al. (2013), it was shown that adding a small amount of noise to an image can trick a classifier into making a nonsensical prediction, even though the image looks exactly the same to the human eye. The opposite is also possible, images that are white noise can be classified with almost complete certainty to belong to a specific class (Nguyen et al. 2015).

Figure 1.1 illustrates a dense neural network, where each neuron is connected to all the neurons of the previous layer. The output of each neuron (or activation) is given by an affine transformation followed by a non-linear element-wise activation function in the following way,

$$b_j^L = \sigma^L \left(\sum_{i=1}^{N} W_{ij}^L b_i^{L-1} + c_j^L \right) \tag{1.1}$$

where b_j^L denotes the j-th activation at layer L, σ^L is the activation function at layer L and W_{ij}^L is the weight matrix where i and j enumerate all the edges between layer $L-1$ and layer L. For each activation, j at each layer L, we also have a bias node, c_j^L. Note that if we are doing classification, then the last layer will use the softmax activation function, which is defined as

$$f(\mathbf{x})_i = \frac{e^{x_i}}{\sum\limits_{j=1}^{K} e^{x_j}},$$

where each of the K elements (classes) of the input vector $\mathbf{x}$ can be a real number. Dividing by the sum of the exponentials ensures that sum of the elements of the output vector will be one, and each element between zero and one, therefore the outputs can be interpreted as probabilities. Thus, if we assume that $\mathbf{y}$ is discrete and $p(\mathbf{y}|\mathbf{x})$ follows a multinomial distribution, we minimize the cross-entropy loss between the output $\hat{\mathbf{y}}$ and the true label $\mathbf{y}$, with

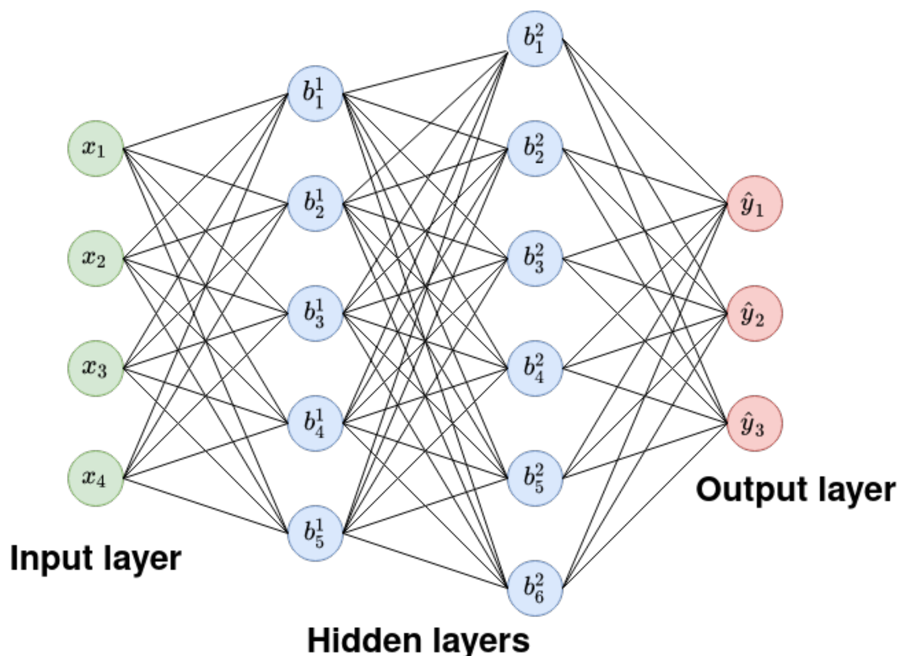

Figure 1.1: fully connected neural network, with four input features, two hidden layers with five and six neurons respectively, and an output layers consisting of three neurons.

respect to the parameters (all the weights and biases of the neural network), which is equivalent to maximizing the log-likelihood. We then compute the derivative of the loss with regards to the parameters, which then are updated with (stochastic) gradient descent or another gradient based optimization routine. The gradients are computed via the chain rule of vector calculus, and then propagated back to the parameters of the network.

Instead of using the full dataset on each forward and backward pass, one typically draws a random minibatch from the training data. One epoch is defined as one pass through the entire training dataset. After some stopping criterion is reached, prediction is made on new data, using the values of the weights and biases that minimized cross-entropy loss. Optimization with the gradient estimated from a random mini-batch of data is referred to as stochastic gradient descent (SGD). This has some advantages. One is to avoid getting stuck in local minima, since there is randomness introduced into the gradient. Secondly, with huge datasets, it might not be possible to fit the entire dataset into memory at the same time. Furthermore, SGD is much more computationally efficient since we only use a fraction of the dataset to compute the (approximate) gradient. A disadvantage of using SGD compared to doing gradient descent on the whole training set is that the gradients will be more noisy, since they are approximated using a small batch. More formally, if we have a gradient estimator, $\widehat{\nabla} g(\theta)$ where θ denotes the set of all parameters of our neural network, and g is the

loss function, we then have the following update rule for the parameters,

$$\theta_{t+1} = \theta_t - \lambda \widehat{\nabla} g(\theta_t),$$

where λ denotes the learning rate. This is an important hyperparameter, since it will determine the rate of convergence. With a large update, SGD may converge too quickly to a sub-optimal local minimum or jump over the optimal solution. Conversely, convergence will be slow with a minuscule learning rate. As mentioned in Bottou et al. (2018), we must also have an unbiased gradient estimator, i.e. $\mathbb{E}\left[\widehat{\nabla} g(\theta)\right] = \nabla g(\theta)$.

In this thesis we will also be using the Adam (Kingma and Ba 2014) optimizer, which can be considered an extension of SGD. Here we take into consideration the exponentially decaying mean of the gradient, and the squared gradient,

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \end{aligned}$$

where g_t denotes the gradient at step t, m_t and v_t denote the estimates of the first and second moment of the gradient respectively. In addition, we have the exponential decay rates β_1 and $\beta_2 \in [0, 1)$. Since these estimates are biased we also compute the bias corrected estimates,

$$\begin{aligned} \widehat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \widehat{v}_t &= \frac{v_t}{1-\beta_2^t}. \end{aligned}$$

This gives the following update rule,

$$\theta_t = \theta_{t-1} - \frac{\alpha \widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon},$$

with α being the learning rate, and ϵ a small constant added for numerical stability. Intuitively, it is an advantage to let past gradient information affect how much to update the current gradient. If the loss surface is currently very flat, we would want a large update in order to get out of it quickly; conversely, if it is steep, we want a small update in order to not overshoot a potential minimum. In addition, using exponentially decaying moment estimators means that the most recent gradients have much larger influence than ones in the past.

2 Background

2.1 Bayesian neural networks (BNN)

In the frequentist setting, we use maximum likelihood estimation to find the values of the model parameters that maximize the probability of observing the given data. In the Bayesian framework, we instead assume that the model parameters follow some probability distribution, instead of being unknown constants. If we let $\mathbf{w}$ denote all the parameters of the model (i.e. the weights and biases of each layer of our neural network), we can then define a prior

distribution over the parameters, $p(\mathbf{w})$, that is meant to incorporate any prior information we may have about the parameters before observing the data. After observing the data, we use Bayes rule to update our prior belief and obtain the posterior distribution of the parameters,

$$p(\mathbf{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\mathbf{w})p(\mathbf{w})}{p(\mathcal{D})},$$

where $p(\mathcal{D}|\mathbf{w})$ is the likelihood of the data, and $p(\mathcal{D})$ is the marginal distribution of the data, realized by integrating out all the parameters:

$$p(\mathcal{D}) = \int p(\mathcal{D}|\mathbf{w})p(\mathbf{w})\, d\mathbf{w}. \tag{2.1}$$

In the frequentist setting, we would have used the maximum likelihood estimates of $\mathbf{w}$ to predict on new data, $\tilde{\boldsymbol{y}}$. In the Bayesian framework, we instead use the posterior predictive distribution to do inference on new data,

$$\begin{aligned} p(\tilde{\boldsymbol{y}}|\mathcal{D}) &= \int p(\tilde{\boldsymbol{y}}, \mathbf{w}|D)\, d\mathbf{w} \\ &= \int p(\tilde{\boldsymbol{y}}|\mathbf{w}, \mathcal{D})p(\mathbf{w}|\mathcal{D})\, d\mathbf{w} \\ &\approx \frac{1}{N}\sum_{i=1}^{N} p(\tilde{\boldsymbol{y}}|\mathbf{w}_{(i)}). \end{aligned}$$

Here, $\mathbf{w}_{(i)}$ denotes the i-th Monte Carlo sample (Shapiro 2003) from the posterior distribution, $p(\mathbf{w}|\mathcal{D})$. Monte Carlo sampling theory tells us that given enough samples, this sum will converge to the true distribution. This equation says that posterior inference is done by averaging over all possible values of the parameters $\mathbf{w}$, approximated by Monte Carlo samples, and we have thus incorporated parameter uncertainty into the model. This is advantageous, because it provides more realistic estimation of predictive uncertainty compared to classical neural networks, that tend to be overconfident.

However, there are some caveats. Due to the ultra high dimensionality of $\mathbf{w}$, the integral (2.1) is not tractable, which makes sampling directly from the posterior not possible; therefore it is not straightforward to use the posterior predictive distribution for inference. This is often the case in Bayesian statistics in general, so there exist well established methods to deal with it. The classical method is Markov chain Monte Carlo (Geyer 1992), which is able to generate samples from the true posterior. A more recent addition is variational inference (Blei et al. 2017), which can only approximate the posterior, but is more scalable. These methods will be discussed in more detail in the following sections. Another issue is the choice of prior. It may not be easy to come up with a reasonable prior for a distribution over millions of neural network parameters, but one possible choice is a prior that incorporates the prior information that we believe many of the neural network parameters should be zero. There is also an added computational burden with BNNs, since typically they will have two parameters (if we are using variational inference) to train per weight instead of just one as in classical neural networks.

2.2 Markov chain Monte Carlo (MCMC)

As mentioned earlier, one possible way to obtain samples from the posterior distribution is using Markov chain Monte Carlo. A Markov chain, or Markov process (Brooks et al. 2011) is a stochastic process, describing a sequence of states $X_1, X_2, X_3, ...$ where the probability of *transitioning* to the next state only depends on the present state,

$$p(X_{i+1} = x_{i+1}|X_i = x_i, \cdots, X_1 = x_1) = p(X_{i+1} = x_{i+1}|X_i = x_i). \tag{2.2}$$

If X is discrete, we refer to this sequence of states as a Markov chain, otherwise it is called a Markov process. If certain assumptions are met, then we can find the unique stationary distribution of the chain, which is a probability distribution that does not change with time. For a Markov chain, these assumptions are:

- Irreducibility; all states communicate with each other, i.e. it is possible to get from any state to any other state
- Aperiodicity; this assumption is met if all the states in the chain are aperiodic, meaning that each state gets visited at irregular time intervals.

If the first property is not met, the chain can get trapped in a subset of states without being able to explore the rest of the possible states. If the chain is periodic, it will have one or more states that can only be visited at regular time intervals (e.g. $2, 4, 6, ...$ steps). The key idea behind MCMC is to create a Markov chain with a stationary distribution that is equal to the probability distribution that we are interested in sampling from, and then use that chain to generate samples from the intractable posterior distribution.

There are several algorithms for constructing such chains. Metropolis-Hastings (Chib and Greenberg 1995) is a method where one uses a proposal distribution to simulate values that are either accepted or rejected:

$$\begin{aligned} &\text{1: sample } \mathbf{x}^\star \text{ from the proposal density } q(\mathbf{x}^\star|\mathbf{x}) \\ &\text{2: compute acceptance probability} \\ &A(\mathbf{x}, \mathbf{x}^\star) = \min\left(1, \frac{p(\mathbf{x}^\star)q(\mathbf{x}|\mathbf{x}^\star)}{p(\mathbf{x})q(\mathbf{x}^\star|\mathbf{x})}\right) \end{aligned} \tag{2.3}$$

Here $\mathbf{x}$ is the current state and $p(\cdot)$ is the target distribution that we want samples from. Due to the ratio in the above formula, the typically intractable denominator in the formula for the Bayesian posterior distribution will get cancelled out. This simple procedure will generate a (correlated) sample from p, given that q is constructed in a way that it is irreducible and aperiodic. The choice of q and how long to run the chain is not obvious, and Metropolis-Hastings suffers from the curse of dimensionality if $\mathbf{x}$ is very high dimensional. However it does have the advantage of not needing to know any conditional distributions, as we must with Gibbs sampling, which shall be described next.

With Gibbs sampling (Gelfand 2000), we sample sequentially from the univariate conditional distributions of the target density. This then requires

enough knowledge about the target distribution in order to be able to derive the conditionals. In the bivariate case, the algorithm is as follows,

$$\begin{aligned}
&\text{for } t = 1, 2, ..., N \\
&x_2^t \sim p(x_2|x_1^{t-1}) \\
&x_1^t \sim p(x_1|x_2^t)
\end{aligned} \tag{2.4}$$

where we always sample conditionally on the current value of the other variable. This sequence generates a Markov chain with stationary distribution $p(x_1, x_2)$. The algorithm can easily be extended to more than two dimensions, and to sampling blocks of variables at a time.

Finally we will mention the state of the art algorithm for MCMC, Hamiltonian Monte Carlo (HMC) (Neal et al. 2011, Betancourt 2017). HMC uses Hamiltonian dynamics in order to generate proposals that are better able to explore the high dimensional space of the target distribution compared to Metropolis-Hastings, and therefore the chain will converge quicker. The main idea is to introduce an auxiliary momentum variable τ with the same dimensionality as the position variable x, and then sample from the joint density $p(x, \tau)$. The Hamiltonian is defined as,

$$H(x, \tau) = -\log p(x|\tau) - \log p(\tau),$$

and we use Hamilton's equations to update to the new position and momentum,

$$\begin{aligned}
\frac{dx}{dt} &= \frac{\partial H}{\partial \tau} \\
\frac{d\tau}{dt} &= -\frac{\partial H}{\partial x}.
\end{aligned}$$

Solving this system of differential equations is done with the leapfrog integrator, an algorithm for numerical integration.

In practice, most BNN implementations use variational inference (discussed in the next section) rather than MCMC, since the computational cost of MCMC in ultra-high dimensions and with big datasets is too great (Papamarkou et al. 2019). A few exceptions exist, such as the NIPS 2003 feature selection challenge (Guyon et al. 2004), where the winners used a combination of BNNs and Dirichlet diffusion trees. According to Neal and Jianguo Zhang (2003), the BNNs consisted of two hidden layers with 25 and 8 neurons respectively, where running the model with MCMC took around one day. A more recent work (Izmailov et al. 2021) is able to use HMC on modern deep learning architectures, using parallel computation on TPU (Tensor Processing Unit) devices. This is the first work that tries to directly sample the posterior in a modern BNN. However, it is not possible to be certain that the chain has converged and that the samples obtained are from the true posterior distribution. Still, some interesting results are obtained from this paper such as BNNs usually performing better in terms of predictive accuracy compared to non-Bayesian models, and that the cold posterior effect (Wenzel et al. 2020), which refers to the empirical observation that predictive performance can be improved by scaling the likelihood and the prior by $\frac{1}{T}$, with $T < 1$, may be a result of data augmentation. Furthermore, the paper posits that the choice of prior has little impact on performance, and that the common independent Gaussian prior is not particularly bad.

2.3 Variational Inference (VI)

We have already established that sampling from the posterior distribution directly is not possible, and furthermore that MCMC methods are too slow when we have millions of weights and potentially large datasets. However we can avoid these problems by adopting VI, where instead of trying to handle the posterior directly, we define a variational posterior distribution, $q_\theta(\mathbf{w})$. This is a probability distribution over all the weights, $\mathbf{w}$, parameterized by the vector θ. Intuitively, the goal is to find the variational posterior that most closely resembles the true posterior, and then use this distribution to do posterior inference instead. By defining what 'closeness' should mean, we can optimize (i.e. take the gradient) of the parameters θ to move the variational posterior closer to the true posterior, $p(\mathbf{w}|\mathcal{D})$.

The most common way to measure closeness between two probability distributions is to use the Kullback-Leibler (KL) divergence, which is defined as,

$$
\begin{aligned}
\mathrm{KL}\left[q_\theta(\mathbf{w})||p(\mathbf{w}|\mathcal{D})\right] &= \int q_\theta(\mathbf{w}) \log \frac{q_\theta(\mathbf{w})}{p(\mathbf{w}|\mathcal{D})}\, d\mathbf{w} \\
&= \mathbb{E}_{q_\theta(\mathbf{w})}\left[\log \frac{q_\theta(\mathbf{w})}{p(\mathbf{w}|\mathcal{D})}\right] \\
&= \mathbb{E}_{q_\theta(\mathbf{w})}\left[\log q_\theta(\mathbf{w})\right] - \mathbb{E}_{q_\theta(\mathbf{w})}\left[\log p(\mathbf{w}|\mathcal{D})\right] \\
&= \mathbb{E}_{q_\theta(\mathbf{w})}\left[\log q_\theta(\mathbf{w})\right] - \mathbb{E}_{q_\theta(\mathbf{w})}\left[\log p(\mathbf{w},\mathcal{D})\right] \\
&+ \mathbb{E}_{q_\theta(\mathbf{w})}\left[\log p(\mathcal{D})\right] \\
&= \mathbb{E}_{q_\theta(\mathbf{w})}\left[\log q_\theta(\mathbf{w})\right] - \mathbb{E}_{q_\theta(\mathbf{w})}\left[\log p(\mathbf{w},\mathcal{D})\right] + \log p(\mathcal{D}).
\end{aligned}
$$

Since $\log p(\mathcal{D})$ does not depend on θ, minimizing the KL-divergence is then equivalent to maximizing the evidence lower bound (ELBO),

$$
\begin{aligned}
\mathrm{ELBO}(q) &= \mathbb{E}_{q_\theta(\mathbf{w})}\left[\log p(\mathbf{w},\mathcal{D})\right] - \mathbb{E}_{q_\theta(\mathbf{w})}\left[\log q_\theta(\mathbf{w})\right] \\
&= \mathbb{E}_{q_\theta(\mathbf{w})}\left[\log p(\mathcal{D}|\mathbf{w})\right] + \mathbb{E}_{q_\theta(\mathbf{w})}\left[\log p(\mathbf{w})\right] - \mathbb{E}_{q_\theta(\mathbf{w})}\left[\log q_\theta(\mathbf{w})\right] \\
&= \mathbb{E}_{q_\theta(\mathbf{w})}\left[\log p(\mathcal{D}|\mathbf{w})\right] - \mathrm{KL}\left[q_\theta(\mathbf{w})||p(\mathbf{w})\right].
\end{aligned} \tag{2.5}
$$

We see that in order to maximize the ELBO, we want to find the parameters θ that maximize the expected log likelihood, and at the same time minimize the KL divergence between the prior and the variational posterior over the weights. Note that since we can not compute the expectations directly, the ELBO is *approximated* with samples from the variational posterior, $q_\theta(\mathbf{w})$. In BNNs this can be quite expensive, since one forward pass is required for each sample, therefore most applications use only one sample (Hubin and Storvik 2019, Louizos and Welling 2017). A further complication is that when we want to compute the gradient of the ELBO with regards to θ,

$$
\nabla_\theta \mathrm{ELBO}(q) = \nabla_\theta \left[\mathbb{E}_{q_\theta(\mathbf{w})}\left[\log p(\mathbf{w},\mathcal{D})\right] - \mathbb{E}_{q_\theta(\mathbf{w})}\left[\log q_\theta(\mathbf{w})\right]\right],
$$

we can not simply interchange the order of the gradient and the expectation, since the distribution we are sampling from is also parametrized by θ. Therefore using Monte Carlo samples to estimate the gradient is not guaranteed to

converge. To sample from a distribution that is independent of θ, we use the reparametrization trick (Kingma and Welling 2014), where instead of sampling $\mathbf{w}$ from $q_\theta(\mathbf{w})$, we sample a noise variable (typically standard normal) $\epsilon \sim p(\epsilon)$ and compute $\mathbf{w}$ deterministically as $\mathbf{w} = g(\epsilon, \theta)$. We then get the following expression for the gradient of the ELBO

$$\begin{aligned}\nabla_\theta \text{ELBO}(q) &= \nabla_\theta \left[\mathbb{E}_{p(\epsilon)}\left[\log p(\mathbf{w} = g(\epsilon,\theta), \mathcal{D})\right] - \mathbb{E}_{p(\epsilon)}\left[\log q_\theta(\mathbf{w} = g(\epsilon,\theta))\right]\right] \\ &\approx \frac{1}{M}\sum_{i=1}^{M} \nabla_\theta \log p(\mathbf{w} = g(\epsilon^{(i)},\theta), \mathcal{D}) - \nabla_\theta \log q_\theta(\mathbf{w} = g(\epsilon^{(i)},\theta)).\end{aligned}$$

For the rest of this thesis, we will be using $M = 1$.

A common choice for the variational posterior is to use the product of independent Gaussian distributions, where for any independent layer we have,

$$q_\theta(\mathbf{w}) = \prod_{i,j} \mathcal{N}(w_{ij}; \mu_{ij}, \sigma_{ij}),$$

where i and j are used to index the neurons from the previous layer and the current layer respectively. Applying the reparametrization trick here means sampling ϵ_{ij} as a standard normal variable, and then obtain $w_{ij} = \mu_{ij} + \sigma_{ij}\epsilon_{ij}$. Furthermore, if the prior is also a product of independent Gaussians, the KL-divergence between the prior and the variational posterior $\text{KL}\left[q_\theta(\mathbf{w})||p(\mathbf{w})\right]$ can be computed analytically. In this case, it is advantageous to also employ the local reparametrization trick (LRT) (Kingma et al. 2015). If the weights are distributed as independent Gaussians, the activations, (before a non-linearity is applied) will also be Gaussian (Weisstein 2000). Therefore, it is possible to sample the implied Gaussian activations directly without sampling the weights. This is more computationally efficient, since the activations are of much lower dimension than the weights (a layer with 1000 input and 1000 output neurons will have a weight matrix with 1 million entries, but only 1000 activations). It also leads to a gradient estimator that has lower variance. However, it restricts the prior and the posterior distributions to be Gaussian, or other distributions where the KL-divergence between the prior and the variational posterior can be computed analytically. If the analytical form of the KL-divergence is not available, then the second line of (2.5) must be used to estimate the ELBO.

2.4 Sparsity in neural networks

A sparse neural network is defined to be a neural network in which a large proportion of the weights are equal to 0. This is good for two main reasons. Firstly, reducing the number of trainable parameters reduces the risk of overfitting. Another benefit is that only non-zero weights need to be stored, thus reducing the memory footprint. In addition, NVIDIA, the leading GPU manufacturer for machine learning, has developed hardware that is capable of accelerating computation time on sparse matrices (Salvator 2020). Sparsity is thus desirable in applications where computation speed and storage constraints are concerns (e.g. mobile devices) (Ablavatski and Dukhan 2021).

The most common way to induce sparsity is with pruning. This is usually done with the dense-to-sparse method (Han et al. 2017). Here, a dense network is trained, while the importance of the weights (i.e. their magnitude) is recorded. Then, the weights that fall below the sparsity threshold (a hyperparameter) are removed. In Frankle and Carbin (2019), it is hypothesized that in randomly initialized dense networks, there exists a sparse sub-network (the winning lottery ticket) that can be trained in isolation and that gives the same test accuracy as the original dense network. Instead of training and pruning once, referred to as one-shot pruning, this process is repeated several times, removing a certain percentage of the remaining weights each time, which then results in networks that have a higher degree of sparsity than the ones found with one-shot pruning. However, this comes at a higher computational cost, due to the iterative nature of this algorithm. Further refinements to this are done in Evci et al. (2020), where the network starts off sparse, and dynamically removes the weights with the smallest magnitude, while also adding new connections based on gradient information.

Nevertheless, the approaches mentioned here are heuristic – they do not have a solid theoretical underpinning. It is therefore interesting to consider sparsity inducing methods from a Bayesian perspective. This is typically done by using sparsity inducing priors, as in variational dropout (Kingma et al. 2015, Molchanov et al. 2017), which use independent log uniform priors on the weights,

$$p(\log(|w_{ij}|)) \propto c.$$

This is an improper prior, meaning that there is no value for c in which it integrates to 1, and thus it can not be a valid probability distribution. As noted in Hron et al. (2017), using this prior, combined with commonly used likelihood functions leads to an improper posterior, which means that the obtained results can not be explained from a Bayesian modelling perspective. It is argued that variational dropout should instead be interpreted as penalized maximum likelihood estimation of the variational parameters. In Gale et al. (2019), it is also found that while variational dropout works well on smaller networks, it gets outperformed by the heuristic methods on bigger networks. Another type of sparsity inducing prior is the independent scale mixture prior, where the author of Blundell et al. (2015) proposes a mixture of two Gaussian densities,

$$p(w_{ij}) = \pi\mathcal{N}(0, \sigma_1^2) + (1-\pi)\mathcal{N}(0, \sigma_2^2)$$

where using a small variance for the second mixture component leads to many of the weights having a prior around 0. Another possibility is to use the independent spike-and-slab prior,

$$\begin{aligned} p(w_{ij}|\gamma_{ij}) &= \gamma_{ij}\mathcal{N}(0, \sigma^2) + (1-\gamma_{ij})\delta(w_{ij}) \\ p(\gamma_{ij}) &= \text{Bernoulli}(\alpha) \end{aligned}$$

where δ is the Dirac delta function, which is a function over the real line that is defined to be 0 everywhere except for at 0, where it has a spike. In Hubin and Storvik (2019), it was shown that using this prior will induce a very sparse network while maintaining good predictive accuracy. The scale mixture and spike-and-slab priors are very similar, but with one key difference. In the scale

mixture prior, the mixture weight, π, is a fixed hyperparameter. When using the spike-and-slab prior, the mixture weights γ_{ij} have a prior themselves, therefore this approach is able to incorporate both model uncertainty and uncertainty in the weights, given the model. By model we mean the network defined by the set of weights corresponding to the γ_{ij} that are equal to one. Note that this model space is enormous, since neural networks typically have millions of weights.

3 Normalizing flows

3.1 Background

The idea behind normalizing flows is that they can be used to construct a complicated probability distribution through a series of learnable transformations of some initial (simple) distribution. Consider the following univariate example. Let $q_0(z_0)$ be distributed according to the standard normal distribution,

$$q_0(z_0) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}z_0^2\right)$$

and define the following transformation,

$$f(z_0) = \mu + \sigma z_0 = z_1,$$

with inverse

$$f^{-1}(z_1) = \frac{z_1 - \mu}{\sigma} = z_0.$$

Using Theorem 2.1.5. from Casella et al. (2002), we can obtain the probability distribution of the transformed random variable z_1 as,

$$\begin{aligned} q_1(z_1) &= q_0(f^{-1}(z_1)) \left|\frac{\partial f^{-1}(z_1)}{\partial z_1}\right| = q_0(f^{-1}(z_1)) \left|\frac{\partial f(z_0)}{\partial z_0}\right|^{-1} \\ &= \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{z_1-\mu}{\sigma}\right)^2\right) \left|\frac{1}{\sigma}\right| = \mathcal{N}(\mu, \sigma). \end{aligned} \tag{3.1}$$

We see that the new variable z_1 follows a normal distribution with parameters μ and σ. This is what we earlier referred to as the reparametrization trick. The formula above can be generalized to a vector random variable $\mathbf{z}$, and chaining together several transformations f as follows,

$$\mathbf{z}_k = f_k \circ ... \circ f_2 \circ f_1(\mathbf{z}_0) \tag{3.2}$$

with the log density of the transformed variable $\mathbf{z}_k$ given as,

$$\log q_k(\mathbf{z}_k) = \log q_0(\mathbf{z}_0) - \sum_{k=1}^{K} \log \left|\det \frac{\partial f_k}{\partial \mathbf{z}_{k-1}}\right|, \tag{3.3}$$

where we now have to compute the log determinant of the Jacobian of the transformation, since $\mathbf{z}$ is a vector.

3.2 Using normalizing flows on the parameters of a BNN

As mentioned previously, the variational posterior in BNNs typically consists of independent Gaussians for each weight in the network. This is a simple distribution that is unable to capture the complexity of the true posterior. If we instead transform it with normalizing flows, we can obtain a more complicated distribution, which in turn should make the variational approximation better. With a better approximation, we expect our network performance to improve, since we are now able to sample from something that more closely resembles the true posterior.

However there are some caveats. Equation (3.2) must be efficient to compute, since $\mathbf{z}$ will be very high dimensional and we might need K to be large in order to make the flow sufficiently complex. In addition, the Jacobian of (3.3) must be tractable and fast to compute, and the transformations f_k should not have too many parameters, as this will again slow down computation time and overparametrize our BNN. To avoid issues of slow computation and overparametrization, Louizos and Welling (2017) only consider using normalizing flows on the activations instead of the weights of the neural network. In section 4.2, this is discussed in detail, and in this thesis we will also only consider normalizing flows in the activation space.

It is however interesting to see how straightforward it is to apply normalizing flows to the variational posterior distribution on the weights of a BNN. Let $\mathbf{w}_0$ denote an initial sample of the weights, and $\mathbf{w}_k = f_k \circ ... \circ f_2 \circ f_1(\mathbf{w}_0)$. We then get the following ELBO,

$$\begin{aligned}\text{ELBO}(q) &= \mathbb{E}_{q_k(\mathbf{w}_k)}\left[\log p(\mathcal{D}|\mathbf{w}_k)\right] + \mathbb{E}_{q_k(\mathbf{w}_k)}\left[\log p(\mathbf{w}_k)\right] - \mathbb{E}_{q_k(\mathbf{w}_k)}\left[\log q_k(\mathbf{w}_k)\right] \\ &= \mathbb{E}_{q_k(\mathbf{w}_k)}\left[\log p(\mathcal{D}|\mathbf{w}_k)\right] + \mathbb{E}_{q_k(\mathbf{w}_k)}\left[\log p(\mathbf{w}_k)\right] \\ &\quad - \mathbb{E}_{q_0(\mathbf{w}_0)}\left[\log q_0(\mathbf{w}_0)\right] - \mathbb{E}_{q_k(\mathbf{w}_k)} \sum_{k=1}^{K} \log \left|\det \frac{\partial f_k}{\partial \mathbf{w}_{k-1}}\right|. \end{aligned} \tag{3.4}$$

Here the dependence on the parameter vector θ has been omitted for clarity of notation, $\mathbf{w}$ denotes the weights for any given layer of our BNN, and different layers are assumed independent of each other. If the biases are also transformed with normalizing flows, the same formula can be applied to them, since they are assumed independent of the weights. Consequently, we note the similarity to the lower bound (2.5) used with regular BNNs. The added computational overhead on each forward pass of the neural network is thus computing the log determinant of the Jacobian, and transforming $\mathbf{w}_0$ to $\mathbf{w}_k$ (computing $\log q_0(\mathbf{w}_0)$ is the same in both cases).

3.3 Types of transformations

The following is a short description of some common transformations used in normalizing flows. By normalizing flow, we refer to the chain of the K transformations that are applied to the initial variable, $\mathbf{z}_0$.

Planar transformation: Introduced by Rezende and Mohamed (2015), this transformation has the form,

$$\mathbf{z}_1 = \mathbf{z}_0 + \boldsymbol{u}h(\mathbf{v}^T\mathbf{z}_0 + \beta),$$

where h is a non-linear elementwise function, $\mathbf{v}$ and $\boldsymbol{u}$ are trainable parameters with the same dimensionality as $\mathbf{z}$, and β is a trainable scalar. This is akin to a layer with a single neuron. As noted in Kingma et al. (2016), a long chain of these transformations would be required to capture high dimensional correlations. This will quickly get expensive, due to the size of $\mathbf{v}$ and $\boldsymbol{u}$.

Radial transformation: Consider a transformation on the following form,

$$\mathbf{z}_1 = \mathbf{z}_0 + \frac{\beta}{\alpha + |\mathbf{z}_0 - \boldsymbol{u}_0|}(\mathbf{z}_0 - \boldsymbol{u}_0);$$

where, α and β are trainable scalars, and $\boldsymbol{u}_0$ is a trainable parameter with the same dimensionality as $\mathbf{z}$. It transforms the initial density around the point $\boldsymbol{u}_0$, hence the name radial transform. Again, we would need many of these transformations, since $\mathbf{z}$ is high dimensional, but in this case there is only one parameter with the same dimensionality as $\mathbf{z}$. Both the planar and the radial transformation have a determinant that can be computed in linear time (see Rezende and Mohamed (2015) for details).

Sylvester transformation: This transformation, introduced by van den Berg et al. (2019) can be considered a generalization of the planar transformation, where instead of having one layer with a single neuron, we have one layer with M neurons:

$$\mathbf{z}_1 = \mathbf{z}_0 + \boldsymbol{A}h(\boldsymbol{B}\mathbf{z}_0 + \mathbf{b}),$$

where h is a non-linear elementwise function as before, with $\mathbf{w} \in \mathbb{R}^D$, $\mathbf{b} \in \mathbb{R}^M$, $\boldsymbol{A} \in \mathbb{R}^{D\times M}$ and $\boldsymbol{B} \in \mathbb{R}^{M\times D}$. This transformation is therefore more expressive than the planar transformation, but it comes at the cost of having many more parameters, since $\boldsymbol{A}$ and $\boldsymbol{B}$ are now matrices.

Householder transformation: In linear algebra, the Householder transformation is a reflection around a hyperplane,

$$\mathbf{z}_1 = \left(\mathbb{I} - 2\frac{\mathbf{v}\mathbf{v}^{\boldsymbol{T}}}{||\mathbf{v}||^{\mathbf{2}}}\right)\mathbf{z}_0$$

with $\mathbf{v}$ being the reflection vector and $\mathbb{I}$ the identity matrix. In Tomczak and Welling (2017), a normalizing flow using a series of these transformations was used to transform the posterior of a variational auto-encoder. For each transformation we will need an additional parameter vector, $\mathbf{v}$, with the same shape as $\mathbf{z}$. An advantage with using this transformation is that the Householder matrix, $\mathbb{I} - 2\frac{\mathbf{v}\mathbf{v}^{\boldsymbol{T}}}{||\mathbf{v}||^{\mathbf{2}}}$, is orthogonal, and therefore the log-determinant of the Jacobian is 0. This is beneficial, as we can now ignore the Jacobian and reduce computation time. On the negative side, this is a *linear* transformation, so it does not have the same expressivity as the non-linear transformations above. Since a linear

transformation of a Gaussian distribution is still Gaussian, starting with $\mathbf{z}_0$ being Gaussian distributed, as we typically do in variational inference, the most complex distribution we can obtain is a multivariate Gaussian distribution with some correlation structure. Instead of using a sequence of only Householder transformations, we could instead alternate with one (or more) of the nonlinear ones mentioned above.

Coupling transformation: To make the notation easier to read, we will denote the initial random variable as $\mathbf{x}$, and the transformed variable as $\mathbf{z}$ (instead of $\mathbf{z}_0$ and $\mathbf{z}_1$). We have $\mathbf{x} \in \mathbb{R}^N$, and then partition it into two disjoint sets, $\mathbf{x}_{1:n}$ and $\mathbf{x}_{n+1:N}$. The basic idea of the coupling transformation is to let the first n dimensions remain the same, and transform the second part as a function of the first part. In NICE (non-linear independent components estimation)(Dinh et al. 2015), the *additive* coupling layer is used, which gives the following transformation

$$\begin{aligned}\mathbf{z}_{1:n} &= \mathbf{x}_{1:n}\\ \mathbf{z}_{n+1:N} &= \mathbf{x}_{n+1:N} + t(\mathbf{x}_{1:n}),\end{aligned}$$

where $t : \mathbb{R}^n \to \mathbb{R}^{N-n}$. To compute the determinant, we note that for the upper left square of the Jacobian, we have

$$\frac{\partial \mathbf{z}_{1:n}}{\partial \mathbf{x}_{1:n}} = \mathbb{I},$$

i.e. the identity matrix. For the bottom right square, we also have the identity matrix since

$$\frac{\partial \mathbf{z}_{n+1:N}}{\partial \mathbf{x}_{n+1:N}} = \mathbb{I}.$$

Finally,

$$\frac{\partial \mathbf{z}_{1:n}}{\partial \mathbf{x}_{n+1:N}} = \mathbf{0}$$

and therefore the Jacobian is triangular with determinant equal to 1, which means the log determinant will be 0. This transformation is thus volume preserving, like the Householder transformation discussed earlier. Since the Jacobian does not depend on $t(\cdot)$, it allows for it to be arbitrarily complex; typically a neural network is used. As one transformation leaves the n first elements of $\mathbf{x}$ unchanged, it is necessary to chain several of these transformations together (and exchange the order), for all of the elements to interact with each other. Again, there is a trade-off between complexity and computational resources. Having a flow with many neural networks chained together means slower computation time, in addition to all the flow parameters being included in the variational lower bound.

RealNVP, (Dinh et al. 2017) is an extention of NICE, with both a scale and shift transformation,

$$\begin{aligned}\mathbf{z}_{1:n} &= \mathbf{x}_{1:n}\\ \mathbf{z}_{n+1:N} &= \mathbf{x}_{n+1:N} \odot \exp\left(s(\mathbf{x}_{1:n})\right) + t(\mathbf{x}_{1:n}).\end{aligned}$$

The Jacobian of this transformation is

$$\boldsymbol{J} = \begin{bmatrix} \mathbb{1} & 0 \\ \frac{\partial \mathbf{y}_{n+1:N}}{\partial \mathbf{x}_{1:n}} & \operatorname{diag}(\exp\left(s(\mathbf{x}_{1:n})\right)), \end{bmatrix} \tag{3.5}$$

where $\odot$ denotes the elementwise product between two vectors. The determinant is easy to compute,

$$\det(\boldsymbol{J}) = \exp \sum_{i=n+1}^{N} s(\mathbf{x}_{1:n})_i.$$

Again, this determinant does not depend on the Jacobian of s and t, and therefore they can be modelled by neural networks. Furthermore, the partitioning of $\mathbf{x}$ is done using a binary mask, where a mask is defined as a binary vector that is multiplied elementwise with the input, $\mathbf{x}$, such that $\mathbf{x}$ is split into two parts. In Dinh et al. (2017) the purpose of this is to design masks that are able to capture local correlation structures in images, and alternating different patterns so that all the elements of $\mathbf{x}$ get changed. In our setting, where we do not have any prior knowledge about the correlation structure in our neural network weights, it makes more sense to do as Louizos and Welling (2017) and use a *random* mask that is re-sampled on each forward pass, as this allows for all the elements of $\mathbf{x}$ to interact with each other quickly. In contrast, using a pre-defined constant mask for each transformation would mean that many more transformations would be needed to get the same effect. Finally, using the numerically stable updates from Kingma et al. (2016), we can let the scale and shift parameters both be outputs from the same neural network,

$$\begin{aligned} \mathbf{m} &\sim \text{Bernoulli}(p) \\ \boldsymbol{\mu}, \boldsymbol{s} &= \text{NeuralNetwork}(\mathbf{m} \odot \mathbf{x}) \\ \boldsymbol{\sigma} &= \text{sigmoid}(\boldsymbol{s}) \\ \mathbf{z} &= (1 - \mathbf{m}) \odot \mathbf{x} \odot \boldsymbol{\sigma} + (1 - \boldsymbol{\sigma}) \odot \boldsymbol{\mu} + \mathbf{m} \odot \mathbf{x}, \end{aligned} \tag{3.6}$$

where $\boldsymbol{s}$ and $\boldsymbol{\mu}$ are the shift and scale parameters, $\mathbf{m}$ is the binary mask and p is a hyperparameter. In addition, $\odot$ denotes the elementwise product. Note that the factor $(1 - \boldsymbol{\sigma}) \odot \boldsymbol{\mu}$ is included for *both* parts of the output vector $\mathbf{z}$, whereas in Louizos and Welling (2017), (see equation 6) it is only included for the part where $\mathbf{m} = 0$. During experimentation, the latter version performed worse, so in this thesis we will be using (3.6), with the implementation from Riebesell (2020). To compute the Jacobian, we note that since the elements of $\mathbf{m}$ are either 0 or 1, we get the following two possibilities, $\frac{\partial \mathbf{z}}{\partial \mathbf{x}}$,

$$\begin{aligned} m_i = 0 &\Rightarrow \frac{\partial z_i}{\partial x_i} = \sigma_i \\ m_i = 1 &\Rightarrow \frac{\partial z_i}{\partial x_i} = 1, \end{aligned}$$

meaning that the Jacobian will be diagonal, with the entries on the diagonal 1 if $m_i = 1$, and σ_i if $m_i = 0$, which leads to the following log-determinant,

$$\log \left| \frac{\partial \mathbf{z}}{\partial \mathbf{x}} \right| = \sum_{i=1}^{N} (1 - m_i) \log \sigma_i. \tag{3.7}$$

4 The latent binary Bayesian neural network (LBBNN)

In this chapter, we will start with a brief description of the LBBNN method, which was introduced in Hubin and Storvik (2019), and thereafter we will discuss the two extensions. The core idea behind this method is that it tries to incorporate uncertainty in both parameters and model structure. By multiplying each weight with a binary variable, we can turn each weight on or off. The set of weights that are turned on will then be considered a particular model structure. Uncertainty around the weights is incorporated as usual with a prior distribution, and we also use a prior distribution on the binary variables. Conditional on the inclusion variable, we have a Gaussian prior on the weights w and a Bernoulli prior on the inclusion variable γ,

$$
\begin{aligned}
p(w|\gamma) &= \gamma \frac{1}{\sqrt{2\pi\sigma_p^2}} \exp\left(-\frac{(w-\mu_p)^2}{2\sigma_p^2}\right) + \delta(w)(1-\gamma) \\
p(\gamma) &= \alpha_p^{\gamma}(1-\alpha_p)^{1-\gamma}.
\end{aligned}
$$

This is called the spike-and-slab prior, most commonly used in Bayesian linear regression models. With probability α_p, the prior is a Gaussian distribution (slab) with parameters μ_p and σ_p, and with probability $1-\alpha_p$, the prior is a (small) delta spike. In Hubin and Storvik (2019), it is shown that this induces a high degree of sparsity, while maintaining good predictive accuracy. In addition, there is an inverse-gamma hyper-prior on σ_p^2, and a beta hyper-prior on α_p. Using empirical Bayes, these hyper-priors are used to find the values of σ_p^2 and α_p.

For the variational posterior distribution, the same form is used,

$$
\begin{aligned}
p(w|\gamma) &= \gamma \frac{1}{\sqrt{2\pi\sigma_q^2}} \exp\left(-\frac{(w-\mu_q)^2}{2\sigma_q^2}\right) + \delta(w)(1-\gamma) \\
p(\gamma) &= \alpha_q^{\gamma}(1-\alpha_q)^{1-\gamma},
\end{aligned}
$$

and we use the subscript q to denote a parameter from the variational distribution, in addition to p for a parameter from the prior. Hubin and Storvik (2019) compare methods with different choices of priors and posteriors, but here we will only compare results against the one where they use a Gaussian prior (GP), and mean field (MF) variational posterior. We will therefore refer to this as LBBNN-GP-MF. Compared to standard BNNs, this method has an added computational burden, due to the extra γ matrix. In order to use the reparametrization trick and compute the gradient of the binary variables, Hubin and Storvik (2019) use the concrete distribution (Maddison et al. 2016), which is a continuous approximation of the Bernoulli distribution. Instead of sampling $\gamma \sim \text{Bernoulli}(\alpha)$, we instead sample $\nu \sim \text{Uniform}[0,1]$ and, using the reparametrization trick we obtain the continuous approximation,

$$
\widehat{\gamma} = \text{sigmoid}\left[\frac{\text{logit}(\alpha) - \text{logit}(\nu)}{\delta}\right], \tag{4.1}
$$

with δ a hyperparameter that is chosen to be a small value. When $\delta \to 0$, $\widehat{\gamma}$ approaches a Bernoulli random variable. This is because $\text{sigmoid}(x) \to 0$ when

$x \to -\infty$ and sigmoid$(x) \to 1$ when $x \to +\infty$, and the sign is determined by the sign in the numerator in (4.1).

4.1 LBBNN-GP-MF with the local reparametrization trick (LRT)

In this section we will discuss how to combine the LBBNN-GP-MF method with the local reparametrization trick. This is not to be confused with the *reparametrization trick*, which as mentioned earlier, is a way to estimate the gradient of the ELBO using samples from a noise distribution instead of directly from the variational posterior. The local reparametrization trick refers to the idea that if one has an independent Gaussian posterior on the weights, then the sum of these weights will also follow a Gaussian distribution, which allows us to sample the activations directly rather than sampling the weight and gamma matrices. This is advantageous, since the activations are of much lower dimension, so we should expect a big computational gain. Secondly, as shown in Kingma et al. (2015), using the LRT reduces variance in the gradient estimator, and could therefore lead to faster convergence of the optimization. Thirdly, since the binary parameters are never sampled directly, there is no need to use the concrete distribution to approximate the Bernoulli distribution. It is however not certain how much (if anything) is lost by using this approximation. Bai et al. (2020) that follows Hubin and Storvik (2019), argues that there is little difference when applied to a toy example in a linear regression model.

We will now detail how to apply the LRT to the LBBNN-GP-MF method. Since the layers are assumed to be independent, we drop the layer subscript here for ease of notation. Assume we have an input vector, $\boldsymbol{x}$ from the previous layer with N neurons. Here we denote b_j as the activation *before* the non-linear elementwise activation function,

$$b_j = \sum_{i=1}^{N} x_i W_{ij} + c_j.$$

Furthermore, we can find the marginal density of any (independent) weight as,

$$\begin{aligned}
q(w) &= \sum_{\gamma} q(w, \gamma) \\
&= \sum_{\gamma} q(w|\gamma) q(\gamma) \\
&= \sum_{\gamma} \left[\gamma \frac{1}{\sqrt{2\pi\sigma_q^2}} \exp\left(-\frac{(w-\mu_q)^2}{2\sigma_q^2} \right) + \delta(w)(1-\gamma) \right] \alpha_q^{\gamma} (1-\alpha_q)^{1-\gamma} \\
&= \delta(w)(1-\alpha_q) + \alpha_q \frac{1}{\sqrt{2\pi\sigma_q^2}} \exp\left(-\frac{(w-\mu_q)^2}{2\sigma_q^2} \right),
\end{aligned}$$

where the subscript q denotes that the parameter is from the variational posterior. We then compute the expectation and variance of w as,

$$\begin{aligned}
\mathbb{E}(w) &= \alpha_q \mu_q \\
\text{Var}(w) &= \alpha_q^2 \sigma_q^2.
\end{aligned}$$

With this, it is straightforward to compute the expectation and the variance of the activation, b_j (before the non-linearity) as,

$$\mathbb{E}(b_j) = \mathbb{E}\left[\sum_{i=1}^{N} x_i W_{ij} + c_j\right] = \sum_{i=1}^{N} x_i A_{ij} M_{ij} + \mu_{c_j}$$
$$\text{Var}(b_j) = \text{Var}\left[\sum_{i=1}^{N} x_i W_{ij} + c_j\right] = \sum_{i=1}^{N} x_i^2 A_{ij}^2 \Sigma_{ij}^2 + \sigma_{c_j}^2,$$

where A_{ij}, M_{ij} and Σ_{ij} denote the matrices of the variational parameters α_q, μ_q and σ_q respectively, and squaring is done elementwise. In addition, μ_{c_j} and $\sigma_{c_j}^2$ denote the mean and variance of the bias parameter, c_j. We can now sample the activation directly as,

$$\begin{aligned} b_j &= \sum_{i=1}^{N} x_i A_{ij} M_{ij} + \mu_{c_j} + \sqrt{\sum_{i=1}^{N} x_i^2 A_{ij}^2 \Sigma_{ij}^2 + \sigma_{c_j}^2} * \epsilon, \\ \epsilon &\sim \mathcal{N}(0, 1). \end{aligned} \tag{4.2}$$

These computations can be vectorized, and with a minibatch of inputs $\mathbf{X}$ we get

$$\mathbf{B} = \mathbf{X}(\mathbf{A} \odot \mathbf{M}) + \boldsymbol{\mu}_c + \sqrt{\mathbf{X}^2(\mathbf{A}^2 \odot \boldsymbol{\Sigma}^2) + \boldsymbol{\sigma}_c^2} \odot \mathbf{E}, \tag{4.3}$$

with $\mathbf{E}$ sampled from the independent standard normal distribution. Notice that this matrix has the same shape as the activation matrix $\mathbf{B}$, and therefore we have independent noise across the activations. In the case when we do *not* use the LRT, (i.e. sample the weights instead), the noise matrix has the same shape as the weight matrix, meaning that each datapoint in the minibatch will use the same noise matrix, so we have correlated noise. This is why it is argued in Kingma et al. (2015) that using the LRT reduces variance in the gradient estimator.

As mentioned previously, to use the LRT it is necessary to have an analytical expression for the KL-divergence between the variational posterior and the prior, $\text{KL}\left[q_\theta(\mathbf{w}) || p(\mathbf{w})\right]$. We will therefore only consider the LBNNN-GP-MF method *without* hyper-priors, and again assume independence between different layers and weights, we get the joint prior, and joint variational density between w and γ

$$\begin{aligned} p(w, \gamma) &= p(w|\gamma)p(\gamma) \\ &= \left[\gamma \frac{1}{\sqrt{2\pi\sigma_p}} \exp(-\frac{(w-\mu_p)^2}{2\sigma_p^2}) + \delta(w)(1-\gamma)\right] \alpha_p^\gamma (1-\alpha_p)^{1-\gamma} \end{aligned}$$

$$\begin{aligned} q(w, \gamma) &= q(w|\gamma)q(\gamma) \\ &= \left[\gamma \frac{1}{\sqrt{2\pi\sigma_q}} \exp(-\frac{(w-\mu_q)^2}{2\sigma_q^2}) + \delta(w)(1-\gamma)\right] \alpha_q^\gamma (1-\alpha_q)^{1-\gamma}. \end{aligned}$$

We can then proceed with deriving the analytical expression for the KL-divergence,

$$\text{KL}\left[q(w,\gamma)||p(w,\gamma)\right] = \int\limits_w \sum_\gamma q(w,\gamma)\log\frac{q(w,\gamma)}{p(w,\gamma)}dw$$
$$= \int\limits_w \sum_\gamma q(w,\gamma)\log q(w,\gamma)dw - \int\limits_w \sum_\gamma q(w,\gamma)\log p(w,\gamma)dw.$$

We compute these two terms separately. For the first one we have,

$$\int\limits_w \sum_\gamma q(w,\gamma)\log q(w,\gamma)dw = \int\limits_w \delta(w)(1-\alpha_q)\left[\log(\delta(w)) + \log(1-\alpha_q)\right]dw$$
$$+ \int\limits_w \frac{1}{\sqrt{2\pi\sigma_q^2}}\exp\left(-\frac{(w-\mu_q)^2}{2\sigma_q^2}\right)\alpha_q\left(-\frac{1}{2}\log(2\pi\sigma_q^2) - \frac{(w-\mu_q)^2}{2\sigma_q^2} + \log(\alpha_q)\right)dw$$
$$= \int\limits_w \delta(w)(1-\alpha_q)\left[\log(\delta(w)) + \log(1-\alpha_q)\right]dw$$
$$+ \alpha_q\left(-\frac{1}{2}\log(2\pi\sigma_q^2) - \frac{1}{2} + \log(\alpha_q)\right).$$

Similarly, for the second term (skipping the intermediate step) we obtain,

$$\int\limits_w \sum_\gamma q(w,\gamma)\log p(w,\gamma)dw = \int\limits_w \delta(w)(1-\alpha_q)\left[\log(\delta(w)) + \log(1-\alpha_p)\right]dw$$
$$+ \alpha_q\left(-\frac{1}{2}\log(2\pi\sigma_p^2) - \frac{\sigma_q^2 + (\mu_q-\mu_p)^2}{2\sigma_p^2} + \log(\alpha_p)\right).$$

By combining these two terms, letting $\int\limits_w \delta(w)dw = 1$, and simplifying, we arrive at the following expression for the KL-divergence, which is equivalent to the bound proposed in Bai et al. (2020).

$$\begin{aligned}\text{KL}\left[q(w,\gamma)||p(w,\gamma)\right] = \alpha_q\left(\log\frac{\sigma_p}{\sigma_q} + \log\frac{\alpha_q}{\alpha_p} - \frac{1}{2} + \frac{\sigma_q^2 + (\mu_q-\mu_p)^2}{2\sigma_p^2}\right)\\ + (1-\alpha_q)\log\frac{1-\alpha_q}{1-\alpha_p}\end{aligned} \tag{4.4}$$

For the biases, we assume they are independent of each other and of the weights. We then get the following expression for the KL-divergence between the prior and the variational posterior, that are both Gaussian,

$$\text{KL}\left[q(c)||p(c)\right] = \log\frac{\sigma_{c_p}}{\sigma_{c_q}} - \frac{1}{2} + \frac{\sigma_{c_q}^2 + (\mu_{c_q}-\mu_{c_p})^2}{2\sigma_{c_p}^2}. \tag{4.5}$$

Notice that (4.4) reduces to the KL-divergence between two Gaussians if $\alpha_p = \alpha_q = 1$, i.e. a regular BNN without sparsity.

4.2 Multiplicative normalizing flows (MNF)

As discussed earlier, using normalizing flows to transform the variational posterior distribution is fairly straightforward, but it comes at the cost of increasing the number of parameters and slowing down computations. Since the LBBNN-GP-MF method already has an extra parameter per weight compared to standard BNNs, it would get very costly to further increase this. Therefore, it is beneficial to apply the flows in the space of the activations instead, which was first done in Louizos and Welling (2017). It was shown that this improved significantly on both predictive accuracy, and predictive uncertainty (although the way this is measured is questionable, as we will discuss in the chapter on experiments), compared to just using the mean field Gaussian variational posterior. MNFs have later been applied with success in the domain of deep reinforcement learning (Touati et al. 2020).

Following is a summary of how MNF work, before we will discuss how to combine MNFs with the LBBNN-GP-MF method. In Louizos and Welling (2017), the variational posterior distribution is defined in the following way,

$$q(\mathbf{W}|\mathbf{z}) = \prod_i \prod_j \mathcal{N}(z_i \mu_{ij}, \sigma_{ij}^2),$$

where the weights are conditioned on an additional density, $q(\mathbf{z})$. In this section we will use $\mathbf{W}$ to denote the weight matrix, and lower case $\mathbf{z}$ to denote the vector $\mathbf{z}$. By applying the transformations to $q(\mathbf{z})$ we can increase the flexibility of the variational posterior, while maintaining the benefits of sampling the activations. In the original paper, $q(\mathbf{z}_0)$ is initialized as independent Gaussian, (using the reparametrization trick), and the transformed variable $\mathbf{z}_k$ is obtained using Equation (3.2). The activations, using similar notation as in (4.3) are then given by

$$\mathbf{B} = (\mathbf{X} \odot \mathbf{z}_k)\mathbf{M} + \boldsymbol{\mu}_c + \sqrt{\mathbf{X}^2 \boldsymbol{\Sigma}^{2^T} + \boldsymbol{\sigma}_c^2} \odot \mathbf{E}, \tag{4.6}$$

where $\mathbf{X} \odot \mathbf{z}_k$ denotes that we do elementwise multiplication between each sample in the minibatch, $\mathbf{X}$ (i.e. each row vector) with the vector $\mathbf{z}_k$. Finding the expression for the lower bound, $\mathrm{KL}\left[q(\boldsymbol{W})||p(\boldsymbol{W})\right]$ is not straightforward, since $q(\boldsymbol{W}) = \int q(\boldsymbol{W}|\boldsymbol{z})q(\boldsymbol{z})d\boldsymbol{z}$ and thus intractable. However, by using Bayes formula, we get

$$q(\boldsymbol{W}) = \frac{q(\boldsymbol{W}|\mathbf{z})q(\mathbf{z})}{q(\mathbf{z}|\boldsymbol{W})},$$

which implies that

$$\log q(\boldsymbol{W}) = \log q(\boldsymbol{W}|\mathbf{z}) + \log q(\mathbf{z}) - \log q(\mathbf{z}|\boldsymbol{W}).$$

We can then get to the bound proposed by Louizos and Welling (2017):

$$\begin{aligned} \mathrm{KL}\left[q(\boldsymbol{W})||p(\boldsymbol{W})\right] &= \mathbb{E}_{q(\boldsymbol{W},\mathbf{z}_k)}[\log q(\boldsymbol{W}|\mathbf{z}_k) + \log q(\mathbf{z}_k) \\ &\quad - \log q(\mathbf{z}_k|\boldsymbol{W}) - \log p(\boldsymbol{W})] \\ &= \mathbb{E}_{q(\boldsymbol{W},\mathbf{z}_k)}[\mathrm{KL}[(\boldsymbol{W}|\mathbf{z}_k)||p(\boldsymbol{W})] \\ &\quad + \log q(\mathbf{z}_k) - \log q(\mathbf{z}_k|\boldsymbol{W})] \end{aligned} \tag{4.7}$$

The last term, $\log q(\mathbf{z}_k|\boldsymbol{W})$, is approximated with an auxiliary distribution, $\log r(\mathbf{z}_k|\boldsymbol{W})$. In order to make r flexible, and closer to the true distribution, it is transformed with an inverse normalizing flow, and then the following approximation is used,

$$\log r\left(\mathbf{z}_k|\boldsymbol{W}\right) = \log r\left(\mathbf{z}_b|\boldsymbol{W}\right) + \sum_{t=k+1}^{b} \log \left|\det \frac{\partial f_t}{\partial \mathbf{z}_{t-1}}\right|. \tag{4.8}$$

With inverse flow, we mean that $\mathbf{z}_k = \mathrm{NF}^{-1}(\mathbf{z}_b) \implies \mathbf{z}_b = \mathrm{NF}(\mathbf{z}_k)$. In other words, we need two normalizing flows, one to get from $\mathbf{z}_0$ to $\mathbf{z}_k$, and another from $\mathbf{z}_k$ to $\mathbf{z}_b$. Furthermore, $\log q(\mathbf{z}_k)$ can be computed using Equation (3.3), and by letting $p(\boldsymbol{W})$ be independent Gaussian, $\mathrm{KL}\left[q(\boldsymbol{W}|\mathbf{z}_k)||p(\boldsymbol{W})\right]$ is straightforward to compute, since it will just be the KL-divergence between two Gaussian distributions.

4.3 Combining LBBNN-GP-MF with MNF

Now that we have established how to use the LBBNN-GP-MF method with the local reparametrization trick, in addition to detailing how MNFs work, we will discuss how to combine these. For the variational posterior distribution, we will consider the following form,

$$q(\mathbf{W}|\boldsymbol{\Gamma}, \mathbf{z}) = \prod_i \prod_j \gamma_{ij} \mathcal{N}(z_i \mu_{ij}, \sigma_{ij}^2) + \delta(\mathbf{W})(1 - \gamma_{ij}), \tag{4.9}$$

where we have used $\boldsymbol{\Gamma}$ to denote the matrix of the inclusion variables, γ_{ij}, and we have the activations sampled as

$$\mathbf{B} = (\mathbf{X} \odot \mathbf{z}_k)(\mathbf{A} \odot \mathbf{M}) + \boldsymbol{\mu}_c + \sqrt{\mathbf{X}^2(\mathbf{A}^2 \odot \boldsymbol{\Sigma}^2) + \boldsymbol{\sigma}_c^2} \odot \mathbf{E}. \tag{4.10}$$

The variance term is the same as in (4.3), since $\mathbf{z}$ only affects the mean of the weights. In Louizos and Welling (2017), it is argued that letting it affect both the mean and the variance leads to the lower bound getting stuck in local optima, due to high variance gradients.

To compute the KL-divergence between $q(\boldsymbol{W}, \boldsymbol{\Gamma})$ and $p(\boldsymbol{W}, \boldsymbol{\Gamma})$, we use the same approach as before by noting that

$$q(\boldsymbol{W}, \boldsymbol{\Gamma}) = \frac{q(\boldsymbol{W}, \boldsymbol{\Gamma}|\mathbf{z})q(\mathbf{z})}{q(\mathbf{z}|\boldsymbol{W}, \boldsymbol{\Gamma})},$$

implying that

$$\log q(\boldsymbol{W}, \boldsymbol{\Gamma}) = \log q(\boldsymbol{W}, \boldsymbol{\Gamma}|\mathbf{z}) + \log q(\mathbf{z}) - \log q(\mathbf{z}|\boldsymbol{W}, \boldsymbol{\Gamma}).$$

Now we get the following for the KL-divergence between the variational posterior and the prior,

$$\begin{aligned}\mathrm{KL}\left[q(\boldsymbol{W}, \boldsymbol{\Gamma})||p(\boldsymbol{W}, \boldsymbol{\Gamma})\right] = \mathbb{E}_{q(\boldsymbol{W}, \boldsymbol{\Gamma}, \mathbf{z}_k)}[&\mathrm{KL}\left[q(\boldsymbol{W}, \boldsymbol{\Gamma}|\mathbf{z}_k)||p(\boldsymbol{W}, \boldsymbol{\Gamma})\right] \\ &+ \log q(\mathbf{z}_k) - \log q(\mathbf{z}_k|\boldsymbol{W}, \boldsymbol{\Gamma})],\end{aligned} \tag{4.11}$$

where the first term is computed as follows,

$$\begin{aligned}\mathrm{KL}\left[q(\boldsymbol{W},\boldsymbol{\Gamma}|\mathbf{z}_k)||p(\boldsymbol{W},\boldsymbol{\Gamma})\right] = \sum_{ij}\alpha_{q_{ij}}\Bigg(&\log\frac{\sigma_{p_{ij}}}{\sigma_{q_{ij}}}+\log\frac{\alpha_{q_{ij}}}{\alpha_{p_{ij}}}\\ &-\frac{1}{2}+\frac{\sigma^2_{q_{ij}}+(\mu_{q_{ij}}z_{k_i}-\mu_{p_{ij}})^2}{2\sigma^2_{p_{ij}}}\Bigg)\\ &+(1-\alpha_{q_{ij}})\log\frac{1-\alpha_{q_{ij}}}{1-\alpha_{p_{ij}}},\end{aligned}$$

which is very similar to how we computed the KL-divergence in equation (4.4). For the other two terms, we compute $\log q(\mathbf{z}_k)$ using equation (3.3), where again we use the independent Gaussian distribution for $q_0(\mathbf{z}_0)$. The last term, $\log q(\mathbf{z}_k|\boldsymbol{W},\boldsymbol{\Gamma})$ is approximated similarly as in (4.8) using an auxiliary distribution $\log r(\mathbf{z}_k|\boldsymbol{W},\boldsymbol{\Gamma})$. For the starting point of the inverse flow, we follow Louizos and Welling (2017) and use

$$r(\mathbf{z}_b|\boldsymbol{W},\boldsymbol{\Gamma}) = \prod_{i=1}^{D}\mathcal{N}(\tilde{\mu}_i,\tilde{\sigma}_i^2),$$

where D is the dimensionality of $\mathbf{z}$. The dependence on $\boldsymbol{W}$ and $\boldsymbol{\Gamma}$ is defined the same way as in Louizos and Welling (2017), with

$$\begin{aligned}\tilde{\boldsymbol{\mu}} &= \left(\mathbf{d}_1\otimes\tanh\left(\mathbf{e}^T\boldsymbol{W}\right)\right)\left(\mathbf{1}\odot D_{\text{out}}^{-1}\right)\\ \log\tilde{\boldsymbol{\sigma}}^{\mathbf{2}} &= \left(\mathbf{d}_2\otimes\tanh\left(\mathbf{e}^T\boldsymbol{W}\right)\right)\left(\mathbf{1}\odot D_{\text{out}}^{-1}\right),\end{aligned}\tag{4.12}$$

where $\mathbf{d}_1$, $\mathbf{d}_2$ and $\mathbf{e}$ are trainable parameters with the same size as $\mathbf{z}$, and $\otimes$ denotes the outer product between two vectors, which results in a matrix. We use $\mathbf{1}\odot D_{\text{out}}^{-1}$ to denote that we compute the mean of this matrix across its rows, such that the result is a vector with the same size as $\mathbf{z}$. To avoid sampling the large $\boldsymbol{W}$ and $\boldsymbol{\Gamma}$ matrices, we use the local reparametrization trick to sample the activations $\boldsymbol{b}=\mathbf{e}^T\boldsymbol{W}$ directly:

$$\boldsymbol{b} = \mathbf{e}^T\left(\mathbf{z}_k\odot\mathbf{A}\odot\mathbf{M}\right)+\sqrt{\mathbf{e}^{2T}\left(\mathbf{A}^2\odot\boldsymbol{\Sigma}^2\right)}\odot\boldsymbol{\epsilon},\tag{4.13}$$

where as before, $\mathbf{A}$, $\mathbf{M}$ and $\boldsymbol{\Sigma}$ denote the matrices of the μ_{ij}, α_{ij} and σ_{ij} parameters respectively. In addition $\boldsymbol{\epsilon}$ is sampled from $\mathcal{N}(0,1)$, and $\mathbf{z}_k = NF(\mathbf{z}_0)$. For the flows of both r and q, we use realNVP with the procedure described in equation (3.6).

5 Experiments

5.1 Background

In this section we will be comparing the three different methods considered in this thesis. We emphasise that these methods have the same underlying *model*, that is, we assume a Bayesian model with a prior distribution on the weights and the inclusion probabilities, and we assume the response y is multinomially distributed. The methods differ in that they give a different variational approximation to the true posterior distribution. The three different methods will be

denoted:

LBBNN-GP-MF: The baseline method that we will be comparing against, with priors, hyper-parameters and (variational) parameter initializations as described in Hubin and Storvik (2019) unless otherwise noted. This method uses a hyper-prior on the inclusion probabilities, and another on the variances of the weights. These are trained using empirical Bayes for the first 20 epochs, which thereafter will remain fixed.

LBBNN-GP-MF-LRT: The local reparametrization trick applied to the baseline method. See section 4.1 for more details. For the prior distributions on the weights and the biases, we use the independent standard Gaussian distribution. The prior for the inclusion probabilities are independent Bernoulli distributed, with $p = 0.05$. We initialize the variational parameters using the same procedure as in the baseline method. We do not use any hyper-priors here, and thus the optimization scheme is a little different to the baseline method, here we use the Adam optimizer with learning rate 0.001 for all the variational parameters.

LBBNN-GP-MF-MNF: The last method we consider will be the one that combines the latent binary Bayesian neural networks with multiplicative normalizing flows, as detailed in section 4.3. We use the same priors as with the LBBNN-GP-MF-LRT method. For the normalizing flows, for both $q(\mathbf{z})$ and $r(\mathbf{z}|\boldsymbol{W}, \boldsymbol{\Gamma})$, we use masked RNVP with numerically stable updates using equation (3.6). For the binary mask, we use $p = 0.5$, and re-sample the mask on each forward pass. Each coupling layer uses a fully connected neural network with four hidden layers, each with 75 neurons and leaky ReLU (Maas et al. 2013) activation functions, and we use two of these coupling layers for the normalizing flows. For the optimizer, we use Adam with learning rate 0.0001.

The code for the experiments is available on GitHub: https://github.com/LarsELund/Bayesian-Neural-Nets

5.2 Logistic regression simulation study

The key advantage of a simulation study is that it allows to carefully compare performance, and gain a deeper understanding of how different algorithms/methods work. This is possible because the process generating the data is known, and thus in the Bayesian framework, we have access to the true posterior distribution. In this section we will consider the dataset that was used in the simulation study of Hubin and Storvik (2018). It consists of a mix of 20 binary and continuous variables, with a binary outcome. More specifically, there are 6 binary variables, and 14 continuous variables. The data, $\boldsymbol{X}$, is generated in a way that there is a strong correlation between some of the variables (see Figure 5.1). For more details on exactly how the data is generated, see appendix B of Hubin and Storvik (2018). The response variable, Y, is generated as a logit

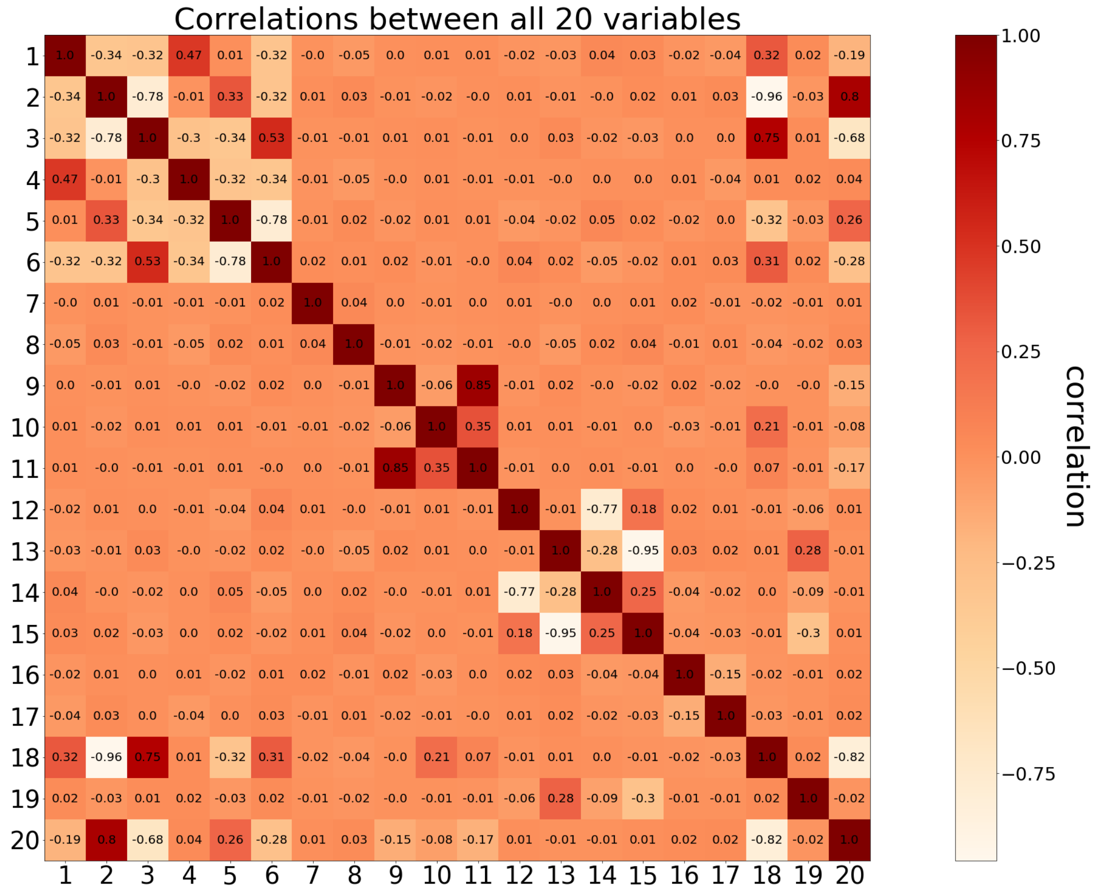

Figure 5.1: Plots showing the correlation between different variables in the logistic regression simulation study.

transformation of the linear predictor in the following way,

$$\eta \sim \mathcal{N}(\boldsymbol{\beta X}, 0.5)$$
$$Y \sim \text{Bernoulli}\left(\frac{\exp(\eta)}{1+\exp(\eta)}\right)$$

with

$$\boldsymbol{\beta} = (-4, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1.2, 0, 37.1, 0, 0, 50, -0.00005, 10, 3, 0).$$

In Hubin and Storvik (2018), the purpose of the study is to see how efficiently the mode jumping MCMC algorithm can explore the space of possible models, which is any combination of the 20 variables. Since each variable can either be included or excluded, this gives a huge space of 2^{20} possible models to explore. Note that using the variational approximation to the posterior avoids this issue, as we are not trying to sample from the exact posterior. Here, we would like to compare the method with normalizing flows (LBBNN-GP-MF-MNF), to the baseline LBBNN-GP-MF method of Hubin and Storvik (2019). To assess

performance, we will be interested in measuring how well the methods can detect the true weight parameters, $\boldsymbol{\beta}$. The weights vary in absolute value, and we have nine weights that are non-zero. We will consider the weight β_j to be included if the posterior inclusion probability $\alpha_j > 0.5$, i.e. the median probability model (Barbieri and Berger 2004). We will train 100 models, and for each model compute the true positive rate (TPR), and the false positive rate (FPR):

$$\text{TPR} = \sum_{j=1}^{M} \frac{\text{TP}_j}{\text{TP}_j + \text{FN}_j}$$
$$\text{FPR} = \sum_{j=1}^{M} \frac{\text{FP}_j}{\text{FP}_j + \text{TN}_j}.$$

Where $M = 20$ variables, and TP = true positive, meaning that the model correctly included a non-zero weight. FN = false negative, meaning that the model failed to include a non-zero weight. FP = false positive, meaning the model included a weight that was zero. TN = true negative, meaning that the model did not include a weight that was zero. Thus, TPR measures the proportion of variables with non-zero weights correctly included into the model, whereas FPR measures the proportion of variables with zero weights that were wrongly included in the model.

Since logistic regression is just a special case of a neural network with one neuron with a sigmoid activation function, it is straightforward to adjust the algorithms. We use the same normalizing flows as described in the background section, but with the coupling layers consisting of dense neural networks with five hidden layers having 50 neurons each (instead of four layers with 75 neurons). For the LBBNN-GP-MF method, it did not seem to converge with the default Adam optimizer, therefore we use stochastic gradient descent (SGD) instead. In general, both methods are very sensitive to different choices of hyper-parameters and parameter initializations. For both methods, we use a batch size of 400, as this works better than smaller ones, and train for 500 epochs.

Figure 5.2 shows a boxplot of the true positive rate and false positive rate for the 100 different runs with the two methods. In terms of TPR and FPR, the method with normalizing flow performs better, overall the mean TPR and mean FPR over the 100 runs are 0.80 and 0.13 respectively. For the LBBNN-GP-MF method, we obtain a mean TPR of 0.68 and mean FPR of 0.28. Furthermore, in figure 5.3, we look at each weight individually over 100 runs and compute the same metrics. Here we see that for the LBBNN-GP-MF-MNF method, there are many weights that are never included, and some that are always included. From the correlation plot in figure 5.1, we see that β_7 and β_8 are almost independent of all the other variables. The method with normalizing flows is able to detect this, as we see that it correctly includes β_7 and excludes β_8 in each of the 100 runs. In addition to this, we also have β_{16} and β_{17} that have some correlation with each other, but are nearly independent of all the others. Again, the method with flows correctly includes both of these in all the 100 runs. The LBBBN-GP-MF method does slightly worse in this respect. It is a little strange that β_{11} is always (wrongly) excluded in the flow method, and

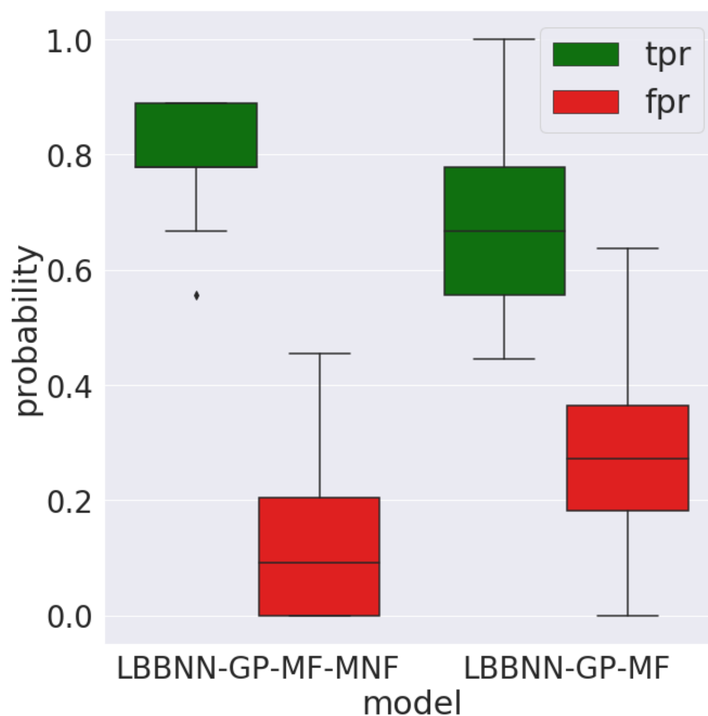

Figure 5.2: Boxplot showing the different true positive and false positive rates obtained with the two different methods, when they are trained 100 times.

rarely included in the baseline method. It is highly correlated with β_9 and β_{10}, but these are also rarely included. In general, the flow method has higher TPR and lower FPR, but it is difficult to assess how good this performance actually is without comparing directly with the full Bayesian solution, i.e. using mode jumping MCMC to sample from the true posterior.

5.3 Classification experiments

In the experiments, we will be doing classification on MNIST (Deng 2012), FMNIST (Fashion MNIST) (Xiao et al. 2017) and KMNIST (Kuzushiji MNIST) (Clanuwat et al. 2018), and compare performance of the three different methods. MNIST is a database of handwritten digits from 0 to 9. FMNIST consists of ten different fashion items from the Zalando (Europe's largest online fashion retailer) database. Lastly, KMNIST also consists of ten classes, with each one representing one row of Hiragana, a Japanese syllabary. All of these datasets contain 28x28 grayscale images, divided into a training and validation set with 60000 and 10000 images respectively. MNIST and FMNIST are well known and often utilized datasets, so it is easy to compare performance when testing novel algorithms. KMNIST is a somewhat recent addition, and is considered a more challenging task than the classic MNIST digits dataset, because each Hiragana can have many different symbols. The experiments were coded in Python, using the PyTorch deep learning library (Paszke et al. 2019).

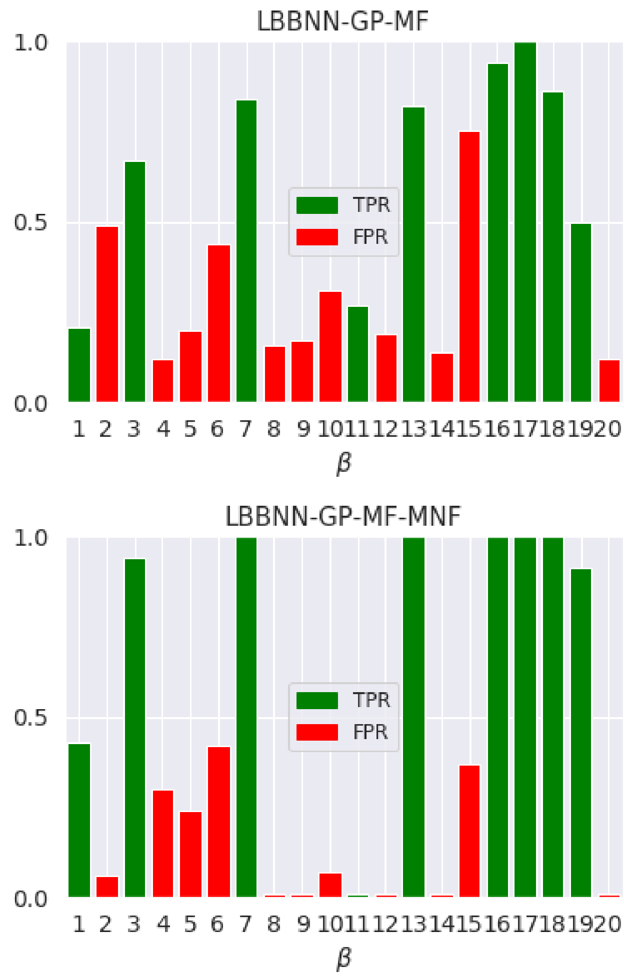

Figure 5.3: Bar plots showing true positive rate and false positive rate for the 20 different weights on the baseline method (top) and the method with normalizing flows(bottom). The bars where the true weights are non-zero are colored green, and the bars where the true weights are zero are colored red.

For the three different methods, we will be using the same structure on the network as in Hubin and Storvik (2019), i.e. a fully connected neural network with three hidden layers with 400, 400 and 600 neurons in the respective layers, and ReLU activation functions. We will also use the Adam optimizer, 250 epochs of training and a batch size of 100 for all three methods. The experiments will be run ten times, and we will consider the following performance metrics,

$$\text{Accuracy} = \frac{\text{number of correct predictions}}{\text{total number of predictions}},$$

i.e. the fraction of predictions (on the validation data) that are correct. We will follow Hubin and Storvik (2019) and report the minimum, median and maximum accuracy over the ten runs, for both the posterior mean and the ensemble accuracies. For the baseline method, we obtain the posterior mean of the weight matrix (for all the layers independently) using,

$$\mathbb{E}(w_{ij}) = \alpha_{ij}\mu_{ij}.$$

For the other two methods, we compute the expectation of the activations, $\mathbf{B}$, directly. For the LBBNN-GP-MF-LRT method, this is given as

$$\mathbb{E}(\mathbf{B}) = \mathbf{X}(\mathbf{A} \odot \mathbf{M}) + \boldsymbol{\mu}_c,$$

and for the LBBNN-GP-MF-MNF method we have

$$\mathbb{E}(\mathbf{B}) = (\mathbf{X} \odot \mathbf{z}_k)(\mathbf{A} \odot \mathbf{M}) + \boldsymbol{\mu}_c.$$

To obtain the ensemble accuracy (the fully Bayesian approach) we do the following: First we sample from the variational posterior distributions. For the baseline method, this means sampling the $\boldsymbol{W}$ and $\boldsymbol{\Gamma}$ matrices, whereas for the other two methods we sample the activations directly, using equations (4.3) and (4.10) respectively. Then we compute the probability that the samples belong to each class, for all the ten models in the ensemble, and finally use the average probability of these ten in order to make the prediction.

In addition to this, we will follow Hubin and Storvik (2019) and report the density for each of the three methods. We get an estimate for the density (of each layer) by first sampling from the variational posterior distribution,

$$\gamma_{ij} \sim \text{Bernoulli}(\alpha_{ij})$$

and computing the density as,

$$\text{density} = \frac{1}{nm}\sum_{ij}\gamma_{ij},$$

where n and m denote the number of input neurons and output neurons of the layer. We then compute an average over all the layers, and the ten runs. Lastly, we will also discuss the computation times for our three methods.

In Table 1, 2 and 3, we have the performance results of the three different methods on the three datasets. We observe that the LBBNN-GP-MF-LRT

	posterior mean accuracy			ensemble accuracy			
method	min	median	max	min	median	max	density
LBBNN-GP-MF	90.21	90.56	90.73	89.35	90.01	90.44	0.118
LBBNN-GP-MF-LRT	92.11	92.38	92.73	91.97	92.28	92.62	0.049
LBBNN-GP-MF-MNF	92.47	92.85	93.36	91.96	92.59	92.99	0.064

Table 1: Performance metrics on the KMNIST validation data, for the three different methods considered. For the accuracies (%), we report the minimum, maximum and median over the ten different runs. Density is computed as an average over the ten runs.

	posterior mean accuracy			ensemble accuracy			
method	min	median	max	min	median	max	density
LBBNN-GP-MF	98.00	98.11	98.25	97.88	98.01	98.14	0.092
LBBNN-GP-MF-LRT	98.35	98.46	98.60	98.33	98.49	98.54	0.049
LBBNN-GP-MF-MNF	98.40	98.49	98.68	98.36	98.50	98.63	0.057

Table 2: Performance metrics on MNIST, see caption in table 1 for more details.

	posterior mean accuracy			ensemble accuracy			
method	min	median	max	min	median	max	density
LBBNN-GP-MF	87.91	88.21	88.69	88.03	88.46	88.64	0.118
LBBNN-GP-MF-LRT	88.99	89.57	89.88	89.25	89.56	89.90	0.049
LBBNN-GP-MF-MNF	89.89	90.04	90.20	89.87	90.06	90.37	0.064

Table 3: Performance metrics on FMNIST, see caption in table 1 for details.

method has significantly better predictive accuracy than the baseline method across all three datasets. In fact, the worst performing instance of the LBBNN-GP-MF-LRT method is better than the best performing instance of the LBBNN-GP-MF method on all three datasets. This result is perhaps a little surprising, since these two methods are very similar. In addition, density is roughly half of the baseline method. For the method with normalizing flows, LBBNN-GP-MF-MNF, we see further improvements in both posterior mean and ensemble accuracies across all three datasets. In terms of maximum accuracy obtained from either the posterior mean or the ensemble, the flow method improves the baseline method's accuracy with 0.43%, 1.68% and 2.63% on MNIST, FMNIST and KMNIST respectively. It is also evident that the baseline method is worse in terms of minimizing the negative log-likelihood on the training data (see figure 5.8). Using the reparametrization trick and normalizing flows, the negative log-likelihood quickly approaches zero on MNIST and KMNIST, whereas the baseline method never gets that low. On FMNIST, we see a similar pattern, where the baseline method has much larger negative log-likelihood than the other two methods. When it comes to computation time, we observe during training that the LBBNN-GP-MF method takes around 20 seconds to train one epoch.

Applying the LRT makes it much quicker, here it only takes approximately six seconds. Using normalizing flows is the slowest, with around 30 seconds per epoch. As expected, increasing the complexity comes at a cost, since it takes around five times longer to train the method with flows compared to the simpler one that only uses the LRT. We use the NVIDIA RTX 2080TI GPU to train all the methods.

5.4 Uncertainty estimation

An important aspect of Bayesian statistics is that it allows us to obtain better uncertainty estimates on unseen data than using frequentist methods, that often make overconfident predictions. We will use this section to explore both in domain and out of domain uncertainty.

To assess predictive out of domain uncertainty, we classify images that the network has never seen before, i.e. classify MNIST images on a network trained on FMNIST. According to Louizos and Welling (2017), the ideal method is one that is able to manifest maximum uncertainty across the ten classes. To measure uncertainty, for each sample, we compute the entropy of the prediction vector:

$$\begin{aligned} \mathrm{E}(\hat{\mathbf{y}}) &= -\sum_{k=1}^{K} \hat{y}_k \log \hat{y}_k \\ \hat{y}_k &\in (0,1), \sum_{k=1}^{K} \hat{y}_k = 1. \end{aligned} \tag{5.1}$$

Where $K = 10$ classes. In general, the uniform distribution maximizes entropy, in this case when $\forall \hat{y}_k = 0.1$. This is the ideal case, when the method outputs complete uncertainty about its predictions. Minimum entropy is attained when the output is 1 for one of the $\hat{y}_k$, and 0 for all the others. Therefore the minimum entropy situation is when there is no uncertainty about a (wrong) prediction. For $\hat{\mathbf{y}}$, we will use the fully Bayesian model averaging approach with ten samples from the approximate posterior. We will follow Hubin and Storvik (2019) and plot the empirical cumulative distribution function of the entropies on $N = 10000$ out of distribution samples (i.e. the entire validation set). The empirical CDF is a step function that increases with 1/N for each sample. Therefore, curves that are towards the bottom right are better, since it means that a large proportion of the samples will have high entropy. In Louizos and Welling (2017), it was argued that multiplicative normalizing flows perform better than non-Bayesian alternatives such as dropout, deep ensembles and variational dropout.

This approach does however have a few problems. It violates the basic assumption that only classes seen during training can exist for new data as well (the closed world assumption) (Fei and Liu 2016). If the method is only trained for ten classes under the assumption that nothing else exists, then whatever it has to say about out of distribution data is not very meaningful. In order to properly incorporate the out of distribution data, the method should be trained to predict whether data belongs to any of the ten classes, *and* have the

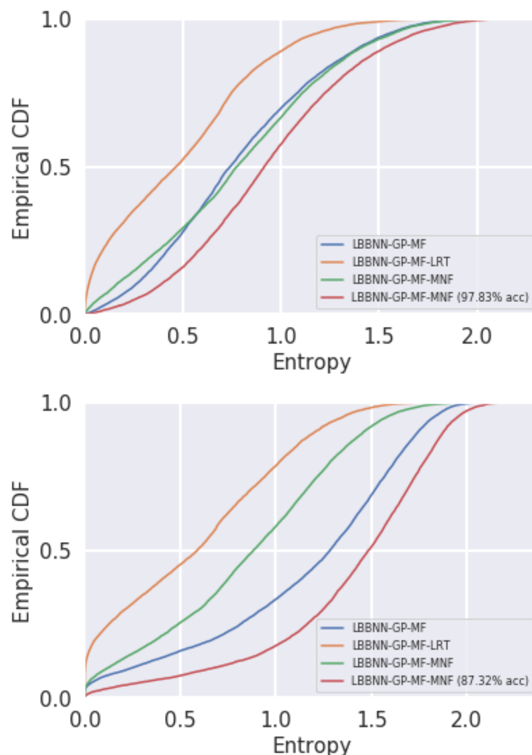

Figure 5.4: Entropy of the predictive distribution on the FMNIST validation set, with the model trained on MNIST (top), and the MNIST validation set with the model trained on FMNIST (bottom).

option to classify to an unknown class. In Figure 5.4, we plot the empirical CDF of the FMNIST validation data with the method trained on MNIST and vice versa. This highlights another issue with this approach, namely that the entropy on out of distribution data strongly depends on performance on *in distribution* data. In the figure, the green and red curves are both the LBBNN-GP-MF-MNF method, but the red curve represents an instance of the method where some hyperparameters are changed in order to induce much lower predictive accuracy on in distribution data. We observe that this leads to much higher (better) entropy on out of distribution data, implying that in general, methods that have high predictive accuracy tend to have lower entropy on out of distribution data, making it questionable whether this should be used as a metric to compare how well different methods can handle out of distribution uncertainty. Furthermore, in Hubin and Storvik (2019) where the empirical CDF is also plotted for all the methods considered in that paper, we observe a similar pattern. High accuracy on in domain data tends to imply lower entropy on out of distribution data. We also plot the entropy on in distribution data, i.e. when we use the same dataset for both the training data and the validation data (see Figure 5.5). We observe that the methods with high predictive accuracy have lower entropy. Furthermore, on MNIST, where all the methods have very high predictive accuracy, the entropies are also much lower than on

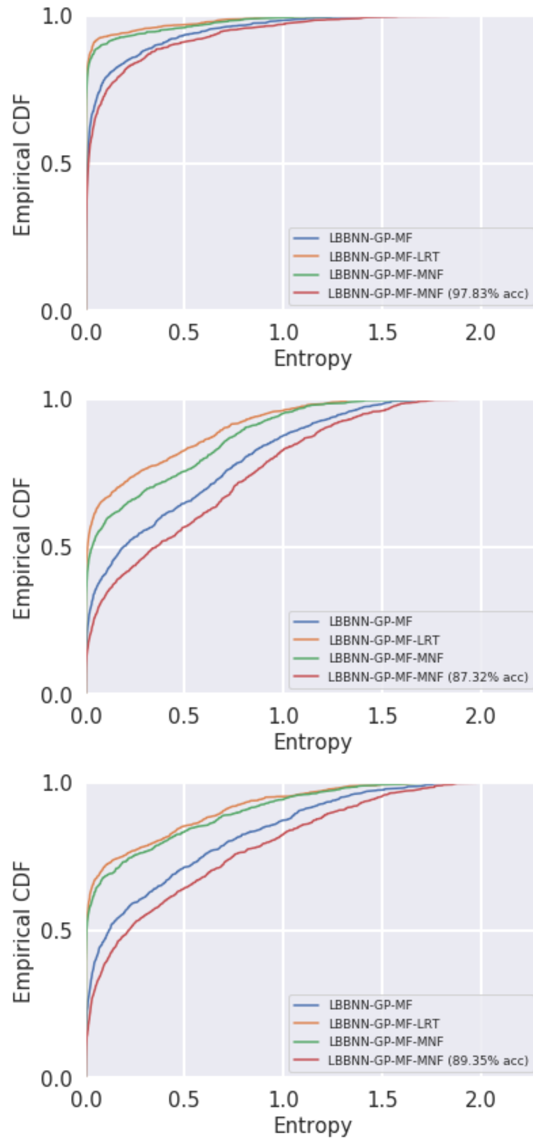

Figure 5.5: Entropy of the predictive distribution on in-distribution data on MNIST, FMNIST and KMNIST from top to bottom

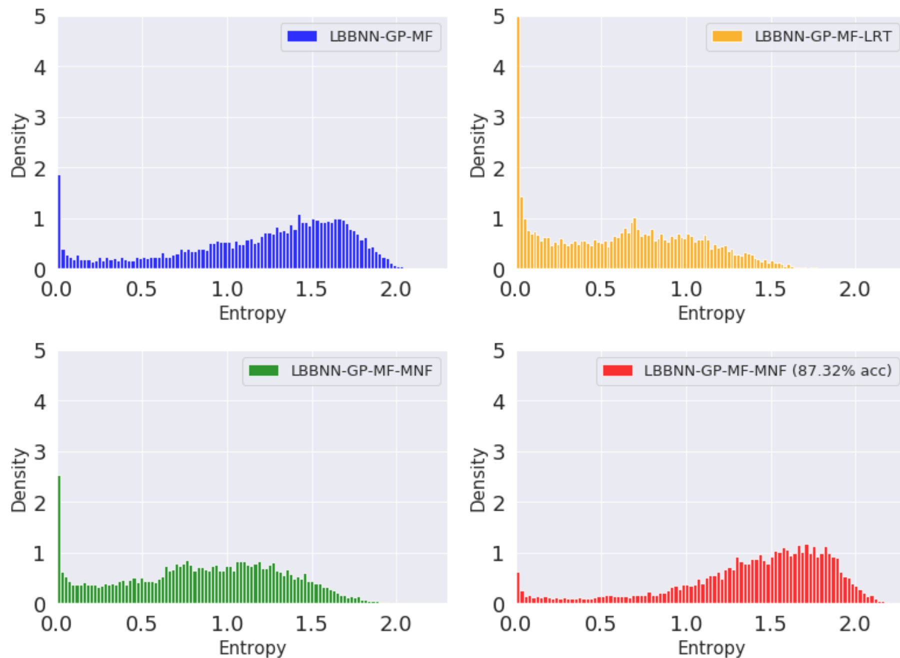

Figure 5.6: Density of the entropies of the MNIST validation set with the model trained on FMNIST. In addition to the three different methods considered, we have the LBBNN-GP-MF-MNF method trained with lower in-domain accuracy

the other two datasets. Thus, it seems evident that entropy (whether it is for in distribution or out of distribution data), strongly depends on predictive power.

To visualize the out of distribution entropies in a different way, we plot a histogram of the entropies obtained from the MNIST validation data, with the method trained on FMNIST and vice versa (see Figure 5.6 and 5.7). In the first one, we see something that is not so easy to spot in the empirical CDF plots, namely that we get very large densities for entropy close to zero. This means that an overconfident prediction is made on a large proportion of the out of distribution data. Note that the methods with the worst in domain accuracy (LBBNN-GP-MF, and LBBNN-GP-MF-MNF trained with lower predictive accuracy), also have much fewer of these overconfident predictions.

Finally, for the LBBNN-GP-MF-MNF method, we will take a closer look at how it handles *in domain* uncertainty. To do this, we compute the entropy of the predictive distribution on a subset (1000 samples). We are interested to see if there is a visual difference between images where the predictive distribution has high and low entropy. We plot ten images from each class, where for the first five images the predictive distributions have low entropy (high confidence predictions), and the last five images had high entropy (low confidence predictions). On the KMNIST dataset, (Figure 5.9) it is not easy to tell the difference between a low and high entropy prediction, but one could perhaps argue that the low entropy predictions have thicker lines. On MNIST (Figure 5.10), the

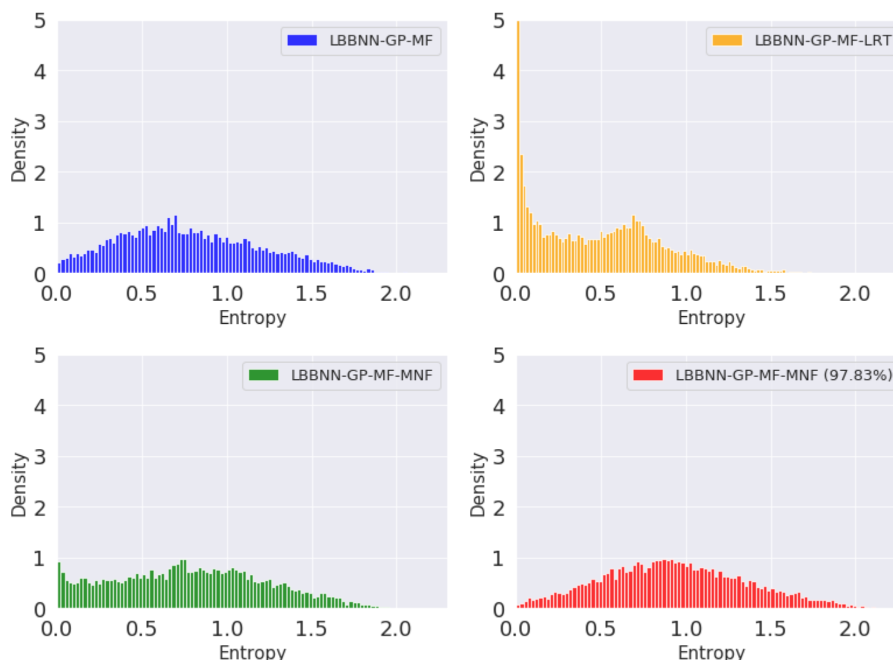

Figure 5.7: Density of the entropies of the FMNIST validation set with the model trained on MNIST. In addition to the three different methods considered, we have the LBBNN-GP-MF-MNF method trained to induce lower in-domain accuracy.

low entropy images also seem to have thicker lines. With higher uncertainty, the images become more indistinct, and also somewhat rotated. To a human eye, it might be challenging to classify some of these correctly. We see that the method seems to be uncertain about sevens that are written with a dash, even though these are easy to classify for a human. Perhaps this is because they look too close to the fours. For FMNIST (Figure 5.11), the high confidence predictions (on the left) within each class look very similar. They tend to have the same shape and often share a distinct feature. For example, the sandals (row six) and boots (row ten) have high heels, and the bags (row nine) have a strap. Another example is the sneakers on row eight. Here we see that they are all low cut and have white soles. We do not see this sort of uniformity amongst the high entropy predictions. Images within each class can look very diverse, such as the pants (row two) or the bags. It is also difficult to distinguish between different classes. The sneakers and the boots are interchangeable, and in general clothing items can be difficult to distinguish from one another. It therefore makes sense that we get high uncertainty about these items.

6 Discussion

In this thesis we have focused on Bayesian neural networks (BNN), and how they can be made more sparse. The motivation behind using BNNs is that they provide more realistic estimation of predictive uncertainty than classical

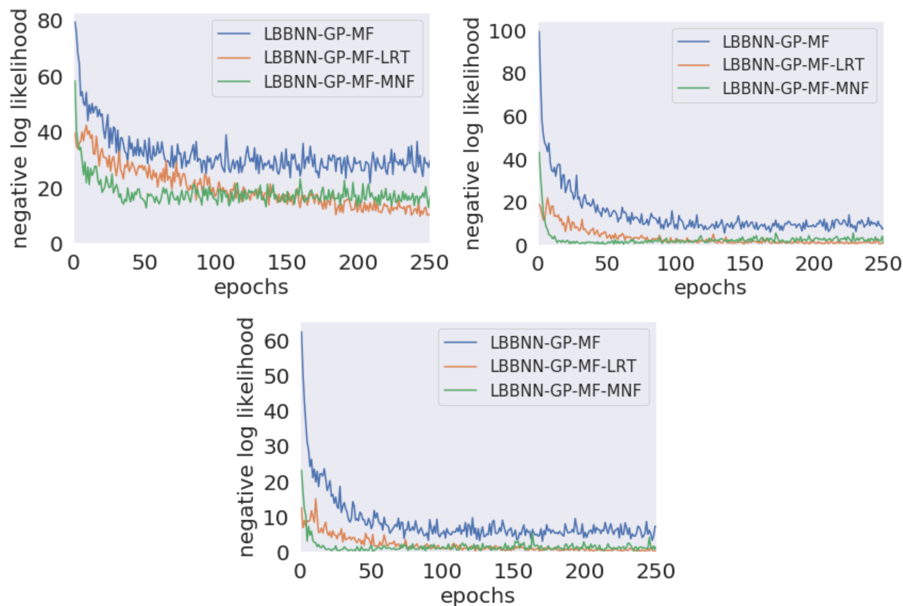

Figure 5.8: The median of the negative log likelihood on the training data over the different runs. Top left is FMNIST, top right is KMNIST and bottom is MNIST.

neural networks, since BNNs take into account uncertainty in the network parameters. Furthermore, neural networks (both Bayesian and classical) have too many parameters, and we are therefore interested in reducing the amount of them, without losing predictive power. The work here is based on the work by Hubin and Storvik (2019), which introduced the idea of considering uncertainty in the neural network structure (i.e. whether weights are included or not), and uncertainty in the weights of a given structure. Note that the network architecture (the number of layers, and the number of neurons within each layer) is fixed. Sparsity is induced by using a spike-and-slab prior on the weights. The results obtained in that paper show that under a fully Bayesian model one can achieve good predictive accuracy, while many (≈ 90 %) of the weights can be removed.

6.1 Contributions

We have built upon the work done by Hubin and Storvik (2019) in two ways. First, by extending the latent binary Bayesian neural network model (LBBNN) with the local reparametrization trick (LRT). Doing this significantly improved predictive accuracy across all three datasets, which may be attributed to the fact that using the LRT gives a gradient estimator with lower variance, and therefore faster convergence. Another advantage is the improved computational efficiency, due to sampling the activations directly. This is twofold (compared to standard BNNs) advantageous in this case, since the LBBNN method requires sampling of *two* matrices for each layer in the network. Furthermore, using

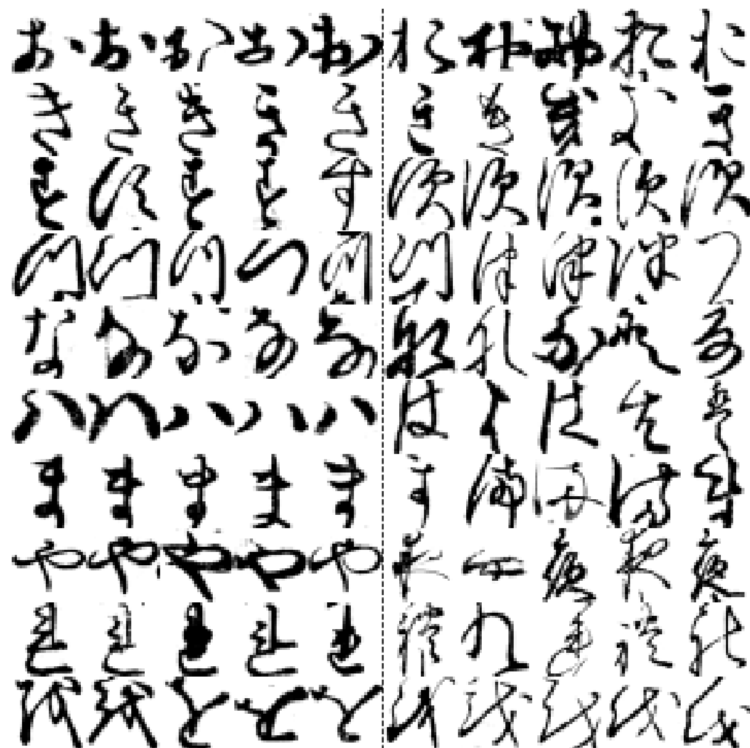

Figure 5.9: Rows are the ten classes (Hiragana) on KMNIST. Low entropy predictions are to the left of the dashed line, and high entropy predictions to the right.

the LRT avoids having to approximate the discrete inclusion variables with a continuous distribution, since these are never sampled directly.

The second main contribution of this thesis is to use normalizing flows on the variational distribution, in order to increase its flexibility and therefore move it closer to the true posterior distribution. This was done first by Louizos and Welling (2017), however that paper only considers dense networks without sparsity. Here, we use normalizing flows in combination with the LBBNN method. Initial experiments using flows directly on the weights and the inclusion parameters of the network showed little improvement in predictive accuracy, while also being very computationally demanding. By instead applying the flows in the space of the activations, the computational burden is manageable, albeit much slower (around five times) than only using the LRT. We also get the highest predictive accuracy on all three datasets with the normalizing flow method, while having around half the density of the baseline method.

When it comes to predictive uncertainty, we argue that the commonly used method of computing the entropy on out of distribution samples to quantify uncertainty is flawed, since we observe a strong dependence on how good the performance is on in distribution data. We also recognize the same pattern on

Figure 5.10: Illustration of the MNIST dataset, with low entropy predictions to the left of the dashed line, and high entropy predictions to the right.

in distribution entropies. We show (visually) that when the normalizing flow method is confident, the images within each class tend to look uniform and share distinct features; additionally, uncertain predictions tend to look different within each class, and images from different classes can often be difficult to separate.

6.2 Limitations

The simulation study was limited in the sense that the data generation process was a simple logistic regression model. While the normalizing flow method performed well, and seemed to be able to select the correct weights to be included in the model, it would perhaps be more interesting to conduct a more complicated simulation study since we are most interested in performance on large neural networks. Moreover, it would be easier to assess the performance of the normalizing flow method if it would be directly compared to the truth, i.e. doing mode jumping Markov chain Monte Carlo.

Another limitation of this thesis is that we do not have any conclusive results with regards to predictive uncertainty. We know that theoretically they should be better than using non-Bayesian neural networks, but since the results from computing the out-of-distribution entropies are questionable, we do not have a

Figure 5.11: Illustration of the FMNIST dataset, with low entropy predictions to the left of the dashed line, and high entropy predictions to the right.

way to compare the results. Additionally, it would have been interesting to see if the normalizing flow method would be an improvement compared to using the method without flows. Since the method with flows has a variational density that is closer to the true density, it would be compelling to see if this would also result in better uncertainty estimates.

6.3 Future research

An interesting avenue of research would be to find a better way to compare uncertainty estimates between the different methods considered here, and non-Bayesian methods. It is also conceivable that the method with normalizing flows could be improved by trying different flows – ones that already exist, or to design new ones that could perhaps be better suited for this particular task. Another possibility is to change the neural network used within the coupling layer of the RNVP transformation to something more complicated than a simple fully connected neural network. Furthermore, it would be interesting to see whether these methods would induce similar levels of sparsity on convolutional neural networks, and also compare the sparsity inducing algorithms of non-Bayesian neural networks to the ones discussed here.

Bibliography

Ablavatski and G. R. Dukhan. Accelerating Neural Networks on Mobile and Web with Sparse Inference . https://ai.googleblog.com/2021/03/accelerating-neural-networks-on-mobile.html, 2021. [Online; accessed 22-January-2022].

J. Bai, Q. Song, and G. Cheng. Efficient variational inference for sparse deep learning with theoretical guarantee. *Advances in Neural Information Processing Systems*, 33:466–476, 2020.

M. M. Barbieri and J. O. Berger. Optimal predictive model selection. *The annals of statistics*, 32(3):870–897, 2004.

M. Betancourt. A conceptual introduction to Hamiltonian Monte Carlo. *arXiv preprint arXiv:1701.02434*, 2017.

D. M. Blei, A. Kucukelbir, and J. D. McAuliffe. Variational inference: A review for statisticians. *Journal of the American statistical Association*, 112(518): 859–877, 2017.

C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra. Weight uncertainty in neural network. In *International conference on machine learning*, pages 1613–1622. PMLR, 2015.

L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311, 2018.

S. Brooks, A. Gelman, G. Jones, and X.-L. Meng. *Handbook of Markov chain Monte Carlo*. CRC press, 2011.

G. Casella, R. Berger, and B. P. Company. *Statistical Inference*. Duxbury advanced series in statistics and decision sciences. Thomson Learning, 2002. ISBN 9780534243128. URL https://books.google.no/books?id=0x_vAAAAMAAJ.

S. Chib and E. Greenberg. Understanding the Metropolis-Hastings algorithm. *The american statistician*, 49(4):327–335, 1995.

T. Clanuwat, M. Bober-Irizar, A. Kitamoto, A. Lamb, K. Yamamoto, and D. Ha. Deep learning for classical japanese literature, 2018.

L. Deng. The MNIST database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, 29(6):141–142, 2012.

L. Dinh, D. Krueger, and Y. Bengio. Nice: Non-linear independent components estimation, 2015.

L. Dinh, J. Sohl-Dickstein, and S. Bengio. Density estimation using real nvp, 2017.

U. Evci, T. Gale, J. Menick, P. S. Castro, and E. Elsen. Rigging the lottery: Making all tickets winners. In *International Conference on Machine Learning*, pages 2943–2952. PMLR, 2020.

G. Fei and B. Liu. Breaking the closed world assumption in text classification. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 506–514, 2016.

J. Frankle and M. Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2019.

T. Gale, E. Elsen, and S. Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

A. E. Gelfand. Gibbs sampling. *Journal of the American statistical Association*, 95(452):1300–1304, 2000.

C. J. Geyer. Practical Markov chain Monte Carlo. *Statistical science*, pages 473–483, 1992.

I. Guyon, S. Gunn, A. Ben-Hur, and G. Dror. Result analysis of the NIPS 2003 feature selection challenge. *Advances in neural information processing systems*, 17, 2004.

S. Han, J. Pool, S. Narang, H. Mao, E. Gong, S. Tang, E. Elsen, P. Vajda, M. Paluri, J. Tran, B. Catanzaro, and W. J. Dally. Dsd: Dense-sparse-dense training for deep neural networks, 2017.

J. Hron, A. G. de G. Matthews, and Z. Ghahramani. Variational Gaussian dropout is not Bayesian, 2017.

A. Hubin and G. Storvik. Mode jumping MCMC for Bayesian variable selection in GLMM. *Computational Statistics Data Analysis*, 127:281–297, Nov 2018. ISSN 0167-9473. doi: 10.1016/j.csda.2018.05.020. URL http://dx.doi.org/10.1016/j.csda.2018.05.020.

A. Hubin and G. Storvik. Combining model and parameter uncertainty in Bayesian neural networks, 2019.

P. Izmailov, S. Vikram, M. D. Hoffman, and A. G. G. Wilson. What are Bayesian neural network posteriors really like? In *International Conference on Machine Learning*, pages 4629–4640. PMLR, 2021.

A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi. A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*, 53(8):5455–5516, 2020.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

D. P. Kingma and M. Welling. Auto-encoding Variational Bayes, 2014.

D. P. Kingma, T. Salimans, and M. Welling. Variational dropout and the local reparameterization trick. *Advances in neural information processing systems*, 28, 2015.

D. P. Kingma, T. Salimans, R. Jozefowicz, X. Chen, I. Sutskever, and M. Welling. Improved variational inference with inverse autoregressive flow. *Advances in neural information processing systems*, 29, 2016.

Y. Li. Deep reinforcement learning: An overview, 2018.

C. Louizos and M. Welling. Multiplicative normalizing flows for variational Bayesian neural networks. In *International Conference on Machine Learning*, pages 2218–2227. PMLR, 2017.

A. L. Maas, A. Y. Hannun, A. Y. Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Citeseer, 2013.

C. J. Maddison, A. Mnih, and Y. W. Teh. The Concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

D. Molchanov, A. Ashukha, and D. Vetrov. Variational dropout sparsifies deep neural networks. In *International Conference on Machine Learning*, pages 2498–2507. PMLR, 2017.

R. M. Neal and U. o. T. Jianguo Zhang. Classification for High Dimensional Problems Using Bayesian Neural Networks and Dirichlet Diffusion Trees . http://clopinet.com/isabelle/Projects/NIPS2003/Slides/Neal-Zhang.pdf, 2003. [Online; accessed 22-March-2022].

R. M. Neal et al. MCMC using Hamiltonian dynamics. *Handbook of Markov chain Monte Carlo*, 2(11):2, 2011.

A. Nguyen, J. Yosinski, and J. Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 427–436, 2015.

T. Papamarkou, J. Hinkle, M. T. Young, and D. Womble. Challenges in Markov chain Monte Carlo for Bayesian neural networks. *arXiv preprint arXiv:1910.06539*, 2019.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

L. Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998.

D. Rezende and S. Mohamed. Variational inference with normalizing flows. In *International conference on machine learning*, pages 1530–1538. PMLR, 2015.

J. Riebesell. RNVP implementation. https://github.com/janosh/torch-mnf/blob/main/torch_mnf/flows/rnvp.py, 2020. [Online; accessed 09-May-2022].

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

D. Salvator. How Sparsity Adds Umph to AI Inference. https://blogs.nvidia.com/blog/2020/05/14/sparsity-ai-inference/, 2020. [Online; accessed 20-April-2022].

A. Shapiro. Monte Carlo sampling methods. *Handbooks in operations research and management science*, 10:353–425, 2003.

C. Shorten and T. M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):1–48, 2019.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

J. M. Tomczak and M. Welling. Improving Variational Auto-Encoders using Householder Flow, 2017.

A. Touati, H. Satija, J. Romoff, J. Pineau, and P. Vincent. Randomized value functions via multiplicative normalizing flows. In *Uncertainty in Artificial Intelligence*, pages 422–432. PMLR, 2020.

R. van den Berg, L. Hasenclever, J. M. Tomczak, and M. Welling. Sylvester normalizing flows for variational inference, 2019.

A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018, 2018.

E. W. Weisstein. Normal Sum Distribution. From MathWorld–A Wolfram Web Resource. https://mathworld.wolfram.com/NormalSumDistribution.html, 2000. [Online; accessed 22-December-2021].

F. Wenzel, K. Roth, B. S. Veeling, J. Świątkowski, L. Tran, S. Mandt, J. Snoek, T. Salimans, R. Jenatton, and S. Nowozin. How good is the Bayes posterior in deep neural networks really? *arXiv preprint arXiv:2002.02405*, 2020.

H. Xiao, K. Rasul, and R. Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

T. Young, D. Hazarika, S. Poria, and E. Cambria. Recent trends in deep learning based natural language processing. *IEEE Computational intelligence magazine*, 13(3):55–75, 2018.