DIFFERENTIABLE BAYESIAN NEURAL NETWORK INFERENCE FOR DATA STREAMS

Anonymous authors
Paper under double-blind review

ABSTRACT

While deep neural networks (NNs) do not provide the confidence of its prediction, Bayesian neural network (BNN) can estimate the uncertainty of the prediction. However, BNNs have not been widely used in practice due to the computational cost of predictive inference. This prohibitive computational cost is a hindrance especially when processing stream data with low-latency. To address this problem, we propose a novel model which approximate BNNs for data streams. Instead of generating separate prediction for each data sample independently, this model estimates the increments of prediction for a new data sample from the previous predictions. The computational cost of this model is almost the same as that of non-Bayesian deep NNs. Experiments including semantic segmentation on real-world data show that this model performs significantly faster than BNNs, estimating uncertainty comparable to the results of BNNs.

1 INTRODUCTION

While deterministic neural networks (DNNs) surpass human capability in some area in terms of prediction accuracy (He et al., 2015; Silver et al., 2016; Ardila et al., 2019), it has been unable to estimate the uncertainty of the predictions until recently. Since the prediction can not be perfect and the misprediction might result in fatal consequences in areas such as medical analysis and autonomous vehicles control, estimating uncertainty as well as predictions will be crucial for the safer application of machine learning based systems.

Bayesian neural network (BNN), a neural network (NN) that uses probability distributions as weights, estimates not only predictive results but also uncertainties. This allows computer systems to make better decisions by combining uncertainty with prediction. Moreover, BNN can achieve high performance in a variety of fields, e.g. image recognition (Kendall et al., 2015; Kendall & Gal, 2017), language modeling (Fortunato et al., 2017), reinforcement learning (Kahn et al., 2017; Osband et al., 2018), meta-learning (Yoon et al., 2018; Finn et al., 2018), and multi-task learning (Kendall et al., 2018), by exploiting uncertainty.

Although BNNs have these theoretical advantages, they have not been used as a practical tool. The predictive inference speed has detained BNNs from wide applications. BNN executes NN inference for dozens of samples from weight distributions. Since sampling and multiple NN executions are difficult to be parallelized, the inference execution takes an order of magnitude more time. Particularly, this is a significant barrier for processing data streams with low-latency.

Most time-varying data streams change continuously, and so do the predictions of BNNs. Thus, we estimate prediction by calculating the increments between two consecutive results, instead of calculating the separate prediction for each input data. This is equivalent to calculating the differentiation of the BNN's prediction for arbitrary data, because the difference of prediction for the new data is the line integration of the gradient of prediction over the new data.

In this work, we propose a *differentiable BNN* (DBNN) inference with respect to an input data. The prediction of the DBNN is given by a Monte Carlo (MC) estimator for the distribution of data streams and weights. We speed up the inference by approximating the distribution using histogram and calculating the gradient for this MC estimator. We show that the time complexity of this model is nearly the same as that of deep DNNs. We evaluate DBNN with semantic segmentation using road scene video sequences and the results show that DBNN has almost no degradation in computational performance compared to deep DNNs. The uncertainty predicted by DBNN is comparable to that of BNN in various situations.

The main contributions of this work are as follows.

- We propose online codevector histogram that estimates the probability of a high-dimensional data stream. Then, we show that this histogram can be used to obtain the MC gradient estimation.
- We propose differentiable Bayesian neural network inference with respect to input data as an approximation of Bayesian neural network inference for data streams. This model is nonparametric and applied to trained BNN without significant modifications.
- We theoretically and empirically show that the computational performance of DBNN is almost the same as that of deep DNNs.

2 Background and Related Work

BNNs are a state-of-the-art method to estimate predictive uncertainty while DNNs based approaches have been developed recently (Lakshminarayanan et al., 2017; Guo et al., 2017). BNNs with probability distributions as weights produce probabilistic results. To make prediction, BNNs sample from the weight probabilities and performs DNN for each sample. In this section, we describe the details and challenges of BNNs inference process.

2.1 Bayesian Neural Network Inference

Suppose that $p(\boldsymbol{x}^1)$ is posterior probability of NN weights $\boldsymbol{x}^1$ and $p(\boldsymbol{y}|\boldsymbol{x}^0, \boldsymbol{x}^1)$ is the BNN model for input data $\boldsymbol{x}^0$. Then, the inference result of BNN is a predictive distribution:

$$p(\boldsymbol{y}|\boldsymbol{x}^0_\star) = \int p(\boldsymbol{y}|\boldsymbol{x}^0_\star, \boldsymbol{x}^1) p(\boldsymbol{x}^1) d\boldsymbol{x}^1 \tag{1}$$

where $\boldsymbol{x}^0_\star$ is observed input data vector and $\boldsymbol{y}$ is output vector. For simplicity, BNNs are usually modeled to have a probability distribution with the mean of the prediction of DNN, e.g. in Hernández-Lobato & Adams (2015); Gal & Ghahramani (2016):

$$p(\boldsymbol{y}|\boldsymbol{x}^0_\star, \boldsymbol{x}^1) = \mathcal{N}(\boldsymbol{y}|\text{NN}(\boldsymbol{x}^0_\star, \boldsymbol{x}^1), \tau^{-1}) \tag{2}$$

where $\text{NN}(\cdot)$ is prediction of DNN and τ is a given parameter. (1) can be approximated using MC estimator:

$$\frac{1}{N_{\boldsymbol{x}^1_\star}} \sum_{\boldsymbol{x}^1_\star} p(\boldsymbol{y}|\boldsymbol{x}^0_\star, \boldsymbol{x}^1_\star) \tag{3}$$

where $\boldsymbol{x}^1_\star \sim p(\boldsymbol{x}^1)$ and $N_{\boldsymbol{x}^1_\star}$ is the number of samples. The expected value of the obtained predictive distribution is the predictive result of BNN and the variance is the predictive uncertainty. The process of calculating the equation consists of two steps: sampling weights from $p(\boldsymbol{x}^1)$, e.g. using Markov chain Monte Carlo (MCMC) (Neal et al., 2011; Hoffman & Gelman, 2014), and executing the DNN for each weight sample. Since real-world data is large and practical NNs are deep, the repetitive computation of NNs is difficult to be fully parallelized due to various problems, such as memory limitations of the GPU (Kendall & Gal, 2017), and the iterative computation of DNNs results in the decrease of computation speed. MCMCs are slow and, despite recent achievements (Tran et al., 2018), it is challenging to achieve linearly scaling performance for multi-GPU MCMC. In addition, it is difficult to check for the convergence of MCMCs. To mitigate these problems, we approximate the difference in the prediction for one new data as one NN calculation, and sample the weights from the posterior before testing.

2.2 Bayesian Neural Network Gradient Estimation

It is standard to use a gradient of MC estimator to optimize the loss function of BNN, e.g. evidence lower bound (ELBO). Several direct and indirect methods are used to obtain the MC gradient estimation: reparametrization trick (Kingma & Welling, 2013; Blundell et al., 2015; Fortunato et al., 2017), score function (Kleijnen & Rubinstein, 1996; Williams, 1992; Glynn, 1990), dropout (Gal & Ghahramani, 2016; McClure & Kriegeskorte, 2016), batch normalization (Teye et al., 2018), expectation propagation (Hernández-Lobato & Adams, 2015), and smooth transformation for samples (Liu & Wang, 2016; Burda et al., 2015; Maddison et al., 2017; Naesseth et al., 2017; Le et al., 2017; van den Oord et al., 2017).

However, it is not appropriate to use these techniques to calculate a gradient of MC estimator $\mathcal{L} = \mathbb{E}_{p(\boldsymbol{x})}\big[f(\boldsymbol{x})\big]$ for arbitrary functions $f(\cdot)$ when a data stream is non-stationary and $p(\boldsymbol{x})$, the distribution of the data stream, is time-variant. For example, the reparameterization trick, one of the most widely used methods for MC gradient estimation, separates the probability $p(\boldsymbol{x})$ into a deterministic function $h(\cdot)$ and an invariant probability $p(\epsilon)$, i.e., $\boldsymbol{x} = h(\epsilon)$ and $\partial_{\boldsymbol{x}} p(\epsilon) = 0$. Then, the $\mathcal{L}$ is expectation over the invariant distribution, i.e., $\mathcal{L} = \mathbb{E}_{p(\epsilon)}\big[f(\boldsymbol{x})\big]$, and the approximated $\mathcal{L}$ becomes differentiable, i.e., $\partial_{\boldsymbol{x}}\mathcal{L} = \mathbb{E}_{p(\epsilon)}\big[\partial_{\boldsymbol{x}} f(\boldsymbol{x})\big]$. To use this technique, we need to adapt $h(\cdot)$ to the non-stationary data stream. If $h(\cdot)$ is modeled in an NN, we may need to continuously train this NN on the data stream; this leads to a decrease in computational performance. To alleviate this problem, we use histogram to estimate probability with nearest neighbor search, instead of using an NN-based generative model. In addition, the histogram is a linear sum of disjoint bins that is analytically useful while simplifying calculations.

3 Online Codevector Histogram

Vector quantization was introduced in Gersho (1982); Gray (1984) in order to compress a probability distribution to a dozens of samples called codevectors. Kotani et al. (2002) showed that the histogram of codevectors (codevectors augmented with counts) can effectively represent the features of face image dataset. We introduce *online codevector histogram* (OCH) to estimate the probability distribution of non-stationary data stream as well as dataset. OCH can add a new codevector and delete old ones, while counting the matches of each codevector for the past data stream. OCH maps the input vector to the codevector using nearest neighbor search. It is a high-dimensional histogram where the Voronoi diagram is the boundary and a codevector represents the corresponding bin. DBNN uses OCH to approximate the probability distributions of input and output vector data streams.

Algorithm 1 shows how OCH operates in three steps. *(a)* Given a new input vector, OCH finds the nearest codevector and increases its count. Then, it divides the corresponding bin by inserting the input vector as a new codevector with probability proportional to the count. *(b)* OCH decrease all counts by the same rate to reduce the contribution of old data. *(c)* To keep the number of codevectors small, OCH deletes codevectors with probability in inverse proportion to its counts.

Algorithm 1: Update OCH

```
input : input vector $\boldsymbol{x}_\star$, count of input vector $n_\star$, OCH $= \{(\boldsymbol{c}_i, n_i)\}$ where $\boldsymbol{c}_i$ is codevector and $n_i$
        is its count, hyperparameters $K$, $\lambda$, and $\phi$
output: updated OCH
1  $\boldsymbol{c}_i \leftarrow$ Search the nearest codevector to $\boldsymbol{x}_\star$ in OCH
2  $n_i \leftarrow n_i + n_\star$ where $n_i$ is count of $\boldsymbol{c}_i$
3  $p \sim \text{Bernoulli}\big(\sigma(\pi_i - \bar{\pi} + \phi)\big)$ where $\sigma(\cdot)$ is sigmoid, $N = \sum_i n_i$, $\pi_i = n_i/N$, and $\bar{\pi} = 1/K$
4  if $p = 1$ then
5      OCH $\leftarrow$ OCH $\cup \{(\boldsymbol{x}_\star, (1-\gamma)\cdot n_i)\}$ where $\gamma = e^{-\lambda/N}$
6      $n_i \leftarrow \gamma \cdot n_i$
7  forall $(\boldsymbol{c}_j, n_j) \in$ OCH do
8      $n_j \leftarrow \gamma \cdot n_j$
9  forall $(\boldsymbol{c}_k, n_k) \in$ OCH do
10     $q \sim \text{Bernoulli}\big(\sigma(\bar{\pi} - \pi_k + \phi)\cdot\bar{\pi}\big)$
11     if $q = 1$ then
12         OCH $\leftarrow$ OCH $\setminus \{(\boldsymbol{c}_k, n_k)\}$
```

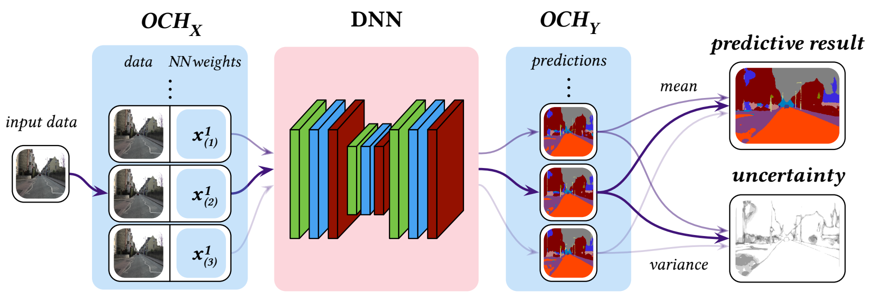


Figure 1: DBNN Inference. OCHs are added to the DNN to estimate the probabilities of input and output vector streams. Given a new data, OCH$_{\boldsymbol{X}}$ adjusts the weight (arrows) of the nearest codevector of OCH$_{\boldsymbol{X}}$ representing the input vector stream. DBNN adjusts weights of the codevectors of the OCH$_{\boldsymbol{Y}}$ representing the predictive distribution based on the results of the inner DNN. DBNN derives predictive result and uncertainty from the weighted ensemble of the codevectors of OCH$_{\boldsymbol{Y}}$.

OCH takes three hyperparameters: K, λ, and ϕ. K is the number of codevectors kept in OCH on average. λ determines how fast OCH depreciates old counts with the rate of γ. ϕ with counts of each bin regulates the probability of adding and deleting codevectors.

Nearest neighbor search is the most computationally intensive step in alg. 1. To lower the computational complexity, we use locality-sensitive hashing with stable distribution, $h = \lfloor {}^{(\boldsymbol{a} \cdot \boldsymbol{x}_\star + b)}/_{w} \rfloor$ for a vector $\boldsymbol{a}$ and scalars b and w. It requires up to $\log K$ hashes for precise search. The upper bound of the computational complexity is $\mathcal{O}(\dim(\boldsymbol{x}) \log K)$. This is faster than continuous training of NN-based generative models for the data stream changing over time.

Meanwhile, OCH approximates probability distribution $p(\boldsymbol{x})$ as:

$$p(\boldsymbol{x}) \simeq \sum_i \pi_i \mathcal{V}(\boldsymbol{x}|\boldsymbol{c}_i) \tag{4}$$

where $N = \sum_i n_i$, $\pi_i = n_i/N$, and $\mathcal{V}(\boldsymbol{x}|\boldsymbol{c})$ is the bin or *neighborhood* of $\boldsymbol{c}$ with $\int \mathcal{V}(\boldsymbol{x}|\boldsymbol{c}) d\boldsymbol{x} = 1$. The right-hand-side of (4) is clearly differentiable with respect to π_j, while a set of random samples from $p(\boldsymbol{x})$ is not differentiable. Given a new input data, OCH changes only one count of the nearest codevector. Therefore, if $n_\star \ll N$, the difference of OCH for a new input vector is approximately:

$$\delta p(\boldsymbol{x}) \simeq \sum_i \delta\pi_i \mathcal{V}(\boldsymbol{x}|\boldsymbol{c}_i) \tag{5}$$

$$\simeq \alpha \mathcal{V}(\boldsymbol{x}|\boldsymbol{c}_\star) \tag{6}$$

where $\boldsymbol{c}_\star$ is the nearest codevector to the input vector, $\alpha = {}^{\delta n}/_{N}$, and δn is the difference of the count of $\boldsymbol{c}_\star$. When OCH creates a new codevector, δn is $n_\star$. Here, we used the fact that codevectors are invariant for input data, so $\mathcal{V}(\boldsymbol{x}|\boldsymbol{c}_i)$ is also invariant. This property is useful when the linear operator $g(\cdot)$ is applied to $p(\boldsymbol{x})$. By definition of linear operator, $g(p(\boldsymbol{x})) = \sum_i \pi_i g(\mathcal{V}(\boldsymbol{x}|\boldsymbol{c}_i))$, then its difference is $\delta g(p(\boldsymbol{x})) = \sum_i \delta\pi_i g(\mathcal{V}(\boldsymbol{x}|\boldsymbol{c}_i)) = \alpha g(\mathcal{V}(\boldsymbol{x}|\boldsymbol{c}_\star))$.

4 Differentiable Bayesian Neural Network Inference

When a data stream $\mathcal{S} = \{\ldots, \boldsymbol{x}^0_\star\}$ is given, where each subsequent data sample changes continuously, the prediction of a NN for $\mathcal{S}$ also changes continuously, i.e., $p(\boldsymbol{y}) \to p(\boldsymbol{y}) + \delta p(\boldsymbol{y})$, because a NN (with continuous activation) is a homeomorphism. We show that $\delta p(\boldsymbol{y})$ is approximately proportional to the prediction of a NN augmented with OCH to the input and output for the recent data $\boldsymbol{x}^0_\star$. Figure 1 shows the structure of DBNN, where DNN is augmented with OCHs as distribution estimator units of input and output stream. DBNN calculates predictive uncertainty as well as the predictive result using the weighted ensemble of output codevectors.

4.1 Differentiation of Bayesian Neural Network Inference as Difference of Prediction

It is challenging to calculate the difference of (1) for a new data $\boldsymbol{x}_\star^0$ from $\mathcal{S}$. The differentiation of $p(\boldsymbol{y}|\boldsymbol{x}_\star^0, \boldsymbol{x}^1)$ with respect to $\boldsymbol{x}_\star^0$ is analytically intractable, because the samples from the probability of the data stream $p(\boldsymbol{x}^0|\mathcal{S})$ are discrete, although $p(\boldsymbol{x}^0|\mathcal{S})$ varies continuously. In other words, $\partial_{\boldsymbol{x}_\star^0} p(\boldsymbol{y}|\boldsymbol{x}_\star^0, \boldsymbol{x}^1) = \int p(\boldsymbol{y}|\boldsymbol{x}^0, \boldsymbol{x}^1) \partial_{\boldsymbol{x}_\star^0} \delta(\boldsymbol{x}^0 - \boldsymbol{x}_\star^0) d\boldsymbol{x}^0$ where $\delta(\cdot)$ is the delta function, and the delta function is not differentiable. To address this issue, DBNN smoothens the delta function of the data sample $\delta(\boldsymbol{x}^0 - \boldsymbol{x}_\star^0)$ to the probability of the data stream $p(\boldsymbol{x}^0|\mathcal{S})$ as follows:

$$p(\boldsymbol{y}|\mathcal{S}) = \int p(\boldsymbol{y}|\boldsymbol{x}^0, \boldsymbol{x}^1) p(\boldsymbol{x}^0|\mathcal{S}) p(\boldsymbol{x}^1) d\boldsymbol{x}^0 d\boldsymbol{x}^1 \tag{7}$$

$$= \int p(\boldsymbol{y}|\boldsymbol{x}) p(\boldsymbol{x}|\mathcal{S}) d\boldsymbol{x} \tag{8}$$

where $\boldsymbol{x} = (\boldsymbol{x}^0, \boldsymbol{x}^1)$ and $p(\boldsymbol{x}|\mathcal{S}) = p(\boldsymbol{x}^0|\mathcal{S})p(\boldsymbol{x}^1)$. $p(\boldsymbol{x}^0|\mathcal{S})$ is separately learned with respect to the data stream $\mathcal{S}$ by another model and is called *data uncertainty*. $p(\boldsymbol{x}^1)$ is the posterior distribution obtained from BNN training and is called *model uncertainty*. DBNN inference decouples data (stream) from the model. Instead, it joins the probability of data and the probability of the NN weights. When the most recent data sample is considered instead of the data stream, i.e., $p(\boldsymbol{x}^0|\mathcal{S}) = \delta(\boldsymbol{x}^0 - \boldsymbol{x}_\star^0)$, DBNN inference is reduced to BNN inference as mentioned before.

To calculate (8), we use OCH and (4) to represent $p(\boldsymbol{x}|\mathcal{S})$, and call it $\text{OCH}_{\boldsymbol{X}}$:

$$p(\boldsymbol{y}|\mathcal{S}) \simeq \sum_i \pi_i \int p(\boldsymbol{y}|\boldsymbol{x}) \mathcal{V}(\boldsymbol{x}|\boldsymbol{c}_i) d\boldsymbol{x} \tag{9}$$

where π_i is weight of codevector $\boldsymbol{c}_i$ and proportional to the count of the codevector. $\mathcal{V}(\boldsymbol{x}|\boldsymbol{c}_i)$ is neighborhood of $\boldsymbol{c}_i$. Then, $\delta p(\boldsymbol{y}|\mathcal{S})$ can be approximated by changing the weight of the codevector nearest to $\boldsymbol{x}_\star^0$ as:

$$\delta p(\boldsymbol{y}|\mathcal{S}) \simeq \sum_i \delta\pi_i \int p(\boldsymbol{y}|\boldsymbol{x}) \mathcal{V}(\boldsymbol{x}|\boldsymbol{c}_i) d\boldsymbol{x} \tag{10}$$

$$\simeq \alpha \int p(\boldsymbol{y}|\boldsymbol{x}) \mathcal{V}(\boldsymbol{x}|\boldsymbol{c}_\star) d\boldsymbol{x} \tag{11}$$

according to the (6). This means that $\int p(\boldsymbol{y}|\boldsymbol{x}) \mathcal{V}(\boldsymbol{x}|\boldsymbol{c}_i) d\boldsymbol{x}$ is invariant with respect to the change of $\mathcal{S}$, i.e., $\delta\big[\int p(\boldsymbol{y}|\boldsymbol{x}) \mathcal{V}(\boldsymbol{x}|\boldsymbol{c}_i) d\boldsymbol{x}\big] = 0$, since histogram only changes its counts, not the codevectors. We approximate $\mathcal{V}(\boldsymbol{x}|\boldsymbol{c}_\star)$ as delta-function distribution at $\boldsymbol{c}_\star$, i.e., $\mathcal{V}(\boldsymbol{x}|\boldsymbol{c}_\star) \simeq \delta(\boldsymbol{x} - \boldsymbol{c}_\star)$:

$$\delta p(\boldsymbol{y}|\mathcal{S}) \simeq \alpha p(\boldsymbol{y}|\boldsymbol{c}_\star) \tag{12}$$

It is safe to assume that $p(\boldsymbol{y}|\boldsymbol{c}_\star)$ is dominantly distributed near $\text{NN}(\boldsymbol{c}_\star)$, i.e., $p(\boldsymbol{y}|\boldsymbol{c}_\star) \simeq \mathcal{V}\big(\boldsymbol{y}|\text{NN}(\boldsymbol{c}_\star)\big)$, because the expected value of $\boldsymbol{y}$ should be equal to $\text{NN}(\boldsymbol{c}_\star)$:

$$\delta p(\boldsymbol{y}|\mathcal{S}) \simeq \alpha \mathcal{V}\big(\boldsymbol{y}|\text{NN}(\boldsymbol{c}_\star)\big) \tag{13}$$

where $\text{NN}(\boldsymbol{c}_\star)$ is prediction of DNN for $\boldsymbol{c}_\star$. This result is equal to the changes of OCH when we update OCH for data, as (6) expresses. In conclusion, the difference of the DBNN prediction contributes only to the neighborhood of the DNN prediction for the codevector in $\text{OCH}_{\boldsymbol{X}}$ approximately. Thus, given the $\text{OCH}_{\boldsymbol{Y}}$ representing $p(\boldsymbol{y}|\mathcal{S})$, $p(\boldsymbol{y}|\mathcal{S}) + \delta p(\boldsymbol{y}|\mathcal{S})$ can be approximated by updating the $\text{OCH}_{\boldsymbol{Y}}$ for the prediction of DNN.

4.2 Implementation of Differentiable Bayesian Neural Network Inference

DBNN is composed of three stages and Algorithm 2 describes the DBNN inference process in detail as follows: *(a)* First, to estimate the probability of the input data stream, the algorithm updates $\text{OCH}_{\boldsymbol{X}}$ for $\boldsymbol{x}_\star = (\boldsymbol{x}_\star^0, \boldsymbol{x}_\star^1)$ where $\boldsymbol{x}_\star^1$ is a random sample from given approximated posterior distribution $\text{OCH}_{\boldsymbol{X}^1}$ which represents the given $p(\boldsymbol{x}^1)$. $\text{OCH}_{\boldsymbol{X}^1}$ consists of weights sampled from trained $p(\boldsymbol{x}^1)$ using MCMC. *(b)* Second, if $\text{OCH}_{\boldsymbol{X}}$ generated a new codevector in $\text{OCH}_{\boldsymbol{X}}$, DBNN generates the prediction for the new codevector using DNN and keeps the prediction in a cache table. As a result,

Algorithm 2: DBNN Inference

```
input  : input data vector $\boldsymbol{x}_\star^0$, deterministic neural network NN$(\cdot)$, posterior distribution OCH$_{\boldsymbol{X}^1}$,
         distribution of input vector OCH$_{\boldsymbol{X}}$, distribution of output vector OCH$_{\boldsymbol{Y}}$, cache table $T$
         initialized to empty set
output : updated distribution of input vector OCH$_{\boldsymbol{X}}$, updated distribution of output vector OCH$_{\boldsymbol{Y}}$
1 $\boldsymbol{x}_\star^1 \sim$ OCH$_{\boldsymbol{X}^1}$
2 $\boldsymbol{x}_\star \leftarrow (\boldsymbol{x}_\star^0, \boldsymbol{x}_\star^1)$
3 OCH$_{\boldsymbol{X}}$ $\leftarrow$ Update OCH$_{\boldsymbol{X}}$ for $\boldsymbol{x}_\star$
4 if new codevector $\boldsymbol{c}_i$ exists in OCH$_{\boldsymbol{X}}$ then
5     $\boldsymbol{y}_i =$ NN$(\boldsymbol{c}_i)$
6     $T \leftarrow T \cup \{\boldsymbol{c}_i \mapsto \boldsymbol{y}_i\}$
7 $\boldsymbol{c}_\star \leftarrow$ Search the nearest neighbor codevector to $\boldsymbol{x}_\star^0$ in OCH$_{\boldsymbol{X}}$
8 $\boldsymbol{y}_\star \leftarrow T(\boldsymbol{c}_\star)$
9 OCH$_{\boldsymbol{Y}}$ $\leftarrow$ Update OCH$_{\boldsymbol{Y}}$ for $\boldsymbol{y}_\star$ with count $\alpha$
```

the cache table contains the results of DNN corresponding to all codevectors in OCH$_{\boldsymbol{X}}$. *(c)* Third, DBNN finds the nearest neighbor codevector to $\boldsymbol{x}_\star^0$ in OCH$_{\boldsymbol{X}}$, looks up the corresponding prediction in the cache table and update OCH$_{\boldsymbol{Y}}$ for the prediction for the codevector. In conclusion, OCH$_{\boldsymbol{Y}}$ estimates the probability of the output data stream.

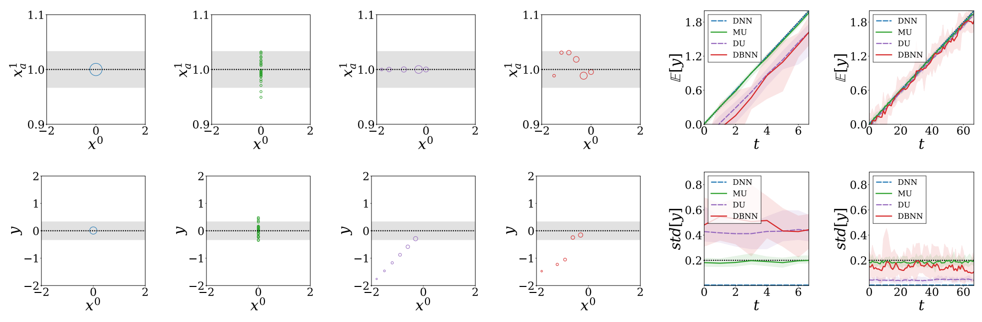
DBNN calculates the difference of prediction for a new data sample from the previous predictions. In the process, it executes DNN once if necessary, which is computationally expensive, in contrast to BNNs' repetitive execution DNNs. Furthermore, DBNN sometimes does not execute DNN, but only updates OCH$_{\boldsymbol{X}}$ and OCH$_{\boldsymbol{Y}}$ for the input and output vectors, by updating the counts and estimates prediction and uncertainty based on cached results. The dominant part of the computational cost of updating OCH$_{\boldsymbol{X}}$ is the inner product of a given input vector, i.e., $\boldsymbol{a} \cdot \boldsymbol{x}$, for nearest neighbor search. Given $\boldsymbol{x} = (\boldsymbol{x}^0, \boldsymbol{x}^1)$ and $\boldsymbol{a} = (\boldsymbol{a}^0, \boldsymbol{a}^1)$, $\boldsymbol{a} \cdot \boldsymbol{x} = \boldsymbol{a}^0 \cdot \boldsymbol{x}^0 + \boldsymbol{a}^1 \cdot \boldsymbol{x}^1$ holds. The codevector samples $\boldsymbol{x}^1$ from OCH$_{\boldsymbol{X}^1}$ is fixed and finite, and DBNN caches all $\boldsymbol{a}^1 \cdot \boldsymbol{x}^1$. Therefore, the average computational cost on $\boldsymbol{a}^1 \cdot \boldsymbol{x}^1$ is minuscule, when updating OCH$_{\boldsymbol{X}}$. In conclusion, the upper bound of the computational complexity of DBNN inference is $\mathcal{O}(\dim(\boldsymbol{x}^0) \log K_{\boldsymbol{X}} + \text{NN}(\cdot) + \dim(\boldsymbol{y}) \log K_{\boldsymbol{Y}})$ where $\mathcal{O}(\text{NN}(\cdot))$ is computational complexity of NN and $K_{\boldsymbol{X}}$ and $K_{\boldsymbol{Y}}$ are hyperparameters K of OCH$_{\boldsymbol{X}}$ and OCH$_{\boldsymbol{Y}}$, respectively. Modern deep NN performs vector operations dozens of times, if not hundreds, so the computation time of OCH is very small compared to DNN execution.

DBNN uses the flexible parametric probability estimator OCH to represent the distributions of input and output vector streams. Unlike the most BNNs that depend on a parameterized model described in (2), DBNN does not depend on the specific model. OCH represents both continuous and discrete vector spaces, and so does DBNN. If DNN and posterior are given, we can easily convert them to DBNN without significant modifications: just add OCH to the input and output of a DNN to estimate the probability of the input and output vector streams.

5 Experiments

Although DBNN uses assumptions that seem reasonable, it is necessary to measure empirical performances to show that this assumption works for real-world problems. This section evaluates the performance of DBNN in three set of experiments. The first experiment visualizes the characteristics of DBNN by performing simple linear regression on synthetic data. The second experiment classifies various real-world multivariate datasets using shallow and narrow NNs. This experiment shows the difference between computational and predictive performance of BNN and DBNN under various conditions when converting BNN to DBNN using small NN. The third experiment performs semantic segmentation on real-world video sequences. This experiment compares the performances of DBNN with other baselines when using a deep and wide modern NNs in practical situation.

We use the following four models as baselines:



(a) DNN (b) MU (c) DU (d) DBNN (e) $v = 0.3$ (f) $v = 0.03$

Figure 2: Simple linear regression on synthetic dataset. In figs. 2a to 2d, the top is the approximated input probability OCH$_{\boldsymbol{X}}$ (x_b^1 is not displayed) and the bottom is the approximated output probability OCH$_{\boldsymbol{Y}}$ (with x_0) at $t = 0$ with $v = 0.3$. The sizes of the circles represents the importance weights of each codevector. figs. 2e to 2f are the expected values and standard deviations of y depending on the timestamp for different v. The black dotted lines represent true values. The error is 90% confidence interval.

- **DNN.** Let Softmax($\boldsymbol{z}$) be probability of NN where $\boldsymbol{z}$ is DNN logits. It is easy to implement, but it differs from the actual classification probability when the NN is deepened, broadened, and regularized well (Guo et al., 2017).
- **Model Uncertainty (MU).** BNN is referred to as MU in this section since it introduces model uncertainty to NN. MU have to calculate (3) to predict the result. It is difficult to analytically determine the number of samples that the prediction converges. Instead, we experimentally set the number of samples to 30 so that the accuracy and the negative log-likelihood (NLL) converge.
- **Data Uncertainty (DU).** DU is a model that assigns only data uncertainty to DNN by adding OCHs to input and output of trained DNN. In other words, this model uses one weight instead of weight distribution in the DBNN, i.e., (8) with $p(\boldsymbol{x}^1) = \delta(\boldsymbol{x}^1 - \boldsymbol{x}^1_\star)$. This model shows the effect of OCH on prediction when it is applied to the NN. Also, it shows that DNNs modified using OCH can achieve higher uncertainty than vanilla DNNs in semantic segmentation experiment.
- **DBNN.** DBNN takes both data uncertainty and model uncertainty. DBNN adds OCH to the input and output of trained BNN. The approximated posterior distribution of this model OCH$_{\boldsymbol{X}^1}$ consists of 30 samples from the BNN's posterior.

5.1 Simple Linear Regression

A neural network with a linear activation function is a linear regression model $y = x_a^1 x^0 + x_b^1$. The posterior is given by $p(x_a^1) = \mathcal{N}(1.0, 0.02^2)$ and $p(x_b^1) = \mathcal{N}(0.0, 0.2^2)$. The distribution of time-varying input data streams is given by $p(x^0|t) = \delta(x^0 + vt)$ where t is integer timestamp from $t = -10$. v is 0.3 and 0.03.

The top of figs. 2a to 2d are distributions of input samples $p(x^0, x_a^1)$ approximated by OCHs and bottom are approximated distributions of output samples with data $p(x^0, y)$. x_b^1 is omitted from x^1 in these figures, but it behaves like x_a^1. As shown in these figures, The input distributions of DNN, MU, and DU are degenerated with respect to $\boldsymbol{x}^0$ and/or $\boldsymbol{x}^1$, while that of DBNN is non-degenerated. DBNN without distribution of $\boldsymbol{x}^0$ is equivalent to MU, and DBNN without distribution of $\boldsymbol{x}^1$ is equivalent to DU.

The figs. 2e to 2f show expected value and standard deviation of DBNN prediction on time. First, as shown in top of fig. 2e, the result of DBNN lags behind since DBNN smoothens its prediction with respect to time.[1] Second, unlike MU uses dozens of samples, DBNN uses a small number of

[1]The delay of the expected value of $\boldsymbol{y}$ over time does not mean that the regression result is biased. DBNN estimates the correct value on average since $\boldsymbol{x}$ is also delayed just as $\boldsymbol{y}$ is delayed.

Table 1: Predictive performance of MU and DBNN for classification on multivariate datasets.

Dataset	RMSE		RMSE-90		NLL		Cov-90	
	MU	DBNN	MU	DBNN	MU	DBNN	MU	DBNN
Localization	0.230	0.229	0.137	0.130	1.89	2.30	3.50	9.90
EMG	0.374	0.375	0.293	0.330	1.21	1.49	1.98	8.83
Occupancy	0.209	0.234	0.109	0.176	0.38	0.41	95.0	91.9

samples, so the error is relatively large. Third, as shown in bottom of fig. 2e, the DBNN prediction is under-confident because DBNN considers not only MU but also DU. Fourth, as shown in the comparison of fig. 2e and fig. 2f, DBNN converges to BNN and DU converges to DNN as the input data stream changes more slowly.

5.2 Classification on Multivariate Datasets

A classification experiment is designed to compare computational and predictive performance changes when converting BNNs to DBNNs using shallow and narrow NNs in various situations. In this experiment, we use three time-series real-world datasets with input dimensions between 4 and 8. Localization and EMG dataset have relatively large number of classes, eleven and eight respectively while Occupancy dataset has only two classes. The NNs consist of 2 fully-connected hidden layers with 50 units. The distributions of weights are optimized using variational inference. See appendix A for more information about experimental settings and datasets.

Computational Performance. Execution times for one batch of MU and DBNN are 26.7$\pm$10.6ms and 19.5$\pm$10.0ms, respectively. MU parallelizes execution by using a batch size of 30. This result shows that DBNN is 37% faster than MU even though MU is parallelized and DBNN uses additional OCHs. DBNN improves the computational performance of DBNN by sampling the weights from the posterior before testing. As NN gets deeper and larger, the execution time of the DBNN will be reduced compared to the MU, because the larger the NN, the greater the burden of sampling weights.[2]

Predictive Performance. As shown in Table 1, we compared DU and DBNN in terms of the root-mean-square error (RMSE), RMSE for predictive results with confidence above 90% (RMSE-90), negative log-likelihood (NLL), and the percentage of predictive results with confidence above 90% (Cov-90) of the MU and DBNN for three datasets. DBNN provides the predictive performance comparable with DU for both confident results (RMSE-90) and all results (RMSE). Higher NLL of DBNN shows that DBNN is relatively inaccurate for uncertain predictions. Since DBNN uses a small number of samples compared to MU, the errors of expected values and that of variances are larger, which is consistent with the results of simple linear regression.

5.3 Semantic Segmentation

Semantic segmentation experiment shows the computational and predictive performance of DBNN with a modern deep NN in practical situation. We use CamVid dataset (Brostow et al., 2009) consisting of real-world day and dusk road scenes with 480$\times$360 resized 30 frame-per-second (fps) video sequence. We use the U-Net (Ronneberger et al., 2015) as the backbone architecture. In this experiment, similar to Kendall et al. (2015), BNN contains six MC dropout (Gal & Ghahramani, 2016) layers behind the layers that receives the smallest input vector sizes. Therefore, the overhead of sampling weights is negligible and DBNN always uses a new dropout every time it predicts a result. For more information about experimental settings, see appendix A.

Computational Performance. The throughput column of table 2 shows the number of video frames processed by each model per second. This table shows that DNN takes 124$\pm$17ms on average

[2]Execution times of MU and DBNN with 10 fully-connected hidden layer NN are 98.9$\pm$20.1ms and 49.0$\pm$23.6ms, respectively. In this case, DBNN is 2.02$\times$ faster than MU. See appendix B for more information about the relationship between NN depth and execution time.

Table 2: Computational and predictive performance with semantic segmentation for each model.

Model	Throughput (fps)	Acc	Acc-90	IoU	IoU-90	Cov-90
DNN	8.04	82.9	85.1	45.7	48.6	94.8
MU	0.595	84.2	91.0	49.8	60.0	83.9
DU	7.14	82.9	88.6	44.9	52.0	85.3
DBNN	7.03	83.7	91.0	47.7	56.7	81.1

to process one frame. In comparison, DBNN takes 142$\pm$60ms on average to process one frame, which is only 15% higher than DNN. According to the difference between the execution time of DNN and DU, the average execution time of one OCH is 8ms, which is only 6% of the total. Besides, due to GPU memory limitations, the batch size of MU is limited to 3 or less. Thus, the MU predicts results for 10 batches of size 3. In conclusion, the execution time of MU is 1680$\pm$50ms, which is 14$\times$ higher than that of DNN, and 12$\times$ higher than that of DBNN.[3] Moreover, DBNN can increase throughput at the expense of accuracy by adjusting a hyperparameter. See appendix C for more information.

Predictive Performance. The Acc to Cov-90 columns of the table 2 show the quantitative comparison of the predictive performance for each model. We measured global pixel accuracy (Acc) and mean intersection over unit (IoU). At the same time, similar to section 5.2, we selected only those pixels with confidence greater than 90% and measure the accuracy and IoU, called Acc-90 and IoU-90. We also measured the percentage of pixels with a confidence of 90% or more, called Cov-90.

MU, DU, and DBNN show 1.6%, -0.066%, and 0.97% higher accuracy than DNN, for all pixels, respectively. For pixels with ninety-percent or higher certainty, DNN, MU, DU, and DBNN predicts with 2.6%, 8.1%, 6.9%, and 8.7% higher accuracy, respectively, compared to the accuracy for all pixels. MU and DBNN improves the accuracy more than DNN and DU for confident pixels, which means that MU and DBNN estimates high uncertainty for misclassified region. IoUs for certain pixels increase by 6.3%, 20%, 16%, 19%, respectively, compared to the IoUs for all pixels, which shows the same trend as in accuracy.

As shown in this results, DNN is the most improper way to distinguish uncertain pixels because it has the least performance improvement compared to other methods for pixels with high confidence. On the other hand, MU has the highest performance improvement compared to DNN when considering all pixels, and it also has the highest performance improvement for certain pixels. DBNN has improved performance compared to DNN for all pixels, and has improved performance for certain pixels, similar to MU. The predictive performance of DU for all pixels is similar to the predictive performance of DNN, but for certain pixels, the performance is much better than that of DNN. See appendix D for a qualitative evaluation of predictive results.

6 Conclusion

We present a differentiable BNN (DBNN) inference with respect to input data, which is a novel approximation of BNN inference, to improve the computational performance of BNN inference for data streams. The derivative of DBNN predictive inference with respect to input data derives the increment of prediction when one data is newly given from the data stream. However, the inference of vanilla BNN cannot be differentiated with respect to data. To address this issue, DBNN introduce a new term that is the probability of data streams in the BNN inference. Then, it approximate its prediction with a histogram for high-dimensional vector streams. Consequently, the DBNN inference executes DNN only once to calculate the prediction changed by a new data from data streams. This results in an order of magnitude times improvement in computational performance compared to deep BNN. Experiments with semantic segmentation using real-world datasets show that the computational performance of DBNN is almost the same as that of DNN, and uncertainty is comparable to that of BNN.

[3]If the batch size is 1 because of memory limitations, the execution time of the MU is 4180$\pm$130ms, which means that the MU is 34$\times$ and 29$\times$ slower than DNN and DBNN, respectively.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

Diego Ardila, Atilla P Kiraly, Sujeeth Bharadwaj, Bokyung Choi, Joshua J Reicher, Lily Peng, Daniel Tse, Mozziyar Etemadi, Wenxing Ye, Greg Corrado, David P Naidich, and Shravya Shetty. End-to-end lung cancer screening with three-dimensional deep learning on low-dose chest computed tomography. *Nature Medicine*, 2019.

Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight Uncertainty in Neural Networks. *arXiv.org*, May 2015.

Gabriel J Brostow, Julien Fauqueur, and Roberto Cipolla. Semantic object classes in video: A high-definition ground truth database. *Pattern Recognition Letters*, 30(2):88–97, 2009.

Yuri Burda, Roger Grosse, and Ruslan Salakhutdinov. Importance weighted autoencoders. *arXiv preprint arXiv:1509.00519*, 2015.

Luis M Candanedo and Véronique Feldheim. Accurate occupancy detection of an office room from light, temperature, humidity and co2 measurements using statistical learning models. *Energy and Buildings*, 112:28–39, 2016.

Chelsea Finn, Kelvin Xu, and Sergey Levine. Probabilistic model-agnostic meta-learning. In *Advances in Neural Information Processing Systems*, pp. 9516–9527, 2018.

Meire Fortunato, Charles Blundell, and Oriol Vinyals. Bayesian recurrent neural networks. *arXiv preprint arXiv:1704.02798*, 2017.

Yarin Gal and Zoubin Ghahramani. Dropout as a Bayesian Approximation - Representing Model Uncertainty in Deep Learning. *ICML*, 2016.

Allen Gersho. On the structure of vector quantizers. *IEEE Transactions on Information Theory*, 28 (2):157–166, 1982.

Peter W Glynn. Likelihood ratio gradient estimation for stochastic systems. *Communications of the ACM*, 33(10):75–84, 1990.

R Gray. Vector quantization. *IEEE ASSP Magazine*, 1(2):4–29, 1984.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1321–1330. JMLR. org, 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

José Miguel Hernández-Lobato and Ryan P Adams. Probabilistic Backpropagation for Scalable Learning of Bayesian Neural Networks. *NIPS*, stat.ML, 2015.

Matthew D Hoffman and Andrew Gelman. The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *Journal of Machine Learning Research*, 15(1):1593–1623, 2014.

Gregory Kahn, Adam Villaflor, Vitchyr Pong, Pieter Abbeel, and Sergey Levine. Uncertainty-aware reinforcement learning for collision avoidance. *arXiv preprint arXiv:1702.01182*, 2017.

Boštjan Kaluža, Violeta Mirchevska, Erik Dovgan, Mitja Luštrek, and Matjaž Gams. An agent-based approach to care in independent living. In *International joint conference on ambient intelligence*, pp. 177–186. Springer, 2010.

Alex Kendall and Yarin Gal. What uncertainties do we need in bayesian deep learning for computer vision? In *Advances in neural information processing systems*, pp. 5574–5584, 2017.

Alex Kendall, Vijay Badrinarayanan, and Roberto Cipolla. Bayesian segnet: Model uncertainty in deep convolutional encoder-decoder architectures for scene understanding. *arXiv preprint arXiv:1511.02680*, 2015.

Alex Kendall, Yarin Gal, and Roberto Cipolla. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 7482–7491, 2018.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Jack PC Kleijnen and Reuven Y Rubinstein. Optimization and sensitivity analysis of computer simulation models by the score function method. *European Journal of Operational Research*, 88 (3):413–427, 1996.

Koji Kotani, Chen Qiu, and Tadahiro Ohmi. Face recognition using vector quantization histogram method. In *Proceedings. International Conference on Image Processing*, volume 2, pp. II–II. IEEE, 2002.

Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in Neural Information Processing Systems*, pp. 6402–6413, 2017.

Tuan Anh Le, Maximilian Igl, Tom Rainforth, Tom Jin, and Frank Wood. Auto-encoding sequential monte carlo. *arXiv preprint arXiv:1705.10306*, 2017.

Qiang Liu and Dilin Wang. Stein variational gradient descent: A general purpose bayesian inference algorithm. In *Advances In Neural Information Processing Systems*, pp. 2378–2386, 2016.

Sergey Lobov, Nadia Krilova, Innokentiy Kastalskiy, Victor Kazantsev, and Valeri Makarov. Latent factors limiting the performance of semg-interfaces. *Sensors*, 18(4):1122, 2018.

Chris J Maddison, John Lawson, George Tucker, Nicolas Heess, Mohammad Norouzi, Andriy Mnih, Arnaud Doucet, and Yee Teh. Filtering variational objectives. In *Advances in Neural Information Processing Systems*, pp. 6573–6583, 2017.

Patrick McClure and Nikolaus Kriegeskorte. Representing inferential uncertainty in deep neural networks through sampling. 2016.

Christian A Naesseth, Scott W Linderman, Rajesh Ranganath, and David M Blei. Variational sequential monte carlo. *arXiv preprint arXiv:1705.11140*, 2017.

Radford M Neal et al. Mcmc using hamiltonian dynamics. *Handbook of markov chain monte carlo*, 2(11):2, 2011.

Ian Osband, John Aslanides, and Albin Cassirer. Randomized prior functions for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 8617–8629, 2018.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241. Springer, 2015.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

Mattias Teye, Hossein Azizpour, and Kevin Smith. Bayesian uncertainty estimation for batch normalized deep networks. *arXiv preprint arXiv:1802.06455*, 2018.

Dustin Tran, Matthew W Hoffman, Dave Moore, Christopher Suter, Srinivas Vasudevan, and Alexey Radul. Simple, distributed, and accelerated probabilistic programming. In *Advances in Neural Information Processing Systems*, pp. 7598–7609, 2018.

Aaron van den Oord, Oriol Vinyals, et al. Neural discrete representation learning. In *Advances in Neural Information Processing Systems*, pp. 6306–6315, 2017.

Table 3: Input dimensionalities ($\dim(\boldsymbol{x}^0)$), output dimensionalities ($\dim(\boldsymbol{y})$), and number of training sets (N) of dataset used in the experiments.

Dataset	$\dim(\boldsymbol{x}^0)$	$\dim(\boldsymbol{y})$	N
Localization	4	11	148373
Occupancy	5	2	8143
EMG	8	8	3793345
CamVid	360×480	32	421

Yeming Wen, Paul Vicol, Jimmy Ba, Dustin Tran, and Roger Grosse. Flipout: Efficient pseudo-independent weight perturbations on mini-batches. *arXiv preprint arXiv:1803.04386*, 2018.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Jaesik Yoon, Taesup Kim, Ousmane Dia, Sungwoong Kim, Yoshua Bengio, and Sungjin Ahn. Bayesian model-agnostic meta-learning. In *Advances in Neural Information Processing Systems*, pp. 7332–7342, 2018.

A Experimental Setup and Datasets

We conducted all the experiments with the Intel Xeon W-2123 Processor, 32GB memory, and a single GeForce RTX 2080 Ti. NN models are implemented in TensorFlow (Abadi et al., 2016).[4] We trained NNs with Adam with a constant learning rate of 0.001. The BNN used in section 5.2 consists of fully-connected layers with Flipout estimator (Wen et al., 2018). DNN and BNN are trained using categorical cross-entropy loss and ELBO, respectively. When training NNs in section 5.2, we set the batch size to 3, which gives the most accurate results in our experiences. In section 5.3, batch size is limited to 3 because of memory limitations. We used the following time-sequential real-world dataset for classification in section 5.2: Localization Data for Person Activity Dataset (Kaluža et al., 2010), Occupancy Detection Dataset (Candanedo & Feldheim, 2016), and EMG Data for Gestures Dataset (Lobov et al., 2018). If there are no distinction between the test set and the training set, NNs use 90% of the dataset for training and the rest for testing. NNs test the datasets in time-sequential order. See table 3 for more information about datasets. In section 5.2, to optimize the predictive performances, we set hyperparameters K, λ, and $\sigma(\phi)$ to 10, 0.01, and 1.0, respectively, in all OCHs of DBNN. In section 5.3, to optimize the predictive performances, we set hyperparameters K, λ, and $\sigma(\phi)$ to 5, 5.0, and 1.0, respectively, in all OCHs of DBNN. See appendix C for the change in IoU with hyperparameters.

B Computational Performance for Neural Network Depth

Figure 3 shows a trend that the execution time of MU and DBNN increases as hidden layers get deeper without changing the input and output dimensions. In this case, as in section 5.2, NN consists of 50 units of fully-connected hidden layers, with input and output dimensions of 4 and 11, respectively. MU and DBNN use batches of size 30 and 1, respectively, but execution time is the same even if MU uses batch size of 1. According to this figure, when the number of hidden layers is 0, the execution time of DBNN is 15.1±6.7ms, which is 4.6ms slower than MU. This is because DBNN uses two additional OCHs compared to MU. However, unlike MU, DBNN does not sample the weight in the posterior when it predicts the result. Therefore, the increase of execution time of DBNN per layer is lower than that of MU. As a result, the execution time of the MU increases by 8.6ms while that of DBNN increases by 3.5ms when one layer is added. When the number of hidden layers is 10, the execution time of MU and DBNN is 98.9±20.1ms and 49.0±23.6ms, respectively—DBNN is 2.02× faster than MU.

[4] Code available at https://anonymous.4open.science/r/dbnn/

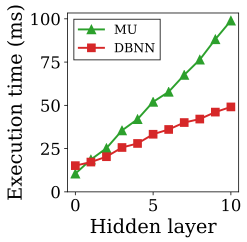


Figure 3: Prediction execution time of MU and DBNN for the number of hidden layers. The execution times increase linearly as the layer increases. If the number of hidden layers is 0, DBNN is slower than MU, but as the number of hidden layers increases, DBNN is faster than MU. The increase rates of execution times of MU and DBNN are 8.6ms/layer and 3.5ms/layer, respectively.

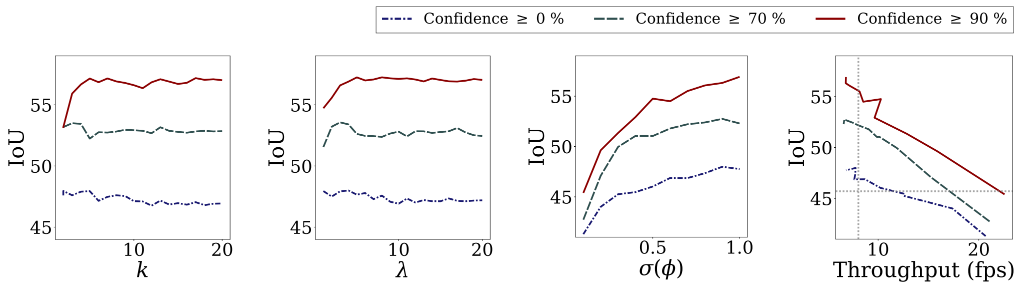


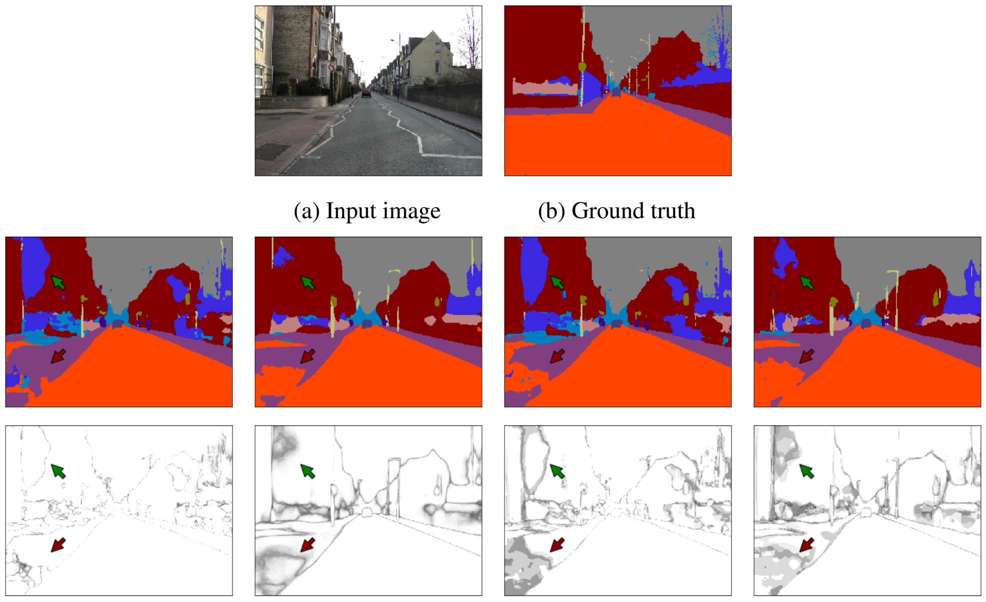
Figure 4: IoU for hyperparameters of $\text{OCH}_{\boldsymbol{Y}}$, K, λ, and $\sigma(\phi)$ and IoU for throughput (from left). In the fourth figure, the IoU and throughput of the DNN are represented as horizontal and vertical dotted lines, respectively. The higher the value of K and λ, the higher the IoU until 5 and 5.0 respectively, and then the higher the value of K and λ, the lower the IoU. Meanwhile, as $\sigma(\phi)$ increases, IoU decreases but throughput increases. DBNN can trade off IoU against throughput by changing $\sigma(\phi)$.

C Predictive Performance for Hyperparameters

DBNN takes three hyperparameters, i.e., K, λ, and $\sigma(\phi)$, in the input and output OCHs. The first to third figures in fig. 4 shows the IoUs for the hyperparameters of the output OCH in semantic segmentation. In this figure, the higher the value of K and λ, the higher the value of IoU until 5 and 5.0 respectively, and then the higher the value of K and λ, the lower the value of IoU. The higher the $\sigma(\phi)$, the higher the IoU.

If K and λ are larger, DBNN maintains more codevectors—recent frames in video sequence. In this case, DBNN obtains more accurate model uncertainty using more codevectors. However, the data uncertainty is too high to estimate results. As a result, at low K and λ, IoU is low because model uncertainty is not precise. At high K and λ, IoU is low again because the data uncertainty increases and the assumption of DBNN is no longer held. IoU is maximized when model uncertainty and data uncertainty are balanced. On the other hand, if $\sigma(\phi)$ is low, DBNN mostly adjusts the weights of codevectors and occasionally adds new codevectors. This results in the DBNN becoming inaccurate, but the calculation is faster. K and λ of input OCH did not affect the predictive performance. The effect of $\sigma(\phi)$ of the input OCH is the same as that of the output OCH.

Accuracy Throughput Trade-off. If $\sigma(\phi)$ is less than 1.0, DBNN occasionally adds a new codevector to the input OCH and executes NN, at the rate of $\sigma(\phi)$ on average. Since NN occupies the most of the execution time of DBNN, throughput decreases when $\sigma(\phi)$ decreases. The fourth figure in fig. 4 shows the changes in throughput and IoU as $\sigma(\phi)$ changes. According to this figure, DBNN can increase throughput up to 12fps when it achieves same IoU as DNN. In this case, DBNN is 50% faster than DNN. If the confidence is more than 90%, the throughput of DBNN is increased up to 23fps, which is 2.8× higher than that of DNN. Conversely, DBNN can update a batch of two or more


(a) Input image (b) Ground truth

(c) DNN (d) MU (e) DU (f) DBNN

Figure 5: Qualitative results on the CamVid dataset for each model. In figs. 5c to 5f, the top is the predictive results and the bottom is the predictive uncertainty. A darker background corresponds to higher uncertainty. The areas indicated by the upper left and lower left arrows show different predictive uncertainty for misclassified region of each model.

sizes instead of one. In this case, throughput of DBNN decreases but IoU increases (not shown in this figure).

D Qualitative Results of Semantic Segmentation

Figure 5 shows the qualitative comparison of the predictions for each model. According to this figure, DNN is overconfident, i.e., uncertainty is generally low, and is mostly distributed at the boundaries of the classified chunks. Even when DNN generates wrong prediction, arrowed in the result figure, the confidence level is very high. MU predicts a similar predictive result to ground truth than DNN. The uncertainty is distributed on the boundaries as in the case of the softmax probability, but is also distributed in the misclassified areas. The predictive result of DU is similar to the result of DNN. However, the uncertainty differs from the uncertainty of DNN. First, DU is under-confident compared to softmax probability of DNN. Second, although DU does not identify all of the misclassifications compared to MU (upper left area), it sometimes estimates high uncertainty in the misclassified areas (lower left area). The predictive result and uncertainty of DBNN is similar to the model uncertainty as we expected.