Hands-on Bayesian Neural Networks – Supplementary material

I. Practical example – Bayesian MNIST

Over the years, MNIST [1] has become the most renown toy dataset in deep learning. Coding a handwritten digit classifier based on this dataset is now the Hello World of deep neural network programming. The first practical example we introduce for this tutorial is thus just a plain old classifier for MNIST implemented as a BNN. The code is available on github [https://github.com/french-paragon/BayesianMnist]. The setup is as follows;

The purpose is to show a Bayesian Neural Network hello world project.

The problem is to train a BNN to perform hand-written digit recognition.

The dataset we are going to use is MNIST. However, to be able to test how the proposed BNN reacts to unseen data, we will remove one of the classes from the training set so that the network never sees it during training.

The stochastic model is the plain Bayesian regression presented in Section IV-B of the main paper. We use a normal distribution as a prior for the network parameters $\boldsymbol{\theta}$, with a standard deviation of 5 for the weights and 25 for the bias as we expect the scale of the bias to have a bit more variability than the weights. This is because, in a RELU or leaky-RELU network, the weights influence the local change in slope of a layer function while the bias indicates the position of those inflection points.

The functional model is a standard convolutional neural network with two convolutional layers followed by two fully connected layers (Fig. 1).

A. Training

We used Variational Inference to train this BNN. We use a Gaussian distribution, with diagonal covariance, as a variational posterior (see Section V-C of the main paper). Since we expect the posterior to be multi-modal, we train an ensemble of BNNs instead of a single one (see Section V-E2b of the main paper). Note that the actual top performing method on the MNIST leaderboard is actually an ensemble-based method [2]. (The authors refer to a multi-column neural network, but the idea is the same.)

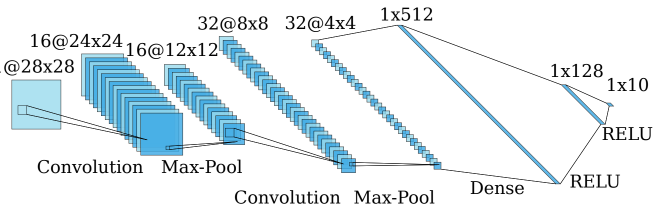


Fig. 1: The neural network architecture used for the Bayesian MNIST practical example.

B. Results

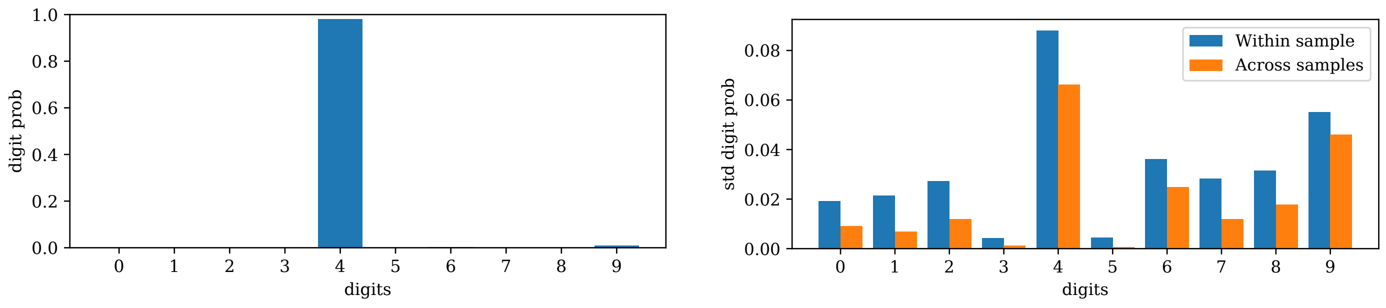
We tested the final BNN against **(1)** the test set restricted to the classes the network has been trained on, **(2)** the test set restricted to the class the network has not been trained on, and **(3)** a pure white noise. For each case, we report the average probabilities predicted by the BNN for all input images in the considered class (Fig. 2), along with the average standard deviation of the network on a single sample, and the standard deviation of the prediction for all samples. While not as informative and rigorous than a collection of calibration curves, it does indicate if the BNN uncertainty matches the natural variability of the dataset. The reported results show that for a class seen during training, the network gives the correct prediction and seems to be quite confident about its results. The variability per sample and across samples are coherent with one another and quite small. When presented with a digit from a class unseen during training, the network apparently tries to match it to different digits that might share some similarities. However, the low predicted probabilities along with the high per sample and across samples standard deviation show that the network is aware that it does not know about these unseen digits. As for the white noise, the average output is not perfectly flat but quite constant.

II. Practical example – Sparse measure

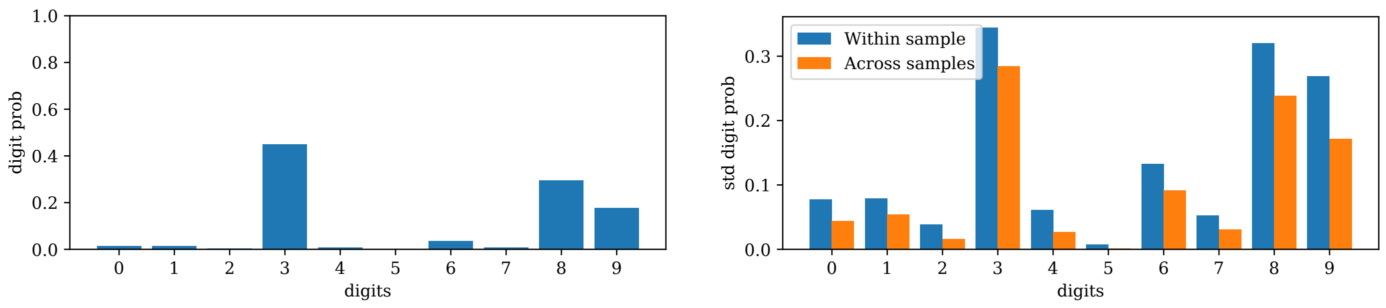
The second practical example we introduce, titled ”Sparse measure”.

The purpose is to present a small model illustrating how different training strategies can be integrated in a BNN. We also provide a Python implementation of this example in order to show how the different hypotheses we present in this Appendix translate to actual code [https://github.com/french-paragon/sparse_measure_bnn].

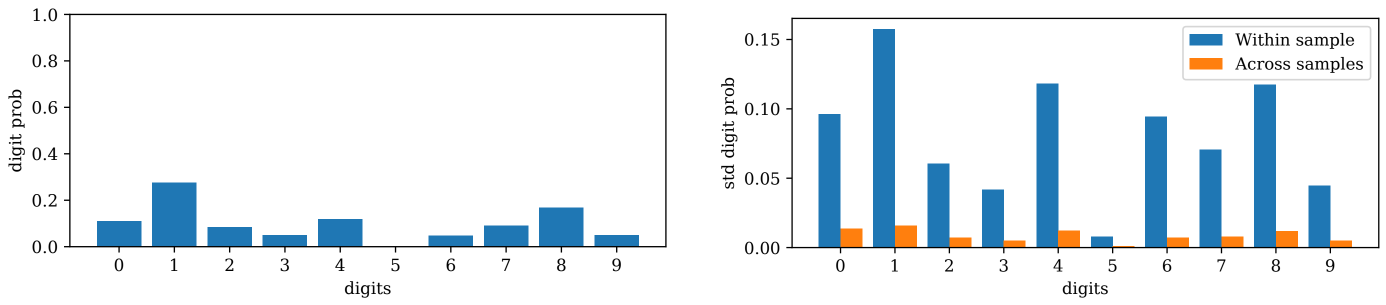
The problem is to learn a function $f : \mathbb{R}^n \to \mathbb{R}^m$ based on a dataset $D = \{(\boldsymbol{x}_i, \boldsymbol{y}_i) | i \in [1, N], \boldsymbol{x}_i \in \mathbb{R}^n, \boldsymbol{y}_i \in \mathbb{R}^m\}$ where for each tuple $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ only some elements of $\boldsymbol{y}_i$ are actually measured. This scenario corresponds to any application of machine learning were measuring a raw signal, represented by the $\boldsymbol{x}_i$'s, is easy but one want to train an algorithm to reconstruct a derived signal, represented by the $\boldsymbol{y}_i$'s, which is much harder to acquire. On top of that, the elements of $\boldsymbol{y}_i$ are measured with a certain level of uncertainty. Also, one element of $\boldsymbol{y}_i$, which is constant but unknown, has a much higher level of uncertainty (*e.g.*, it has been found that one of the instruments used for data acquisition is defective, but no one remembers for which experiment it has been used). It is



(a) Results on a seen class



(b) Results on the unseen class



(c) Results on white noise

Fig. 2: Average prediction for a seen class (a) the unseen class (b) and just white noise input (c) of the Bayesian MNIST practical example.

also known that the function f is continuous and will, with very high probability map any given input to a point near a known hyperplane of $\mathbb{R}^m$.

The dataset is generated at random for each experiment using python. Inputs are sampled from a zero-mean multivariate normal distribution with a random correlation matrix. The input is then multiplied by a random projection matrix to $\mathbb{R}^3$ before a non-linear function is applied. A random set of three orthonormal vectors in $\mathbb{R}^m$ is then generated and used to reproject the values in the final output space, constrained on a known hyperplane. A final non-linear function is applied on the input and its results are added to the previously generated outputs with a very low gain factor to ensure the actual output is still likely to be near the known hyperplane. Finally, noise is added to the training data, with one channel receiving much more noise than the other channels. We set $n = m = 9$. The measured values for the training set are then selected at random, the other values being set to 0. A set of binary masks are returned with the data to indicate which values have actually been measured.

The stochastic model is the most interesting part of this case study. Note that learning one function $f : \mathbb{R}^n \to \mathbb{R}^m$ is equivalent to learning m functions $f : \mathbb{R}^n \to \mathbb{R}$, meaning a standard Bayesian regression would be enough to address this learning problem. But since doing so ignores the fact that the functions under consideration can be correlated with one another it is possible to do much more.

First, it is possible to extend the model to account for the **noisy labels**. To learn which output channel is actually more noisy than the others, a **metalearning** approach can be used with **hierarchical Bayes** applied not on the model parameters but on the noise model. If applied to standard point estimate networks, this approach would be seen as learning a loss. The prior can be extended with a **consistency condition** to account for the fact that the unknown function projects points near a known hyperplane. Last but not least, a **semisupervised learning** model can be used to share information across the training samples. The PGM corresponding to this approach is depicted in Fig. 3. We kept the plate for the dataset D as we will use a second consistency condition to implement the

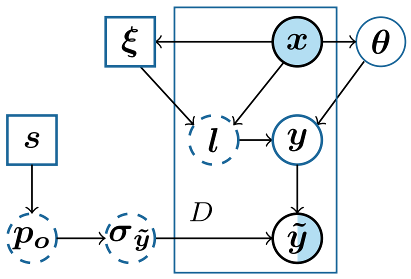


Fig. 3: BBN corresponding to the stochastic model we used for the sparse measures example.

semisupervised learning strategy. We also chose to use a last layer-only BNN, mostly to illustrate how it can be done. We will also consider point estimates for the hierarchy of noise quantification variables.

The functional model $\Phi_{\boldsymbol{\xi},\boldsymbol{\theta}}$ is a feed-forward artificial neural network with five hidden layers (Fig. 4). The first four layers are point-estimate layers, with parameters $\boldsymbol{\xi}$, while the last hidden layer and the output layer are stochastic, with parameters $\boldsymbol{\theta}$. We also have three Monte-Carlo dropout layers at the end of the network. The hourglass shape of the network is supposed to match the expected behaviour of the studied function, where the information is approximately located on a smaller dimensional space.

A. *Base stochastic model*

The base stochastic model is built as a Bayesian regression with a last-layer BNN architecture. The base probability of $\boldsymbol{\theta}$, without the consistency conditions that will be applied later on, has been set to a normal distribution with mean 0 and a standard deviation of 1 for the weights and 10 for the bias. The prior distribution of the output $\boldsymbol{y}$ knowing the input $\boldsymbol{x}$ and the parameters $\boldsymbol{\theta}$ and $\boldsymbol{\xi}$ is then given by:

$$p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta},\boldsymbol{\xi}) = \mathcal{N}(\Phi_{\boldsymbol{\xi},\boldsymbol{\theta}}(\boldsymbol{x}), \sigma_{\boldsymbol{y}}), \tag{1}$$

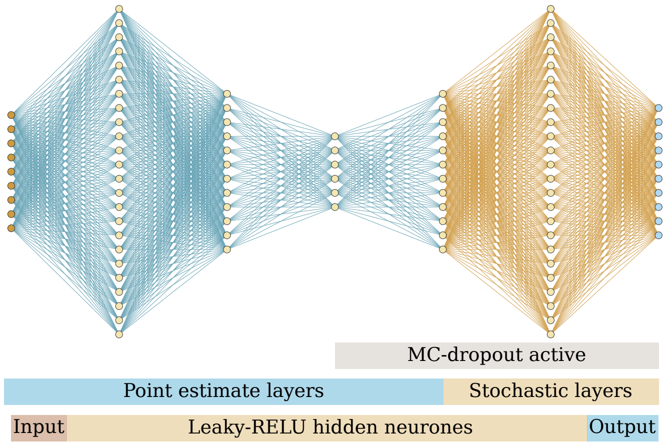


Fig. 4: The neural network architecture we used for the sparse measure example.

where $\sigma_{\boldsymbol{y}}$ is the small uncertainty due to the fact that the functionnal model will never be able to perfectly fit the actual function f. We set $\sigma_{\boldsymbol{y}} = 0.1$.

B. *Noisy labels model*

For this regression problem, we assume that the measurements have been corrupted by a zero-mean Gaussian noise, which is a convenient formulation and more than often a reasonable approximation of the true noise model for continuous measurements (due to the central limit theorem). The mean of the noisy measurements $\tilde{\boldsymbol{y}}$ is thus the unobserved, true function value y. For the standard deviation $\boldsymbol{\sigma_{\tilde{y}}}$, we assume that it is $\sigma_{in} = 0.1$ for inliers and $\sigma_{out} = 5$ for outliers. The problem here is that we do not know which channel has been corrupted by the additional noise. We could just set a constant standard deviation for all channels sligthly above σ_{in}, but instead we added an additional variable, $\boldsymbol{p_o}$, a vector representing the probability that a given channel is an outlier. Since a single channel is corrupted by noise, we could constrain $\boldsymbol{p_o}$ to lie on the probability simplex. Instead, to let the model generalize to cases with more than a single noisy channel, we will impose that $\boldsymbol{p_o} \in [0,1]^m$. For convenience, we will consider an unconstrained variable $\boldsymbol{s} \in \mathbb{R}^m$ and define $\boldsymbol{p_o}$ as a function of s to simplify the optimisation:

$$\boldsymbol{p_o} = \frac{1}{1+e^{-\boldsymbol{s}}}. \tag{2}$$

The value of σ is then determined by a Bernoulli distribution with parameter $\boldsymbol{p_o}$:

$$\begin{aligned} &\boldsymbol{b} \sim \text{Bernoulli}(\boldsymbol{p_o}), \\ &\boldsymbol{\sigma_{\tilde{y}}} = \boldsymbol{b} \cdot \sigma_{out} + (1-\boldsymbol{b}) \cdot \sigma_{in}. \end{aligned} \tag{3}$$

The only thing left is to determine the prior for $\boldsymbol{s}$. Assuming we have no idea which channel is noisy and which is not, a reasonable hypothesis is a Gaussian with mean 0 and large covariance matrix $\boldsymbol{\Sigma_s}$ (we choose $\boldsymbol{\Sigma_s} = 2500\boldsymbol{I}$)). Yet, since we know that only one channel is noisy, we can add a consistency condition to enforce that $\|\boldsymbol{p_o}\|_1 \approx 1$, or any expected number of noisy channels:

$$p(\boldsymbol{s}) \propto \mathcal{N}(0, \boldsymbol{\Sigma_s}) exp\left[-\gamma_s\left(\left\|\frac{1}{1+e^{-\boldsymbol{s}}}\right\|_1 - 1\right)^2\right], \tag{4}$$

where γ_s is a scaling factor representing the confidence one has in the model capacity to fit the correct number of noisy channels. We set $\gamma_s = 3000$ as we are almost certain the model should be able to fit the single noisy channel. The probability of $\tilde{\boldsymbol{y}}$ given $\boldsymbol{y}$ and $\boldsymbol{\sigma_{\tilde{y}}}$ is then given by:

$$p(\tilde{\boldsymbol{y}}|\boldsymbol{y},\boldsymbol{\sigma_{\tilde{y}}}) = \mathcal{N}(\boldsymbol{y},\boldsymbol{\sigma_{\tilde{y}}}). \tag{5}$$

Since we are not so interested in the complete posterior distribution for $\boldsymbol{s}$ or even $\boldsymbol{\sigma_{\tilde{y}}}$, we will learn point estimates of those variables instead and then derive a regularization term from the log of Equation (4). The final contribution for the learnable parameters s to the total loss is $\log(p(\boldsymbol{s}))$ plus an unknown constant one can just ignore to, is:

$$\frac{1}{2}\left(\boldsymbol{s}^\top \boldsymbol{\Sigma_s}^{-1} \boldsymbol{s}\right) + \gamma_s\left(\left\|\frac{1}{1+e^{-\boldsymbol{s}}}\right\|_1 - 1\right)^2. \tag{6}$$

This is implemented in the —outlierAwareSumSquareError— learnable loss in the python implementation.

C. Consistency condition

The function f is likely to map a given input near a known subspace of $\mathbb{R}^m$, which can be represented as span($\boldsymbol{S}$) where $\boldsymbol{S} \in \mathbb{R}^{m\times 3}$ is a given orthonormal matrix. The distance $d_{\boldsymbol{y},\boldsymbol{S}}$ between a prediction $\boldsymbol{y}$ and span($\boldsymbol{S}$) is then given by:

$$d_{\boldsymbol{y},\boldsymbol{S}} = \left\| \boldsymbol{y} - \boldsymbol{S}\boldsymbol{S}^\top \boldsymbol{y} \right\|_2 . \tag{7}$$

We then assume a-priori that $d_{\boldsymbol{y},\boldsymbol{S}}$ knowing $\boldsymbol{x}$ follows a normal distribution and use this to derive a consistency condition for $p(\boldsymbol{\theta}, \boldsymbol{\xi}|D_{\boldsymbol{x}})$:

$$p(\boldsymbol{\theta}, \boldsymbol{\xi}|D_{\boldsymbol{x}}) \propto p(\boldsymbol{\theta})p(\boldsymbol{\xi})e^{-\sum_{\boldsymbol{x}\in D_{\boldsymbol{x}}} \frac{1}{\sigma_d^2} \left\| \Phi_{\boldsymbol{\xi},\boldsymbol{\theta}}(\boldsymbol{x}) - \boldsymbol{S}\boldsymbol{S}^\top \Phi_{\boldsymbol{\xi},\boldsymbol{\theta}}(\boldsymbol{x}) \right\|_2^2} \tag{8}$$

where σ_d is the prior standard deviation of $d_{\boldsymbol{y},\boldsymbol{S}}$, which we set to 1.

The additional contribution to the loss by the consistency condition is thus:

$$\sum_{\boldsymbol{x}\in D_{\boldsymbol{x}}} \frac{1}{\sigma_d^2} \left\| \Phi_{\boldsymbol{\xi},\boldsymbol{\theta}}(\boldsymbol{x}) - \boldsymbol{S}\boldsymbol{S}^\top \Phi_{\boldsymbol{\xi},\boldsymbol{\theta}}(\boldsymbol{x}) \right\|_2^2 . \tag{9}$$

D. Semisupervised learning strategy

As stated in Section IV-D1 of the main paper, there are two ways one can implement a semisupervised learning strategy for a BNN. The first one is by using a consistency condition (*i.e.*, data driven regularization). The second one is by assuming some kind of dependence across the samples. We use the former for this practical example as data driven regularization is generally easier to implement.

The intuition behind this example is that if two points from the input space are close to one another then their image by f should also be close to one another. The simplest approach to formally measure this is just to assume that the norm of the difference in the output space normalized by the distance in the input space is small. This sets the consistency condition function $C(\boldsymbol{\theta}, \boldsymbol{\xi}, \boldsymbol{x})$ to be:

$$C(\boldsymbol{\theta}, \boldsymbol{\xi}, \boldsymbol{x}) = \frac{1}{2} \sum_{\boldsymbol{x}'\in D_{\boldsymbol{x}}\setminus\{\boldsymbol{x}\}} \frac{\| \Phi_{\boldsymbol{\xi},\boldsymbol{\theta}}(\boldsymbol{x}) - \Phi_{\boldsymbol{\xi},\boldsymbol{\theta}}(\boldsymbol{x}') \|_2^2}{\| \boldsymbol{x} - \boldsymbol{x}' \|_2^2} . \tag{10}$$

Averaging this consistency condition over the whole training set $D_{\boldsymbol{x}}$ is equivalent to computing the sum over all output channels of $\Phi_{\boldsymbol{\xi},\boldsymbol{\theta}}$ of the quadratic form of the Laplacian matrix of the dense graph spanned by the training set $D_{\boldsymbol{x}}$, where the weights of the edges are given by the inverse Euclidean distances of the points. Graph Laplacian regularization is itself a common method to implement semisupervised learning [3].

This approach can be further improved. Assume the differences between the components of two random input vectors $\boldsymbol{x}$ and $\boldsymbol{x}'$, as well as the differences between the components of their images by f, are normally distributed. This implies that:

$$\frac{\sigma_{\Delta x}^2 \, \| f(\boldsymbol{x}) - f(\boldsymbol{x}') \|_2^2}{\sigma_{\Delta f}^2 \, \| \boldsymbol{x} - \boldsymbol{x}' \|_2^2} \sim F(n, m), \tag{11}$$

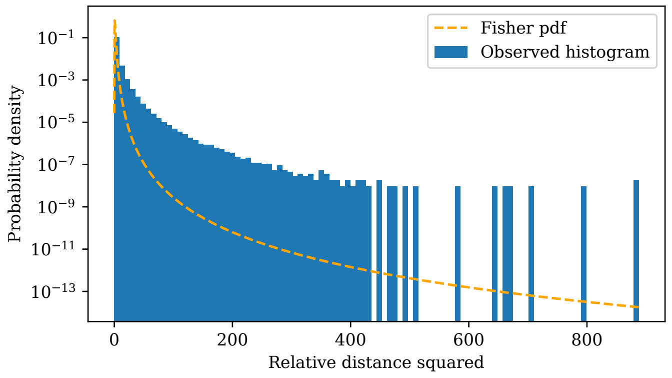


Fig. 5: Comparison between the observed distribution of relative distances squared in the test set and the Fisher–Snedecor distribution of parameters 9 and 9

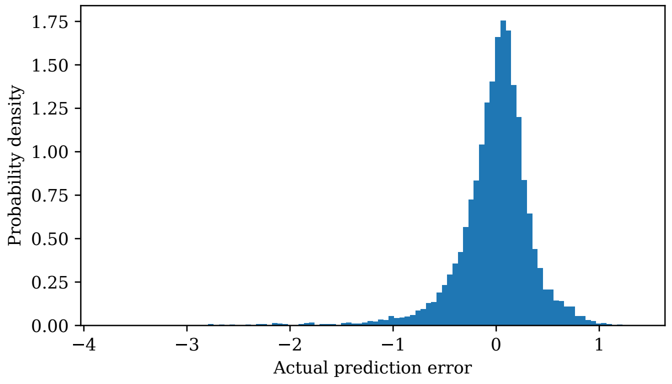


Fig. 6: Prediction error distribution for the sparse measure example

where $F(n, m)$ is a Fisher–Snedecor distribution with parameters n and m, $\sigma_{\Delta x}$ is the prior standard deviation of the difference of two random inputs, and $\sigma_{\Delta f}$ is the prior standard deviation of the difference of the corresponding outputs. As shown in Fig. 5, such assumptions are not perfect but still match reasonably well the data, especially since those assumptions are just a prior and not an exact model. The most important point to notice is that both the observed and prior distributions have a heavy tail. This would not be the case with the consistency condition proposed in Equation 10. This naive approach would thus penalize outliers too heavily. We thus implemented the following, corrected, consistency condition function in the sparse measures example:

$$\sum_{\boldsymbol{x}'\in D_{\boldsymbol{x}}\setminus\{\boldsymbol{x}\}} \left(\frac{n+m}{2} - 1 \right) log(1 + F) - \left(\frac{n}{2} - 1 \right) \log(F), \tag{12}$$

where $F = \lambda_\Delta \frac{\| \Phi_{\boldsymbol{\xi},\boldsymbol{\theta}}(\boldsymbol{x}) - \Phi_{\boldsymbol{\xi},\boldsymbol{\theta}}(\boldsymbol{x}') \|_2^2}{\| \boldsymbol{x} - \boldsymbol{x}' \|_2^2}$ is the Fisher statistic.

The only parameter one has to define for this prior is λ_Δ, which is the ratio of $\sigma_{\Delta x}^2$ and $\sigma_{\Delta f}^2$. We set $\lambda_\Delta = 1$.

E. Results

The estimator for $\boldsymbol{y}$ reaches a RMSE of around 0.4 with a distribution of error approximating a normal distribution.

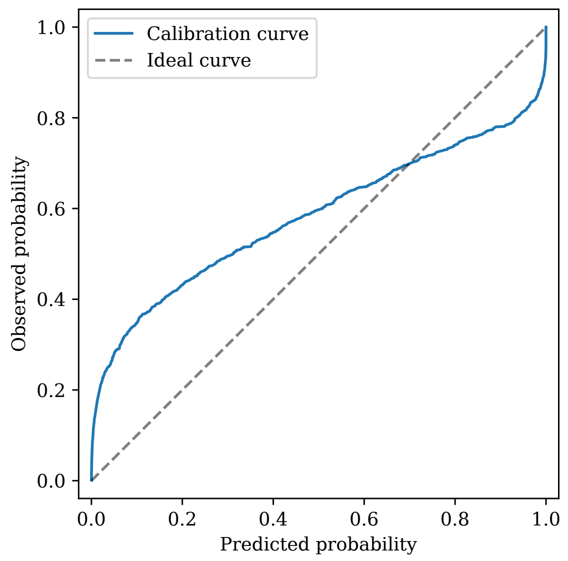


Fig. 7: Calibration curve for the sparse measure example

Variations can be observed over mutliple runs, as a different function f is generated at random each times (Fig. 6). The python code also computes the calibration curve using the hypothesis for regression models presented in Section VII of the main paper. The results (Fig. 7) show that the model tends to be slightly underconfident, except for large outliers where the model is overconfident. This is probably due to the fact that our simple variational inference model cannot fit the exact posterior and thus tends to be a bit too pessimistic about its predictions to compensate for the outliers it is unable to efficiently fit.

III. Practical example – Paperfold

The last practical example we introduce is titled ”Paperfold”. The code is available on github [https://github.com/french-paragon/paperfold-bnn].

The purpose of this case study is to compare different inference approaches for BNNs. We introduce a model, and a corresponding dataset, which is small enough to make all existing training methods for BNNs tractables when used for the corresponding architecture. This makes the number of parameters small enough such that the posterior and the corresponding approximations are low dimensional enough to be easily visualized and sampled without issues with exact MCMC methods. On the other hand, the model is complex enough to display most of the issues one would encounter when studying a BNN posterior.

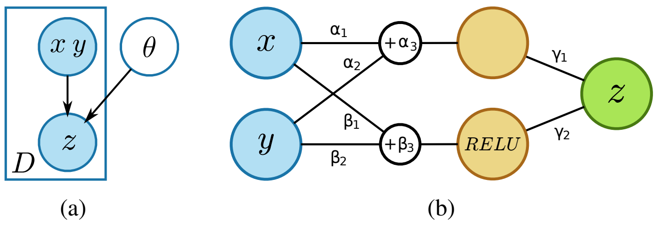


Fig. 8: Stochastic (a) and functional (b) model for the paperfold case study.

The problem is to fit a one dimensional function of two parameters of the form $z = f(x, y)$.

The dataset is made of eight (x, y, z) samples grouped into the vectors $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ with:

$$\boldsymbol{x} = \begin{pmatrix} \sqrt{2} \\ \frac{1}{2}\sqrt{2} \\ 0 \\ -\frac{1}{2}\sqrt{2} \\ -\sqrt{2} \\ -\frac{1}{2}\sqrt{2} \\ 0 \\ \frac{1}{2}\sqrt{2} \end{pmatrix}, \boldsymbol{y} = \begin{pmatrix} 0 \\ \frac{1}{2}\sqrt{2} \\ \sqrt{2} \\ \frac{1}{2}\sqrt{2} \\ 0 \\ -\frac{1}{2}\sqrt{2} \\ -\sqrt{2} \\ -\frac{1}{2}\sqrt{2} \end{pmatrix}, \boldsymbol{z} = \begin{pmatrix} 1 \\ \frac{1}{2} \\ 0 \\ \frac{1}{2} \\ 1 \\ \frac{1}{2} \\ 0 \\ \frac{1}{2} \end{pmatrix}. \tag{13}$$

The stochastic model (Fig. 8a) is the usual fully supervised Baysian regression. We assume each measure in $\boldsymbol{z}$ has a small uncertainty $\varepsilon_i \sim \mathcal{N}(0, \sigma_{\boldsymbol{z}}^2)$, with $\sigma_{\boldsymbol{z}}$ being a small positive standard deviation that one can set depending on the experiment. By default, we assume that $\sigma_{\boldsymbol{z}} = 0.1$.

The functional model (Fig. 8b) is a two-branch and two-layer feedforward neural network. The first layer branch has no non-linearities afterwards, while the second branch is followed by a $RELU$ non-linearity. The second layer just takes the weighed sum of both branches and thus has no bias. This model reduces to a function of the form:

$$z = \gamma_1(\alpha_1 x + \alpha_2 y + \alpha_3) + \gamma_2 \text{RELU}(\beta_1 x + \beta_2 y + \beta_3). \tag{14}$$

Here, $\boldsymbol{\alpha}$ represents the parameters of the first branch of the first layer, $\boldsymbol{\beta}$ the parameters of the second branch of the first layer and $\boldsymbol{\gamma}$ the parameters of the second layer. This simple model represents a planar function with a single fold along the line of equation $\beta_1 x + \beta_2 y + \beta_3 = 0$.

This model has always two and only two non-equivalent ways of fitting the proposed dataset (see Fig. 9). The model also exhibits some weight space symmetry (*i.e.*, the half-planes fitted by the first and second branch of the first layer can be swapped, resulting in a different parametrization but equivalent fitting of the data) and scaling symmetry (between $\boldsymbol{\beta}$ and $\boldsymbol{\alpha}$ against $\boldsymbol{\gamma}$).

To sample the posterior, we first have to choose a prior for the parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ of the BNN. The following procedure works for any choice of prior $p(\boldsymbol{\alpha}, \boldsymbol{\beta}), p(\boldsymbol{\gamma})$ where the probability distribution of the parameters of the first layer is independent of the probability distribution of the parameters of the second layer. For simplicity and if not specified otherwise, we assume a normal prior with diagonal constant covariance:

$$\begin{pmatrix} \boldsymbol{\alpha} \\ \boldsymbol{\beta} \\ \boldsymbol{\gamma} \end{pmatrix} \sim \mathcal{N}(0, \sigma^2 \boldsymbol{I}), \tag{15}$$

with σ a positive standard deviation. By default, we assume we do not know much about the model and set $\sigma = 5$ for this case study.

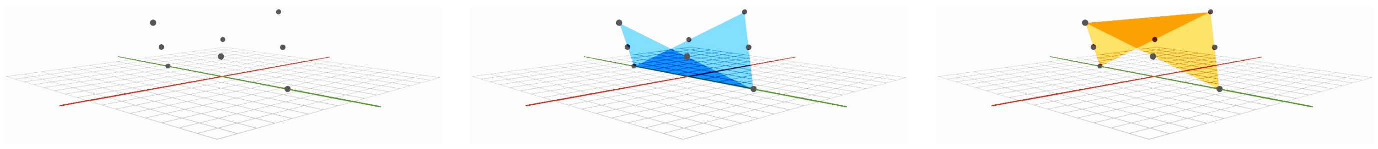

(a) Raw data (b) First possible fold (c) Second possible fold

Fig. 9: The dataset and two possible folding behavior of the model for the paperfold case study.

A. *Comparison of training methods*

We implemented four different samplers for the paperfold example (Fig. 10):

- The NUTS sampler, whose implementation is provided by the Pyro python library. This is a state-of-the-art general purpose MCMC sampler, which we used mainly as a way of generating samples from the true posterior as a base of comparison with other methods. This is because this MCMC sampler is much slower than the other methods we consider (Fig. 13).
- A variational inference model based on a Gaussian approximation of the posterior.
- An ensemble based approach, and
- An ensemble of variational inference based models.

1) MCMC: The MCMC sampler leads to results which should match really well samples from the exact posterior. In Fig. 10, one can get an appreciation of the complexity of the exact posterior, even for such a small and simple model. Each row and column represent one variable with the graph at the intersection showing the samples projected in the 2-dimensional plane spanned by those variables. The samples form perpendicular and diagonal structures, which probably correspond to the two non-equivalent ways of fitting the data. The symmetric lines are caused by the weight-space symmetry. The hyperbolas correspond to equivalent parametrizations of the network under scaling symmetry. Finally, the few outliers that are visibles correspond to a series of small modes in the posterior around the parametrizations fitting the constant function $f(x, y) = 0.5$.

When looking at the actual aggregate results (Fig. 11a), one can see that the average prediction of the BNN fits the data with a regular function similar to a second degree polynomial. The uncertainty, as measured by the standard deviation a posteriori, is low along the lines of the square where the data points lie and increases linearly with distance, creating a diamond shape. Sampling the predicted value for z along a series of lines in the (x, y) plane of coordinates

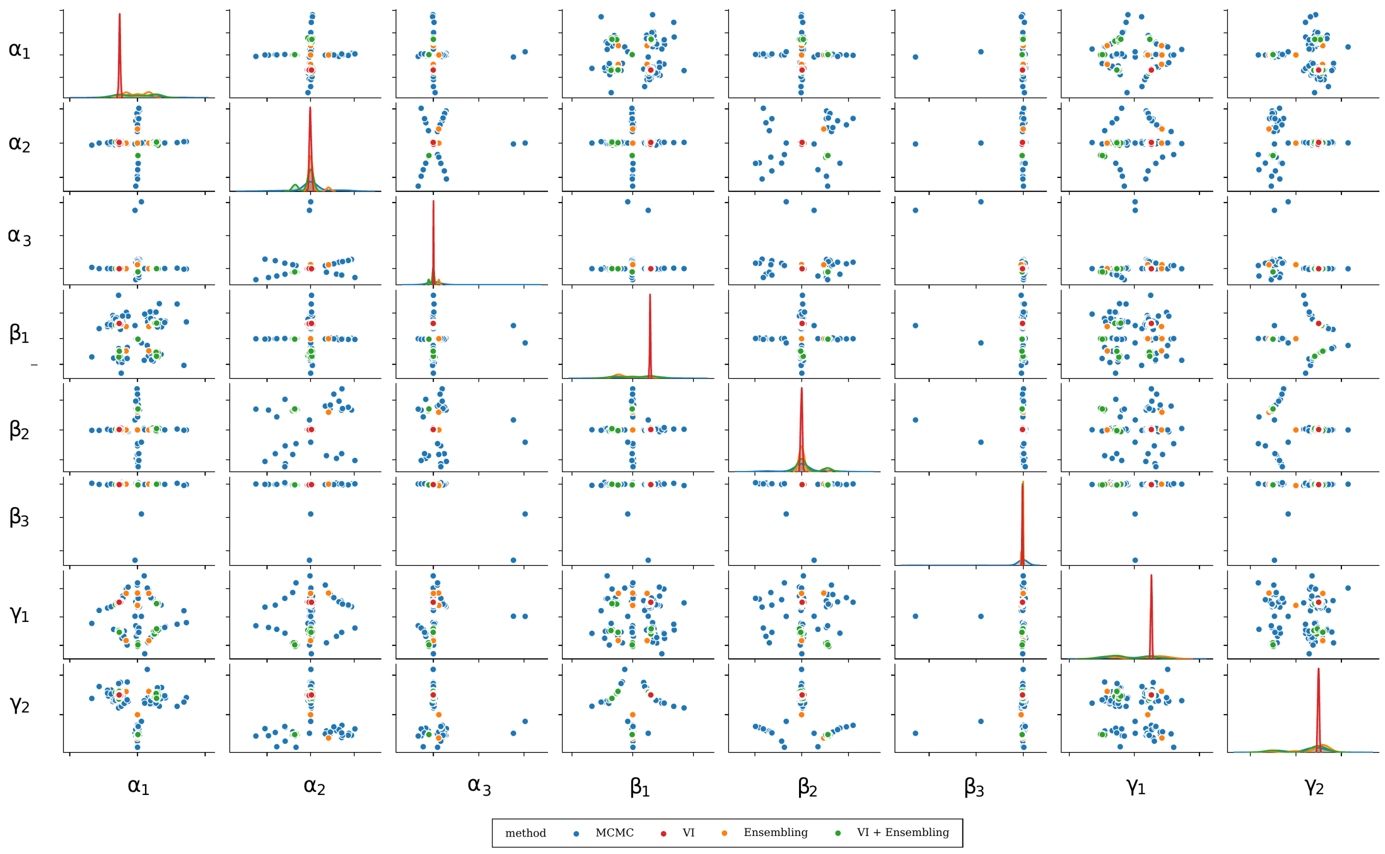
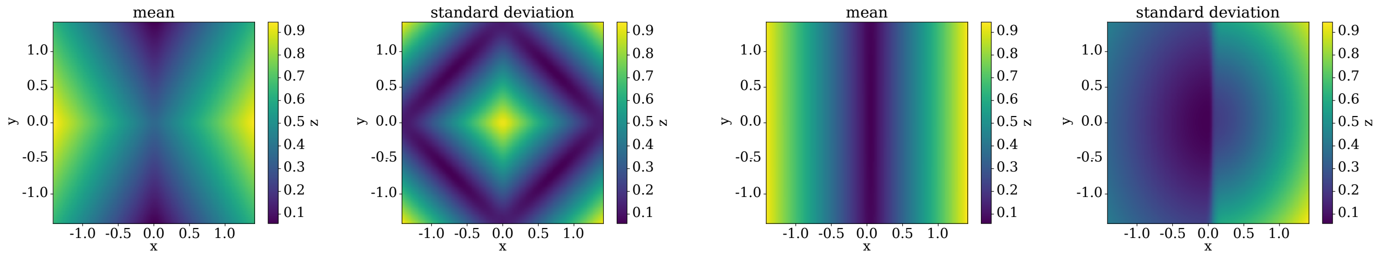


Fig. 10: Pairplot of the samples from the posterior using four different approaches.


(a) (b)

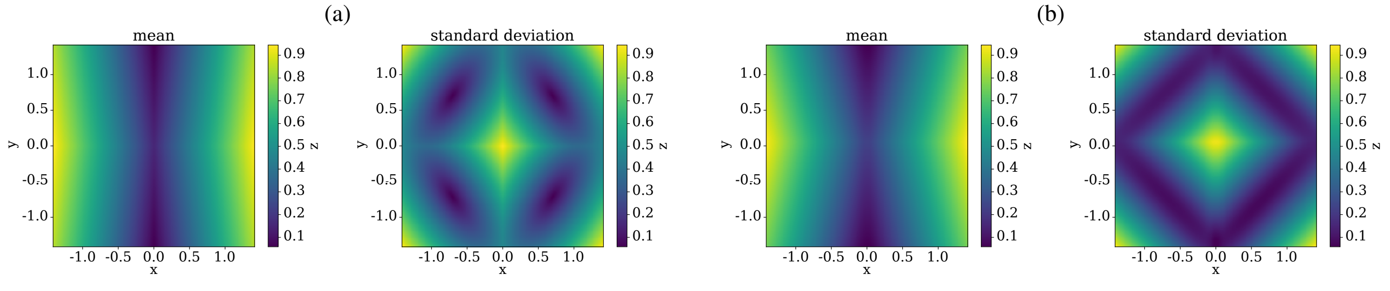
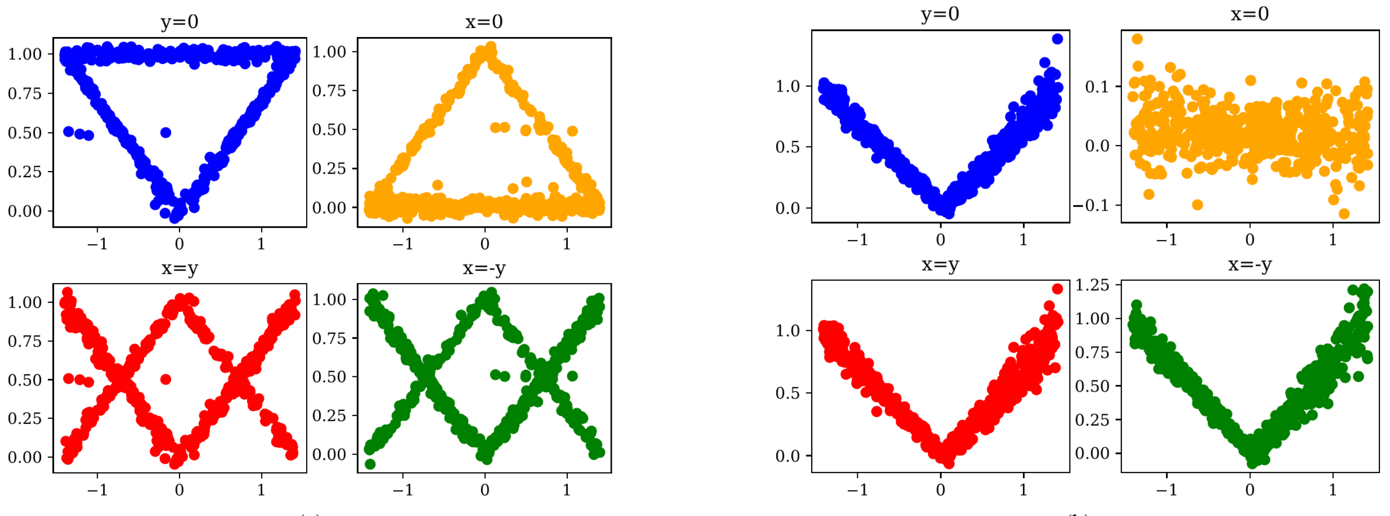

(c) (d)

Fig. 11: Mean and standard deviation of the marginal distribution of z knowing D, x and y for the paperfold model using MCMC (a), variational inference (b), ensembling (c) and variational inference + ensembling (d).


(a) (b)

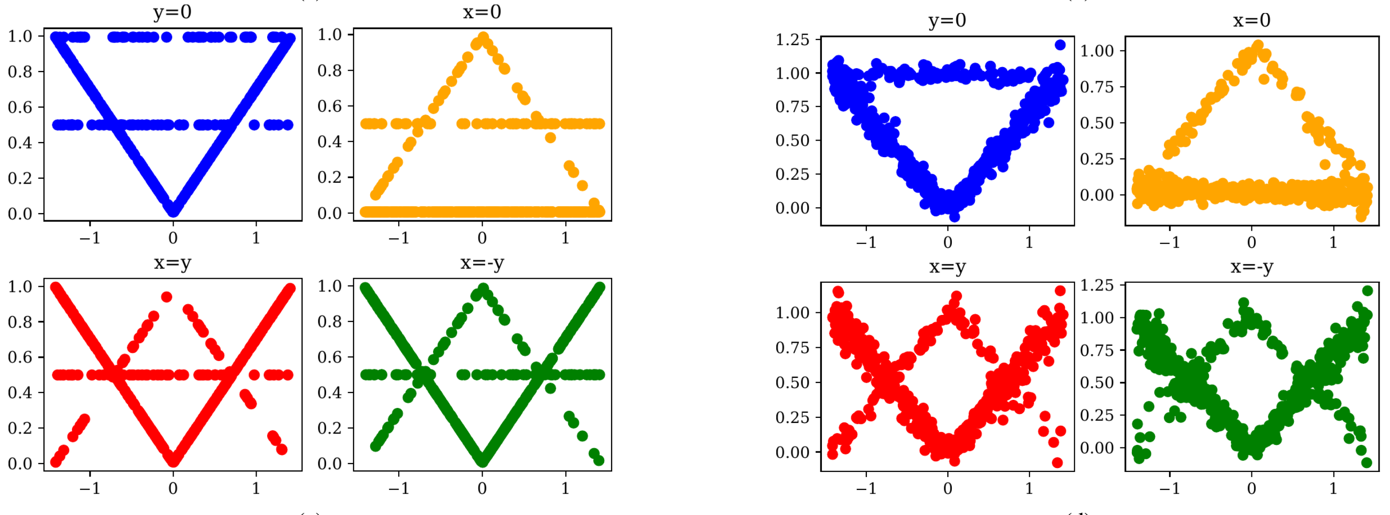
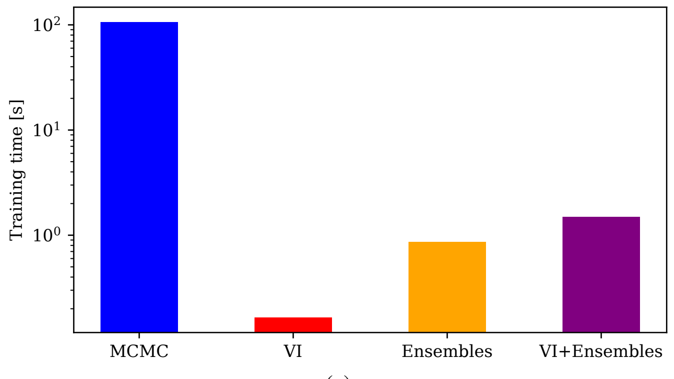
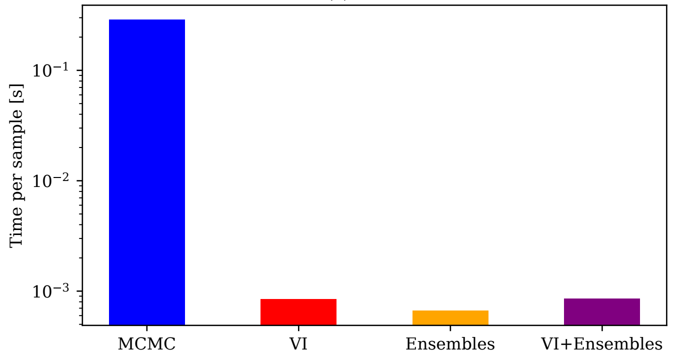

(c) (d)

Fig. 12: Predicted z values across four different lines for the MCMC (a), variational inference (b), ensembling (c) and variational inference + ensembling (d) version of the paperfold BNN.



(a)



(b)

Fig. 13: Comparison between training time (a) and inference time (b) for the paperfold example with standard algorithms for the different training approaches.

(Fig. 12a) shows that both folds appear as clearly distinct from on another, but a bit of uncertainty remains around the exact position of these folds.

2) Variational inference: As a variational inference approximation to the posterior, we used a normal distribution with a learnable mean and a diagonal covariance matrix. It is pretty clear that for this specific example (which, should be noted, has been designed to generate such a specific problem), this method leads to a very poor approximation of the actual posterior, since it can only fit an unimodal distribution (Fig. 10). In terms of the marginal, only one of the two possible folds (Fig. 12b) has been fitted. This translates to a mean estimate for z as well as an uncertainty level and distribution well off the actual posterior (Fig. 11b). The actual side of the fold the RELU branch is fitting is also visible as its uncertainty is higher than on the other side of the fold. This highlights the major limitations of simple variational inference methods for BNN: since the underlying models are complex and the data can be fitted in many non-equivalent manners, simple variational posteriors lack some expressive power to provide a good estimate of the actual BNN uncertainty. It is still important to keep in mind that this example has been designed specifically to make those problems apparent and that in practice mean field Gaussian and similar variational inference approximations can still lead to good estimations of the actual uncertainty (an example is the sparse measure example in Appendix II). Variational inference is also orders of magnitude faster than MCMC (Fig. 13).

3) Ensembling: The most straightforward way to perform ensemble learning with an artificial neural network architecture is just to restart the learning procedure multiple times and each time add the final model to the ensemble. Using this strategy to learn the posterior for the paperfold example already gives quite good results, even better than naive variational inference as the samples can belong to different modes of the posterior (Fig. 10). The marginal mean and standard deviation a-posteriori are clearly different from the ones obtained via MCMC, but the diamond shape can be recongnized in both predictions (Fig. 11c). The training is slightly longer than with basic variational inference as it has to be run multiple times. However, the time per sample when computing the marginal is extremely low (Fig. 13). The main drawback of this approach is that it provides no estimation of the local uncertainty around the different modes of the posterior as shown by Fig. 12c.

4) Variational Inference + Ensembling: Combining the benefits of a simple variational inference and ensembling can be done in a straightforward manner by learning an ensemble of variational approximations. The resulting distribution is a Gaussian mixture, which can approximate the shape of multiple modes of the posterior. While for this example it still has a rough time matching the complex shapes in the exact posterior (Fig. 10), the resulting marginal for z is a very good match with the one generated via a MCMC sampler (Fig. 11 and 12). In addition, the time per sample when computing the marginal is still very similar to the single Gaussian variational approximation (Fig. 13). This shows the huge benefits of ensembling as a tool for approximate Bayesian methods in deep learning.

References

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3642–3649.

[3] M. Belkin, P. Niyogi, and V. Sindhwani, "Manifold regularization: A geometric framework for learning from labeled and unlabeled examples," *J. Mach. Learn. Res.*, vol. 7, pp. 2399–2434, Dec. 2006.