



Bayesian neural networks at scale: a performance analysis and pruning study

Himanshu Sharma¹ · Elise Jennings¹

© This is a U.S. government work and not under copyright protection in the U.S.; foreign copyright protection may apply 2020

Abstract

Bayesian neural networks (BNNs) are a promising method of obtaining statistical uncertainties for neural network predictions but with a higher computational overhead which can limit their practical usage. This work explores the use of high-performance computing with distributed training to address the challenges of training BNNs at scale. We present a performance and scalability comparison of training the VGG-16 and Resnet-18 models on a Cray-XC40 cluster. We demonstrate that network pruning can speed up inference without accuracy loss and provide an open-source software package, *BPrune*, to automate this pruning. For certain models we find that pruning up to 80% of the network results in only a 7.0% loss in accuracy. With the development of new hardware accelerators for deep learning, BNNs are of considerable interest for benchmarking performance. This analysis of training a BNN at scale outlines the limitations and benefits compared to a conventional neural network.

Keywords Bayesian neural networks (BNN) · Distributed training · Model uncertainty · Pruning BNNs

1 Introduction

One important challenge for machine and deep learning (DL) practitioners is to develop a robust and accurate understanding of the model uncertainty. The current state-of-the-art deep learning networks are now able to learn representations in complex high-dimensional data for doing context-informed predictions. However, these predictions are often taken blindly with the provided accuracy metric, which may be

✉ Himanshu Sharma
himanshu90sharma@gmail.com

Elise Jennings
elise.jennings@ichec.ie

¹ Argonne Leadership Computing Facility, Argonne National Laboratory, Lemont, IL, USA

erroneous. Further, for scientific applications of machine learning such as in physics, biology and manufacturing, including accurate model uncertainties, is crucial.

Conventional deep neural networks (DNNs) are deterministic models. These models do not provide uncertainty quantification (UQ), model confidence or a probabilistic framework for model comparison. Typically, a probabilistic model is used to compute these quantities of interest. In a deep learning context, DNNs can be integrated with probabilistic models such as Gaussian processes, which induce probability distribution over functions. A Gaussian process can be recovered from these networks in the limit of an infinite number of weights associated with probabilistic distributions (see [1, 2]). In a finite setting, a Bayesian neural network (BNN) is a DNN with probability distributions instead of point estimates for each weight. Several foundational works on this topic such as Mackay [3] and Neal [1] have led to BNNs gaining in popularity among DL practitioners. In theory these networks can overcome many limitations of DNNs such as overfitting and hyperparameter optimization, but BNNs present additional workload-to-system challenges due to the increased computational costs. To address the computational complexities, techniques like variational inference (VI) are routinely applied [4, 5, 6]. More recently methods such as stochastic VI and sampling-based VI have been developed [7]. The work in [8–12] have further added thrust into the usage of BNN for a wide variety of applications such as autonomous driving control, medical diagnostics, explanatory atmospheric retrieval and uncertainty quantification of turbulence models. Further, comprehensive review on the BNN can also be looked in the work by Shridhar et al. [13]. In addition to BNN there are a variety of other approaches for performing UQ [14, 15], such as methods based on the dropout technique proposed by Hinton et al. [14] for avoiding over fitting. This has been extended by Gal et al. [15] as *Monte Carlo dropout* for quantifying uncertainties.

Carrying out distributed training of BNNs poses several computational challenges on high-performance computing (HPC) systems. Dustin et al. [16] outline some of the computational challenges of training BNNs at scale and show results for a 5 billion parameter ‘Bayesian Transformer’ on 512 TPUv2 cores in machine translation and a Bayesian dynamics model for model-based planning. In Tran et al. work [16] they used mesh-Tensorflow [17] to perform model distributed training, and the authors observe linear scaling from 8–512 TPUs. Yeming et al. [18] present a technique to sample the weights for each layer with minimum covariance for a BNN and report a comparison of computational performance of their techniques compared to conventional techniques on multiple CPUs. However, both of these studies lack a detailed performance analysis of training a BNN benchmark. With the development of new hardware accelerators for DL, such as the massively parallel intelligence processing unit (IPU) [19, 20] or the wafer-scale engine [21], BNNs and probabilistic models are of considerable interest for benchmarking the performance of new architectures. There is a growing interest in scaling probabilistic programming languages (PPL) such as the recent work by Baydin et al. [22] who present an integrated cross-platform probabilistic execution protocol for directly coupling to existing scientific simulations. Given these efforts to analyse the performance of BNNs/NNs on different architectures, to date, there have been limited studies on the training performance on Intel Xeon Phi architectures [23]. The main contributions of this work are

as follows: we present a scalability analysis of Bayesian neural networks on a state-of-the-art computing cluster. We evaluate the performance of TensorFlow and TensorFlow Probability on the 10 PetaFlops Cray XC40 high-performance supercomputer, Theta, at ALCF [24]. We present the training throughput using customized software builds which exploit the multi-core multi-thread system using optimized libraries such as Intel MKL, Intel MKL-DNN, Intel Numpy [25] and Cray MPICH [26]. Until recently the computational overhead of BNNs together with the lack of efficient software stacks has been prohibitive to running at scale.

Accounting for an increase of deep learning algorithms for computer vision applications this work uses popular network architectures. We present a performance and scalability analysis of single-node and data parallel distributed training of BNNs for two classification models, VGG-16 [27] and Resnet-18 [28], applied to the CIFAR-10 dataset and VGG-16 applied to a 0.1 million transformed MNIST dataset with large training batch size of 1024.

To the best of our knowledge, this work presents the first detailed analysis and measurements of BNN distributed training. We present scaling efficiencies up to 128 nodes; a comparison of times to a given accuracy between BNNs and conventional neural networks and detailed profiling results showing a breakdown of time spent in various routines.

On a smaller scale we present results from a study on a NVIDIA-DGX station. We report the training time for the BNN models for the 0.1 million transformed MNIST dataset up to 8 GPUs.

In addition, the work addresses the limitations of BNN's usage for inference. We present an open-source post-training software package *BPrune* which can be used for pruning a BNN model after training. Inference for BNNs can be slow due to the Monte Carlo sampling in each layer, and we demonstrate how post-training pruning of the network is useful in deploying models in case of limited computational resources. Further, details on pruning are presented in Sect. 4.6.

This paper is organized as follows: we discuss the background of Bayesian neural networks in Sect. 2 and present the variational inference methods used for training the network. In Sect. 3 we outline the details of the BNN architecture, and we also describe the dataset used in Sect. 3.1 for training and testing and the computational resources used for training in Sect. 3.2. The results of the performed scaling study and pruning analysis are presented in Sect. 4. Finally, we present the discussion and conclusion in Sects. 5 and 6, respectively.

2 Background on Bayesian neural networks

BNNs represent the integration of a hierarchical Bayesian framework together with a neural network structure composed of recursive applications of linear weighted functions followed by nonlinear transformations. With prior distributions on weights, we are able to approximate their posterior distributions and perform posterior prediction via a variational inference framework. Consider a model that returns a probability distribution over an output y given an input x and parameter θ , that is $p(y|x, \theta)$. The goal is to learn these parameters from the observed data $D = \{D_i\}_{i=1}^N = \{x_i, y_i\}_{i=1}^N$,

where N is the number of samples, and i is the sample index. Following a Bayesian approach, we put a prior distribution $p(\theta)$ over θ and aim to obtain the posterior distribution

$$p(\theta|D) \propto p(\theta)p(D|\theta) = p(\theta) \prod_{i=1}^N p(y_i|\theta, x_i).$$

2.1 Variational inference

In most cases, the posterior distribution is intractable and an approximation is required. Consider using a variational family distribution $q_v(\theta)$ with parameters v to approximate the posterior by minimizing the KL divergence, i.e.

$$\min_v D_{\text{KL}}(q_v(\theta)||p(\theta|D)), \quad (1)$$

where

$$\begin{aligned} D_{\text{KL}}(q_v(\theta)||p(\theta|D)) &= \mathbb{E}_q \left[\log \frac{q_v(\theta)}{p(\theta|D)} \right] = \mathbb{E}_q \left[\log \frac{q_v(\theta)p(D)}{p(D|\theta)p(\theta)} \right] \\ &= -\{ \mathbb{E}_q [\log p(D|\theta)] - D_{\text{KL}}(q_v(\theta)||p(\theta)) \} + \log p(D) \end{aligned}$$

where the first term represents the log-likelihood of the model predictions, while the second part serves as a regularizer KL divergence between the approximate posterior $q_v(\theta)$ and the prior distribution $p(\theta)$. In the case where $\log p(D)$ is constant given prior distribution $p(\theta)$, minimizing the KL divergence in Eq. (1) is equivalent to maximizing the objective function or the Evidence Lower Bound (ELBO) given by

$$L(v) = \sum_{i=1}^N \mathbb{E}_q \log p(y_i|\theta, x_i) - \beta D_{\text{KL}}(q_v||p), \quad (2)$$

where β is a hyperparameter for tuning the degree of regularization during training of the neural network. Typically the β hyperparameter is introduced to address the difficulty of the KL term vanishing which is often observed while training variational autoencoders (VAEs). This hyperparameter was introduced by Bowman et al. [29] as ‘*KL-annealing*’ where the β parameter can be varied from 0 to 1 over the course of training. Extensions and modifications to the annealing procedure can be seen in [30, 31]. Recently an analysis on the scheduling strategies of β was presented by Liu et al. [32]. Once the BNN model is trained the inference procedure is to compute the predictive probability distribution $p(y^*|x^*)$ given by

$$p(y^*|x^*) = \int p_\theta(y^*|x^*)p(\theta)d\theta, \quad (3)$$

where x^* is the unseen (test) sample and y^* is the corresponding predicted class. Finding a closed-form solution to the above integral for nonconjugate pair of

distribution is not possible; hence, the integral can be approximated as an expectation by sampling from $q_v(w|D)$ as

$$\mathbb{E}_q[p(y^*|x^*)] = \int q_v(\theta|D)p_\theta(y|x)d\theta \approx \frac{1}{S} \sum_{i=1}^S p_{\theta_i}(y^*|x^*) \quad (4)$$

where S is the number of samples or Monte Carlo iterations.

2.2 Training algorithm

We summarize the procedure to optimize the loss function described in Eq. (2). The backpropagation algorithm is at the heart of training a neural network and relies on the calculation of gradients of the loss function which are subsequently used to update the weights layer by layer. There are several approaches available for computing the gradients such as score-function gradient estimators shown in [8, 33], reparametrization gradient estimators described in [9, 12] or a combination of the two as described in Naesseth et al. [34]. These methods provide an unbiased stochastic gradient which is used to calculate the optimum for the loss function. In the current work we use the reparametrization procedure for computing the gradients. We use the Gaussian distribution for the variational posterior and initialize the weights, θ , by sampling from the unit Gaussian with mean μ and standard deviation σ . The hyperparameters for the variational posterior are therefore $v = (\mu, \sigma)$. The reparametrization trick is to parameterize weights as a function, t , where $\theta = t(v, \epsilon) = \mu + \log(1 + \exp(\sigma)) \circ \epsilon$, where \circ represents an element-wise multiplication and the parameter free noise is defined by $\epsilon \sim N(0, I)$. Equation (2) can be rewritten in terms of $L(\theta, v)$, and the gradients, $\Delta_\mu L(\theta, v)$ and $\Delta_\sigma L(\theta, v)$, are computed to update the hyperparameters $\mu \leftarrow \mu + \eta \Delta_\mu L(\theta, v)$ and $\sigma \leftarrow \sigma + \eta \Delta_\sigma L(\theta, v)$. We refer readers to [12] for more advanced details on backpropagation in BNNs.

3 Methodology

In Sect. 3.1 we discuss the network architectures and the dataset considered in this study. The details of the hardware, software and the analysis procedure used in the study for benchmarking are discussed in Sect. 3.2.

3.1 Dataset and network architecture

In this study we use the following datasets which are common in the machine learning community for classification tasks: CIFAR-10, MNIST and MNIST transformed 0.1 Million. The CIFAR-10 dataset [35] consists of 60,000 32×32 colour images in 10 classes, with 6000 images per class. There are 50,000 training images and 10,000 test images. The MNIST dataset [36] consists of hand-written gray scale 28×28 images of digits representing the numbers 0–9. There are 50,000 training and 10,000 test images in this dataset. This dataset is used for our pruning study presented in

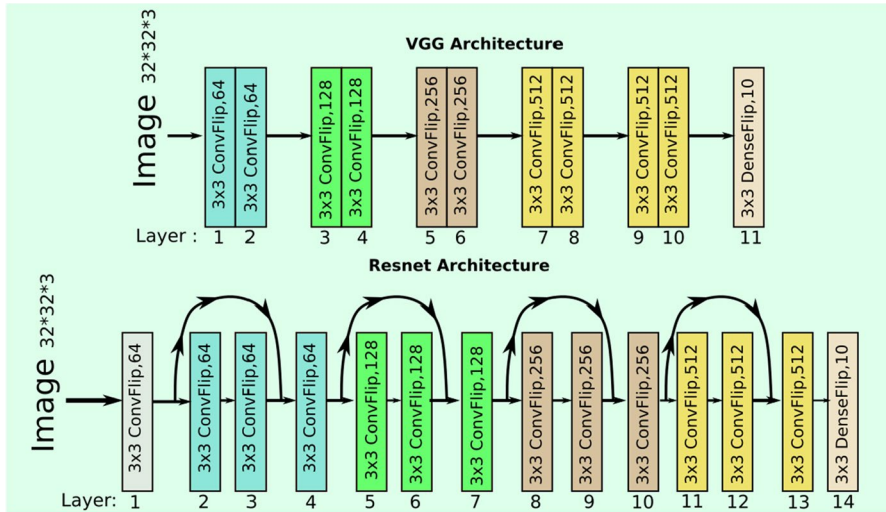


Fig. 1 A schematic of the network architectures used to understand the performance of BNN. For both the VGG and Resnet architectures, batch normalization and activation layer are not shown and only Bayesian flipout convolutional layers are represented

Sect. 4.6. The MNIST transformation 0.1 Million images were generated by pseudo-random deformations and translations of the original MNIST data using the package by Bottou et al. [37]. Given the larger volume this dataset allows us to scale up the batch size for the distributed training of the VGG-16 network.

In this work we demonstrate the use of distributed training of BNNs using data parallelism with Horovod [38] for two image classification models, VGG-16 [27] and Resnet-18 [28], applied to the CIFAR-10 dataset. In the conventional VGG network the architecture consists of convolutions layers with maxpooling and batch normalization operations. In the Resnet architecture the input is passed through a block consisting of convolution, batch normalization and max pooling operations. Subsequent layers in the network are structured as blocks featuring convolution, batch normalization and ReLU activations except in the last operation of a block, and ReLU activation is not performed. So-called shortcut connections are also made between the blocks with an stride of 2. Further details on the architecture and implementation can be found in He et al. [28]. In a BNN implementation the conventional convolutions layers are replaced by the Bayesian convolution layers which, at runtime, are sampled using a so-called *flipout technique* [18]. This technique uses a sign flip operation to sample the weights with minimum covariance for each layer. The fully connected layers are also replaced by fully connected probabilistic layers. The priors are chosen to be standard normal, and VI is performed assuming the mean-field approximation [39]. The total number of trainable parameters for VGG-16 and Resnet-18 is 18 million and 9 million, respectively. The representation diagram of the BNN networks is shown in Fig. 1. For the pruning study, presented in Sect. 4.6, we demonstrate the use of our software package, BPrune, with a network composed of 2 convolution flipout and 1 dense flipout layers (BNN-Conv), and a

second network composed of 3 dense flipout layers (BNN-FC). Further details about the network structures and hyperparameters can be seen in ‘Appendix’. To perform inference on a held-out test set, and to estimate the prediction uncertainty intervals, we Monte Carlo sample from the approximate predictive distribution. This sampling increases the computational cost of the inference step in a BNN which can be significant in comparison with conventional neural networks.

3.2 Setup

A variety of software frameworks have been developed which are keeping pace with recent advances in probabilistic methods. Some of the most notable are Stan [40], PyStan [41], Edward, Pyro [42], Gen [43], Edward2 and TensorFlow Probability [44]. These frameworks are capable of running efficiently on a variety of architectures such as CPUs, GPUs and TPUs and provide programming flexibility and ease of coding probabilistic models. Furthermore, there are various techniques for improving the scaling performance during training for conventional networks, such as prefetching data, the use of mixed precision and data parallel distributed training. An efficient implementation to perform data parallel training is Horovod [38], an open-source library that employs efficient inter-communications via ring reduction.

For the current work we use Theta, a HPC cluster with second-generation Intel Xeon Phi™ processors, code named Knights Landing (KNL), at Argonne Leadership Computing Facility. The important architectural features to note for this study are Theta’s multi-core, hyperthreaded nodes (64 cores per node with 4 hyperthreads per core) [24]. We use TensorFlow (v1.14.1), TensorFlow Probability (v0.7.0) and Horovod (v0.18.1) for our benchmarking study. Tensorflow is compiled using GCC 8.2.0 and was linked to high-performance math libraries such as Intel MKL and MKL-DNN libraries, while Horovod is compiled using GCC 7.3.0 and linked to the Cray MPI Library. Optimal throughput performance was observed using 1 MPI rank per node, 64 cores per node and 2 threads per core for the distributed runs. Previous results [45] have shown that the best/optimal performance is obtained while using 1 MPI rank per node. Scaling studies for the two BNN architectures are performed up to 128 nodes for CIFAR-10 keeping the regularization parameter constant as $\beta = 1$. We considered a range of values $\beta \in \{0 - 1\}$; however, we found that the results did not change significantly. The β the KL-annealing term can play a significant role when working with variational autoencoders (VAE), where the chances of posterior collapse present a major challenge as the model tries to learn the input data representation. We also modified this parameter constant and found no improvement in the model performance. We then use the same setup with $\beta = 1$ to perform the scaling study of VGG-16 network for the MNIST transformed 0.1 million images. To demonstrate the effect of pruning we train two BNNs networks on the MNIST data.

For performing data distributed training the Tensorflow graph scheduler allows the order of operation execution to vary across workers, even in the case of similar models. We found that this is not helpful when performing collective operations during the distributed training of a BNN and can result into a deadlock. Horovod v0.18.1 introduced additional worker co-ordination logic which ensured that

all workers submit collective operations in a common order. A caching scheme is implemented, where the collective operations are processed and gathered by the coordinating ranks only once from the worker pool. Each rank also stores the broadcasted processed results in their cache. Horovod carries out these co-ordination processes at a frequent interval during the training which is referred to as a cycle time. Only collective operation requests are executed at the chosen cycle time across the workers. For effective network utilization Horovod has the ability to fuse individual collective operations. The procedure couples the cycle time and size of the collective messages. To find a optimal trade-off between the two is challenging, and therefore, grouping of collective operations is carried out only at a given cycle when a complete group of requests are present. In the case of multiple complete groups they are fused into a larger message. A lower bound on fusion to complete groups indirectly leads to a minimum message size making it agnostic to the cycle time. Optimal settings here can lead to efficient network utilization. Further details and scaling results of the implementation can be seen in detail by Laanait et al. [46]. For our work, we found the default cycle time of 5ms and fusion buffer size 64 MB to be optimal.

4 Results

Sections 4.1 and 4.2 present the performance analysis and scalability results for the VGG-16 and Resnet-18 networks used in this study. The inference using BNNs is discussed in Sect. 4.3. The results for the MNIST big-data run with 0.1 million images for BNN VGG-16 model are discussed in Sect. 4.4. The graphical processing unit (GPU) performance analysis results are presented in Sect. 4.5. The effect of pruning on a BNNs accuracy and a description of the BPrune pruning library are presented in Sect. 4.6.

4.1 Throughput and MPI statistics

Figure 2a shows the measured samples processed per second verses the number of nodes for a BNN (filled pink histogram) and conventional CNN (convolutional neural network) (unfilled histogram) VGG model. A fixed batch (mini-batch) size of 128 is used. The learning rate is fixed to 1×10^{-4} and is scaled by the number of nodes during the distributed training. The error bars represent the standard deviation of samples processed per second over all iterations. As the number of nodes used increases, the samples processed per second by a BNN network are nearly 50% less than the CNN counterpart. Similar trends are observed in the case of the Resnet architecture as shown in Fig. 2b. These results can be explained by the increased computational overhead of a BNN which contains approximately double the number of trainable parameters compared to a CNN and features the flipout sampling technique for every trainable parameter in the network. The training time based on a fixed number of 64 epochs is also compared for the two architectures as shown in Fig. 2c. It can be seen that training Resnet either with a CNN or BNN takes $\sim 25\%$ more time in comparison with the VGG model for the single-node run because of

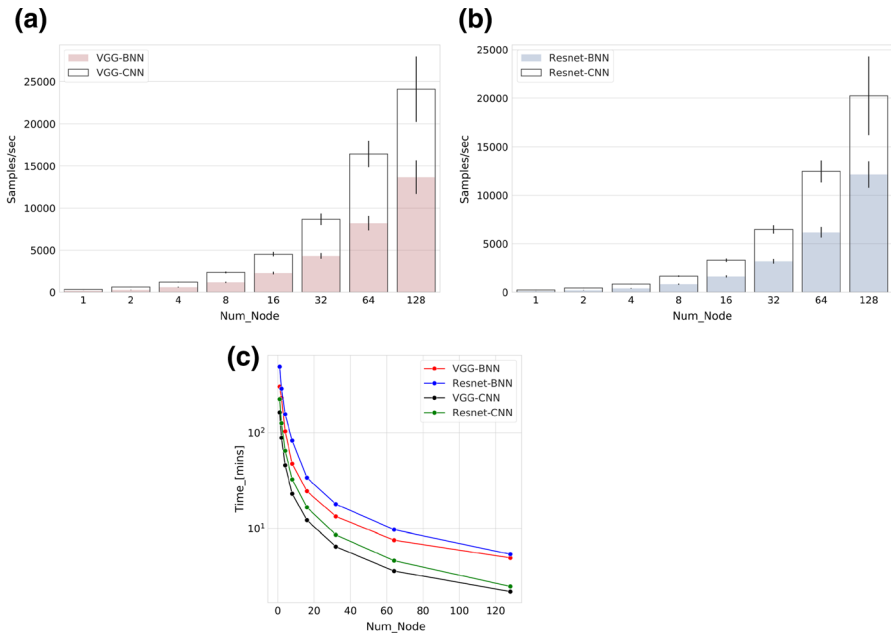


Fig. 2 Number of samples processed per second for a BNN and CNN implementation of the VGG (a) and Resnet (b) models. c A comparison of the training time (mins) for a fixed number of 64 epochs with a batch size of 256 for both a BNN and CNN implementation of the VGG and Resnet models as shown in the legend

the difference in architectures. For the BNN implementation we find that as we increase the number of workers, the time to train a VGG BNN on 128 nodes takes 3.37 min, while the VGG CNN model takes 1.3 min; for the Resnet BNN it takes 3.3 min and Resnet CNN takes 1.4 min. Overall the BNNs are found to take approximately a factor of 2.4 increase in the time to complete a fixed number of epochs for these models. As the number of ranks increases up to 128 ranks, the computation speed-up of 61% is achieved for VGG BNN and 90% for Resnet BNN architecture for same minibatch size.

Table 1 shows the main MPI routines outlining the number of calls, averages bytes and time in each, reported by rank 0 from a 64-node run (1 MPI rank per node) obtained from the MPI profiler HPCTW [47]. Overall the communication time for both models is significantly higher for BNNs than CNNs. In particular the all-reduce operations constitute most of the communication time and Horovod carries out more all-reduce operations for the BNN implementations of both models.

4.2 Scaling efficiency

To further understand the (weak) scalability of a BNN compared to a CNN, we compute the scaling efficiency as $\frac{T_1}{T_N \times N}$, where T_1 is the time to process fixed number of epochs with 1 rank; T_N is time to process fixed number of epochs with N ranks.

Table 1 MPI routine statistics for rank 0 from a 64-node run generated using MPI profiling for the VGG (upper) and Resnet (lower) models

MPI routine	#Calls		Avg. bytes		Time (s)	
	VGG-CNN	VGG-BNN	VGG-CNN	VGG-BNN	VGG-CNN	VGG-BNN
MPI_Bcast	518	641	218,303.9	352,552.1	0.859	1.339
MPI_Allreduce	31,427	62,126	233,707.6	236,335.1	260.842	558.553
MPI_Gather	184	228	4.0	4.0	0.751	1.016
MPI_Gatherv	184	228	0.0	0.0	2.395	2.904
MPI_Allgather	2	2	4.0	4.0	0.0	0.00
CNN			BNN			
Total communication time = 264.847 s			Total communication time = 563.812 s			
Total elapsed time = 294.493 s			Total elapsed time = 613.408 s			
	Resnet-CNN	Resnet-BNN	Resnet-CNN	Resnet-BNN	Resnet-CNN	Resnet-BNN
MPI_Bcast	508	846	116,119.0	139,279.8	0.725	1.590
MPI_Allreduce	26,525	74,667	144,347.7	102,487.3	322.862	709.909
MPI_Gather	174	321	4.0	4.0	1.031	2.064
MPI_Gatherv	174	321	0.0	0.0	2.972	3.299
MPI_Allgather	2	2	4.0	4.0	0.00	0.000
CNN			BNN			
Total communication time = 327.592 s			Total communication time = 716.862 s			
Total elapsed time = 351.908 s			Total elapsed time = 762.892 s			

Figure 3a shows the training efficiency curve based on recorded training time shown in Fig. 2 for VGG and Resnet with a BNN and CNN implementation. The rate of efficiency decline for the VGG BNN (red) is faster than that of the VGG CNN (black). We find the opposite trend for the Resnet BNN and CNN models where the BNN implementation scales better than the corresponding CNN; however, the two models are consistent within the 1σ error bars. We find that the FLOP rate for the BNN VGG-16 model is approximately 573.87 million, while for the CNN VGG it is 18.84 million. The dominant contribution for the VGG BNN model is the effect of the larger parameter set in inducing higher communication costs which makes it scale less efficiently. We record a FLOP rate of 299.02 million for the Resnet BNN model (blue) and 9.82 million for the CNN counterpart (green). The BNN implementation is slightly more compute intensive in both BNN models, and Horovod operates efficiently to overlap the compute and communication. We find that the communication efficiency is higher for the BNN Resnet model compared to the CNN counterpart.¹ Overall we find that communication costs dominate in the VGG

¹ The communication efficiency is calculated as the ratio of communication time (MPI_WTIME) to elapsed time (includes MPI_INIT & MPI_FINALIZE). For 16-node run the efficiency of BNN VGG and Resnet models is 86.81% and 87.59% respectively, while that of the CNN VGG and Resnet model are 80.26% and 88.59%, respectively. For a 128-node run the BNN VGG and Resnet model communication efficiencies are 91.15% and 94.91%, while for the CNN VGG and Resnet model efficiencies are 86.99% and 89.11%, respectively.

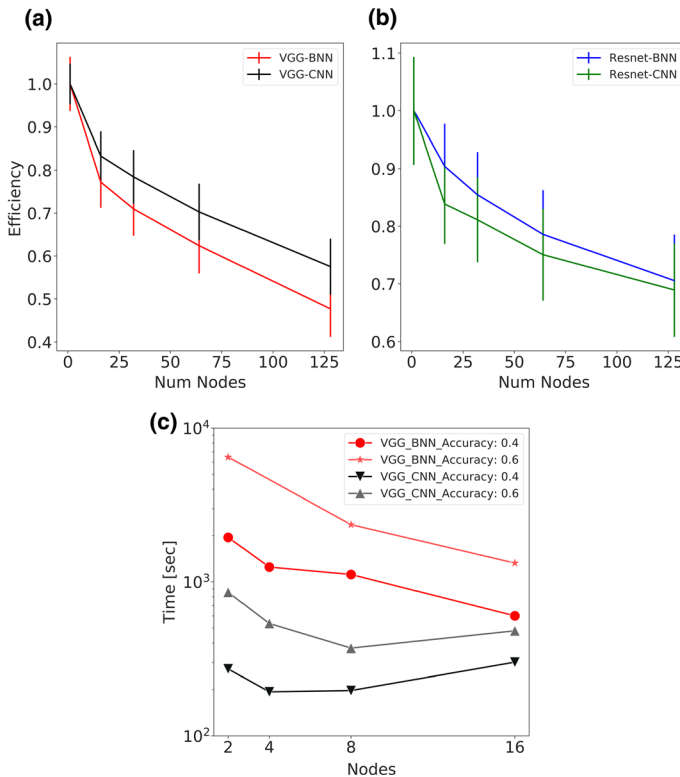


Fig. 3 The scaling efficiency for the BNN and CNN implementation of VGG (a) and Resnet (b). c The time taken to reach a fixed training accuracy with increasing number of nodes for the VGG-BNN and VGG-CNN architectures (colour figure online)

model compared to Resnet due to the larger number of parameters in the model, which contributes to it scaling less efficiently than the Resnet model for both the CNN and BNN implementation. The time to reach a fixed training accuracy is shown in Fig. 3c for the VGG BNN model. It can be seen that with the increase in the number of ranks the time to reach a fixed accuracy decreases. This plot shows the runtime for two training accuracies of 0.4 and 0.6 with different markers for each model as given in the legend. It is clear from this figure that the time to a given accuracy increases substantially for a BNN network shown in red compared to the corresponding CNN in black. We find that training the VGG model to an accuracy of 0.6 (0.4) takes approximately 7.57 (7.14) times longer for the BNN on two nodes. Running on 16 nodes we find runtimes are 2.76 (1.99) times longer for a BNN to reach an accuracy of 0.6 (0.4). Note that the minibatch size used in this case was 256. We find that when scaling up a larger effective batch helps reduce the difference between training a BNN and a CNN to the same level of accuracy. Note that the absolute value of the accuracy is not the focal point of this paper as key parameters such as the learning rate, batch size, warm-up period have not been optimized.

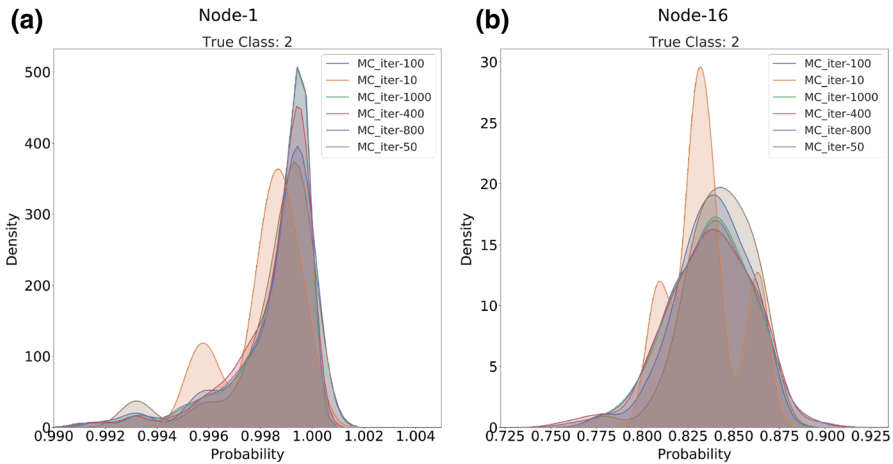


Fig. 4 The BNN VGG-16 predictive pdf of softmax values for the test image to be in class 2 for a given number of MC iterations as indicated in the legend (the true image class label is 2). These are plotted using the kernel distribution estimate method (KDE) for the test CIFAR10 dataset. The Node-1 (a) and Node-16 (b) models have been trained to an accuracy of 0.91 and 0.64, respectively

We would expect these to be different optimal parameters in a Bayesian Neural network compared to the conventional counterpart. By not optimizing these parameters we have attempted to make a strict comparison between the BNNs and conventional convolutional neural networks.

4.3 Inference using BNNs

The BNN VGG-16 and Resnet-18 models were trained for 64 epochs on one node to a training accuracy of 0.923 and 0.942, respectively. Using these trained models we carry out inference on the test data as in Eq. (3). To understand the effect of the number of Monte Carlo (MC) iterations on the predictive probability density function (pdf) we vary the number of iterations and record the softmax outputs for a given true class label. In Fig. 4 the predictive pdf of the softmax values for various MC number of iterations is plotted. Here we are predicting a class label of 2 for a single image when the true class label is 2. We also compare the results for the VGG BNN model trained on (a) 1 node (b) 16 nodes with training accuracies of 0.923 and 0.64, respectively. The range of MC iterations sampled from was 10–1000. From Fig. 4 it is clear that 10 MC iterations result in a noisy pdf in both panels. We find convergent results for the pdfs with MC iteration ≥ 400 . The difference in the accuracy between the Node-1 and Node-16 model is apparent here. An accuracy of 0.923 for Node-1 results in a narrow pdf close to 1 for MC iterations ≥ 400 , while the reduced accuracy of the Node-16 model results in a pdf with larger variance. Note that we see similar trends for other test images. The runtime for inference using 10 MC iterations was $\sim 5\%$ of the runtime for 1000 iterations, which were found to be 152.4 s and 2861.0 s, respectively. Clearly the number of MC iterations plays an important role in the computational cost of inference in a BNN.

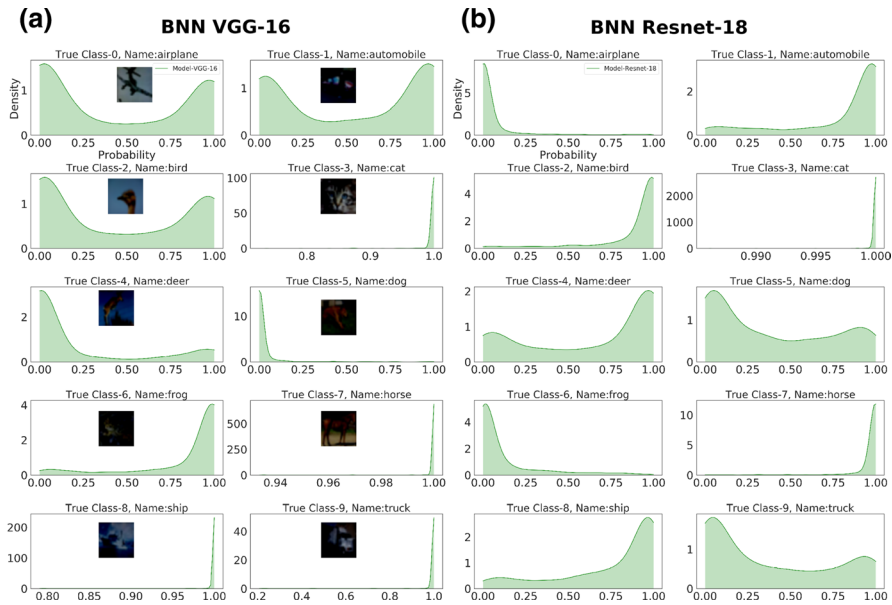


Fig. 5 The predictive pdf of the softmax values output from the VGG-16 model in (a) and the Resnet model in (b) using a sample of test images in each class. These distributions represent the softmax values for class X when the true class is X (where X is in the range 0–9 as in the plot titles) and are obtained by running the trained BNN model for 400 MC iterations

In Fig. 5 we show the predictive pdf of the softmax values output from the VGG-16 model in (a) and the Resnet model in (b) using a sample of test images in each class. These distributions represent the softmax values for class X when the true class is X and are obtained by running the trained BNN model for 400 MC iterations. In this figure if the model was performing perfectly on the test dataset each panel would show a sharp peak at unity. Note that both models were trained on a single node for a fixed number of epochs to a training accuracy of 0.923 and 0.942 for VGG and Resnet, respectively. We include these results as representative outputs from two different model implementations of a Bayesian neural network applied to the same test set.

Comparing the results from each model on the same test image it can be seen that for some of the images both models behave similarly correctly identifying the image, as for test class 7 and 3. For test image labels 0, 4 and 5 when one model precisely misclassifies the image (sharp peak around zero probability) the other model shows large uncertainties in their classification. For test image label 6 the VGG model correctly classifies with little variance, while the Resnet model misclassifies with little variance. Overall the output from a BNN contains a richer set of information compared to a corresponding CNN, which can be used to understand model performance and carry out model comparisons. We shall explore this topic in future work.

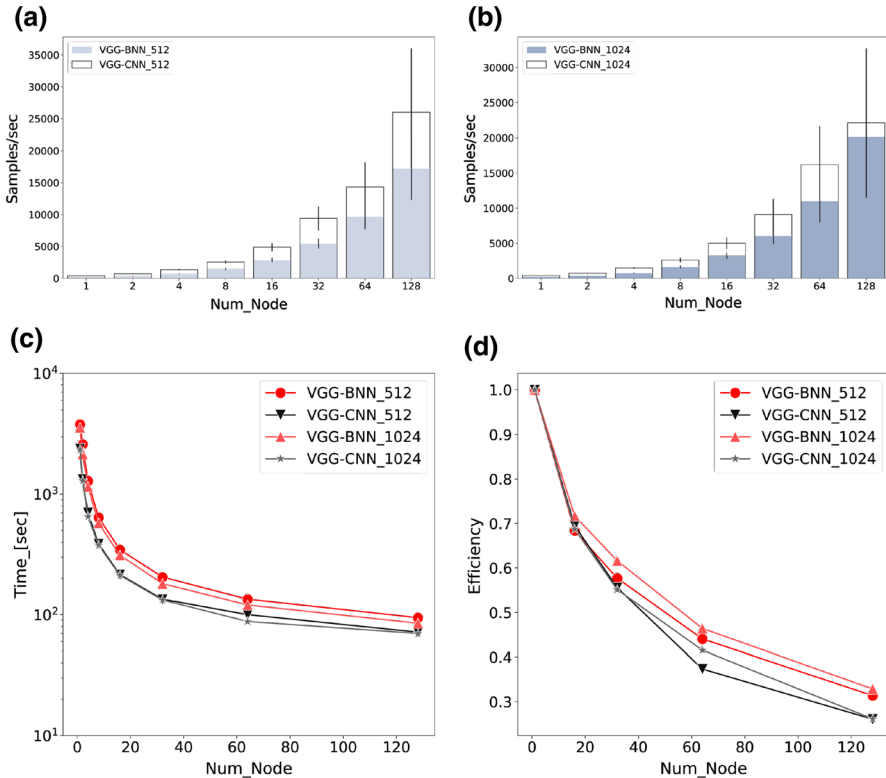


Fig. 6 The measured samples per second processed by the BNN VGG-16 model versus the number of nodes for the MNIST transformed 0.1M images for a batch size of 512 (a) and 1024 (b). c The training time in seconds with the increasing number of nodes for different batch-size run for 12 epochs. d The efficiency curve for BNN, CNN VGG model with two different batch sizes with increasing number of nodes

4.4 Scalability of VGG-16 on MNIST-Transformed

HPC resources at leadership computing facilities are typically used to process and train DL models with big data. Therefore, we extended the distributed training performance evaluation study of BNNs using the VGG-16 model applied to the 0.1 million MNIST transformed image dataset [37]. When scaling up the number of nodes the number of epochs was fixed to 12, and all other hyperparameters and priors are unchanged from the previous study. Figure 6 shows a histogram of the measured samples processed per second versus the number of nodes for the VGG model with a batch size of 512 in panel (a) and 1024 in panel (b). The error bars (black) represent the standard deviation in samples per second over all iterations. As the batch size increases from 512 to 1024 we see a change in the mean sample per second processed in the range of 5–12% for every node from 1 to 128 for VGG BNN model. From this figure it is clear that increasing the throughput for the CNN model saturates at a batch size of 512 with little improvement using 1024, but the throughput

for the BNN does improve using the larger batch size. Figure 6c shows the total runtime to complete 12 epochs for 512 and 1024 minibatch sizes using the CNN and BNN models. As expected using larger minibatches reduces the time to complete a fixed number of epochs with similar improvement going from 512 to 1024 minibatch size in the CNN and BNN. For example using 16 nodes the runtime is reduced by $\sim 10\%$ using a minibatch size of 1024 compared to 512. Making a qualitative comparison with Fig. 2c it is clear that increasing the batch size as we increase the number of nodes improves the performance of the BNN model and can significantly reduce the difference in training time when comparing with the corresponding CNN. This effect of the larger minibatch size can also be seen in the efficiency plot shown in Fig. 6d. In this figure the larger minibatch size reduces the difference in scaling efficiency between the BNN and CNN. However the actual efficiencies for 512 and 1024 minibatch sizes are much lower at 128 nodes compared to those in Fig. 3.

4.5 GPU scaling study

We analyze the performance of the VGG-16 BNN and CNN networks on the NVIDIA-DGX-1 V100 GPU cluster² at ALCF. The model training was performed with a minibatch sizes of 512 and 1024 using the Adam optimizer and a learning rate of 1×10^{-4} . The MNIST transformed dataset is used for training up to 8 GPUs for 12 epochs. All other distributed training settings are the same as in previous sections for comparison. Figure 7a shows the image samples processed per second versus the number of GPUs. It can be seen that the throughput for both the BNN and CNN scales linearly with the increase in the number of GPUs. Figure 7b shows a comparison between the throughput for the VGG BNN model on the GPU (shaded blue histogram) and previous results on the CPU (green filled histogram). Figure 7c shows a similar comparison for the CNN model on the GPU and CPU. It is clear from the figure that the throughput capabilities of GPUs are significantly higher for both the BNN and CNN model. Using 8 GPUs we find ~ 29 times more samples per second compared to running on the same number of Xeon KNL nodes for BNN model and ~ 18.20 times more samples per second for the CNN model on the GPU compared to the CPU. For both the BNN and CNN models we find that nearly 128 Xeon Phi nodes are needed to achieve the same throughput as 4 V100 GPUs. Overall, in the study 8 GPUs gave the best training throughput for a batch size of 512, with $\sim 42\text{K}$ samples for BNN and $\sim 46\text{K}$ samples for CNN.

The runtime for 12 epochs for the VGG model on the CPUs and GPUs is shown in Fig. 7c. We find that the difference in runtime for the BNN is $\sim 10\times$ less on a single GPU compared to the CPU and $\sim 11\times$ less on 8 GPUs compared to 8 CPUs. As a stark comparison we find that the runtime for 12 epochs on 8 GPUs is 1.67 times faster than using 128 KNL nodes for the BNN model, highlighting the difference in

² The configuration for the GPUs used at ALCF is as follows, 8X Tesla V100, GPU total system memory of 128 GB, a CPU with Dual 20-Core Intel Xeon E5-2698 v4 2.2 GHz, 40,960 NVIDIA CUDA Cores, 5120 NVIDIA Tensor Cores, System memory of 512 GB 2133 MHz DDR4 LRDIMM, Storage of 4X 1.92 TB SSD RAID 0, and Dual 10 GbE, 4 IB EDR network.

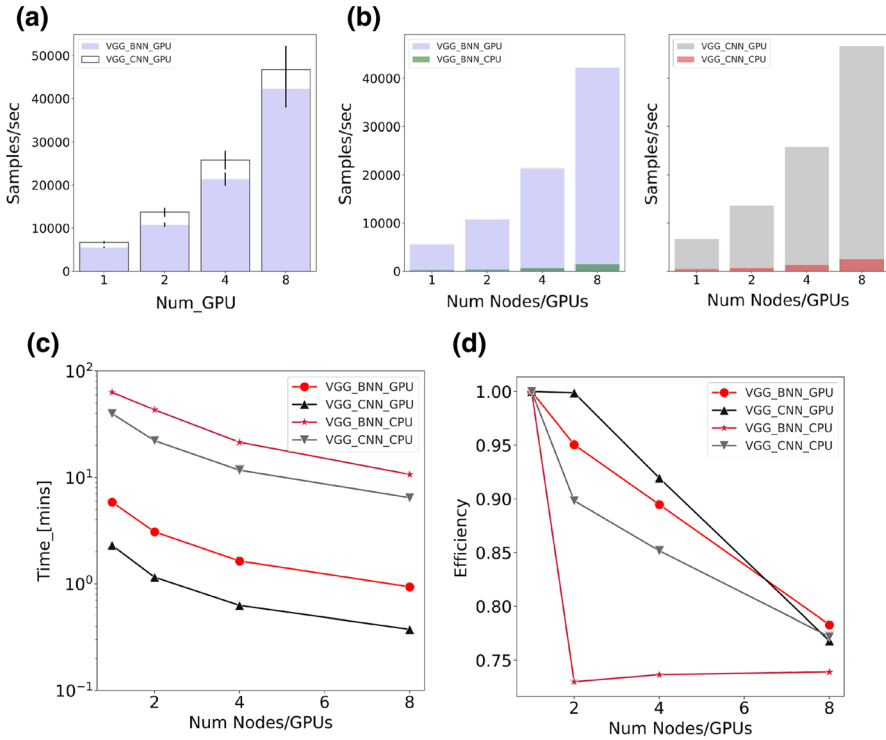


Fig. 7 The VGG-BNN scaling performance on a NVIDIA-DGX V100 work station for the MNIST 0.1M image dataset with a minibatch size of 512. **a** The samples processed per second with increasing number of GPUs. **b** Compared the throughput on GPUs and CPUs for the BNN and CNN VGG models. **c** The total runtime to complete 12 epochs with increasing number of GPUs and CPUs for CNN and BNN model. **d** The efficiency of both the BNN and CNN models with increasing number of GPUs and CPUs (colour figure online)

performance on the different architectures. As shown in previous sections the runtime for the BNN model is larger than the corresponding CNN regardless of whether we are using a CPU or GPU. On 8 GPUs the runtime of the BNN is 56.21 s, while the CNN runtime is 22.51 s; on 8 CPU nodes the runtimes for the BNN and CNN models are 641.79 s and 387.58 s, respectively. Figure 7d shows the efficiency curve for the VGG BNN and CNN models. The scaling trends are similar between the BNN and CNN up to 8 GPUs. We find that the efficiency scaling is better on the GPUs compared to the CPUs with less discrepancy between the BNN and CNN. In future work we will examine the scaling behaviour to higher numbers of GPUs.

4.6 Post-training pruning of BNNs

The process of removing weights from a neural network is referred as pruning. This procedure has been introduced in the past to reduce the network complexity and improve generalization [48, 49]. In the case of BNNs with probabilistic layers

the procedure has been found to be helpful in limiting the computational cost and memory demands. The study by Graves [6] has found that pruning can also improve the final performance by reducing the noise in the gradient estimates. In addition, the work by Blundell et al. [12] has shown in a BNN pruning experiment that by reducing a given network by up to 95% the accuracy is not significantly affected. For various practical applications of BNN models, for example when deployed for inference on edge devices, they will bring with them a higher computational cost. This overhead is due to the number of MC iterations required to produce robust predictions and uncertainty estimates. As shown in Fig. 4 with the VGG-16 BNN network, if someone chooses to run 1000 MC iterations it is 2.5 times more computationally expensive in comparison with 400 iteration and 18.3 times in comparison with 10 iterations. With a pruned smaller network with sparse computations the inference time can be reduced for BNNs without sacrificing accuracy.

In this work we present an open-source software package, *BPrune*,³ which has been developed as an add-on for Tensorflow and Tensorflow Probability. This software automates the post-training pruning procedure using a user-defined signal-to-noise threshold to zero outweights from the graph. The signal-to-noise threshold is set by the user. Although BPrune is intended for use to prune a trained network prior to carrying out inference, it should be installed before training as information on the final graph needs to be extracted and saved. BPrune brings with it the necessary utilities to write the required files after training as well as to reload the trained model for inference or pruning. A high-level description of how BPrune operates is as follows:

- Once training of the BNN is completed the model can be saved in the TensorFlow native file format which writes *model-<iteration>.ckpt* in the checkpoint directory. This binary file contains the trained weight matrices of the BNN. This file is output using the conventional procedure of Tensorflow [50].
- TensorFlow Probability lacks routines to easily reload a saved model for restarts, inference or for any additional model modifications such as pruning. BPrune addresses this by outputting two additional text files, *LayerNames.txt* and *OpsNames.txt*, after training.
- *LayerNames.txt*: This file contains an ordered list of Bayesian and non-Bayesian layer names (also known as training variables) associated with each layer. This is needed for accessing each layers respective weight matrices. For example using a Gaussian prior for the weights, each layer in the network will have two matrices defined in the name scope of the layer, i.e. a matrix for mean (μ) and sigma (σ) named as 'kernel' and 'un-transformed scale', respectively. These names are listed into the file associated with each layer. At inference time BPrune parses this file and identifies the layers associated with training variable for pruning.
- *OpsNames.txt*: This file contains all the operations which are defined by Tensorflow to execute the graph. This information is needed during the re-load to identify the input and output placeholders, the names of the sampling operation

³ <https://github.com/Himscipy/BPrune>.

and other metrics such as accuracy, if defined. For example, in the case of a BNN model trained to perform image classification we define a categorical distribution for the prediction output and compute the log probability. This operation is identified as ‘log_prob’ for a label distribution name scope and is listed as one of the operations in the OpsNames.txt file. BPrune parses the file to automatically identify the minimum required operations for running inference or pruning with the test/held-out dataset.

- To run the model for the inference and pruning the user needs to provide the case directory including the two text files and the checkpoint directory containing the saved checkpoint files, to BPrune. If a user wishes to prune the model a threshold value can be passed as a command line argument which has a default set to 10.0. The user can also set the number of MC iterations to be used for generating prediction samples.
- When pruning BPrune automatically calculates the signal-to-noise (SNR)⁴ ratio for individual layers and zeros the weights below the given threshold.
- BPrune outputs a binary file which contains the used test samples, the labels, the total number of nonzeros in the network, the nonzeros per layer, the predictive probability distributions and the inference runtime.

The motivation for creating the BPrune package is to address the limitation of re-loading the trained Tensorflow Probability-based model for inference and provide a framework for pruning a BNN model by computing the SNR for each layer and then automatically pruning them based on a user-defined threshold. BPrune also computes the signal-to-noise ratio for the complete network by collecting the individual layer ratio as a single array to visualize the global signal-to-noise ratio for the whole network, saving manual intervention and efforts. The code is agnostic to loading model graphs either trained serially or using distributed training. A user can load the trained model at any iteration by simply specifying the name of the Tensorflow checkpoint file (‘model-<iteration>.ckpt’). In addition, the object-oriented programming approach used for developing the framework can be used to run multiple independent inference or pruning jobs using high-throughput computing in parallel.

To demonstrate BPrune in action we train two BNNs on the MNIST dataset. One network has convolutional and fully connected layers, while the second network has only fully connected layers. These models will be referred to as Model-Conv and Model-FC, respectively. The details of these network architectures and training parameters can be found in ‘Appendix’. Both Model-Conv and Model-FC are trained for 23 epochs until the training accuracy is 0.98 and 0.90, respectively. We then output each trained model to a Tensorflow native file format with the full graph and parameter information needed by BPrune. In Fig. 8 we show the effect on the test accuracy when varying the signal-to-noise threshold of weights pruned from the network. This corresponds to a pruning percentage as indicated on the x -axis in

⁴ The ratio for the Gaussian prior overweights can be simply calculated as a ratio $|\mu|/\sigma$ [6]. In the initial BPrune release Gaussian priors are supported. Other choices of distribution will be supported in future releases.

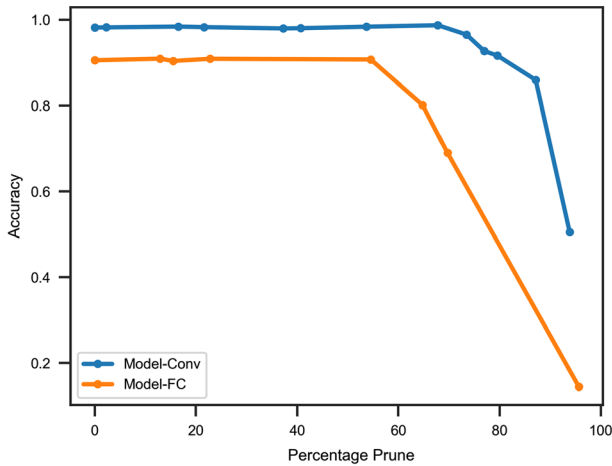


Fig. 8 The test accuracy for Model-Conv and Model-FC versus varying percentage of the network pruned (higher percentages indicate high sparsity in the network) using the MNIST test data

Fig. 8, i.e. retaining all weights regardless of their signal-to-noise would represent zero pruning, while removing all weights below a fixed signal-to-noise threshold corresponds to a certain percentage of the network being removed. For the particular models used here we find that the Model-Conv test accuracy remains unchanged as the pruning percentage increases to approximately 80% of the network. The Model-FC test accuracy is unchanged after pruning to approximately 60% of the network and additional pruning after this reduces the accuracy. To illustrate the effect of pruning on inference, we profiled the Model-FC Bayesian network inference with 0% and 70% of the model pruned for a single MC iteration. We find that the inference time is reduced by $\sim 50\%$ for the 70% pruned network compared to running the full network. It is worth noting that the current Tensorflow graph operations do not support sparse computation which may speed up matrix multiplication in the pruned network. Introducing this functionality in future could significantly speed up the inference for pruned networks and will be explored in future work.

In addition it is worth considering the effect of pruning on the prediction uncertainties for individual classes. We show the predictive pdfs for the softmax values for a sample of test images generated by running the trained models Model-Conv (a) and Model-FC (b) for inference with 200 MC iterations with differing percentages of pruning in Fig. 9. Note that in this plot the pdf of softmax values for a class X from 200 MC iterations is shown when the actual true class is X , for the MNIST dataset. Remarkably for some of the test images chosen, pruning to over 90% of the network has little impact on the predictive pdf as seen for Model-Conv for class 3, 6, 8 and 9; and for Model-FC for class 7. For both models, the test image for the digit 5 we can see that the model loses the confidence in the prediction as the network is pruned. For this image with $\sim 0\%$ pruning we see a peaked pdf at unity, while at $\sim 90\%$ pruning the pdf becomes bimodal with peaks at 0 and 1, showing the uncertainty caused by losing relevant weights from the network. Comparing the effect of

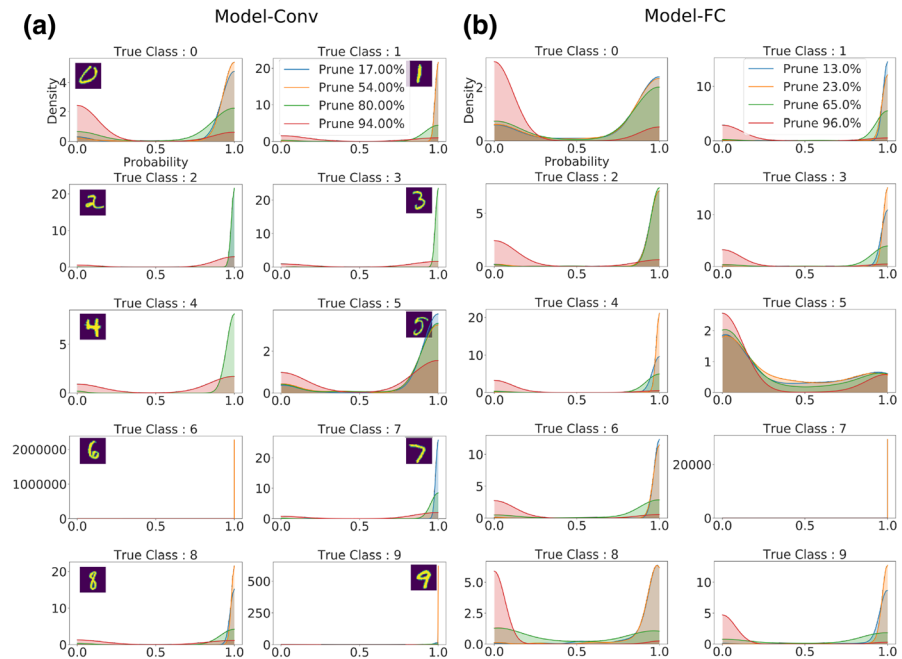


Fig. 9 The predictive pdfs for the softmax values from 200 MC iterations for a sample of test images generated by running the trained models Model-Conv (a) and Model-FC (b) for inference with differing percentages of pruning as given in the legend

pruning on the Model-Conv and Model-FC pdfs for these chosen test images, we find that pruning affects the Model-FC network to a greater extent than Model-Conv as can be seen for test image 8 and 9 in Fig. 9. Convolutional networks have proven to be significantly more robust for computer vision applications compared to fully connected BNNs, and these results are consistent with our results on pruning BNNs.

5 Discussion

The training and inference results presented in this work used a customized build of the distributed training framework, Horovod, which implements data-parallel training. On theta the MPI communication between the Horovod workers uses Cray MPICH libraries. Horovod copies the TensorFlow operation graph across different ranks to process unique mini-batches of training data. To compute the average gradient from each rank an all-reduce operation is inserted at the backpropagation computation.

During this study we used Horovod versions 0.16 and 0.18. Using version 0.16 we encountered failures when running the BNN models on ≥ 4 nodes as some ranks

were stalling due to a failure in negotiating all reduce. On further analysis, we found that the stalls and failures using version 0.16 could be attributed to the following breakdown in Horovod operations: (a) Rank 0 is the central scheduler in Horovod operations and waits on each rank to accumulate tensors for broadcasting and all-reduce. (b) As the Tensorflow process is independent to schedule its operations in the graph, if one or more ranks experience load imbalance and attempts to execute the all-reduce operation in different orders the operation will stall resulting in a deadlock. We found that the BNN model suffered from this load imbalance which was causing the failures.

Horovod version 0.18 [38] features some significant improvements in how the all reduce is carried out among the workers. In this version the developers have introduced a dynamical reordering of all-reduce operations so that consistency can be maintained for all ranks. This is called negotiating all-reduce. This is done by improving the gradient reduction strategy. The worker co-ordination is done by a lightweight BitAllReduce, and grouping of gradient tensors is performed based on the buffer size to perform reduction operations for the buffered tensors at fixed interval (cycle-time). The implementation provides a improvement in various aspects of running deep learning at scale on large models and big data with Horovod. In profiling the BNNs in this work, we found that the network has over a thousand all-reduce operations per step which results in the controller being forced to receive and then send millions of messages per second for larger jobs. This bottleneck for BNN models was reduced by improving the collective reduction implementation. Our findings outline the challenges in running BNNs at scale due to the large reductions operations in conjunction with more compute, which require efficient use of hardware and systems interconnect together with optimize MPI routines.

6 Conclusions

We present a performance analysis for the distributed training of Bayesian neural networks. As the use of machine learning for decision-making increases for scientific and industrial applications, quantifying uncertainties becomes increasingly important. BNNs are compute intensive and in a distributed setting requires significant communication; as a result training large BNN models present unique challenges for future architectures. The following are the main results of this paper for the image classification models VGG-16 and Resnet-18:

- The throughput for BNNs is approximately 50% less than the corresponding CNN for small batch sizes on KNL nodes.
- For BNN small batch sizes we find approximately a factor of 2.4 increase in the runtime for a fixed number of epochs.
- The neural network model size plays a role in the scalability and efficiency with increasing number of nodes, and BNNs can suffer from reduced efficiency as we scale up. BNN efficiency can be improved to match or outperform the cor-

responding CNN by increasing the batch size or varying the Horovod cycle time and buffer size.

- Overall we see a $30\times$ increase in the FLOP rate for BNNs compared to CNNs.
- Runtime to a fixed accuracy can be up to a factor of $\sim 7\times$ longer on a small number of nodes but reduced to a factor of $\sim 3\times$ longer on ≥ 16 nodes.
- Increasing the batch size to 512 or 1024 can improve the BNN's performance and reduce the difference in training time and throughput compared to the corresponding CNN, especially as the number of nodes increases.
- For inference a minimum of 400 MC iterations is needed to produce robust pdfs.
- Using 8 GPUs, we find a $\sim 29(18)\times$ increase in the throughput for BNNs (CNNs) compared to running on an equivalent number of KNL nodes.
- For certain architectures pruning to 60% of the network has little effect on the inference accuracy. The effect of pruning the majority of the network, e.g. 90%, has noticeable effects of the final pdf produced for certain test images.

Our findings indicate that the computational overheads of training a BNN network can be reduced with a distributed training framework which employs efficient MPI communication schemes. The reported results on the Cray XC40 could also be used as a baseline for projections on the next generation of hardware architectures. We hope that this study is useful to the data science community providing insights into the distributed training with Horovod on HPC clusters. We plan to extend this work to understand the scalability of BNNs with different models such as recurrent neural networks, autoencoders and generative adversarial networks with large datasets.

Acknowledgements This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. This research was funded in part and used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the US DOE or the US Government. Declaration of Interests—none.

Appendix: Network information and hyperparameter details

Information about the VGG-16 and Resnet-18 networks together with details of the individual layers is shown in Tables 2, 3 and Tables 4, 5, respectively. For distributed training the learning rate was scaled by the number of nodes. The initial learning rate was fixed to 1×10^{-4} with batch size of 256 and ReLU activation function. A mean field normal distribution with zero mean and unit variance is used to define the prior $p(\theta)$ for the weights; the approximate kernel posterior $q(\theta|x)$ is initialized by normal distribution with mean $\mu = -9.0$ and standard deviation as 0.1. The distributed model training was carried using the Adam optimizer.

The models used for the pruning study are shown in Tables 6 and 7 with ReLU as the activation function in each layer. Model training was done serially, with a learning rate of 1×10^{-3} and a batch size of 100 with RMSProp as the optimizer.

Table 2 BNN VGG-16 architecture used for distributed training

Layer (type)	Output shape	Param #
input_1 (InputLayer)	(None, 32, 32, 3)	0
Conv_I_0 (Conv2DFlipout)	(None, 32, 32, 64)	3520
batch_normalization_v1	(None, 32, 32, 64)	256
activation (Activation)	(None, 32, 32, 64)	0
Conv_II_0 (Conv2DFlipout)	(None, 32, 32, 64)	73,792
batch_normalization_v1_1	(None, 32, 32, 64)	256
activation_1 (Activation)	(None, 32, 32, 64)	0
Max_I_0 (MaxPooling2D)	(None, 16, 16, 64)	0
Conv_I_1 (Conv2DFlipout)	(None, 16, 16, 128)	147,584
batch_normalization_v1_2	(None, 16, 16, 128)	512
activation_2 (Activation)	(None, 16, 16, 128)	0
Conv_II_1 (Conv2DFlipout)	(None, 16, 16, 128)	295,040
batch_normalization_v1_3	(None, 16, 16, 128)	512
activation_3 (Activation)	(None, 16, 16, 128)	0
Max_I_1 (MaxPooling2D)	(None, 8, 8, 128)	0
Conv_I_2 (Conv2DFlipout)	(None, 8, 8, 256)	590,080
batch_normalization_v1_4	(None, 8, 8, 256)	1024
activation_4 (Activation)	(None, 8, 8, 256)	0
Conv_II_2 (Conv2DFlipout)	(None, 8, 8, 256)	1,179,904
batch_normalization_v1_5	(None, 8, 8, 256)	1024
activation_5 (Activation)	(None, 8, 8, 256)	0
Max_I_2 (MaxPooling2D)	(None, 4, 4, 256)	0
Conv_I_3 (Conv2DFlipout)	(None, 4, 4, 512)	2,359,808
batch_normalization_v1_6	(None, 4, 4, 512)	2048
activation_6 (Activation)	(None, 4, 4, 512)	0
Conv_II_3 (Conv2DFlipout)	(None, 4, 4, 512)	4,719,104
batch_normalization_v1_7	(None, 4, 4, 512)	2048
activation_7 (Activation)	(None, 4, 4, 512)	0
Max_I_3 (MaxPooling2D)	(None, 2, 2, 512)	0
Conv_I_4 (Conv2DFlipout)	(None, 2, 2, 512)	4,719,104
batch_normalization_v1_8	(None, 2, 2, 512)	2048
activation_8 (Activation)	(None, 2, 2, 512)	0
Conv_II_4 (Conv2DFlipout)	(None, 2, 2, 512)	4,719,104
batch_normalization_v1_9	(None, 2, 2, 512)	2048
activation_9 (Activation)	(None, 2, 2, 512)	0
Max_I_4 (MaxPooling2D)	(None, 1, 1, 512)	0
flatten (Flatten)	(None, 512)	0
Dense_I_4 (DenseFlipout)	(None, 10)	10,250
Total params: 18,829,066		
Trainable params: 18,823,178		
Nontrainable params: 5888		

Table 3 CNN VGG-16 architecture used for distributed training

Layer (type)	Output shape	Param #
input_1 (InputLayer)	(None, 32, 32, 3)	0
Conv_I_0 (Conv2D)	(None, 32, 32, 64)	1792
batch_normalization (BatchNo	(None, 32, 32, 64)	256
activation (Activation)	(None, 32, 32, 64)	0
Conv_II_0 (Conv2D)	(None, 32, 32, 64)	36,928
batch_normalization_1 (Batch	(None, 32, 32, 64)	256
activation_1 (Activation)	(None, 32, 32, 64)	0
Max_I_0 (MaxPooling2D)	(None, 16, 16, 128)	0
Conv_I_1 (Conv2D)	(None, 16, 16, 128)	73,856
batch_normalization_2 (Batch	(None, 16, 16, 128)	512
activation_2 (Activation)	(None, 16, 16, 128)	0
Conv_II_1 (Conv2D)	(None, 16, 16, 128)	147,584
batch_normalization_3 (Batch	(None, 16, 16, 128)	512
activation_3 (Activation)	(None, 16, 16, 128)	0
Max_I_1 (MaxPooling2D)	(None, 8, 8, 256)	0
Conv_I_2 (Conv2D)	(None, 8, 8, 256)	295,168
batch_normalization_4 (Batch	(None, 8, 8, 256)	1024
activation_4 (Activation)	(None, 8, 8, 256)	0
Conv_II_2 (Conv2D)	(None, 8, 8, 256)	590,080
batch_normalization_5 (Batch	(None, 8, 8, 256)	1024
activation_5 (Activation)	(None, 8, 8, 256)	0
Max_I_2 (MaxPooling2D)	(None, 4, 4, 512)	0
Conv_I_3 (Conv2D)	(None, 4, 4, 512)	1,180,160
batch_normalization_6 (Batch	(None, 4, 4, 512)	2048
activation_6 (Activation)	(None, 4, 4, 512)	0
Conv_II_3 (Conv2D)	(None, 4, 4, 512)	2,359,808
batch_normalization_7 (Batch	(None, 4, 4, 512)	2048
activation_7 (Activation)	(None, 4, 4, 512)	0
Max_I_3 (MaxPooling2D)	(None, 2, 2, 512)	0
Conv_I_4 (Conv2D)	(None, 2, 2, 512)	2,359,808
batch_normalization_8 (Batch	(None, 2, 2, 512)	2048
activation_8 (Activation)	(None, 2, 2, 512)	0
Conv_II_4 (Conv2D)	(None, 2, 2, 512)	2,359,808
batch_normalization_9 (Batch	(None, 2, 2, 512)	2048
activation_9 (Activation)	(None, 2, 2, 512)	0
Max_I_4 (MaxPooling2D)	(None, 1, 1, 512)	0
flatten (Flatten)	(None, 512)	0
Dense_I_4 (Dense)	(None, 10)	5130
Total params: 9,421,898		
Trainable params: 9,416,010		
Nontrainable params: 5888		

Table 4 BNN Resnet-18 architecture used for distributed training

Layer (type)	Output shape	Param #	Connected to
input_1 (InputLayer)	(None, 32, 32, 3)	0	
cond2d_flipout	(None, 32, 32, 64)	3520	input_1[0][0]
batch_normalization	(None, 32, 32, 64)	256	cond2d_flipout[0][0]
activation	(None, 32, 32, 64)	0	batch_normalization[0][0]
conv2d_flipout_2	(None, 32, 32, 64)	73,792	activation[0][0]
batch_normalization_1	(None, 32, 32, 64)	256	conv2d_flipout_2[0][0]
activation_1	(None, 32, 32, 64)	0	batch_normalization_1[0][0]
conv2d_flipout_3	(None, 32, 32, 64)	73,792	activation_1[0][0]
conv2d_flipout_1	(None, 32, 32, 64)	8256	activation[0][0]
add	(None, 32, 32, 64)	0	cond2d_flipout_3[0][0] cond2d_flipout_1[0][0]
batch_normalization_2	(None, 32, 32, 64)	256	add[0][0]
activation_2	(None, 32, 32, 64)	0	batch_normalization_2[0][0]
conv2d_flipout_5	(None, 16, 16, 128)	147,584	activation_2[0][0]
batch_normalization_3	(None, 16, 16, 128)	512	conv2d_flipout_5[0][0]
activation_3	(None, 16, 16, 128)	0	batch_normalization_3[0][0]
conv2d_flipout_6	(None, 16, 16, 128)	295,040	activation_3[0][0]
conv2d_flipout_4	(None, 16, 16, 128)	16,512	activation_2[0][0]
add_1	(None, 16, 16, 128)	0	cond2d_flipout_6[0][0] cond2d_flipout_4[0][0]
batch_normalization_4	(None, 16, 16, 128)	512	add_1[0][0]
activation_4	(None, 16, 16, 128)	0	batch_normalization_4[0][0]
conv2d_flipout_8	(None, 8, 8, 256)	590,080	activation_4[0][0]
batch_normalization_5	(None, 8, 8, 256)	1024	conv2d_flipout_8[0][0]
activation_5	(None, 8, 8, 256)	0	batch_normalization_5[0][0]
conv2d_flipout_9	(None, 8, 8, 256)	1,179,904	activation_5[0][0]
conv2d_flipout_7	(None, 8, 8, 256)	65,792	activation_4[0][0]
add_2	(None, 8, 8, 256)	0	cond2d_flipout_9[0][0] cond2d_flipout_7[0][0]
batch_normalization_6	(None, 8, 8, 256)	1024	add_2[0][0]
activation_6	(None, 8, 8, 256)	0	batch_normalization_6[0][0]
conv2d_flipout_11	(None, 4, 4, 512)	2,359,808	activation_6[0][0]
batch_normalization_7	(None, 4, 4, 512)	2048	conv2d_flipout_11[0][0]
activation_7	(None, 4, 4, 512)	0	batch_normalization_7[0][0]
conv2d_flipout_12	(None, 4, 4, 512)	4,719,104	activation_7[0][0]
conv2d_flipout_10	(None, 4, 4, 512)	262,656	activation_6[0][0]
add_3	(None, 4, 4, 512)	0	cond2d_flipout_12[0][0] cond2d_flipout_10[0][0]
batch_normalization_8	(None, 4, 4, 512)	2048	add_3[0][0]
activation_8	(None, 4, 4, 512)	0	batch_normalization_8[0][0]
average_pooling2d	(None, 1, 1, 512)	0	activation_8[0][0]
flatten	(None, 512)	0	average_pooling2d[0][0]
dense_flipout	(None, 10)	10,250	flatten[0][0]
Total params: 9,814,026			
Trainable params: 9,810,058			
Nontrainable params: 3968			

Table 5 CNN Resnet-18 architecture used for distributed training

Layer (type)	Output shape	Param #	Connected to
input_1 (InputLayer)	(None, 32, 32, 3)	0	
Conv_block_I (Conv2D)	(None, 32, 32, 64)	1792	input_1[0][0]
batch_normalization_1 (BatchNorma	(None, 32, 32, 64)	256	Conv_block_I[0][0]
activation (Activation)	(None, 32, 32, 64)	0	batch_normalization_1[0][0]
Conv_block_I_0 (Conv2D)	(None, 32, 32, 64)	36,928	activation[0][0]
batch_normalization_1 (BatchNor	(None, 32, 32, 64)	256	Conv_block_I_0[0][0]
activation_1 (Activation)	(None, 32, 32, 64)	0	batch_normalization_1[0][0]
Conv_block_II_0 (Conv2D)	(None, 32, 32, 64)	36,928	activation_1[0][0]
conv2d (Conv2D)	(None, 32, 32, 64)	4160	activation[0][0]
add (Add)	(None, 32, 32, 64)	0	Conv_block_II_0[0][0] conv2d[0][0]
batch_normalization_2 (BatchNor	(None, 32, 32, 64)	256	add[0][0]
activation_2 (Activation)	(None, 32, 32, 64)	0	batch_normalization_2[0][0]
Conv_block_I_1 (Conv2D)	(None, 16, 16, 128)	73,856	activation_2[0][0]
batch_normalization_3 (BatchNor	(None, 16, 16, 128)	512	Conv_block_I_1[0][0]
activation_3 (Activation)	(None, 16, 16, 128)	0	batch_normalization_3[0][0]
Conv_block_II_1 (Conv2D)	(None, 16, 16, 128)	147,584	activation_3[0][0]
conv2d_1 (Conv2D)	(None, 16, 16, 128)	8320	activation_2[0][0]
add_1 (Add)	(None, 16, 16, 128)	0	Conv_block_II_1[0][0] conv2d_1[0][0]
batch_normalization_4 (BatchNor	(None, 16, 16, 128)	512	add_1[0][0]
activation_4 (Activation)	(None, 16, 16, 128)	0	batch_normalization_4[0][0]
Conv_block_I_2 (Conv2D)	(None, 8, 8, 256)	295,168	activation_4[0][0]
batch_normalization_5 (BatchNor	(None, 8, 8, 256)	1024	Conv_block_I_2[0][0]
activation_5 (Activation)	(None, 8, 8, 256)	0	batch_normalization_5[0][0]
Conv_block_II_2 (Conv2D)	(None, 8, 8, 256)	590,080	activation_5[0][0]
conv2d_2 (Conv2D)	(None, 8, 8, 256)	33,024	activation_4[0][0]
add_2 (Add)	(None, 8, 8, 256)	0	Conv_block_II_2[0][0] conv2d_2[0][0]
batch_normalization_6 (BatchNor	(None, 8, 8, 256)	1024	add_2[0][0]
activation_6 (Activation)	(None, 8, 8, 256)	0	batch_normalization_6[0][0]
Conv_block_I_3 (Conv2D)	(None, 4, 4, 512)	1,180,160	activation_6[0][0]
batch_normalization_7 (BatchNor	(None, 4, 4, 512)	2048	Conv_block_I_3[0][0]
activation_7 (Activation)	(None, 4, 4, 512)	0	batch_normalization_7[0][0]
Conv_block_II_3 (Conv2D)	(None, 4, 4, 512)	2,359,808	activation_7[0][0]
conv2d_3 (Conv2D)	(None, 4, 4, 512)	131,584	activation_6[0][0]
add_3 (Add)	(None, 4, 4, 512)	0	Conv_block_II_3[0][0] conv2d_3[0][0]
batch_normalization_8 (BatchNor	(None, 4, 4, 512)	2048	add_3[0][0]
activation_8 (Activation)	(None, 4, 4, 512)	0	batch_normalization_8[0][0]
average_pooling2d (AveragePooli	(None, 4, 4, 512)	0	activation_8[0][0]
flatten (Flatten)	(None, 512)	0	average_pooling2d[0][0]
Dense_I_3 (Dense)	(None, 10)	5130	flatten[0][0]

Table 5 (continued)

Layer (type)	Output shape	Param #	Connected to
Total params: 4,912,458			
Trainable params: 4,908,490			
Nontrainable params: 3968			

Table 6 The BNN-Conv model used to demonstrate pruning with BPrune

Layer (type)	Output shape	Param #
input_1 (InputLayer)	(None, 28, 28, 1)	0
Conv_1 (Conv2DFlipout)	(None, 5, 5, 256)	13,056
Max_I_1 (MaxPooling2D)	(None, 2, 2, 256)	0
Conv_2 (Conv2DFlipout)	(None, 5, 5, 256)	3,277,056
flatten (Flatten)	(None, 2560)	0
Dense_I_4 (DenseFlipout)	(None, 10)	512,010
Total params: 3,802,122		
Trainable params: 3,802,122		
Nontrainable params: 0		

Table 7 The BNN-FC model used to demonstrate pruning with BPrune

Layer (type)	Output shape	Param #
input_1 (InputLayer)	(None, 784)	0
den_1 (DenseFlipout)	(None, 256)	401,664
den_2 (DenseFlipout)	(None, 256)	131,328
den_3 (DenseFlipout)	(None, 10)	5130
Total params: 538,122		
Trainable params: 538,122		
Nontrainable params: 0		

References

1. Neal RM (1995) Bayesian learning for neural networks. Technical report

2. Williams C (1996) Computing with infinite networks. In: Advances in neural information processing systems, vol 9. MIT Press, Cambridge, pp 295–301

3. MacKay DJC (1992) A practical Bayesian framework for backpropagation networks. Neural Comput 4(3):448–472

4. Hinton G, Van Camp D (1993) Keeping neural networks simple by minimizing the description length of the weights. In: Proceedings of the 6th Annual ACM Conference on Computational Learning Theory. Citeseer

5. Barber D, Bishop CM (1998) Ensemble learning in Bayesian neural networks. NATO ASI Ser Ser F Comput Syst Sci 168:215–237

6. Graves A (2011) Practical variational inference for neural networks. In: Advances in neural information processing systems. pp 2348–2356

7. Hoffman MD, Blei DM, Wang C, Paisley J (2013) Stochastic variational inference. J Mach Learn Res 14(1):1303–1347

8. Paisley J, Blei D, Jordan M (2013) Variational Bayesian inference with stochastic search. arXiv preprint [arXiv:1206.6430](https://arxiv.org/abs/1206.6430)
9. Kingma DP, Welling M (2013) Auto-encoding variational Bayes. arXiv preprint: [arXiv:1312.6114](https://arxiv.org/abs/1312.6114)
10. Rezende DJ, Mohamed S, Wierstra D (2014) Stochastic backpropagation and approximate inference in deep generative models
11. Titsias M, Lázaro-Gredilla M (2014) Doubly stochastic variational Bayes for non-conjugate inference. In: International Conference on Machine Learning, pp 1971–1979
12. Blundell C, Cornebise J, Kavukcuoglu K, Wierstra D (2015) Weight uncertainty in neural networks. arXiv preprint [arXiv:1505.05424](https://arxiv.org/abs/1505.05424)
13. Shridhar K, Laumann F, Liwicki M (2019) A comprehensive guide to Bayesian convolutional neural network with variational inference. arXiv preprint [arXiv:1901.02731](https://arxiv.org/abs/1901.02731)
14. Hinton GE, Srivastava N, Krizhevsky A, Sutskever I, Salakhutdinov RR (2012) Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint [arXiv:1207.0580](https://arxiv.org/abs/1207.0580)
15. Gal Y, Ghahramani Z (2016) Dropout as a Bayesian approximation: representing model uncertainty in deep learning. In: International Conference on Machine Learning, pp 1050–1059
16. Tran D, Dusenberry M, van der Wilk M, Hafner D (2019) Bayesian layers: a module for neural network uncertainty. In: Advances in neural information processing systems, pp 14633–14645
17. Shazeer N, Cheng Y, Parmar N, Tran D, Vaswani A, Koanantakool P, Hawkins P, Lee H, Hong M, Young C et al (2018) Mesh-tensorflow: deep learning for supercomputers. In: Advances in neural information processing systems, pp 10414–10423
18. Wen Y, Vicol P, Ba J, Tran D, Grosse R (2018) Flipout: efficient pseudo-independent weight perturbations on mini-batches. arXiv preprint: [arXiv:1803.04386](https://arxiv.org/abs/1803.04386)
19. Tsyplikhin A (2019) Graphcore delivers 26x performance gains for finance customers
20. Nabarro S (2019) Probabilistic-modelling-by-combining-markov-chain-monte-carlo-and-variational-inference-with-ipus. <https://www.graphcore.ai/posts/probabilistic-modelling-by-combining-markov-chain-monte-carlo-and-variational-inference-with-ipus>
21. Cerebras Systems (2019) Cerebras wafer scale engine: an introduction. <https://www.cerebras.net/wp-content/uploads/2019/08/Cerebras-Wafer-Scale-Engine-An-Introduction.pdf>. Accessed Nov 2019
22. Baydin AG, Shao L, Bhimji W, Heinrich L, Meadows L, Liu J, Munk A, Naderiparizi S, Gram-Hansen B, Louppe G et al (2019) Etalumis: bringing probabilistic programming to scientific simulators at scale. arXiv preprint [arXiv:1907.03382](https://arxiv.org/abs/1907.03382)
23. Viebke A, Memeti S, Pllana S, Abraham A (2019) Chaos: a parallelization scheme for training convolutional neural networks on Intel Xeon Phi. J Supercomput 75(1):197–227
24. ALCF (2019/2020) Xc40 machine overview. Technical report
25. Vipin Kumar EK, Ying H, Jing X (2017) Numpy/Scipy with Intel® MKL and Intel® compilers. <https://software.intel.com/en-us/articles/numpyscipy-with-intel-mkl>
26. Cray-MPICH. The MPICH source wiki
27. Simonyan K, Zisserman A (2014) Very deep convolutional networks for large-scale image recognition. arXiv preprint [arXiv:1409.1556](https://arxiv.org/abs/1409.1556)
28. He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp 770–778
29. Bowman SR, Vilnis L, Vinyals O, Dai AM, Jozefowicz R, Bengio S (2015) Generating sentences from a continuous space. arXiv preprint [arXiv:1511.06349](https://arxiv.org/abs/1511.06349)
30. Higgins I, Matthey L, Pal A, Burgess C, Glorot X, Botvinick M, Mohamed S, Lerchner A (2017) beta-vae: learning basic visual concepts with a constrained variational framework. ICLR 2(5):6
31. Alemi AA, Poole B, Fischer I, Dillon JV, Saurous RA, Murphy K (2017) Fixing a broken ELBO. arXiv preprint [arXiv:1711.00464](https://arxiv.org/abs/1711.00464)
32. Liu X, Gao J, Celikyilmaz A, Carin L et al (2019) Cyclical annealing schedule: a simple approach to mitigating kl vanishing. arXiv preprint [arXiv:1903.10145](https://arxiv.org/abs/1903.10145)
33. Ranganath R, Gerrish S, Blei DM (2013) Black box variational inference. arXiv preprint [arXiv:1401.0118](https://arxiv.org/abs/1401.0118)
34. Naesseth CA, Ruiz FJR, Linderman SW, Blei DM (2017) Reparameterization gradients through acceptance–rejection sampling algorithms. In: Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017
35. Krizhevsky A et al (2009) Learning multiple layers of features from tiny images. Technical report, Citeseer

36. LeCun Y (1998) The MNIST database of handwritten digits. <http://yann.lecun.com/exdb/mnist/>. Accessed June 2019
37. Loosli G, Canu S, Bottou L (2007) Training invariant support vector machines using selective sampling
38. Sergeev A, Del Balso M (2018) Horovod: fast and easy distributed deep learning in TensorFlow. arXiv preprint [arXiv:1802.05799](https://arxiv.org/abs/1802.05799)
39. Blei DM, Kucukelbir A, McAuliffe JD (2017) Variational inference: a review for statisticians. University of California, Berkeley
40. Carpenter B, Gelman A, Hoffman MD, Lee D, Goodrich B, Betancourt M, Brubaker M, Guo J, Li P, Riddell A (2017) Stan: a probabilistic programming language. *J Stat Softw* 76(1)
41. Stan Development Team et al (2017) PyStan: the Python interface to Stan. Version 2.16. 0.0
42. Bingham E, Chen JP, Jankowiak M, Obermeyer F, Pradhan N, Karaletsos T, Singh R, Szerlip P, Horsfall P, Goodman ND (2019) Pyro: deep universal probabilistic programming. *J Mach Learn Res* 20(1):973–978
43. Cusumano-Towner MF, Saad FA, Lew AK, Mansinghka VK (2019) Gen: a general-purpose probabilistic programming system with programmable inference. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019. ACM, New York, pp 221–236
44. Dillon JV, Langmore I, Tran D, Brevdo E, Vasudevan S, Moore D, Patton B, Alemi A, Hoffman M, Saurous RA (2017) Tensorflow distributions. arXiv preprint [arXiv:1711.10604](https://arxiv.org/abs/1711.10604)
45. Huihuo Z, Elise J. Deep learning with Keras, TensorFlow, PyTorch, and Horovod on Theta. <https://www.alcf.anl.gov/support-center/training-assets/deep-learning-keras-tensorflow-pytorch-and-horovod-theta>
46. Laanait N, Romero J, Yin J, Young MT, Treichler S, Starchenko V, Borisevich A, Sergeev A, Matheson M (2019) Exascale deep learning for scientific inverse problems. arXiv preprint [arXiv:1909.11150](https://arxiv.org/abs/1909.11150)
47. HPCTW. MPI HPC tool. https://www.alcf.anl.gov/files/Knight_Software_JobSubmission_2.pdf
48. LeCun Y, Denker JS, Solla SA (1990) Optimal brain damage. In: Advances in neural information processing systems, pp 598–605
49. Giles CL, Omlin CW (1994) Pruning recurrent neural networks for improved generalization performance. *IEEE Trans Neural Netw* 5(5):848–851
50. Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Watteberg M, Wicke M, Yu Y, Zheng X (2015) TensorFlow: large-scale machine learning on heterogeneous systems. Software available from www.tensorflow.org. Accessed June 2019

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.