Normalizing Flows for Implicit Bayesian Neural Networks

Student: Weijiang Xiong Supervisor: Markus Heinonen

1 Introduction and Motivation

1.1 Types of Uncertainties in Deep Learning

Deep learning methods have led to significant advances in research on Artificial Intelligence. Typically, we will create a model $\mathcal{M}$ to capture the patterns in a given dataset $\mathcal{D}$. More specifically, for supervised learning tasks, the dataset usually consists of pairs of inputs and labels $(\mathbf{X}, \mathbf{y})$. Thus, the problem can be described as: choosing an optimal set of parameters $\mathbf{w}$ for the model $\mathcal{M}$, so the model prediction $\mathcal{M}_{\mathbf{w}}(\mathbf{X})$ is closest to the label $\mathbf{y}$ under a distance measurement (loss function) $\mathcal{L}$.

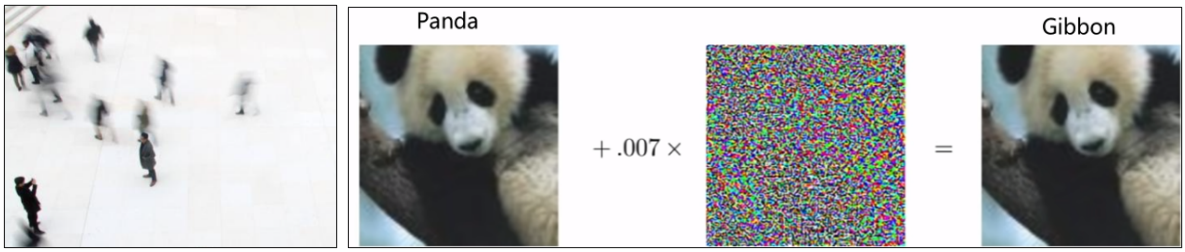
While the optimizing problem of parameter $\mathbf{w}$ can be tackled by error backpropagation and gradient-based approaches, such as ADAM [1], there is no guarantee of an optimal parameter set $\mathbf{w}$. On the contrary, in practice, we will randomly initialize $\mathbf{w}$ and then start a learning algorithm. And it's often the case that the learning algorithm behaves stably, which means if we run the same training program multiple times, the different obtained models will have comparable training losses after several epochs, and also similar accuracies on test data. Those models have different initial parameters, and will end up having different weights after the learning process.

A question that arises is how to make use of those models. In practice, we often pick the one with highest test accuracy, even if those models have almost equal performance. However, with the accuracy on the limited testing data, we are not sure whether the picked model is the *truly best* model that correctly models the true data-generating process. Therefore, in Bayesian Deep Learning, we model this uncertainty with a probability distribution over the weights $p(\mathbf{w}|\mathcal{D})$, which is known as *epistemic uncertainty* or simply *weight uncertainty*. And this kind of neural network is known as Bayesian Neural Network (BNN), which marginalize over the weight posterior to provide predictive distributions [2]:

$$p(\mathbf{y} \mid \mathbf{X}, \mathcal{D}) = \int p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}) p(\mathbf{w} \mid \mathcal{D}) d\mathbf{w}. \tag{1}$$

As we obtain more data, the results on the dataset will be more reliable, and thus weight uncertainty could be explained away with enough data [3].

On the other hand, real-world data are often captured by sensors. For example, images are taken by cameras, and audio waves are recorded by microphones. As a result, sensor noise and motion noise will lead to uncertainties in recorded data. Figure 1 (a) shows an example of motion blur, where the images of some moving people become unclear. This kind of uncertainty is known as *aleatoric uncertainty* or simply *input uncertainty*. Unfortunately, aleatoric uncertainty is inherent in data, and can not be explained away even if we obtain more data [3].



(a) Motion Blur (b) Adversarial Attack

Figure 1

1.2 Motivation for Modelling Uncertainties

In this project, we study the uncertainty modeling problem in the context of image classification, a basic task in computer vision. The motivation comes from two problems in deep neural networks.

First, conventional DNNs are deterministic, because they only learn a single set of parameters, thus providing a point estimation. However, based on the discussions above, uncertainty exists in both the network weight and input data. Moreover, in applications that involve decision-making (such as autonomous driving), we would like to consider all probable results and make sure the decision won't violate any rules or hurt anybody in all these conditions. Thus, uncertainties must be seriously taken into account, and a point estimation won't be sufficient.

Second, DNNs are usually overconfident about their results [4], even if the results are wrong. Figure 1 (b) shows an adversarial example [5], where we mix a carefully chosen noise image to the image of a panda. As a result, the network classifies the mixed image as a gibbon with very high confidence. This error-prone overconfidence makes the output of DNNs even less reliable. Therefore, we would like to learn a model that is *well-calibrated*, which means its confidence should match the actual accuracy. Then the distribution of confidence will reflect the uncertainties of the prediction.

In short, an ideal model should learn the uncertainties of its task and provide trustworthy confidence.

1.3 Project Idea

In theory, we may consider both epistemic uncertainty and aleatoric uncertainty, but previous research has shown that for computer vision tasks, it's more efficient to model input uncertainty rather than weight uncertainty [3, 6]. For a deep neural network with millions of parameters, placing a distribution on weights is computationally expensive, but the input space makes a more suitable option. Because the size of an image or a feature map is usually much smaller than the network itself.

Trinh et al. [6] proposed an implicit Bayesian Neural Network (iBNN) that learns a Gaussian distribution for the input noise in each network layer. Compared to conventional neural networks, iBNN achieves higher classification accuracy and is also better calibrated. Although the variance of Gaussian distribution provides a reasonable estimation for uncertainties, real-world image data have various contents and the true input uncertainty could have more complicated distributions. Therefore, in this project, we will further extend the iBNN with Normalizing Flows [7] to support more flexible distributions.

2 Normalizing Flows for Implicit BNN

2.1 General Network Structure

The general structure of iBNN-style network is similar to a conventional deep neural network. As shown in Figure 2, an iBNN-style network contains a series of deterministic layers and stochastic layers. These two kinds of layers have the same interface, i.e., the same format for input and output, but different internal mechanisms. A deterministic layer refers to the building blocks of conventional neural network, such as convolution layer and linear layer. Meanwhile, A stochastic layer extends a deterministic layer with a stochastic branch that offers a random vector to augment the original input. The workflow can also be expressed with the following equations.

$$\begin{aligned} \mathbf{f}_l(\mathbf{x}) &= layer_l\left(\mathbf{z}_l \circ \mathbf{f}_{l-1}\right) \quad l \geq 1 \\ \mathbf{f}_0 &= \mathbf{x} \\ \mathbf{z}_l &= NF_l(\mathbf{z}_0) \\ \mathbf{z}_0 &\sim q_0(\mathbf{z}) \end{aligned} \tag{2}$$

In the forward pass of a stochastic layer (indexed by l), we first sample a (set of) stochastic vector(s)

$\mathbf{z}_0$ from the base distribution $q_0(\mathbf{z})$ (Base Dist.). Then, the base samples $\mathbf{z}_0$ will go through a stack of Normalizing Flows (NF), which apply a series of invertible transforms to the input samples and output transformed samples $\mathbf{z}_l$. After that, the transformed samples will augment the output of the previous layer, i.e. $\mathbf{f}_{l-1}$, through element-wise multiplication. Finally, we put the augmented input $\mathbf{z}_l \circ \mathbf{f}_{l-1}$ into the deterministic component (Det. Compo) of that stochastic layer, which is literally a deterministic layer.

To represent the flow-based distribution, we will need a number of samples. Therefore an input image will be augmented by different transformed samples, and the output will also consist of different samples, from which we can learn the output uncertainty.

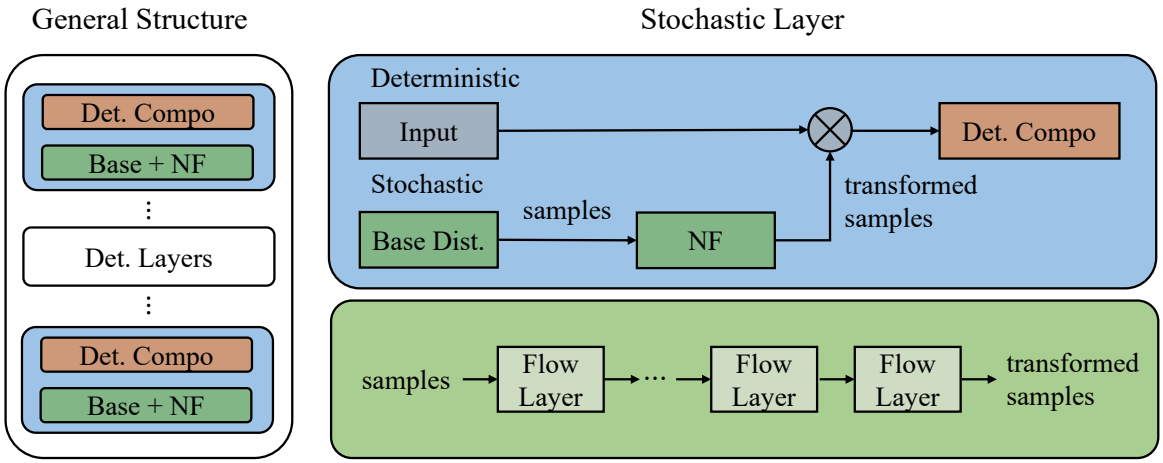


Figure 2: Network Structure

Notably, we assume that each layer has its own stochastic part, which is independent of other layers. Besides, the deterministic and stochastic parts could be trained separately, and we can even migrate the weights for the deterministic part from a pre-trained model. From this perspective, we can also regard the stochastic part as an extension to the original neural network.

2.2 Properties Normalizing Flows

In probabilistic modeling and inference, we often need to find a balance between expressiveness and tractability. While simple distributions, such as Gaussian, have nice analytical properties and are efficient to compute, their expressiveness is also highly limited. Sometimes even if we just move one step more complicated, the problem instantly loses analytical solution. For example, the KL divergence of two Gaussian distributions has closed-form solution, but if we change one of them into a mixture of Gaussian, there won't be a general solution anymore.

Normalizing Flows are a series of bijective (invertible) functions, with which we can construct arbitrarily complex distributions from a simple distribution. In recent years, Normalizing Flows have been successfully applied to density estimation [7] and image generation [8], where the exact value of parameters are learned from data. These research works have shown the expressiveness of NFs, and in this project, we will apply them to image classification tasks and learn the uncertainties of images and feature vectors.

Mathematically, a normalizing flow consists of one or several invertible transforms:

$$\begin{aligned} \mathbf{z}_k &= g_k(g_{k-1}(\ldots g_1(\mathbf{z}_0))) \\ &= g_k \circ g_{k-1} \circ \ldots \circ g_1(\mathbf{z}_0), \end{aligned} \tag{3}$$

where $g_1, \ldots, g_k$ are invertible functions, $\mathbf{z}_0$ is a continuous random variable with simple distribution (such as Gaussian) and $\mathbf{z}_k$ is the transformed random variable with a complex distribution. The key benefit is that we can evaluate the log probability density value of $\mathbf{z}_k$ with the following equation:

$$\log q(\mathbf{z}_k) = \log \left[q_0(\mathbf{z}_0) \prod_{i=1}^{k} |\det J_{g_i}(\mathbf{z}_{i-1})|^{-1} \right] = \log q_0(\mathbf{z}_0) - \sum_{i=1}^{k} \log |\det J_{g_i}(\mathbf{z}_{i-1})| \,. \tag{4}$$

While $\log q_0(\mathbf{z}_0)$ is the log probability of the initial random variable (base distribution), the second term accounts for the transformations made by the k layers of flows. First, $\mathbf{z}_{i-1}$ means the random variable transformed by previous $i-1$ layers of flow, i.e., $\mathbf{z}_{i-1} = g_{i-1} \circ \ldots \circ g_1(\mathbf{z}_0)$. Then, $J_{g_i}(\mathbf{z}_{i-1})$ computes the Jacobian matrix of the i-th layer of flow with respect to its immediate input $\mathbf{z}_{i-1}$. After that, we compute the determinant (det) of the Jacobian matrix, take its absolute value ($|\cdot|$) and then map to log space (log). For a tutorial on normalizing flow, please refer to [9], and for detailed proof please have a look at [10].

But here is a simplified example. Consider a general function $y = f(x)$, where x is a Gaussian random variable. Then a small interval on the x axis $[x, x + dx]$ is mapped to $[y, y + dy]$. If we think of the corresponding probability mass on the two intervals like a physical flow (wind), the total probability mass will stay the same, just like the mass stays the same in a physical flow. Meanwhile, the density and volume could change because of the outer environment. For wind, that would be different pressure in the atmosphere. But the product of density and volume, i.e., mass, must not change. In in our example, density becomes probability density $p(x)$ and $p(y)$, and volume becomes $|dx|$ and $|dy|$. Here we take the absolute value because the volume is always positive, but $|dx|$ and $|dy|$ could be either positive or negative. Therefore we have an equation $p(x)|dx| = p(y)|dy|$, which is even simpler in log space:

$$\log p(y) = \log p(x) - \log |\frac{dy}{dx}|. \tag{5}$$

Similarly, if we have another function $z = g(y)$, then

$$\log p(z) = \log p(y) - \log |\frac{dz}{dy}|. \tag{6}$$

Adding up the two equations we know the log probability density of random variable z:

$$\log p(z) = \log p(x) - (\log |\frac{dz}{dy}| + \log |\frac{dy}{dx}|) \tag{7}$$

In fact, we know the source of z can be traced back to x, because $z = g(y) = g(f(x))$, but the distribution of z could be much more complicated than x because of the changes induced by f and g. If we generalize into vector case from scalar case, the derivatives $\frac{dz}{dy}$ and $\frac{dy}{dx}$ will also become Jacobian matrices. Then we will have Equation 4.

2.3 Examples of Flow

Although Equation 4 provides a straightforward method for density evaluation, the Jacobian determinant $\det J_{g_i}(\mathbf{z}_{i-1})$ in the expression makes it computationally expensive. Specifically, a recent research on matrix multiplication has indicated a complexity at $\mathcal{O}(n^{2.37})$ [11]. Therefore, we still need to carefully choose the functions in the flow stack to have feasible determinant computation. In this section, we will briefly introduce three kinds of flows with tractable Jacobian determinant: planar flow [7], affine coupling flow [12] and 1×1 invertible convolution [8].

Planar Flow belongs to the Residual Flow family [10], which means the transformed vector $\mathbf{z}'$ is a sum of the original vector $\mathbf{z}$ and a residual term. Equation 8 shows the transform, Jacobian matrix and the Jacobian determinant of planar flow. Notably, we need to use the Matrix Determinant Lemma

to get the final expression of Jacobian determinant.

$$\begin{aligned} \mathbf{z}' &= \mathbf{z} + \mathbf{v}\sigma\left(\mathbf{w}^\top \mathbf{z} + b\right) \\ J_g(\mathbf{z}) &= \mathbf{I} + \sigma'\left(\mathbf{w}^\top \mathbf{z} + b\right)\mathbf{v}\mathbf{w}^\top \\ \det J_g(\mathbf{z}) &= 1 + \sigma'\left(\mathbf{w}^\top \mathbf{z} + b\right)\mathbf{w}^\top \mathbf{v} \end{aligned} \tag{8}$$

Affine Coupling Flow splits the input vector into two groups, keep one group unchanged, and apply element-wise transform to the other group.

$$\begin{aligned} \mathbf{z}'_{1:d} &= \mathbf{z}_{1:d} \\ \mathbf{z}'_{d+1:D} &= \mathbf{z}_{d+1:D} \odot \exp\left(s\left(\mathbf{z}_{1:d}\right)\right) + t\left(\mathbf{z}_{1:d}\right) \end{aligned} \tag{9}$$

Here, the first d elements $\mathbf{z}_{1:d}$ belongs to a group, and the rest $\mathbf{z}_{d+1:D}$ belongs to another. Meanwhile, $s(\cdot)$ and $t(\cdot)$ can be any functions. As a result, the Jacobian matrix will be triangular (Equation 10), and thus the determinant will be the product of diagonal elements. Usually, we need the log of determinant, which turns out to be the sum of elements in $s(\mathbf{z}_{1:d})$.

$$\begin{aligned} J_g(\mathbf{z})^\top &= \begin{bmatrix} \mathbf{I}_d & 0 \\ \frac{\partial \mathbf{z}'_{d+1:D}}{\partial \mathbf{z}_{1:d}} & \operatorname{diag}\left(\exp\left[s\left(\mathbf{z}_{1:d}\right)\right]\right) \end{bmatrix} \\ \log\det J_g(\mathbf{z}) &= \operatorname{sum}(s(\mathbf{z}_{1:d})) \end{aligned} \tag{10}$$

1×1 **Invertible Convolution** is specially adapted for image data, and applies a shared weight matrix $\mathbf{W}$ across the height and width dimensions. If we have an input image or feature map with size (C, H, W), the size of weight matrix will be $C \times C$. Then, for each length-C feature vector $\mathbf{z}_{i,j}$ across the height and width dimensions, we have the following equations:

$$\begin{aligned} \mathbf{z}'_{i,j} &= \mathbf{W}\mathbf{z}_{i,j} \\ J_g(\mathbf{z}_{i,j}) &= \mathbf{W} \\ \det J_g(\mathbf{z}_{i,j}) &= \det \mathbf{W} \end{aligned} \tag{11}$$

2.4 Network Optimization

For this part, we will leverage the idea of variational inference [7], which is learning a flow-based distribution that best approximates the true distribution. The general problem formulation is the same as the original iBNN [6], but the flow part has made some difference in the details.

Loss Formulation

Let's assume the parameters θ of the deterministic part have already been trained, or have been migrated from a pre-trained model. With a dataset $\mathcal{D}$, we now want to learn the input uncertainty of each layer, which is described by the posterior distribution of latent random variables $\mathbf{z}_1, ..., \mathbf{z}_l$. That is, we want to know the posterior distribution of the latent variables $p(\mathbf{z}_{1:l}|\mathcal{D};\theta)$, but the true model is so complicated that we decide to use a flow-based posterior $q(\mathbf{z}_{1:l};\theta)$ to approximate it. To this end, we minimize the KL divergence between the true posterior $p(\mathbf{z}_{1:l}|\mathcal{D};\theta)$ and its approximation $q(\mathbf{z}_{1:l};\theta)$:

$$\begin{aligned} KL[q(\mathbf{z}_{1:l};\theta)||p(\mathbf{z}_{1:l}|\mathcal{D};\theta)] &= \mathbb{E}_q[\ln q(\mathbf{z}_{1:l})] - \mathbb{E}_q[\ln p(\mathbf{z}_{1:l}|\mathcal{D})] \quad \text{(omit } \theta \text{ for simplicity)} \\ &= \mathbb{E}_q[\ln q(\mathbf{z}_{1:l})] - \mathbb{E}_q[\ln \frac{p(\mathcal{D}|\mathbf{z}_{1:l})p(\mathbf{z}_{1:l})}{p(\mathcal{D})}] \quad \text{(Bayes's rule)} \\ &= \mathbb{E}_q[\ln q(\mathbf{z}_{1:l})] - \mathbb{E}_q[\ln p(\mathcal{D}|\mathbf{z}_{1:l})] - \mathbb{E}_q[\ln p(\mathbf{z}_{1:l})] + \mathbb{E}_q[\ln p(\mathcal{D})] \\ &= KL[q(\mathbf{z}_{1:l})||p(\mathbf{z}_{1:l})] - \mathbb{E}_q[\ln p(\mathcal{D}|\mathbf{z}_{1:l})] + \mathbb{E}_q[\ln p(\mathcal{D})]. \end{aligned} \tag{12}$$

By rearranging terms, the equation becomes

$$\mathbb{E}_q[\ln p(\mathcal{D})] = \underbrace{\mathbb{E}_q[\ln p(\mathcal{D}|\mathbf{z}_{1:l})] - KL[q(\mathbf{z}_{1:l})||p(\mathbf{z}_{1:l})]}_{ELBO} + KL[q(\mathbf{z}_{1:l};\theta)||p(\mathbf{z}_{1:l}|\mathcal{D};\theta)]. \tag{13}$$

Because KL divergence is always larger than 0 and the probability of dataset is constant, minimizing the KL term is equivalent to maximizing ELBO. Since we assume different layers have independent stochastic parts, $\mathbf{z}_1, ..., \mathbf{z}_l$ are independent, then $q(\mathbf{z}_{1:l})$ could be factorized into a product $q(\mathbf{z}_{1:l}) = \prod_{i=1}^{l} q(\mathbf{z}_i)$. Further, we can decompose the second term in $ELBO$ into a sum of $\ln p(\mathbf{z}_i)$, where we use β-weighted KL term to provide a more flexible bound:

$$ELBO = \mathbb{E}_q[\ln p(\mathcal{D}|\mathbf{z}_{1:l})] - \beta \sum_{i=1}^{l} KL[q(\mathbf{z}_i)||p(\mathbf{z}_i)]. \tag{14}$$

In this equation, the first term represents data likelihood (averaged over samples), which can be calculated directly with the prediction results. Specifically, if we capture the likelihood with a categorical distribution, and the network outputs are processed by a Softmax function, the log likelihood is literally equal to the cross entropy loss [13]. The second term is the KL divergence between the flow based posterior and the prior, which controls the complexity of the flows.

KL Divergence Let's analyze the KL of just one layer. For simplicity, we remove the layer index and use a subscript to indicate the level of flow (k in total). Take care that the prior is chosen for the *transformed samples* $\mathbf{z}_k$ not the *base samples* $\mathbf{z}_0$, although they could be the same distribution. Then, the KL term of a layer could be expressed as follows:

$$\begin{aligned} KL[q(\mathbf{z}_k)||p(\mathbf{z}_k)] &= \mathbb{E}_q[\log q(\mathbf{z}_k)] - \mathbb{E}_q[\log p(\mathbf{z}_k)] \\ &= \mathbb{E}_{q_0}\left[\log q_0(\mathbf{z}_0) - \sum_{i=1}^{k} \log|\det J_{g_i}(\mathbf{z}_{i-1})|\right] - \mathbb{E}_{q_0}[\log p(\mathbf{z}_k)] \\ &= \mathbb{E}_{q_0}[\log q_0(\mathbf{z}_0)] - \mathbb{E}_{q_0} \sum_{i=1}^{k} \log|\det J_{g_i}(\mathbf{z}_{i-1})| - \mathbb{E}_{q_0}[\log p(\mathbf{z}_k)] \end{aligned} \tag{15}$$

We first expand the KL divergence by definition, and then plug in the property of normalizing flows in Equation 4. Meanwhile, we can use the change of variable technique to move the expectation from flow posterior q to the base distribution q_0. But in practice, they are both the arithmetic average over samples. As a result, the first term is the log probability of initial samples on the **base** distribution. The second term reflects the property of the learned flow, and can be named "log det jacobian". The third term is the log probability of transformed samples on the **prior** distribution.

While it's tempting to write the first and third term as $KL[q_0||p]$, it's actually wrong, because q_0 and p does not apply to the same random variable. When we write $KL[q_0||p]$, it in fact means, the two distributions should share the same variable $\mathbf{z}$, which is then integrated out.

$$KL[q_0||p] = \int q_0(\mathbf{z}) \log \frac{q_0(\mathbf{z})}{p(\mathbf{z})} d\mathbf{z} \tag{16}$$

However, that is not our case, because q_0 is the base distribution and applies to the base random variable $\mathbf{z}_0$, while p is placed on the transformed variable $\mathbf{z}_k$.

In summary, with Equation 14 and 15, we can add up the KL divergence layer by layer, and combine it with the data likelihood term.

3 Experiments and Analysis

Expected Calibration Error (ECE) [4] measures how much the model confidence deviates from the test accuracy, and we will use this metric in our experiments to evaluate the calibration of our models. For detailed derivation, please refer to [4], but here we provide a short description. Generally, the ECE is a weighted average over the absolute difference between the confidence and accuracy, and the weight is the proportion of samples whose confidence lies within a range. After obtaining the prediction results for all data, we first evenly divide the interval $[0, 1]$ into n bins, and assign the results to these bins according to the prediction confidence. Then we calculate the average confidence $Conf$ and actual classification accuracy Acc for each bin, and the ECE could be expressed by:

$$ECE = \sum_{i=1}^{n} \frac{1}{B_i} |Acc - Conf|, \tag{17}$$

where B_i is the percentage of samples in the i-th bin.

Throughout the experiments, we will compare the performance of a basic neural network and its corresponding (flow-based) stochastic variant. That is, the stochastic model is obtained by extending one or more layers in the basic model into stochastic layers. Meanwhile, the deterministic parts of the stochastic model are directly migrated from the basic model.

3.1 2D Classification Example

In this section, we demonstrate the effect of flow with a simple 2D classification example. The three plots in the first row show the classification results, sample percentage for each confidence range, and the confidence-accuracy gap of a basic 2-layer MLP model. And the second row shows the corresponding properties of a 2-layer MLP with stochastic layers.

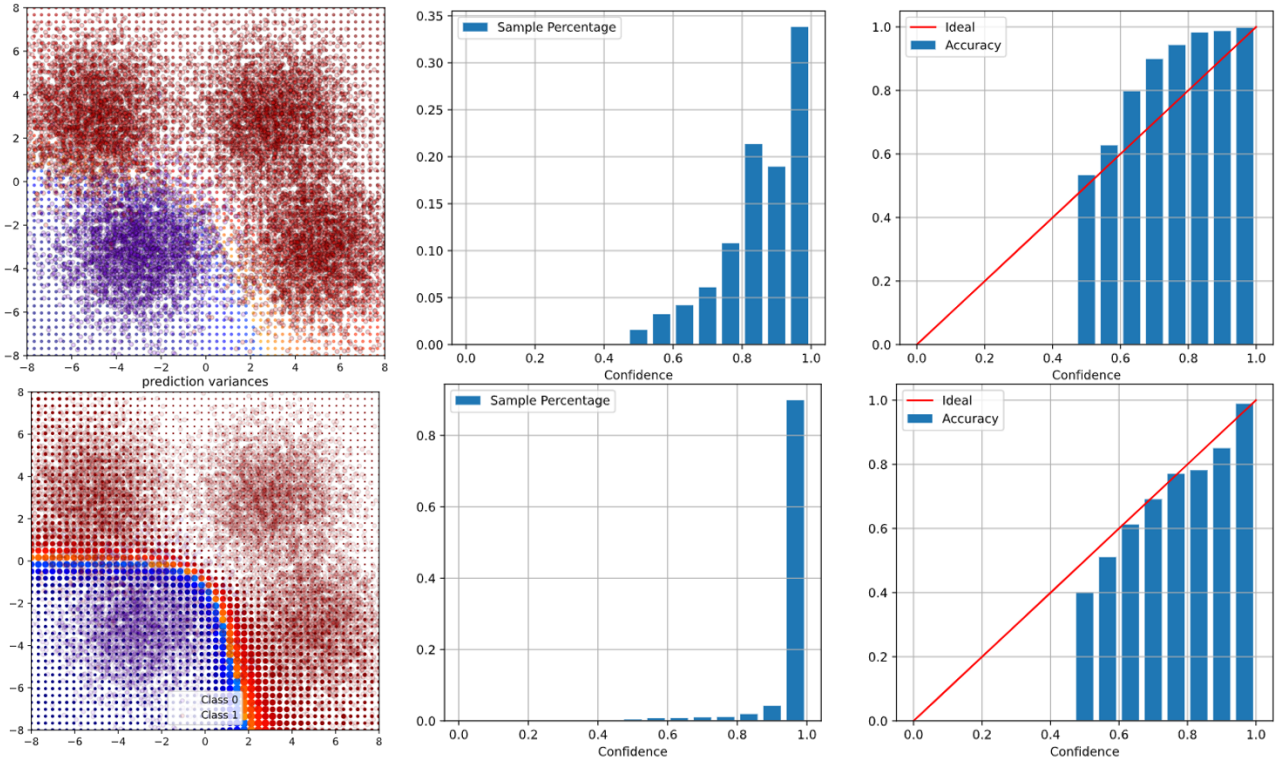


Figure 3: Classification example for basic MLP (first row) and stochastic MLP (second row)

The upper left plot in Figure 3 shows the data clusters and the decision boundary of a basic 2-layer MLP. The red clusters have some overlap with the purple cluster, and the color of the grid points indicates the classification results. We can see the grid points near the purple cluster are assigned blue

color, while the rest are assigned to red color. The decision boundary is a curved line that roughly splits the two classes, and the classification accuracy is 95.3%. Meanwhile, the lower-left plot shows the results from a 2-layer stochastic MLP. A clear difference is a measurement of uncertainty, which is indicated by the size of the grid points. Since the stochastic model is migrated from a deterministic one, the stochastic boundary is very close to the deterministic one. From the plot, we observe reasonable high uncertainty near the boundary, and low uncertainty elsewhere.

In the plots for confidence-accuracy gap, the red line represents the ideal case, where the confidence perfectly matches the actual accuracy. From the plots, we observe better calibration results in the stochastic model (lower right plot). The main reason is the stochastic model correctly classified over 80% of data points, with matching confidence. As a result, the deterministic MLP has an ECE at 0.094, while the stochastic model has only 0.009.

To investigate the effect of flow, we compare the training progress of stochastic MLPs with different flow lengths. Figure 4 shows the trends of ELBO, likelihood and KL divergence at each training epochs. For this example, we use planar flow, and only change the length. As a special case, length 0 means the stochastic part has merely a learnable Gaussian distribution. From the plots, we observe that flow length 12 results in the highest likelihood and ELBO, and flow length 0 gives the lowest values. Therefore, the flow part with a reasonable length (12 here) could improve the model performance.

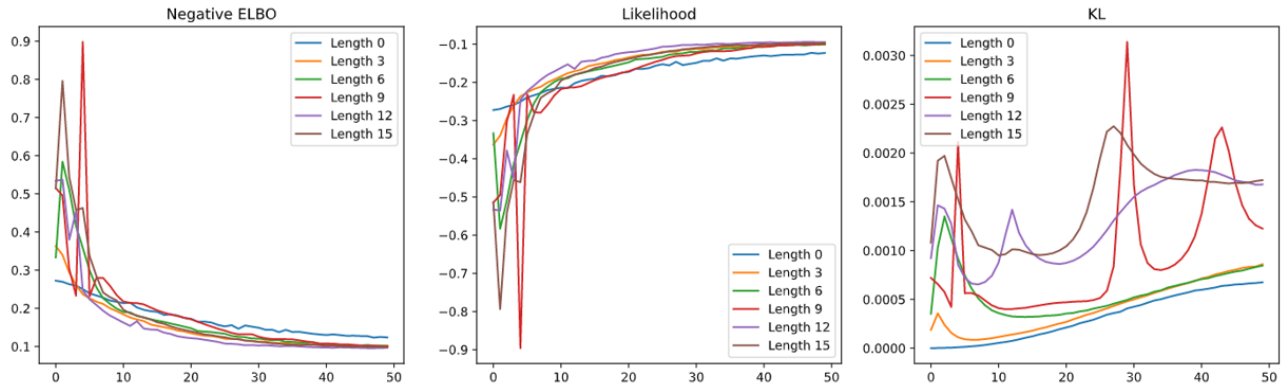


Figure 4: Training progress of stochastic MLPs with different flow lengths (x axis is epoch)

3.2 Image Classification

In this section, we compare flow-based LeNet [14] and VGG [15] with their deterministic baseline. Since images are high-dimensional data, we don't have good methods to visualize the output uncertainty. But we can still observe the training progress in Figure 5 and Figure 6.

Figure 5 presents the training progress of LeNet-based models on FashionMnist [16] and CIFAR-10 [17]. The three plots in the first row show the trend of ELBO, log-likelihood and KL Divergence. Compared to the "no flow" version, the flow-based model has higher data likelihood, higher KL divergence and larger ELBO for both datasets. Same as before, the models labeled with "no flow" have only a learnable Gaussian distribution in the stochastic part of each layer. Higher data likelihood means the flow-based models can better fit the data, and higher KL divergence indicates that the flow-based posterior deviates farther from the prior (Gaussian here), compared to the learnable Gaussian.

The two plots in the second row record the testing accuracy and ECE in each training epoch. Both the flow-based model and its "no flow" version have outperformed the deterministic baseline after a few training epochs. Compared to the baseline, the improvements of stochastic models on CIFAR-10 are more evident than those on FashionMnist. A possible reason is the capacity of basic LeNet is enough to handle the complexity of FashionMnist (nearly 90% baseline accuracy). Therefore the extra capacity introduced by the flows won't have significant effects. If we move to CIFAR-10, the effects of

flows begin to manifest as the classification problem becomes more difficult. And the flow-based model has about 2% higher accuracy than the "no flow" version. However, the expected calibration loss is more unpredictable. While the baseline model is nicely calibrated, with 1.5% ECE on both datasets, the stochastic models could have even higher ECE.

Besides, the input augmentation in our stochastic layer has similar mechanisms to Dropout. But the difference is, dropout applies binary random variables to the input, while our stochastic layer adopts real-valued augmentation. Since the basic LeNet does not have dropout layers, we also extended it with dropout and recorded the best accuracy and ECE as another baseline. As a result, dropout didn't improve the classification accuracy, and the ECE (up to 5%) is also higher than the basic LeNet.

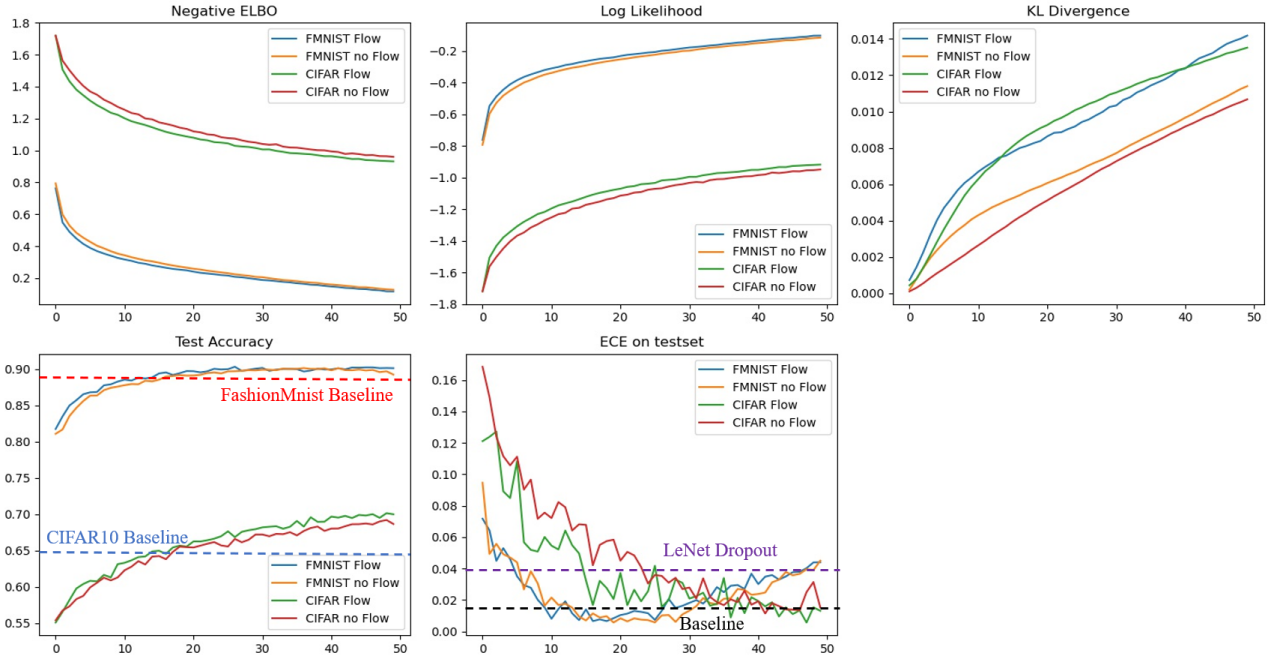


Figure 5: Training progress of basic and stochastic LeNet (x axis is epoch)

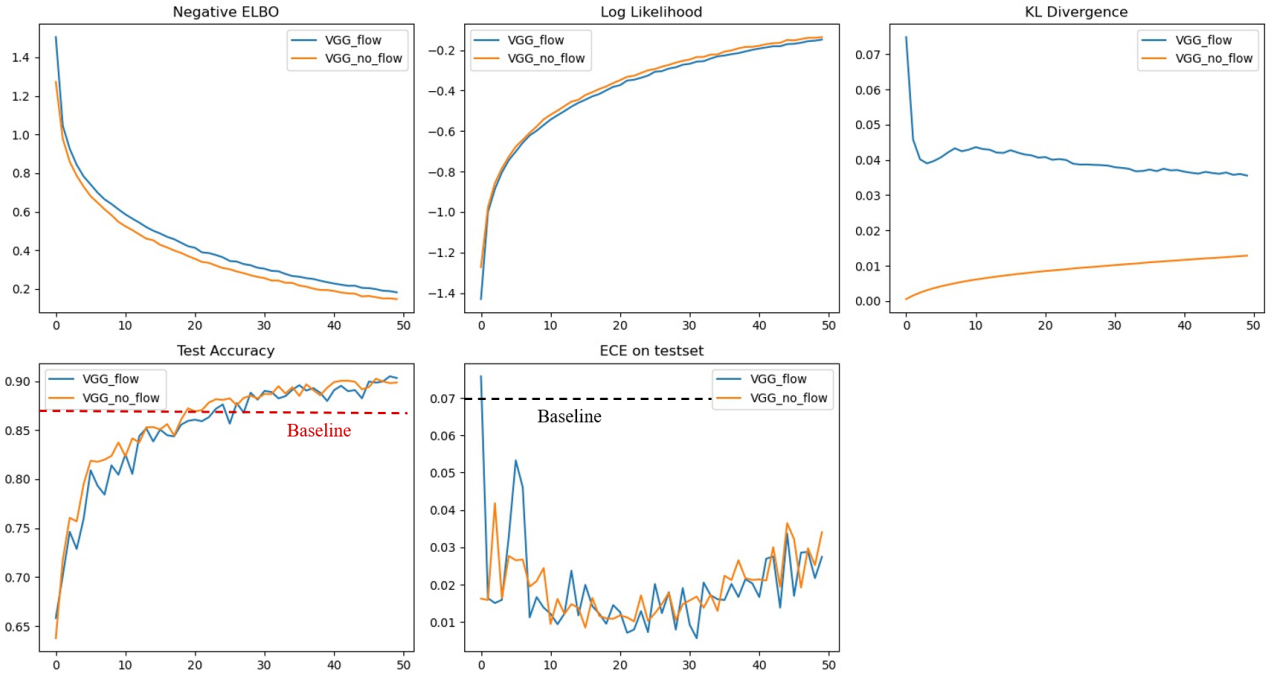


Figure 6: Training progress of basic and stochastic VGG16 (x axis is epoch)

Figure 6 shows the training progress of basic and stochastic VGG16. From the plots, we can see the flow part has almost no effect, although both stochastic models have higher accuracies and lower ECE compared to baseline. Again, these experiment results are also in favor of our conjecture above. The basic VGG16 model has enough capacity itself, because it can reach a high testing accuracy (93.15%) on CIFAR10 [18]. Therefore, the flow part, which introduces extra complexity, won't have a substantial effect on VGG16. Another evidence is the KL divergence in the flow-based model, which decreases as training goes on. This trend means the training process is gradually leading the flow posterior back to the prior, which indicates the flow part is not necessarily helpful for higher classification accuracy. As for ECE, both the flow-based model and "no flow" model have generally smaller ECE than the deterministic baseline.

4 Implementation Details

The Stochastic Part in a stochastic layer could be very flexible, and in fact, we can even regard it as a small neural network. We can choose the building blocks for the flow, the depths of all blocks, and which layers to use the flows. To prevent too much overhead on the flow choices, we predefined a few basic blocks that could be used as the stochastic part of a stochastic layer. For example:

- NormalAffine: a normal base distribution with an affine transformation (learnable Gaussian)
- NormalPlanar1d: add 2 layers of 1d planar flows after NormalAffine block
- NormalInvConv: based on a NormalAffine block, sequentially add 2 layers of 2D planar flow, 3 invertible 1×1 convolution and another 2 layers of 2D planar flow.

For detailed definition of flows, please refer to the flow folder of the codes. In the "no flow" version of stochastic VGG and LeNet, we actually used "NormalAffine" for all the layers. Since the flow part will introduce extra computation, we can not afford to use deep flows for all layers, and therefore we just extend part of the layers with flows. For LeNet model (3 conv + 2 linear), we used NormalInvConv for the second conventional layer, and NormalPlanar1d for the first linear layer. For VGG16 (13 conv + 3 linear), we used NormalInvConv at the 4th, 8th and 11th layers, and NormalPlanar1d at the second linear layer.

2D Planar Flow is an adaptation for image data. The idea is quite similar to invertible 1×1 convolution in Glow, that is using shared parameters across the width and height dimensions. Notably, the original Glow implementation considers the whole image/feature map as a vector, and the features at different height and width positions are independent parts in the vector, resulting in H*W parts in total. The joint probability of the whole vector will be the multiplication of all H*W parts, and thus it becomes a sum in the log space. Therefore the original Glow implementation multiplied the log determinant of the shared matrix $\mathbf{W}$ with a coefficient H*W. But in our implementation for both Glow and 2D Planar Flow, we regard each position as an individual vector, and consider them separately. Thus, we don't multiply the log determinant by a coefficient.

Samples are used to describe the base distribution and flow-based posterior, both in training and testing. During training, we use sample size 1, and in testing 128.

Numerical Problem was found when trying to use the "step of flow" block from the Glow paper. The problem may come from affine coupling layer, but the exact cause is yet not found.

5 Discussions

In this project, we applied normalizing flows to learn a flexible distribution for the input uncertainty of a neural network. The flow-based stochastic parts could provide a reasonable estimation of uncertainty and increase the capacity of the model. However, if the basic model already has sufficient capacity, the flow part may not help increase the classification accuracy. This project belongs to Bayesian Deep Learning, which is still an evolving research domain, and there remain some unsolved problems and

possible ways for improvements.

First is suitable benchmarks and metrics. Back in the problem formulation stage, we started by approximating the true posterior with a flow-based distribution in Equation 12. Therefore, the best criterion to evaluate the flow posterior is to see how well it approximates the true posterior. However, right now we don't have a good benchmark or metric to evaluate the quality of the posterior, and have to resort to classic metrics like accuracy and ECE.

Then, we also need a better way to exploit the output distribution. In our classification problem, each of the 10 classes has a corresponding output distribution, from which we compute the mean and variance of the class. However, the distribution itself contains more information than mean and variance, and it would be better if we can find a way to extract more useful statistics from the distribution.

Besides, the posterior often contracts to a single point, and the flow will just help the network overfit the dataset. To encourage the flow to learn a diverse distribution, we can add the entropy of posterior to ELBO in 14:

$$NEW\_LOSS = \mathbb{E}_q[\ln p(\mathcal{D}|\mathbf{z}_{1:l})] - \beta \sum_{i=1}^{l} KL[q(\mathbf{z}_i)||p(\mathbf{z}_i)] + \gamma \sum_{i=1}^{l} H(q(\mathbf{z}_i)). \tag{18}$$

In fact, this change can be covered in the KL term. In the first step in Equation 15, we expand the KL term by definition, and the first term is exactly the negative entropy of the flow posterior. As a general notation, we have

$$KL[q||p] = cross\_entropy(q, p) - entropy(q). \tag{19}$$

Therefore, if we want to emphasize the entropy of the flow posterior, we can simply multiply the first two terms in 15 with an coefficient larger than 1.

References

[1] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980.

[2] A. G. Wilson, P. Izmailov, Bayesian deep learning and a probabilistic perspective of generalization, arXiv preprint arXiv:2002.08791.

[3] A. Kendall, Y. Gal, What uncertainties do we need in bayesian deep learning for computer vision?, arXiv preprint arXiv:1703.04977.

[4] C. Guo, G. Pleiss, Y. Sun, K. Q. Weinberger, On calibration of modern neural networks, in: International Conference on Machine Learning, PMLR, 2017, pp. 1321–1330.

[5] I. Goodfellow, Adversarial examples and adversarial training, CS 231N Lecture Slides, Stanford University (2017).

[6] T. Trinh, S. Kaski, M. Heinonen, Scalable bayesian neural networks by layer-wise input augmentation, arXiv preprint arXiv:2010.13498.

[7] D. Rezende, S. Mohamed, Variational inference with normalizing flows, in: International conference on machine learning, PMLR, 2015, pp. 1530–1538.

[8] D. P. Kingma, P. Dhariwal, Glow: Generative flow with invertible 1x1 convolutions, arXiv preprint arXiv:1807.03039.

[9] E. Jang, Normalizing flows tutorial, https://blog.evjang.com/2018/01/nf1.html (2018).

[10] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, B. Lakshminarayanan, Normalizing flows for probabilistic modeling and inference, arXiv preprint arXiv:1912.02762.

[11] J. Alman, V. V. Williams, A refined laser method and faster matrix multiplication, in: Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, 2021, pp. 522–539.

[12] L. Dinh, J. Sohl-Dickstein, S. Bengio, Density estimation using real nvp, arXiv preprint arXiv:1605.08803.

[13] F. K. Gustafsson, M. Danelljan, T. B. Schon, Evaluating scalable bayesian deep learning methods for robust computer vision, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops, 2020, pp. 318–319.

[14] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE 86 (11) (1998) 2278–2324.

[15] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, arXiv preprint arXiv:1409.1556.

[16] H. Xiao, K. Rasul, R. Vollgraf, Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, arXiv preprint arXiv:1708.07747.

[17] A. Krizhevsky, G. Hinton, et al., Learning multiple layers of features from tiny images.

[18] SeHwanJoo, cifar10-vgg16, https://github.com/SeHwanJoo/cifar10-vgg16 (2020).