Distributed Deep Learning for Broadly Deployed Health Care

Beng Chin Ooi  
School of Computing  
National University of Singapore  
Singapore  
ooibc@comp.nus.edu.sg

Gang Chen  
College of Computer Science and Technology  
Zhejiang University  
Hangzhou, China  
cg@cs.zju.edu.cn

Kee Yuan Ngiam  
Department of Surgery  
National University Hospital  
Singapore  
kee_yuan_ngiam@nuhs.edu.sg

Sheng Wang  
Database and Storage Lab  
Alibaba Group  
Singapore  
sh.wang@alibaba-inc.com

Qian Lin  
School of Computing  
National University of Singapore  
Singapore  
linqian@comp.nus.edu.sg

Wei Wang  
School of Computing  
National University of Singapore  
Singapore  
wangwei@comp.nus.edu.sg

James Yip  
Department of Cardiology  
National University Health Systems  
Singapore  
james_yip@nuhs.edu.sg

Mehui Zhang  
School of Computer Science and Technology  
Beijing Institute of Technology  
Beijing, China  
mehui_zhang@bit.edu.cn

Abstract—While chatbots and self-driving cars may be the most prominent exemplars of deep learning systems deployed in society, personalized health care is perhaps the application that promises the greatest benefits in the foreseeable future. The central barrier to widespread adoption of dynamic personalized health care is the high cost and inadequate performance of the deep learning systems required. This paper describes a breakthrough in the scalability and usability of a deep learning system, providing a way to get past this barrier. The consequence is better health outcomes through care personalized not just based on static personal characteristics, such as genetic make up, but also dynamic personal characteristics, such as health events and behaviors.

I. JUSTIFICATION

The team has achieved cloud-based distributed deep learning at scale. Thousands of users, having a wide range of technical sophistication, are able to use the system and obtain real-time assistance in caring for health in the field.

II. PERFORMANCE ATTRIBUTES

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category of achievement</td>
<td>Scalability and time-to-solution</td>
</tr>
<tr>
<td>Type of method used</td>
<td>Both explicit and implicit</td>
</tr>
<tr>
<td>Results reported on the basis of</td>
<td>Whole application including I/O</td>
</tr>
<tr>
<td>Precision reported</td>
<td>Single precision</td>
</tr>
<tr>
<td>System scale</td>
<td>Results measured on full-scale system</td>
</tr>
<tr>
<td>Measurement mechanism</td>
<td>Timers, FLOP count</td>
</tr>
</tbody>
</table>

III. OVERVIEW OF THE PROBLEM

There is tremendous interest today in the possibility of using AI for a broad range of applications. Deep learning requires a great deal of data-intensive computation, which has spurred considerable research on systems that can provide this capability. However, actually deploying AI in practice at scale has remained challenging. This paper describes the first real system that had been deployed at scale.

The current excitement with AI is primarily on account of machine learning with deep neural networks, which have demonstrated remarkable capability in many tasks, including in particular tasks involving natural language or image understanding. A neural network is simply a collection of interconnected nodes, each with several inputs and one output. The output value of a node is computed as a weighted sum of the inputs received. In a typical set up, these nodes are arranged in layers with the output of a node being made available as input to all nodes in the next layer.

Deep learning [24], with a neural network that has several such layers of nodes, has shown impressive results in various domains in the last decade. In fact, neural networks were invented more than 30 years ago. The resurgence of deep learning is mainly fueled by deep models and big (labeled) training data. These impose a huge computational burden, which requires powerful computing units, e.g., Graphic Processing Units (GPU).
A. Problem 1: Large-scale Training

A critical set of parameters in a neural network are the weights used at each node to compute the weighted sum of inputs. These are learned in an automated manner, with training data, usually by performing a gradient descent computation. Not only is this computation expensive, it has to be performed many times for each weight, and the number of weights to set is very large, being the number of nodes times the average number of inputs at each node. In consequence, the training of a deep learning model could take days to finish. For example, training ResNet-50 [18] over ImageNet dataset [15] takes about 6 days on Nvidia P100 GPU. Distributed training that exploits GPUs across multiple nodes has the potential to speed up the training. Algorithm 1 shows the pseudo-code for one distributed training scheme, where the data is partitioned onto each worker (e.g., a GPU) and the model is replicated on every worker. In each iteration, the workers read a mini-batch of data (e.g., 256 images) from its partition and run BackPropagation [25] algorithm to compute the gradients of the weights, which are averaged together for weight update (Line 8 and 9) following stochastic gradient descent algorithms (SGD). For distributed training, scalability is the most important performance attribute. There are two scalability challenges in training large deep learning models.

- Convergence. When we distribute the training over multiple workers, we can run them either synchronously (like Algorithm 1) or asynchronously. Although asynchronous training can reduce the synchronization overhead, the training algorithm may not converge [38]. Synchronous training can converge, but it may converge to a lower accuracy [22] because the effective batch size (i.e., $N \cdot |B_i|$ in Algorithm 1) is large when there are many nodes.

- Communication. During training, the nodes have to synchronize the information, i.e., the weight values or gradients. For large models, the weight size could be 100 millions. When the computation on each node is fast, the communication is likely to become the bottleneck.

B. Problem 2: Efficient Inference

After training, the deep learning models are deployed on various devices for inference, ranging from workstations in computer centers to edge devices like mobile phones. On all of these, efficient inference is required to meet the latency service level objective (SLO) [12] and reduce resource consumption e.g., power. However, deeper and larger models tend to have better accuracy at higher computational cost. For example, the NasNet model [50] for image classification requires more than 20 GFLOPs for the inference of a single image. The BERT model [16] for natural language processing has about 110 million weights. How to maximize efficiency without degrading accuracy is a challenge.

C. Problem 3: Efficient Model Adaptation

Research on machine learning, including deep learning, usually has the assumption that the training distribution is the same as the test distribution. However, in real deployment, we often see that the new data does not follow the training distribution. Consequently, the model accuracy may drop significantly. This is called dataset shift [21]. A simple solution to dataset shift is to fine-tune the model, i.e., train the model over the new data. However, the cost of fine-tuning a large model can be too high for online deployment. How to adapt the model efficiently is a challenge.

D. Problem 4: Usability

The life-cycle of developing a deep learning application includes multiple phases, including data preparation, model construction, training, inference and adaption. Deep learning models are notoriously complex to implement and debug. Model construction and configuration require rich experience of deep learning. Distributed training is difficult to conduct as it needs to manage the cluster resources, like machines, networking and storage. Inference and training share the model but have different workflow and optimization targets. For example, we may train multiple models separately and deploy them together as an ensemble to improve the inference accuracy. These difficulties are a barrier for the broad adoption of deep learning. How to make deep learning easy to use is a challenging problem.

IV. CURRENT STATE OF THE ART

In this section, we introduce the state-of-the-art solutions towards the challenges mentioned in Section III.

A. Large-scale Training

As larger training datasets and bigger models are being used to improve accuracy [11], [36], memory requirement for training the model may easily exceed the capacity of a single CPU or GPU. In addition, the computational cost of training may be too high for a single commodity server, which results in unreasonably long training time. For instance, it takes 10 days [32], [42] to train the DCNN [23] with 1.2 million training images and 60 million weights using one GPU.

Algorithm 1 DistTrain(Model $M$, Data $D$, NumWorker $N$, Learning rate $\alpha$)

1: $D_1, D_2, \cdots D_N = \text{Partition}(D, N)$
2: $P \leftarrow \text{Random()} // \text{initialize weights}$
3: while Not converge do
4:   for each worker do //run in parallel
5:     $B_i = \text{GetOneBatch}(D_i)$
6:     $\Delta P_i = \text{BackPropagation}(M, P, B_i)$
7:   end for
8:   $\Delta P = \text{average}(\Delta P_1, \Delta P_2, \cdots, \Delta P_N)$
9:   $P = \alpha \ast \Delta P$
10: end while

Footnotes:
1 https://mlperf.org/results/
2 GFLOPs stands for Giga floating point operations; GFLOPS stands for Giga floating point operations per second.
Pytorch\(^3\) and Tensorflow [6] are the two most popular deep learning libraries. Each provides a comprehensive set of APIs for neural network components with which users can implement their deep learning models. In Pytorch, users need to explicitly manage the communication between the workers, which adds some difficulties and requirements to the users. In Tensorflow, the model is represented as a graph with each node as an execution device. For distributed training, as shown in Figure 1, the graph (and nodes) are replicated and assigned with different device location, i.e., worker IP:Port. The communication is then fully controlled by Tensorflow. Both are designed as general purpose libraries, without application-specific optimization in terms of convergence and communication.

Some ad-hoc optimization techniques have been proposed for special deep learning models, like convolutional neural networks (ConvNet), trying to improve the convergence with large batch size under synchronous training. Priya et. al [17] propose to adjust the learning rate (i.e., \(\alpha\) in Algorithm 1) based on the batch size and use a warm-up stage to make the training converge to better accuracy. They manage to scale the training of ResNet-50 [18] to 256 GPUs with the batch size of 8K. When applying the techniques to AlexNet [23], Yang et. al [43] can only scale the batch size to 2K. They propose to improve the convergence accuracy by applying adaptive learning rates for weights from different layers. They manage to scale the training of AlexNet from batch size of 128 to 32K without accuracy loss. The scalability w.r.t the number of workers is not reported. Currently, synchronous training is more stable and popular than asynchronous training. But researchers are still working on improving asynchronous training [27], [48].

### B. Efficient Inference

There is a stream of research on model compression for efficient inference. Han et al. [19] compress AlexNet [23] by reducing the convolution filters, replacing 3x3 convolution with 1x1 convolution, etc. The resulting network, named SqueezeNet, has the same accuracy as AlexNet, but is 500x smaller in terms of the parameter size. Since the publication of SqueezeNet, various compression techniques [10] have been proposed, including parameter pruning and sharing, low-rank factorization, quantization, low precision floating number, etc. These methods typically compress an existing network architecture, and thus may incur some accuracy loss due to compression. There are also attempts to design efficient network architectures [34], [37]. Again, there is a tradeoff between accuracy and efficiency.

### C. Efficient Model Adaptation

For deep learning models, the simplest model adaptation approach is to train (refine) the model over the new data. This process is called fine-tuning, which requires a label for each new example. However, the label may not be available or may be expensive to collect. Black Box Shift Estimation (BBSE) [28] adapts the model without the labels. However, it needs to fine-tune the model over the new data for multiple (> 10) epochs. The cost is non-negligible for online deployment.

### D. Usability

Deep learning is a "black art". There are many choices to be made in constructing the network architecture and configuring the training knobs (a.k.a hyper-parameters), and the results obtained can be very sensitive to these choices. For instance, the deep convolution neural network (DCNN) is suitable for image classification [23], recurrent neural network (RNN) for language modelling [35], and deep auto-encoders for multi-modal data analysis [40]. However, for a certain task and dataset, a specific DCNN or RNN model should be created and configured. Yet, there are few principled methods for making these choices. Rather, experts try neural network configurations they think will succeed, and get to one that performs well enough by trial and error.

To ease the burden on AI users, the concept of AutoML (automated machine learning) has been suggested. Mainstream cloud platform providers like Amazon AWS, Microsoft Azure and Google Cloud offer services (a.k.a, AI/ML as a service) to realize AutoML. They either provide the APIs for a specific application, e.g., image classification and sentiment analysis, or provide a service to train machine learning (including deep learning) models over user uploaded data. The first service is simple for both the users and the providers, but it may deliver low accuracy because the models are trained by Amazon, Microsoft and Google with a different dataset than users’ data. The second service has the potential to create a customized neural network over the user uploaded data. However, it is very complex as it includes the model construction, configuration, training and deployment services. There are research papers on each stage, e.g., Neural architecture search (NAS) [50] for model construction, random search for model hyper-parameter tuning [7], Clipper [12] and Tensorflow Serving for efficient inference. However, there is no open-source system providing the complete second service.

### V. Innovations Realized

#### A. Large-scale Training

Addressing both usability and scalability challenges requires a distributed training platform that supports various deep

---

\(^3\)https://pytorch.org/
learning models, that comes with an intuitive programming model (similar to MapReduce [14], Spark [44] and epiC [20] in spirit), and that is scalable. In this paper, we describe an open source distributed deep learning platform, called SINGA (Malay word for Lion), which tackles both usability and scalability challenges at the same time. SINGA is an Apache Software Foundation (ASF) open source project [1] – the first ASF distributed deep learning and machine learning platform. We introduce optimization techniques and GPU support for SINGA. SINGA provides a simple, intuitive programming model which makes it accessible even to non-experts. SINGA’s simplicity is driven by the observation that both the structures and training algorithms of deep learning models can be expressed using a simple abstraction: the neuron layer (or layer). In SINGA, the user defines and connects layers to form the neural network model, and the runtime transparently manages other issues pertaining to the distributed training such as partitioning, synchronization and communication. SINGA’s scalability comes from its flexible system architecture and specific optimization. Both synchronous and asynchronous training frameworks are supported, which enables users to readily explore and find an optimal training configuration for maximizing the convergence rate. Optimization techniques, including minimizing data transferring and overlapping computation and communication, are implemented to reduce the communication overhead from distributed training.

SINGA [38] provides a flexible distributed architecture as shown in Figure 2. One worker represents one model replica that runs Line 5 and 6 of Algorithm 1. The workers are assigned into groups. Each group has a master that is connected to the masters in other groups. In other words, the masters form a group. The communication within each group can be customized, e.g., using parameter server where the master is the server, or ring structure AllReduce. Two functions are provided for the communication

```c
put(key, var) // non-blocking
get(key) // blocking
```

where the put() function puts the gradients of weights, e.g., the weight matrix of a convolution layer, for synchronization and the get() function gets the “averaged” gradient for SGD updating (Line 9 of Algorithm 1). They together make the underlying communication protocol including the optimization transparent to the user. Specifically, put() buffers the data and does the real synchronization until there is a big chunk of data. In this way, we avoid transferring small messages which cannot fully utilize the network bandwidth. In addition, by letting put() return immediately, we can overlap the computation of other layers and the synchronization of data in the buffer.

It is flexible to customize the communication protocol. For example, we can customize the master group with the ring structure AllReduce and the normal group with parameter server structure, which has a lower latency than the flat organization of the workers. Alternatively, we can customize both types of groups with ring structure AllReduce, i.e., a hierarchical AllReduce. In SINGA, we can also delay the synchronization across groups. In other words, we do frequent synchronization within each group, e.g., per iteration, and periodic synchronization across groups [27], [45], e.g., after every 3 iterations. In this way, we can save the synchronization cost significantly.

B. Efficient Inference

The most time consuming operations in deep neural networks are from the convolution and dense layers. Therefore, in [8], we propose a technique called model slicing to reduce the computational cost of these two layers, which consequently shortens the inference latency. The convolution and dense layers transform the input features via matrix multiplication as shown in Figure 3, where the first column vector represents the input feature of one layer, the second column vector is the output feature and the matrix stands for the weights of the layer. Suppose the length of input (resp. output) feature vector is \( l_i \) (resp. \( l_o \)), then model slicing selects the sub-matrix, \((r \times l_i , rows, r \times l_o , columns)\) to transform the sub-feature vector (of length \( r \times l_i \)), where \( r \) is the slicing rate determined based on the latency budget. According to [8], \( r \leq \min\left( \sqrt{\frac{C_o}{C_t}}, 1 \right) \), where \( C_o \) is the latency for running the full model and \( C_t \) is the given budget. Various techniques [8] are proposed to retain the accuracy, including output re-scaling.
Besides slicing existing models, we also propose to design new models that can adaptively change its architecture against different input instances to reduce the inference cost [9]. It is based on the assumption that different instances have different complexities for analysis. Taking image classification as an example, some instances like balls are easy to recognize, whereas some instances like food are complex and thus difficult to recognize. Correspondingly, we need a larger model to process difficult instances and a smaller model to handle easy instances. In [9], we extend the NAS algorithm [29] by adding a lightweight branching network to implement this idea. The branching network estimates the importance weight of each path in the (big) neural network for the input instance. If the importance weight is small, then we can skip the path to get a smaller model for inference. As a result, the model architecture is adaptive to the instance and the overall inference efficiency is improved. The model is called instance-aware branching network, denoted as ISBNet.

C. Efficient Model Adaptation

In [49], we propose two simple and efficient methods for model adaptation in terms of label shift, which is a common type of dataset shift encountered in practice. Label shift happens when the real-world or test label distribution is different from the training label distribution; In other words, \( P(Y) \) is different. For label shift, it is typically assumed that the conditional distribution \( P(X|Y) \) is the same between the training distribution and deployment/test distribution [28]. We use image classification as an example to illustrate label shift [28]. Suppose we are building a pneumonia predictor [33] based on chest X-ray images. The predictor is a binary classifier trained over patients’ chest X-ray images with labels from doctors’ diagnosis on whether the patient has pneumonia. When we collect the data from a time point, the percentage of patients with pneumonia is 1%. We then deploy the model to help doctors to make diagnosis. After a while, there is a pneumonia outbreak and the percentage of patients with pneumonia goes up to 5%. Due to the fact that the existing model is quite conservative, we may miss many patients with pneumonia.

To solve the label shift problem mentioned above, we propose to transform the original classifier directly for the testing data without re-training. First, the importance weights vector is estimated using the ratio vector between the testing label distribution (estimated using the original classifier) and training label distribution. Second, for each example from the test dataset, it is fed into the original classifier, which generates a probability vector as the prediction result. Third, the prediction result is adapted by: 1) exponentiating each probability value with the corresponding reciprocal importance weights vector value; 2) multiplying each probability value with the corresponding importance weights vector value. For each adaptation method, the theoretical analysis on the optimality is given. Both methods cost almost no extra computation and no extra storage.

D. Usability

We develop the Rafiki [41] system to realize AutoML. Rafiki, which has been released together with Apache SINGA v2.0, provides services for model construction, hyperparameter tuning, model training and inference. Users simply upload their datasets and configure the service to conduct training and then deploy the model for inference. As a cloud service system, Rafiki manages the hardware resources, failure recovery, etc. By integrating these services into a single system, Rafiki efficiently utilize the underlying storage, communication protocols and computation resource. For example, Rafiki speeds up hyper-parameter tuning easily by distributed tuning, which has almost linear scalability.

In addition, Rafiki effectively manages the tradesoff between accuracy and efficiency for inference. One problem with practical deployment is the variability in load on the system over time. It is not uncommon to have high usage levels that are 10 times the average load, with extreme peaks being even higher. Existing solutions attempt to size models to the computing power, in a static manner. In consequence, computational resources could be wasted during off-peak hours. On the other hand, the inference latency could increase significantly when the workload exceeds system design. Rafiki proposes an adaptive ensemble modelling strategy to fully utilize the computational resource, maximize the prediction accuracy and minimize the latency. Ensemble modelling aggregates the prediction results from multiple models. The inference accuracy is typically better when there are more models. During off-peak hours, Rafiki feeds each example into all models for ensemble modelling to improve the prediction accuracy; During peak hours, Rafiki processes multiple examples in parallel, one per model. In this way, the throughput is increased and latency is decreased.

VI. REAL APPLICATIONS

The combination of SINGA and Rafiki technologies (Apache SINGA v2.0) was bundled in a customized package for health care, called GEMINI (GEneralizable Medical Information aNalysis and Integration system) [26] (shown in Figure 4). The data cleaning component of GEMINI [13] has been used to clean 13 years of historical health care data of National University Hospital (NUH) in Singapore. The GEMINI software stack has been integrated into NUH production system since December 2018. Now, the whole hospital data, including demographics, inpatient visits, outpatient visits, discharge summary, diagnoses, lab tests, medications and procedures, is fed to GEMINI for analytics every day. Every day, GEMINI needs to process about 1500 inpatient visits and 15000 outpatient visits. There are two health care applications running on top of GEMINI currently, namely, Readmission [30] and Disease Progression Modeling (DPM) for Chronic Kidney Disease (CKD) [46], [47].

For the Readmission application, GEMINI now goes through the data of all the inpatient patients every day and alerts doctors of high readmission risk patients through a user interface as shown in Figure 5. With the predicted explainable
risk factors, such as diseases and risk of readmission, provided by GEMINI for each patient, doctors are now able to categorize patients into different groups and provide personalized post-discharge care recommendation. One concrete example is cooperating with NUH Carehub: Doctors who are responsible for patient discharge are categorizing the discharged patients according to predicted disease risk factors and send them to NUH Carehub for corresponding post-discharge interventions and follow-ups.

For the Disease Progression Modeling application, GEMINI now goes through the data of all patients diagnosed with CKD stage 3, 4 and 5 every day and provides predicted progression trajectories for each patient. With the predictions provided by GEMINI, personalized care recommendation is now able to be given to doctors. The current recommendations are as follows: (1) For patients who are predicted to be in a less severe stage of CKD in the next 12 weeks, such as stage 1, 2 and 3, GEMINI recommends doctors to guarantee monitoring for these patients. (2) For patients who are predicted to have deterioration trends in the next 12 weeks, GEMINI recommends doctors to take more aggressive interventions for these patients. (3) For patients who are predicted to be in the most severe stage of CKD stably, i.e., stage 5, in the next 12 weeks, GEMINI recommends doctors to take actions like dialysis or kidney transplant for these patients. Figure 6 shows the predictions by the model for 3 patients corresponding to the 3 cases (recommendations).

The GEMINI system is the first widely deployed system providing personalized health care, as discussed further in the Implications section below.

VII. HOW PERFORMANCE WAS MEASURED

Ideally, we would like to measure performance of the GEMINI system deployed in the field. However, such measurement is difficult to do without impacting the delivery of care. Furthermore, no other system actually has the performance to be deployed, so we would not be able to present a comparison baseline even if we were able to measure our system in field. For these reasons, we measure the performance of our system in the lab on standard benchmarks also used by competing systems.

For large-scale training, we use a deep convolutional neural network, ResNet-50 [18], as the application. ResNet-50 is has 50 convolution layers for image classification. It requires 3.8 GFLOPs to pass a single image (of size 224x224) through the network. The training is more expensive. We train the model using the ImageNet dataset, which has 1.2 million...
To evaluate the performance of the adaptive network (IS-BNNet) for efficient inference, we compare our model with baseline models in terms of the model parameters, inference FLOPs and test accuracy for image classification over the CIFAR10 dataset. The baseline models include ResNet with 101 convolution layers which has no special optimization for inference, MoibleNetV2 [34] and ShuffleNetV2 [31] which are optimized with respect to the inference latency.

For model adaptation test, we use image classification as the application. We train ResNet-18 [18] (18 convolution layers) over the CIFAR10 dataset, which has 50,000 (resp. 10,000) training (resp. test) images over 100 categories. Multiple subsets are sampled from the test dataset to simulate different degrees of label shifts. We use the KL divergence to measure the difference between the training and generated testing datasets. We compare our proposed model adaptation methods with the baseline adaptation methods in terms of the testing accuracy.

To test the performance of Rafiki, we use it to tune hyper-parameters of ResNet-18 for image classification over the CIFAR10 dataset. We measure the running time and test accuracy for different cluster sizes to evaluate the scalability of distributed hyper-parameter tuning.

VIII. PERFORMANCE RESULTS

Fig. 7: Scalability test for distributed synchronous training.

A. Scalability Evaluation

The scalability test results are shown in Figure 7. The bars are for the throughput and the lines are for the communication cost. Note that there are 8 GPUs on each AWS instance. When the number of GPUs is less or equal to 8, it means a single node is used. TensorFlow has lightly better throughput for single node training. However, when we increase the number of nodes, SINGA exhibits better throughput and thus better scalability. This is mainly due to the communication and synchronization cost between nodes as shown in the figure (the blue lines). The training using TensorFlow on 128 GPUs is not stable due to connection errors; hence, we estimate the performance based on the results from 8, 16, 32 and 64 GPUs;
Overall, SINGA achieves 78.9% parallel efficiency from one instance (8 GPUs) to 16 instances (128 GPUs).

In addition, to demonstrate that SINGA is flexible to support different distributed training schemes, we train the AlexNet model\(^4\) over CIFAR10 using the asynchronous training scheme on CPUs. The results are shown in Figure 8. We can see that asynchronous training on a single node scales well, whereas the training in a cluster becomes unstable when there are more workers (nodes) involved. This is likely caused by the delay of the update [39], [48], which is more severe for training in a cluster than on a single node due to the inter-node communication.

**B. Inference Evaluation**

Figure 9 shows the results measured by the FLOPs and test accuracy. First, the tradeoff between accuracy and efficiency is clear. With more FLOPs, the accuracy is generally better. Second, we can observe that model ensemble methods are strong baselines which trade off accuracy for lower inference cost and that the varying width ensemble of ResNet-164 performs better than varying depth, especially in lower budget prediction. The finding demonstrates the superiority of width slicing over depth slicing by inserting multiple classifiers to the network. Third, with model slicing, we are able to approach the performance of the ensemble of varying width networks, but the computational (FLOPs) cost is significantly reduced.

---

\(^4\) https://code.google.com/p/cuda-convnet

Fig. 8: Distributed asynchronous training. Images from [39].

---

Multiple versions of ISBNet are compared in Table II against the baseline methods. First, we can see the tradeoff between accuracy and efficiency among the 6 ISB Nets. Second, ResNet-101 is very slow and large in comparison to the neural networks with optimized inference efficiency. Third, for the ISBNet instances that have similar test error as other networks, they are much more efficient, in terms of model parameter size and inference FLOPs.

**C. Model Adaptation Evaluation**

The results for model adaptation are shown in Figure 10, which show the testing accuracy of different adaptation methods with varying degree of label shift measured by KL divergence. Various baseline methods are compared: Applying the original classifier without adaptation, denoted as ‘orig’; Applying the exponentiation adaptation method, denoted as ‘m1’; Applying the multiplication adaptation method, denoted ‘m2’; Retrain the classifier for k epochs following BBSE [28], denoted as ‘rk’. Note that for ‘orig’, ‘m1’, and ‘m2’, there is no fine-tuning and thus there is (almost) no adaptation cost in terms of time and memory; for ‘rk’, the larger the k, the higher the computational cost is incurred.

First, as the KL divergence between the test distribution and training distribution becomes larger, the accuracy of the ‘orig’ method has a tendency to go down. A more interesting fact is that the accuracy of the adaptation methods goes up. This is because, as the KL divergence becomes larger, the testing distribution actually becomes easier to learn. Second, for retraining based methods, in this experiment, more re-training epochs seem to lead to better accuracy results. However, this will cost more computation time. Third, in all the experiments, our proposed transformation based methods work better or at least on par with re-training based adaptation methods. Note
TABLE II: Comparison of ISBNet with existing models. Data from [9].

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Test Error (%)</th>
<th>Model Params (M)</th>
<th>Inference FLOPs (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet101 [18]</td>
<td>6.25</td>
<td>42.51</td>
<td>2519.71</td>
</tr>
<tr>
<td>MobileNetV2 1.0× [34]</td>
<td>5.56</td>
<td>2.30</td>
<td>94.42</td>
</tr>
<tr>
<td>ShuffleNetV2 1.5× [31]</td>
<td>6.36</td>
<td>2.49</td>
<td>95.70</td>
</tr>
<tr>
<td>ISBNet(S)-R-0.1-T-0.8 + cutout</td>
<td>6.97 ± 0.12</td>
<td>0.57</td>
<td>48.62</td>
</tr>
<tr>
<td>ISBNet(S)-R-0.1-T-0.8 + cutout</td>
<td>6.20 ± 0.16</td>
<td>0.57</td>
<td>44.16</td>
</tr>
<tr>
<td>ISBNet(S)-R-1.0-T-0.8 + cutout</td>
<td>8.26 ± 0.08</td>
<td>0.57</td>
<td>27.60</td>
</tr>
<tr>
<td>ISBNet(M)-R-0.0-T-0.8 + cutout</td>
<td>4.56 ± 0.10</td>
<td>1.86</td>
<td>123.23</td>
</tr>
<tr>
<td>ISBNet(M)-R-0.1-T-0.8 + cutout</td>
<td>5.04 ± 0.12</td>
<td>1.86</td>
<td>105.41</td>
</tr>
<tr>
<td>ISBNet(M)-R-1.0-T-0.8 + cutout</td>
<td>6.67 ± 0.11</td>
<td>1.86</td>
<td>67.55</td>
</tr>
</tbody>
</table>

Fig. 10: Comparison of model adaption methods. Image from [49].

our adaptation methods incur almost no additional computational cost and storage.

D. Rafiki Evaluation

We study the scalability of Rafiki for distributed hyper-parameter tuning by varying the number of workers (GPUs). Figure 11a compares the tuning time running over 1, 2, 4 and 8 GPUs respectively. Each point on the curves in Figure 11b represents the best validation performance among all trials that have been tested so far. The x-axis is the wall clock time. We can see that with more GPUs, the tuning becomes faster. It scales almost linearly since only hyper-parameters are transferred, which incur small communication cost. In fact, workers do not communicate with each other and worker-master communication happens once per trial.

IX. IMPLICATIONS

There is a great deal of discussion of personalized health care these days, due to the recognition that we are all different so that the exact same treatment may not be optimum for all. Almost all the work in personalized medicine today is static, usually based on the individual’s genome. While this is valuable, it is likely the case that dynamic personalization is even more important: even where two people have the same disease with the same progression, the specific time point at which particular events occur could differ greatly, and hence require very different medical interventions. Furthermore, individual environment, behaviors, and interventions can all affect disease progression, and this should be taken into account. The GEMINI system provides the promise of dynamic personalization for the first time.

The Singapore government is so impressed by the results observed in the deployment in one hospital that they are funding the deployment of this technology to an entire district, with the expectation of a 20% reduction in the prevalence of hypertension, hyperlipidemia, and diabetes within the next five years. In addition to using this technology to decide on
health care interventions, it will also be used for personalized lifestyle monitoring with a "health coach" application driven by AI. Furthermore, all this computation will occur on a small portable device, called Meddi, that can be used widely, including in homes of ordinary citizens.

GEMINI is being deployed at Singapore General Hospital, and has been used to form two startups, MediLOT Technologies in Singapore [3] and MZH Technologies in China [4]. Both have been funded by the government agency that supports entrepreneurship and translation.

As part of efforts in reducing diabetes, Apache SINGA has also been used to develop and train food recognition model as part of a FoodLG app [2] for helping pre-diabetes and diabetes patients to manage their diet. FoodLG, as illustrated in Figure 12, has been deployed at five hospitals in Singapore, National University Hospital, Singapore General Hospital, KK Hospital for Children and Women, Jurong Health (JH) Hospital and Tan Tok Seng Hospital, which are the major public hospitals in Singapore as shown in Figure 13. JHFoodLG app (a customized version of FoodLG) has gone through clinical trial and has been used for the study of behavioral change of pre-diabetes patients.

In short, GEMINI as a whole, and its SINGA kernel, have both already shown benefits in applications other than the main deployment described above. These systems, and their underlying technologies, are being used by the industry in many other application domains such as social networking, e-commerce, computer games, investment and security.

Fig. 12: Use cases in Singapore public hospitals.

Fig. 13: Singapore public hospitals.

REFERENCES


