AdaLearner: An Adaptive Distributed Mobile Learning System for Neural Networks

Jiachen Mao[†], Zhuwei Qin[‡], Zirui Xu[‡], Kent W. Nixon[†], Xiang Chen[‡], Hai Li[†], and Yiran Chen[†]
[†]Duke University, USA; [‡]George Mason University, USA
[†]{jiachen.mao, kwn8, hai.li, yiran.chen}@duke.edu; [‡]{zqin, zxu21, xchen26}@gmu.edu

***Abstract*—Neural networks hold a critical domain in machine learning algorithms because of their self-adaptiveness and state-of-the-art performance. Before the testing (inference) phases in practical use, sophisticated training (learning) phases are required, calling for efficient training methods with higher accuracy and shorter converging time. Many existing studies focus on the training optimization on high-performance servers or computing clusters, e.g. GPU clusters. However, training neural networks on resource-constrained devices, e.g. mobile platforms, is an important research topic barely touched. In this paper, we implement *AdaLearner*– an adaptive distributed mobile learning system for neural networks that trains a single network with heterogenous mobile resources under the same local network in parallel. To exploit the potential of our system, we adapt neural networks training phase to mobile device-wise resources and fiercely decrease the transmission overhead for better system scalability. On three representative neural network structures trained from two image classification datasets, *AdaLearner* boosts the training phase significantly. For example, on LeNet, 1.75-3.37× speedup is achieved when increasing the worker nodes from 2 to 8, thanks to the achieved high execution parallelism and excellent scalability.**

I. Introduction

Triggered by the unprecedented development in neural networks, a broad range of applications have emerged because of their state-of-the-art performance, e.g. object classification, language precessing, and motion estimation [1][2][3]. Due to the plethora of embedded sensors available on the platform, mobile devices represent the largest market where these applications are fed into practical use, enabling mobile navigation systems [4], speech-based human-device interaction, and augmented reality games [5]. However, a significant limitation of neural networks is the requirement for the neural network models to be trained before their practical use (testing). In comparison with the testing phase, the training phase has significantly higher computational complexity and memory consumption.

Many works have been done to minimize the time required for the training phase, ranging from algorithm-level optimization (e.g. low variance Stochastic Gradient Descent (SGD) [6]) to hardware-level acceleration (e.g. Resistive Random Access Memory (ReRAM) [7]), allowing DNN training to be applied to increasingly challenging real-world problems. Recent developments in deep learning software also shifts the focus away from frequency scaling and towards massive parallelization. In [8], Abadi *et al.* built TensorFlow, which enables distributed training by converting the neural network into an executable graph, with each device responsible for a unique subgraph. In [9], Li *et al.* proposed parameter server, featuring many architecture-level optimizations resulting in excellent execution scalability during the training phase of neural networks.

However, these previous works focus mainly on designing around powerful, dedicated server clusters, neglecting that it is mobile devices which are the primary source for the majority of data collected from the outside world. Thus, in this paper, we explore an important research topic that is barely touched: distributed learning on multiple mobile devices under wireless local area network (WLAN). By utilizing such a solution, neither private data nor trained models will be accessible by the outside world, and no extra cost is needed for establishment of cloud-based resources. Unlike previous works regarding distributed mobile computing systems for neural networks [10], we advocate a distributed mobile system that can not only test, but also train, a single neural network. Concretely, our contributions include:

- We propose *AdaLearner*–an adaptive distributed mobile system for neural networks training, consisting of two core distributed training architectures, which are discussed in Section IV;
- We design an adaptive scheduler for *AdaLearner* in Section VII, which adapts the training configuration (e.g. number of worker nodes (WNs), batch size, transmission data size) to heterogeneous mobile resources and network circumstances.
- We adopt and improve 1-bit quantization in *AdaLearner*, which can increase the system scalability by fiercely compressing the transmission data size to only 1-bit with little or no accuracy drop, which is detailed in Section VI;
- We implement and experiment *AdaLearner* on ARM-based smartphones by modifying and combining the state-of-the-art libraries for neural network training in Section VIII.

We evaluate *AdaLearner* on three neural network models which are trained from two well-known image classification datasets (MNIST, CIFAR-10). Take the experimental results of LeNet on MNIST as an example, compared with local training, *AdaLearner* accelerates the training by 1.75-3.37×, when increasing the WNs from 2 to 8.

II. Background

A. Distributed Training Methods

There are two ways to train a neural network distributedly:

Model parallelism: In model parallelism, all the WNs are provided with the same data, but with each holding only a part of the model [11]. As an example in the left part of Fig. 1, each WN processes only a single layer of the original neural network, with intermediate results from each layer being transmitted WN-to-WN as calculations are performed. Model parallelism is highly useful when training extremely large-scale models which require more memory than any single device could provide.

Data parallelism: In data parallelism, as illustrated in the right part of Fig. 1, all WNs maintain a local copy of the entire model to be trained, but with each being fed a different minibatch of the overall data [9]. After all WNs complete training with their allocated minibatches, computation results are collected by the group owner (GO) and combined into an updated model. Whether there is an independent node acting as the scheduler varies in different training scenarios [11][12]. While it incurs a larger impact on memory utilization, data parallelism allows for less inter-node communication frequency and partitioning complexity compared to model parallelism. Considering communication as a critical overhead, in *AdaLearner*, we leverage data parallelism for neural network training.

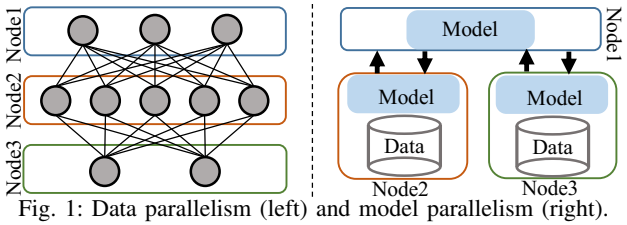


Fig. 1: Data parallelism (left) and model parallelism (right).

B. Backgroupagation with SGD

Stochastic Gradient Descent (SGD) is a popular gradient descent approach for training a wide range of models, including linear support vector machines, logistic regression and Bayesian graphical models. When training a neural network on n examples: $(x_1, y_1), ..., (x_n, y_n)$, we often target at optimizing the following problem:

$$\text{Min}\{F(w)\}, \quad F(w) = \frac{1}{n}\sum_{i=1}^{n} L(w; x_i, y_i), \tag{1}$$

where $L(\cdot)$ represents the loss function and w denotes the weights. For instance, if we use a square loss in the training phase, then the loss function will be: $L(w) = (w^\top x_i - y_i)^2$. In order to update the weights for the minimization of $F(w)$, a partial derivative with respect to each weight in w is calculated by backpropagating through all the layers in the neural network. The update procedure from iteration l to $l+1$ can be formulated as:

$$w_{l+1} = w_l - \eta \frac{\partial F(w)}{\partial w}, \tag{2}$$

where η is the learning rate. However, in each iteration of gradient descent, backpropagation is applied to all the n training examples, encountering expensive computation cost. SGD is thus introduced where each iteration only consists a mini-batch of the whole training examples. Hence, the $F(w)$ in Eq. 2 can be approximated as:

$$F^{'}(w) = \frac{1}{t}\sum_{i=1}^{t} L(w; x_i, y_i)), \tag{3}$$

where t is the minibatch size for a single update.

In this paper, we adopt traditional backpropagation with SGD and migrate this procedure to the distributed scenario.

III. System Design Motivation

In parallel training, the total execution time consumed in the training phase with N WNs can be expressed as:

$$T_{total} = \text{Max}(\frac{W_{comp[i]}}{C_{[i]}}) + \frac{W_{trans}}{B} \cdot N + \sum_{j=1}^{N} D_{[j]}, i = 1, 2...N. \tag{4}$$

Where W_{comp} and W_{trans} respectively denote the device-wise minibatch size and the transmission data size of node i in a single training iteration; $C_{[i]}$ describes the computing capability of node i, which illustrates the minibatch size that can be completed per unit of time; B is the available WiFi bandwidth; while $D_{[j]}$ stands for the wakeup time for WN j. In *AdaLearner*, we set W_{comp} and W_{trans} in Eq. 4 as our main optimization target.

Adaptive Configuration for Computation: To minimize $\text{Max}(\frac{W_{comp[i]}}{C_{[i]}})$ in Eq. 4, we need to excellently balance the computing workload within each WN in Eq. 3. From Eq. 3, we can find that there is a significant solution space w.r.t. the optimal size t=W_{comp} of the minibatch. Therefore, in *AdaLearner*, we adaptively determine the minibatch size allocated to each WN as a function of the mobile device's available computing resources for optimal parallelism.

Beyond this, it can be seen from Eq. 4 that as the number of WNs increases, the contribution of the actual training computations decreases due to their parallel nature, causing the total execution time to be dominated by the transmission overhead. In such cases, we also need to adaptively choose the optimal number of WNs.

Adaptive Configuration for Communication: Communication is always a big bottleneck for distributed training system because of the tremendous amount of data to be transmitted between WNs, which are unavoidable for each update iteration. Such data includes model gradients, parameters, and etc. Traditionally, each number in the model is a full-precision, 32 bit floating point value, which is actually unnecessarily accurate in certain situations.

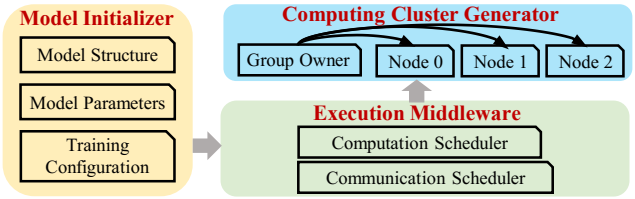


Fig. 2: System overview of *AdaLearner*.

In response to this, *AdaLearner* adopts a fierce quantization technology–1-bit quantization [12], aimed at compressing the precision of model parameters in order to significantly reduce transmission size. Besides greatly minimized transmission data size, 1-bit quantization need little extra compression overhead.

IV. AdaLearner System Framework

A. Software Components

Fig. 2 gives an overview of *AdaLearner* including 3 components:

Model Initializer: The model initializer generates the necessary files for training a model: symbol files and parameter files. Symbol files record the structure of the layers while parameter files define connection weights. Additionally, the model initializer of *AdaLearner* generates a configuration file which contains meta data such as learning rate, batch size, and etc.

Execution Middleware: Execution middleware is the fundamental component in *AdaLearner*. This execution middleware includes both the computation and communication scheduler, which is deployed to each WN and be responsible for their identical computation and communication works in the training phase.

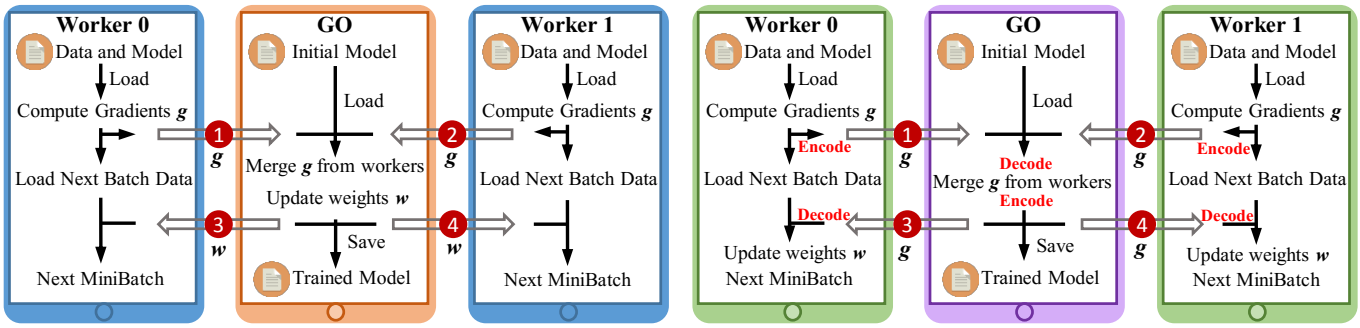
Computing Cluster Generator: When training a neural network, the mobile device held by the user is designated as the GO and all the other assisting devices are designated as WNs. The computing cluster generator runs on the GO, which first enables its Wi-Fi hotspot feature and determines the number of WNs which subsequently connect to it. The GO then collects the IP addresses of all connected WNs for future communication. Last, the GO collects meta data, which include the computing capabilities from each of the WNs.

B. System Architecture of AdaLearner

In *AdaLearner*, we provide two training architectures to deal with the tradeoff between computation cost and communication cost.

1) Architecture for Efficient Computation: Fig. 3(a) details the distributed architecture in *AdaLearner* for efficient computation, which is leveraged when the communication network is less congested, all WNs first load the training data and model into memory. Then, the model is trained on a minibatch of data using forward and backward propagation. This results in the generation of model gradients, which are then transmitted back to the GO. As data is being transmitted to the GO, each WN begins to load the training data for the next minibatch. After gathering all the gradients from the WNs, GO merges the gradients, updates its local model parameters, and transmits the updated model parameters to all the WNs. Once the WNs have received the updated model parameters, a new iteration is triggered. The total training procedure here are similar to the local training scheme with high parallelism and computation efficiency.

2) Architecture for Efficient Communication: Scalability is a critical optimization target for distributed systems, with communication costs always being the main bottleneck which restricts this. Compared with traditional data centers, communication in mobile networks suffers greater limitations due to the limited radio channels and comparatively low transmission bandwidth. Under such circumstances,



(a) Distributed architecture for efficient computation.

(b) Distributed architecture for efficient communication.

Fig. 3: Two training architectures in *AdaLearner* for computation-communication tradeoff.

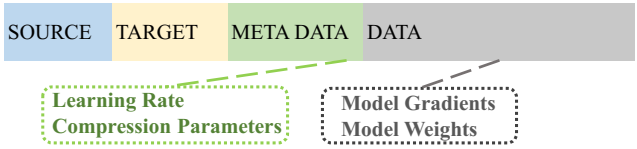


Fig. 4: Message format in *AdaLearner*.

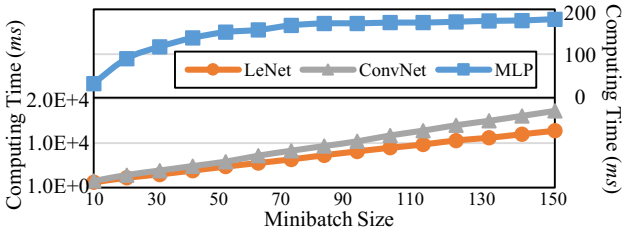


Fig. 5: Time consumption with different minibatch sizes.

Minibatch Size			
Node1	*Node2*	*Node3*	Time Threshold
20	25	30	200 *ms*
30	35	40	300 *ms*
40	55	75	400 *ms*

Fig. 6: Lookup table for minibatch size and computing time.

an alternative training architecture is proposed in Fig. 3(b), which is tailored to minimize communication overhead. Different from Fig. 3(a), there are additional steps of data encoding and a decoding performed by both the GO and WNs before and after every data transmission. The encoding process is utilized to compress the data which will be transmitted, thus reducing transmission time overhead. The detailed compression technology utilized and the corresponding tradeoff between the transmission data size and the extra cost for encoding decoding procedures will be described in Section VI.

3) Synchronization Mechanism: As a result of hardware limitations in the radio module of mobile devices, the communication between GO and WNs can only occur sequentially. Only one pair of devices can communicate at any given time. To maintain high parallelism, the GO maintains a First In First Out (FIFO) synchronization mechanism. For example, in Fig. 3(a), as WN0 is the first to transmit gradients to GO, and is followed by WN1, the GO sends back weights in the same sequence, first to WN0, then to WN1.

4) Message Format: The message format utilized in *AdaLearner* is detailed in Fig. 4. The head of the messages in *AdaLearner* is the identifier of the sender and the receiver of the message. The following number of bytes contain the meta data such as the adaptive learning rate for a minibatch. If the data is compressed, the compression parameter for decoding is also included in the meta data section. The last part is the main body of the message, containing the gradients or the weights of the neural network.

V. Computation Cost in Adaptive Scheduler

In Eq 4 and Eq 3, we assume the computing time is linearly related to minibatch size, which is consistent with our experimental results. Fig 5 shows the time consumption for the forward and backward procedures with increasing minibatch size on Nexus 5X. For LeNet and ConvNet, the time consumption is proportional to the minibatch size. An exception occurs on MLP, where the computation efficiency keeps increasing with the increase of minibatch size. It is because that MLP is a neural network with low computation cost and memory usage. When the minibatch size is small, MLP does not utilize the maximum resources in the mobile device and thus showing a lower efficiency. However, after a minibatch size of approximately 50, it, too, begins to reflect the expected linear relationship.

Without loss of generality, we utilize lookup table to record the device-wise relationship between minibatch size and their corresponding time consumption on heterogeneous mobile devices. The table in Fig. 6 presents an example of such a lookup table containing computing capability meta data from three WNs. For example, in order to train a model with a total minibatch size of 75, our adaptive scheduler searches the lookup table for WN combinations for each number of WN, favoring devices with higher computing capability. First, our adaptive scheduler will search the worker node with the best computation capability, which is WN3 in Fig. 6 and the corresponding estimation time is $400ms$. Secondly, it searches the second line of Fig. 6, where the total minibatch size of WN2 and WN3 is 75. In this scenario, the total computation time is $300ms$. Last, the first line in Fig. 6 will be searched and the corresponding result is $200ms$. In this way, *AdaLearner* can quickly estimate that $400ms$, $300ms$, $200ms$ is needed for one, two and three WNs, respectively.

VI. Communication Cost in Adaptive Scheduler

Transmission efficiency is critical for mobile networks scalability. Among many data compression technologies targeting at efficient neural network training [12][13]. We adopt the concept of 1-bit quantization in our communication efficient architecture in Fig. 3(b). The choice is based on two reasons: (1) The compression rate is the highest among all the existing technologies. Each number to be transmitted is quantized to 1 bit, the compression ratio of which is $1/32$ because the original data are represented as 32 bits floating point numbers; (2) 1-bit quantization enables less extra computation cost because of its simple quantization function.

A. 1-bit Quantization with Error Feedback

In 1-bit quantization, we set 0 as the quantization threshold. All the number larger than or equals 0 is set as 1 while all the negative numbers are set as 0. In order to make up of the loss introduced by 1-bit quantization, the quantization error of last minibatch will be

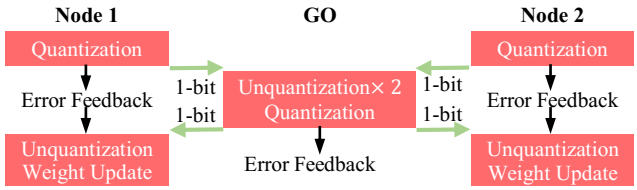


Fig. 7: Tradeoff between computation and communication costs.

added to the current one [12]. Assume the gradients calculated by k-th minibatch is $G_{[k]}$, the quantization can be expressed as:

$$G_{quantized[k]} = \mathscr{Q}(G_{[k]} + \Delta_{[k-1]}), \tag{5}$$

where $\mathscr{Q}(\cdot)$ stands for the quantization function. $\Delta_{[k-1]}$ represents the quantization error of (k-1) minibatch and $G_{quantized[k]}$ is the quantized 1-bit results to be transmitted. After the quantization procedure in Eq 5, a new quantization error feedback is calculated for next minibatch:

$$\Delta_{[k]} = G_{[k]} - \mathscr{Q}^{-1}(G_{quantized[k]}). \tag{6}$$

With the effect of error feedback, the quantization errors are accumulated and expressed in the subsequent iterations of minibatch. Therefore, we can losslessly compress the gradients between nodes to 1 bit with little or no accuracy drop and the experimental results will be presented in experiment section.

B. Gradients Aggregating Scheme

Conventionally, 1-bit SGD is realized in GPUs cluster without central scheduler and each GPU in N GPUs is responsible for aggregating a $1/N$ subset of the model parameters [12]. Such training topology is not suitable in our distributed mobile system. Different from conventional realization, *AdaLearner* incorporates 1-bit SGD to the training architecture in Fig. 3(b). After deriving the gradients on each WN, the gradients experiences the quantization procedure in Eq 5 and then transmit them to GO. GO gathers all the gradients in 1-bit format and sums them up in unquantized form. After post-processing the gradients in GO, a second quantization is applied before transmitted back to each WN. Each WN will then unquantize the received 1-bit gradients and updates the model parameters locally. The quantization and unquantization modules are embedded in the encoder and decoder in Fig. 3(b). In order to conduct unquantization procedure in GO, two numbers are required to be transmitted along with the 1-bit gradients for each column of the original weight matrix, which represents the multiplication scalar for positive and negative gradients. They are calculated by averaging all the original positive and negative gradients. Mind that original 1-bit quantization is utilized in Recurrent Neural Networks (RNNs), which only contains fully-connected layers. However, *AdaLearner* targets at all the neural network structures, including convolutional layers, for which we innovatively calculate the positive and negative scalar for each filter.

C. Computation / Communication Tradeoff

Such gradients aggregation scheme introduces the tradeoff between computation cost and communication cost. The procedures with red square background in Fig. 7 detail the extra computation cost yielded by reduced communication cost utilizing 1-bit quantization. Before and after each communication, quantization and unquantization procedures are necessary. In order to reach high parallelism, the procedure for calculating the new error feedback is executed in parallel without influencing the total execution time.

If we transmit the 32-bit floating point numbers between N nodes, the communication overhead can be formulated as:

$$T_{comm} = N \cdot \frac{W_{trans}}{B} + D, \tag{7}$$

where W_{trans} means the original transmission data size under the network of bandwith B. Here D denotes the total wakeup time during the communication, which is assumed to be a constant in our formula. Alternatively, by applying 1-bit quantization technology, the communication can be expressed as:

$$T^{1bit}_{comm} = 2 \cdot \frac{W_{trans}}{S_q} + (N+1)\frac{W_{trans}}{S_{uq}} + \frac{T_{comm}}{Z} + D, \tag{8}$$

where S_q and S_{uq} respectively denote the quantization and unquantization speed (bits per unit time), which are also pre-tested on each mobile device. In Eq 8, Z equals 32 here because the transmission data size is $1/32$ of the original transmission data size. Because the unquantization procedure of GO in Fig. 7 is done sequentially, the time consumption on unquantization is linearly related to the number of WNs. Thus, the decision function of whether to leverage communication efficient scheme with N WNs is given by:

$$\text{Decision} = T^{1bit}_{comm} < T_{comm} \ ? \ True : False, \tag{9}$$

which simply compares between the communication costs for these two training schemes and chooses the one with less communication time. If Eq 9 holds, the scheduler will use communication efficient training because the extra computation overhead is smaller than saved communication overhead and vice versa.

VII. Incorporate Adaptive Scheduler to AdaLearner

In this sections, we give the comprehensive scheme, which adapts the optimal configuration (number of WNs, minibatch size, communication scheme) to the current mobile resources. As described in Algorithm 1, the outside loop iteratively estimate the execution time with different WNs. For each iteration, our scheme estimates the time consumption with and without data compression and pick the one with less time estimation. The scheme keeps running until it reaches the max WNs number or the minimal total estimated time. In *AdaLearner*, we only focus on the partition scheme with maximal parallelism of total minibatch size (W_{comp}), where $W_{comp} = \sum_{i=1}^{N} W_{comp[i]}$. The total minibatch size in *AdaLearner* is predefined by the users and it is worth to mention that we test our system with the most commonly used configuration where $W_{comp} = 128$.

Algorithm 1 Mobile resource-based adaptive scheduling scheme.

Input: Number of available WNs: N, Lookup table: T_{lookup}, which contains minibatch size with respect to the time consumption of each node $i \in [1, N]$, Transmission data size: W_{trans}, Quantization and Unquantization speed: S_q and S_{uq}, Network bandwidth: B.
Initialization: Estimation time: T_{est}=∞, Total minibatch size: W_{comp}=$C_{constant}$ ($C_{constant} = 128$ in our experiments),

1: **for** each int $n \in [1, N]$ **do**
2: Partition W_{comp} into n WNs based on the lookup table T_{lookup} and get the estimated computing time: T_{comp}.
3: **if** Eq 9 is $true$ **then**:
4: Estimate communication overhead using Eq 7.
5: Estimated training time: T_{temp} = T_{comp}+T^{1bit}_{comm}.
6: **else**:
7: Estimate communication overhead using Eq 8.
8: Estimated training time: T_{temp} = T_{comp}+T_{comm}.
9: **if** $T_{temp} > T_{est}$ **then** : break the loop.
10: **else**: T_{est} = T_{temp}.

Output: The optimal number of WNs: n, the minibatch size for each WN, and the communication scheme to use.

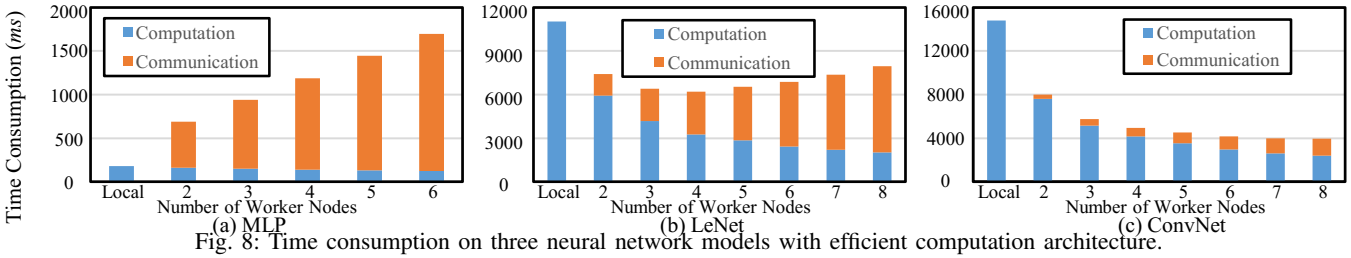


Fig. 8: Time consumption on three neural network models with efficient computation architecture.

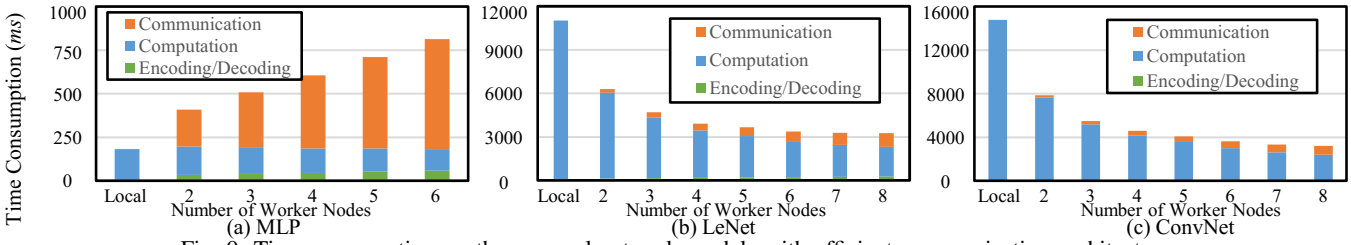


Fig. 9: Time consumption on three neural network models with efficient communication architecture.

VIII. Experiments

A. System Implementation and Experiments Setup

In *AdaLearner*, we adopt and modify three cutting-edge software libraries for the computation of neural network training and data communication. Targeting at efficient computation, the local training procedure is implemented based on MXNet [14], which is built under the concept of [9]. MXNet is a high performance framework for neural network training for desktops and servers realized in C++. However, MXNet only provides the mobile platforms with testing feature. In this work, we modify the neural network training code in MXNet and recompile them to ARM-based Android devices. For efficient communication, Message Passing Interface (MPI) is adopted in *AdaLearner*. However, none currently existing MPI implementations explicitly offer support for Android devices. As such, we modify the source code and build process for an existing MPI implementation called Open MPI [15] such that it compiled and ran on Android devices. Open MPI relies on an SSH server and client implementation to serve as its underlying communication channel between nodes. Again, while multiple SSH implementations exist for standard Linux distributions, none of them offer direct support for Android platforms due to the difference in security and account implementations. To get around this, we modify the source code of Dropbear [16], an open source SSH implementation that specifically targets embedded device with low storage, computing, and memory resources.

By merging the aforementioned libraries with the implementation logic in *AdaLearner*, we realized a local distributed mobile system for neural network training on mobile devices. Our experiments are conducted on LG Nexus 5X running Android 6.0.1 with a 1.8 GHz processor and 2GB RAM. The experimental setup is depicted in Fig. 10. We conduct the experiments on two image classification datasets: MNIST [17], CIFAR-10 [18]. In order to show the robustness of *AdaLearner* on different model complexities, three neural network model with increasing scales are tested based on these two datasets, which are called Multi-layer Perceptron (MLP), LeNet [19], and ConvNet [20], respectively. MLP only contains fully-connected layers while LeNet and ConvNet are two representative Convolutional Neural Networks (CNNs) on MNIST and CIFAR-10.

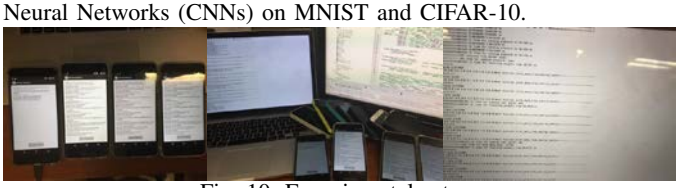


Fig. 10: Experimental setup.

B. Evaluation of Efficient Computation Architecture

The bars in Fig. 8 show the relationship between the number of WNs and their corresponding execution time utilizing efficient computation training architecture. For efficient computation architecture, the total execution time for training is made up of the computation time denoted by the blue bars and the communication time which is presented as the orange part of the stacked column in Fig. 8. Here we define the baseline as the training time on local device. Additionally, the total training time in the experimental results indicate the time consumption as the time spent in a whole distributed training procedure of a single minibatch iteration. The total number of iterations is defined by the users. We can find that these three neural network models shows different results compared with the baseline:

Fig. 8(a) shows the distributed training time on MLP, which keeps increasing with the increase of number of WNs. On the one hand, MLP is a model with only fully-connected layers and is of extreme small scale and thus requiring less computation cost. On the other hand, the model size is comparatively big ($0.44MB$), increasing the communication overhead. Therefore, compared with the local training time ($183ms$), the communication overhead accounts for the main part in the total training time, which occupies from 76% to 93% of the total training time for 2 to 6 WNs. In this scenario, our scheduler assigns all the work to local device for the shortest training time.

The experimental results on LeNet in Fig. 8(b) show an obvious reduction in total training time when the WNs increases from 2 to 4. The optimal setting of 4 WNs costs $6208ms$, reaching $1.78\times$ speedup compared with the local training time of $11031ms$. However, if we further increase the number of WNs, the total training time is immersed by the communication cost, leading to a worse performance. Hence, 4 WNs is optimal for LeNet in our efficient computation architecture of *AdaLearner*.

Because of the intensive computation cost of ConvNet on Cifar-10 and small parameter size, the total training time keeps decreasing from 2 to 8 WNs in Fig. 8(c). For example, the total training time of ConvNet is $3976ms$ while it costs $14780ms$ for local execution, demonstrating $3.72\times$ acceleration.

C. Evaluation of Efficient Communication Architecture

The bars in Fig. 9 demonstrate the total training time using efficient communication architecture with 1-bit quantization technology. Besides computation cost and communication cost, Fig. 9 also includes the time of the encoding and decoding procedures. It can be clearly viewed that the extra overhead for encoding and decoding procedures

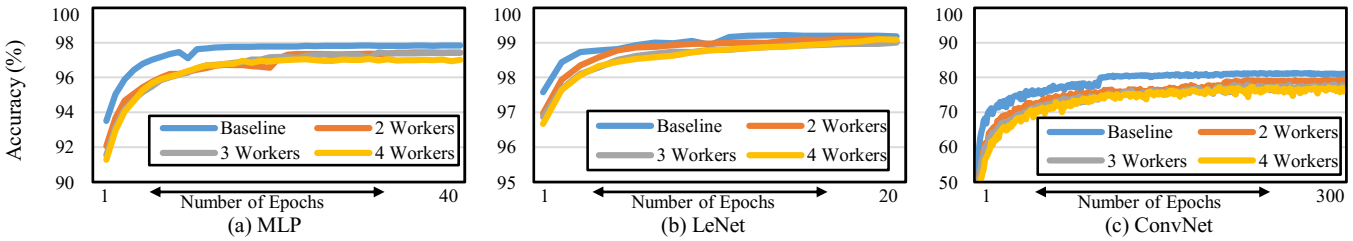


Fig. 11: Accuracy results using 1-bit quantization under different number of WNs.

is low when compared with computation cost and communication cost, which is at most 10%, 8.5%, and 1.3% of the total training time for MLP, LeNet, and ConvNet.

Mind that the communication time is the summation of transmission time and wakeup time. The wakeup time in our experiments is tested to be about $100ms$ in average, which affects the parallelism of small-scale neural network model like MLP: In Fig. 9(a), although the transmission time decreases from $1572ms$ to $630ms$, the communication overhead is still the bottleneck with 1-bit quantization.

Thanks to the effectiveness of 1-bit quantization, when performing LeNet in *AdaLearner* with efficient communication architecture, the scalability is better than that of efficient computation architecture. As illustrated in Fig. 9(b), the total training time keeps decreasing from 2 to 8 WNs. In 8 WNs scenario, the total training time comes to $3276ms$, boasting the execution by $3.37\times$, which is better than the optimal scenario in efficient computation architecture ($1.78\times$). The same result occurs for ConvNet in Fig. 9(c), where the total training time is decreased to $3262ms$, which is lower than 3976 ms in efficient computation architecture, achieving $3.53\times$ speedup.

D. Performance / Accuracy Tradeoff

The use of 1-bit quantization will may cause the tradeoff between training performance and potential accuracy drop. Fig. 11(a), Fig. 11(b), and Fig. 11(c) presents the accuracy results with the increase of training epoch with 2, 3, and 4 WNs on MLP, LeNet, and ConvNet. respectively.

Fig. 11(a) depicts the accuracy on MLP. Compared with the baseline accuracy without 1-bit quantization (97.8%), 1-bit quantization results in the accuracy of 97.5%, 97.5%, and 97% respectively for 2, 3, and 4 WNs. The accuracy drop is limited in less than 0.8%. For LeNet on MNIST, the corresponding accuracies of 2, 3, and 4 WNs in the $20th$ epochs are 99.12%, 99.00%, and 99.01%. The accuracy drop is constrained within 0.2% in comparison to the original baseline of 99.2%. However, the accuracy drops higher in ConvNet in Fig. 11(c). Compare with the baseline accuracy of 81.2% without using 1-bit quantization, our communication efficient training with 1-bit quantization on 2, 3, and 4 WNs can only reaches the accuracy 79.4%, 78.1%, and 77.3%, incurring 2.9% accuracy loss in average. The high accuracy loss is due to the complexity of the dataset and the comparatively simple model structure adopt in our experiment. Hence, whether to utilize 1-bit quantization is according to the user real demand of performance-accuracy tradeoff.

IX. Conclusion

In this work, we propose *AdaLearner* - an adaptive distributed mobile learning system for neural networks to enable parallel training of neural networks on mobile platforms. In *AdaLearner*, we first design the architectures tailored for distributed mobile devices, namely, efficient computation architecture and efficient communication architecture. Then, 1-bit quantization technology is adopted in efficient communication architecture to largely compress the transmission data size for better system scalability. Additionally, an adaptive scheduler is designed to adapt the training configuration to the mobile resources so as to realize high execution parallelism. Finally, we realize all the functionality of *AdaLearner* on distributed Android mobile devices and provide the total execution speedup with respect to different number of worker nodes.

X. Acknowledgment

This work was supported in part by NSF grants CNS-1717657 and SPX-1725456.

References

[1] J. Qiu *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 26–35.

[2] S. Han *et al.*, "Ese: Efficient speech recognition engine with compressed lstm on fpga," *arXiv preprint arXiv:1612.00694*, 2016.

[3] A. Jain and *et al*, "Modeep: a deep learning framework using motion features for human pose estimation," in *Proceedings of Asian Conference on Computer Vision (ACCV)*, 2014, pp. 302–315.

[4] M. Limmer and *et al*, "Robust deep-learning-based road-prediction for augmented reality navigation systems at night," in *Proc. of Intelligent Transportation Systems (ITSC)*. IEEE, 2016, pp. 1888–1895.

[5] V. Mnih and *et al*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, 2015.

[6] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Proc. of Advances in neural information processing systems (NIPS)*, 2013, pp. 315–323.

[7] L. Song and *et al*, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *High Performance Computer Architecture*, 2017.

[8] M. Abadi and *et al*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv:1603.04467*, 2016.

[9] M. Li and *et al*, "Scaling distributed machine learning with the parameter server," in *Operation System Design and Implementation*, vol. 14, 2014.

[10] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "Modnn: Local distributed mobile computing system for deep neural network," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 1396–1401.

[11] J. Dean and *et al*, "Large scale distributed deep networks," in *Proc. of Advances in neural information processing systems (NIPS)*, 2012, pp. 1223–1231.

[12] F. Seide and *et al*, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns," in *Interspeech*, 2014.

[13] T. Dettmers, "8-bit approximations for parallelism in deep learning," *CoRR*, vol. abs/1511.04561, 2015. [Online]. Available: http://arxiv.org/abs/1511.04561

[14] T. Chen and *et al*, "Mxnet: a flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[15] E. Gabriel and *et al*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *Proc. of European Parallel Virtual Machine/Message Passing Interface Users'Group Meeting*. Springer, 2004, pp. 97–104.

[16] M. Johnston, "Dropbear ssh server and client," 2005.

[17] Y. LeCun and *et al*, "The mnist database of handwritten digits," 1998.

[18] A. Krizhevsky and *et al*, "Learning multiple layers of features from tiny images," 2009.

[19] Y. LeCun and *et al*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[20] A. Krizhevsky and *et al*, "Imagenet classification with deep convolutional neural networks," in *Proc. of Advances in neural information processing systems (NIPS)*, 2012, pp. 1097–1105.