MQGrad: Reinforcement Learning of Gradient Quantization in Parameter Server

Guoxin Cui, Jun Xu*,Wei Zeng, Yanyan Lan, Jiafeng Guo, Xueqi Cheng
[1]University of Chinese Academy of Sciences, Beijing, China
[2]CAS Key Lab of Network Data Science & Technology, Institute of Computing Technology, Chinese Academy of Sciences
{cuiguoxin,zengwei}@software.ict.ac.cn,{junxu,lanyanyan,guojiafeng,cxq}@ict.ac.cn

ABSTRACT

One of the most significant bottleneck in training large scale machine learning models on parameter server (PS) is the communication overhead, because it needs to frequently exchange the model gradients between the workers and servers during the training iterations. Gradient quantization has been proposed as an effective approach to reducing the communication volume. One key issue in gradient quantization is setting the number of bits for quantizing the gradients. Small number of bits can significantly reduce the communication overhead while hurts the gradient accuracies, and vise versa. An ideal quantization method would dynamically balance the communication overhead and model accuracy, through adjusting the number bits according to the knowledge learned from the immediate past training iterations. Existing methods, however, quantize the gradients either with fixed number of bits, or with predefined heuristic rules. In this paper we propose a novel adaptive quantization method within the framework of reinforcement learning. The method, referred to as MQGrad, formalizes the selection of quantization bits as actions in a Markov decision process (MDP) where the MDP states records the information collected from the past optimization iterations (e.g., the sequence of the loss function values). During the training iterations of a machine learning algorithm, MQGrad continuously updates the MDP state according to the changes of the loss function. Based on the information, MDP learns to select the optimal actions (number of bits) to quantize the gradients. Experimental results based on a benchmark dataset showed that MQGrad can accelerate the learning of a large scale deep neural network while keeping its prediction accuracies.

CCS CONCEPTS

• **Theory of computation → Reinforcement learning**; • **Software and its engineering → Client-server architectures**;

KEYWORDS

Gradient Quantization; Reinforcement Learning; Adaptive System

* Corresponding author: Jun Xu

ICTIR '18, September 14–17, 2018, Tianjin, China

ACM ISBN 978-1-4503-5656-5/18/09…$15.00
https://doi.org/10.1145/3234944.3234978

ACM Reference Format:
Guoxin Cui, Jun Xu*,Wei Zeng, Yanyan Lan, Jiafeng Guo, Xueqi Cheng. 2018. MQGrad: Reinforcement Learning of Gradient Quantization in Parameter Server. In *2018 ACM SIGIR International Conference on the Theory of Information Retrieval (ICTIR '18), September 14–17, 2018, Tianjin, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3234944.3234978

1 INTRODUCTION

With the rapid growth of the training data and the resulting machine learning model complexity, distributed optimization has become a popular solution for scaling up the machine learning problems. Parameters sever (PS) [14] is one of the most popularly adopted distributed computing framework tailored for large scale machine learning. PS splits the computers in a cluster into worker nodes and server nodes. The data and workload are distributed to worker nodes and the globally shared model parameters are maintained by the server nodes. During the training of machine learning models, the worker nodes process data and calculate the gradients while server nodes synchronize parameters and perform global updates. PS can scale up a number of machine learning algorithms such as LDA and logistic regression.

It has been observed that the communication overhead is one of the major bottleneck in PS[4, 10]. At each of the optimization iteration, after finishing the local computations, multiple worker nodes need to push the resulting gradients of the parameters to the corresponding server nodes for parameter updating, and then pull the updated parameters to local for next iteration computations. Since the optimization procedure needs to execute a large number of iterations, the communication volume between the worker nodes and server nodes is huge and time consuming. Thus, how to reduce the communication volume becomes a key problem for accelerating the training of machine learning algorithms on PS.

Gradient quantization has been proposed as one of the most effective approach to reduce the communication overhead in distributed systems [1, 17, 18]. It reduces the number of bits used to transmit each parameter through quantizing (compressing) the transmitted values. When applying gradient quantization in PS, how to determine the number of bits used to transit each parameter, aka the quantization bits, is a critical issue. On one hand, one may want to set a small quantization bits for significantly reducing the communication overhead. On the other hand, the quantization bits cannot be too small because heavily compressing the gradients inevitably makes the model inaccurate, which may slow down the decreasing of the loss or even make the optimization not converge. How to balance between the communication overhead and gradient accuracy is one of the key issues in gradient quantization.

Ideally, for choosing optimal quantization bits at each iteration, PS system would dynamically adjust the bits with the knowledge

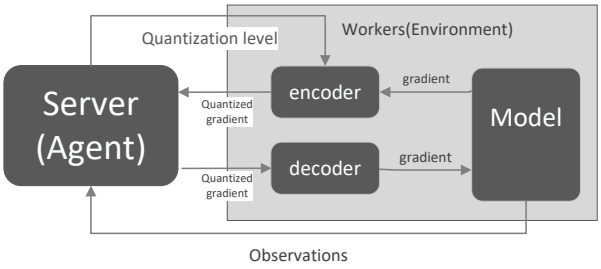

Figure 1: Agent-environment interaction in MQGrad.

learned from the past optimization iterations (e.g., the decreasing rates of the loss function at the past a few iterations). Existing approaches, however, usually choose a fixed quantization bits before running the machine learning algorithm [1, 18]. In recent years methods are proposed in which the quantization bits can be dynamically adjusted with predefined heuristic rules. For example, Øland and Raj proposes to adjust the quantization bits according to the norm of gradient vector[17]. The experience from the past optimization iterations is not fully utilized.

In this paper, we aim at developing a new method that can learn to adjust the quantization bits, on the basis of the information collected from the past optimization iterations. Inspired by the success of learning to learn [2] and reinforcement learning, we propose to use the Markov decision process (MDP) to learn the quantization bits in PS, referred to as MQGrad. The agent-environment interaction framework of MQGrad is shown in Figure 1. MQGrad formalizes the adjustment of the quantization bits as actions in an MDP. At each iteration of training the machine learning algorithm, the MQGrad agent repeatedly monitors the values of the machine learning loss function for updating its state and calculating the rewards. Then, it chooses the best action (quantization bits) and sends it to the workers for quantizing the gradients at the current iteration. The reinforcement algorithm of SARSA [20] is utilized here for determining the quantization bits and updating the parameters of the MDP model.

MQGrad offers several advantages: ease in implementation, ability of balancing the communication overhead and model accuracy automatically, and effectively accelerating the large scale machine learning algorithms.

Experimental results indicate that MQGrad can outperform the baseline quantization methods including the fixed quantization methods and the adaptive quantization, in terms of accelerating a deep learning algorithm trained on CIFAR-10 dataset.

2 RELATED WORK

In this section, we introduce the backgrounds of the paper, including the Markov decision process in reinforcement learning, the parameter server system for distributed machine learning, the current way of reducing communication overhead in distributed computing system and the learning to learn method.

2.1 Markov decision process

In the paper, we employ MDP[20], a widely used sequential decision making model, for choosing the quantization bits in the machine learning optimization. An MDP consists of several key components:

States S is a set of states. For instance, in this paper we define the state to track the status of the optimization, including the values of the loss function in the past iteration, the best number of bits for current optimization etc.

Actions A is a discrete set of actions that an agent can take. The actions available may depend on the state s, denoted as $A(s)$.

Transition T is the state transition function $s_{t+1} = T(s_t, a_t)$ which specifies a function which maps a state s_t into a new state s_{t+1} in response to the action selected a_t.

Policy π is a mapping from each state, $s \in S$, and action, $a \in \mathcal{A}$, to the probability $\pi(a|s)$ of taking action a when in state s.

Reward $r = R(s, a)$ is the immediate reward, also known as reinforcement. It gives the immediate reward of taking action a at state s.

Value function $v = V^{\pi}(s)$ is a function of states that estimates how good it is for the agent to be in a given state s, under the policy π (state-value function for policy π). The goodness is measured in terms of future rewards that can be expected (expected return).

$$V^{\pi}(s) \doteq \mathbf{E}_{\pi}\left[\sum_{k=1}^{\infty} \gamma^k r_{t+k+1} | S_t = s\right],$$

where $\gamma \in [0, 1]$ is the discount factor.

Q **function** $Q^{\pi}(s, a)$ is a function of state-action pairs that estimates the value of taking action a in state s under a policy π (action-value function for policy π). It is defined as the expected return starting from s, taking the action a, and thereafter following policy π:

$$Q^{\pi}(s, a) \doteq \mathbf{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | S_t = s, A_t = a\right].$$

$Q^{\pi} : S \times A \rightarrow \mathbf{R}$ is the value of taking action a in state s under a policy π. It equals to the expected return when starting from s taking the action a and thereafter following policy π. It easy to show that $Q(s, a) = r(s, a) + \gamma V * (T(s, a))$, where $\gamma \in [0, 1]$ is the discount factor.

The agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, \cdots$. At each time step t the agent receives some representation of the environment's state, $s_t \in S$, and on that basis selects an action $a_t \in A(s_t)$, where $A(s_t)$ is the set of actions available in state s_t. One time step later, in part as a consequence of its action, the agent receives a numerical reward, $r_{t+1} \in \mathbb{R}$ and finds itself in a new state $s_{t+1} = T(s_t, a_t)$.

The value function V^{π} and Q function Q^{π} can be estimated from experience. A number of reinforcement learning algorithms have been proposed. Among these algorithms, the on-policy TD control algorithm SARSA [20] is one of the most widely method.

2.2 Distributed learning with parameter server

In distributed machine learning, the training data may be extremely large. For instance, an Internet company may use one year of an ad impression log to train an ad click predictor, which would

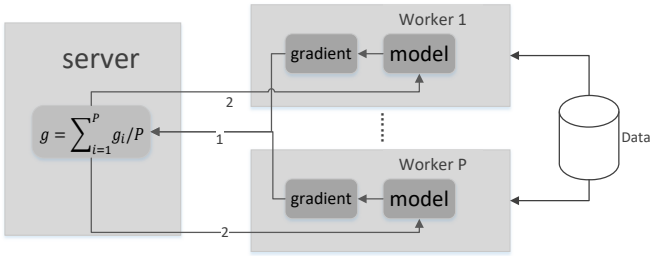

Figure 2: Distributed training of machine learning algorithms in parameter server.

involve trillions of training examples [14]. Also, in machine learning there is an important relationship between the amount of training data and the model size. If there is enough training data to avoid the overfitting problem, a more detailed model typically improves accuracy. Thus, a distributed machine learning would contain huge number of parameters.

The goal of many machine learning (including the deep learning) algorithms can be expressed via an "loss function", which captures the properties of the learned model, such as the error in terms of the training data and the complexity of the learned model. Given a set of training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$, where $\mathbf{x}_i$ and y_i are the feature vector and the label for the i-th instance, respectively, and N is the number of training instances, the machine learning algorithm typically minimizes the loss function to obtain the model:

$$L(\mathbf{w}) = \sum_{i=1}^{N} \ell(\mathbf{x}_i, y_i; \mathbf{w}) + \Omega(\mathbf{w}),$$

where ℓ represents the prediction error on the training data and regularizer Ω penalize the model complexity. In general, there is no closed-form solution; instead, learning starts from an initial model. It iteratively refines the model by processing the training data and stops when a (near) optimal solution is found or the model is considered to be converged.

Iteratively processing large scale data and updating huge number of parameters require enormous memory, computing, and bandwidth resources. Parameter server are proposed to solve the problem. As shown in Figure 2, the nodes in a cluster is splited into servers nodes and worker nodes. the training data is partitioned among all of the workers, which jointly learn the parameter vector $\mathbf{w}$. The parameter vector $\mathbf{w}$ is maintained on sever nodes.

The training optimization algorithm operates iteratively. In each iteration, every worker independently uses its own training data to determine what changes should be made to $\mathbf{w}$ in order to get closer to an optimal value. Because each worker's updates reflect only its own training data, the system expresses the updates as a subgradient, a direction in which the parameter vector $\mathbf{w}$ should be shifted, and aggregates all subgradients before applying them to $\mathbf{w}$. Figure 2 shows the steps required in performing distributed machine learning in parameter server.

One of the bottlenecks in PS is that all workers have to communicate with the server. The huge communication would significantly slow down the training process as the server needs to wait all of the gradient vectors for aggregation and updating parameters. Even using the fastest interconnects, such as InfiniBand (IB) or PCI Express 3.0 (PCIe), this communication overhead can still become a serious bottleneck and greatly slow down the optimization.

To alleviate the commnication overhead's impact on slowing down the system, reseachers proposed Asynchronous Parallel (ASP) [4] and Staleness Synchronous Parallel (SSP) [10]. Although the total information needed to trasmit is not reduced, the workers don't need to wait for each other and communication and computing can be overlapped to accelerating the training process.

2.3 Reducing the communication overhead in distributed learning

One direct approach to minimize communication overhead is just to reduce the number of parameters need to be exchanged, e.g. by having fewer parameters in the first place by sparsifying them. In the early works[8, 9, 13, 19], network pruning has been proved as a valid way to reduce the complexity of the network. Recently, Han et al. pruned state-of-the-art CNN models with no loss of accuracy[6] and Han et al. prunes the network's connections by removing all connections with weights below a threshold to reduce the parameters[7].

Another way to reduce the communication overhead is using less bits to represent the parameters or gradients, called parameter or gradient quantization. For example, Seide et al. uses 1 bit to quantize the gradients which greatly reduce the communication overhead while it needs the quantization error to be carried forward across mini-batches[18]. Alistarh et al. proposed quantized SGD(QSGD) which is a family of compression schemes with convergence guarantees and good practical performance[1]. Wen et al. used similar stochastic quantization like QSGD but focused on using three possible values to represent each value of gradient[21]. Øland and Raj reduced the communication overhead by nearly an order of magnitude through adaptively choosing the bits to quantize the weights according to gradient's mean square error[17]. The method is based on the simple hypothesis: when the gradient's norm is large, more bits are needed to represent the gradient because relatively small perturbations can result in relatively large changes in the error.

2.4 Learning to learn

All existing parameter quantization methods cannot utilize the knowledge from the optimization history. In this paper, we propose to learn to set the quantization bits, on the basis of the data collected from the past training iterations. The idea is similar to that of "learning to learn" [2] which automatically learns the updating rule of optimization in machine learning. A number of learning to learn algorithms has been proposed in the past a few years and reinforcement learning is also used for the task. For example, reinforcement learning algorithms are used for tuning the learning rate [3, 5, 22], for optimizing device placement for Tensorflow computational graphs[15], and for generating network architecture to maximize the expected accuracy of the validate set [23]. In this paper, we make use of the reinforcement learning model of MDP to learn the quantization bits for compressing the gradients.

3 OUR APPROACH: MQGRAD

In this section, we describe the proposed MQGrad model for reducing the communication overhead in PS.

3.1 MQGrad system architecture

We extend the PS architecture shown in Figure 2 with an MDP module on the sever side and gradient quantize/de-quantize modules on all of the nodes, achieving the MQGrad system shown in Figure 3 and the functions executed on the scheduler, workers, and servers are shown in Algorithm 1.

Suppose that a large scale machine learning model is being trained on the PS. After distributing the training data and the model parameters (necessary working set) to each worker node, the training algorithm executes an iterative optimization of its loss function. At each iteration m, given the current model parameters, the training algorithm calculates the local gradients at each worker node. At the same time, each worker also calculates the local value of the loss function based on the local data (step 1 in Fig. 3, line 28 in Alg. 1). The local values at all of the workers are collected by the sever MDP module (step 2 in Fig. 3, line 38 of Alg. 1). After that, the MDP module at server restores the overall global loss, updates its state, calculates the reward, determines the action (the quantization bits), and finally broadcasts the number to all worker nodes (step 3 in Fig. 3, line 39-47 in Alg. 1). Given the quantization bits, the worker nodes quantize[1] their local gradients (step 4 in Fig. 3, line 30 in Alg. 1) and send the quantized local gradients to the parameter server (step 5 in Fig. 3, line 31 in Alg. 1). The server nodes de-quantize and summarize all of the received local gradients to a global gradient for updating the model parameters (step 6 and 7 in Fig. 3, line 51-52 in Alg. 1). Then the server broadcasts the quantized global gradient (step 8 in Fig. 3, line 53 in Alg. 1) and the workers receive it, de-quantize the gradient, and update the local model parameters (step 9 in Fig. 3, line 32-33 in Alg. 1).

Receiving the signal that the model parameters have been updated, the machine learning training algorithm moves to iteration $m+1$ and re-estimates the local gradients and local losses. The process repeats until converge or the number of iterations reaches a predefined maximum number.

3.2 Learning for gradient quantization with MDP

The key component in MQGrad is the MDP module which determines the quantization bits. The configuration of the MDP is as follows:

Time step t: $t \in Z^+ \cup \{0\}$ is the discrete time step of the MDP. To avoid adjusting the quantization bits too frequently, which may result in an unstable training process, the MDP model in MQGrad is configured to update the quantization bits every T training iterations. That is, the server will broadcast the identical quantization bits used in the last iteration to worker nodes (step 3 in Figure 3) if $m\%T \neq 0$, where m is the iteration number of the machine learning training algorithm. During these iterations, the MDP module only collects the losses for constructing its state. The MDP module will be activated to update the quantization bits when $m\%T = 0$. Thus

[1]MQGrad uses the Quantize (Encode) and De-quantize (Decode) functions in https://www.tensorflow.org/performance/quantization.

Algorithm 1 MQGrad functions

1: **Task scheduler:**
2: issue LoadData( ) to all workers
3: init $\mathbf{w}$ and issue it go all workers
4: **for** iteration $m = 0$ to M **do**
5: Issue WorkerIterate(p, m), where p is worker ID
6: **end for**
7:
8: **Worker** $p = 1, \cdots, P$**:**
9: **function** LoadData( )
10: load a batch of training data $\{\mathbf{x}_{ip}, y_{ip}\}_{i=1}^{N_p}$
11: **end function**
12:
13: **function** SendLossThenReceivebits($L_{p,m}$)
14: send $L_{p,m}$ to server
15: remote call server function ReceiveLossThenSendbits($L_{p,m}$)
16: receive quantize bits K from server
17: **return** K
18: **end function**
19:
20: **function** SendQGThenReceiveQG($\bar{\mathbf{g}}_{p,m}$)
21: send $\bar{\mathbf{g}}_{p,m}$ to server
22: remote call server function ReceiveQGThenSendQG($\bar{\mathbf{g}}_{p,m}$)
23: receive updated gradient $\bar{\mathbf{g}}^m$ from server
24: **return** $\bar{\mathbf{g}}^m$
25: **end function**
26:
27: **function** WorkerIterate(p, m)
28: $(\mathbf{g}_{p,m}, L_{p,m}) \leftarrow$ Gradient/loss w.r.t. a batch of data
29: $K \leftarrow$ SendLossThenReceivebits($L_{p,m}$)
30: $\bar{\mathbf{g}}_{p,m} \leftarrow$ Quantize($\mathbf{g}_{p,m}, K$)
31: $\bar{\mathbf{g}}^m \leftarrow$ SendQGThenReceiveQG($\bar{\mathbf{g}}_{p,m}$)
32: $\mathbf{g}^m \leftarrow$ De-quantize($\bar{\mathbf{g}}^m$)
33: $\mathbf{w} \leftarrow \mathbf{w} - \eta \mathbf{g}^m$
34: **end function**
35:
36: **Servers:**
37: **function** ReceiveLossThenSendbits($L_{p,m}$)
38: receive loss $L_{p,m}$'s from workers
39: **if** all $L_{p,m}, p = 1, \cdots, P$ are received **then**
40: $L^m \leftarrow \frac{\sum_{p=1}^{P} L_{p,m}}{P}$
41: **if** $m\%T = 0$ **then**
42: $t \leftarrow m/T$
43: $K \leftarrow$ MQGrad-SARSA(t)
44: send K to all workers
45: **end if**
46: **end if**
47: send the last iteration K to workers
48: **end function**
49:
50: **function** ReceiveQGThenSendQG($\bar{\mathbf{g}}_{p,m}$)
51: $\mathbf{g}_{p,m} \leftarrow$ De-quantize($\bar{\mathbf{g}}_{p,m}$)
52: **if** all $\mathbf{g}_{p,m}, p = 1, \cdots, P$ are received **then**
53: $\mathbf{g}^m \leftarrow \frac{\sum_{p=1}^{P} \mathbf{g}^{p,m}}{P}$
54: $\bar{\mathbf{g}}^m \leftarrow$ Quantize($\mathbf{g}^m, K$)
55: send $\bar{\mathbf{g}}^m$ to all workers
56: **end if**
57: **end function**

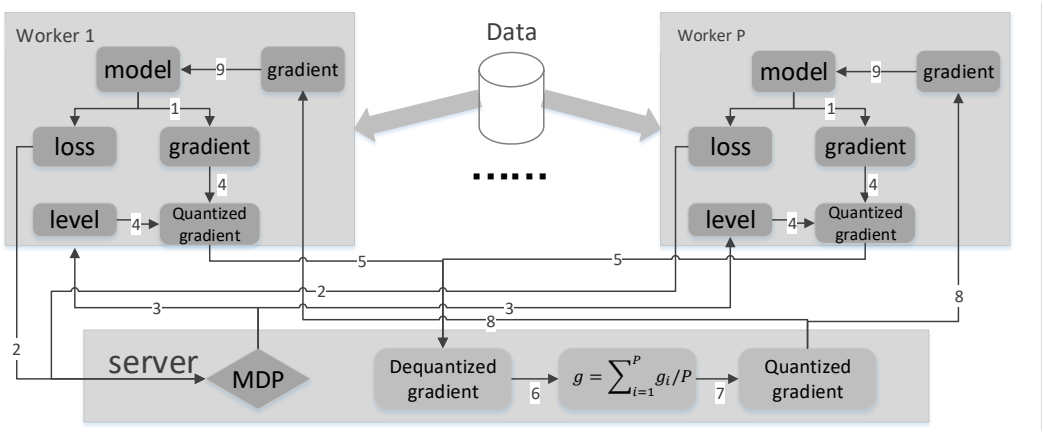

Figure 3: MQGrad system architecture.

Algorithm 2 MQGrad-SARSA

Input: MDP time step t

1: $s_t \leftarrow T(s_{t-1}, a_{t-1})$
2: $r_{t-1} \leftarrow R(s_{t-1}, a_{t-1})$
3: Choose action a_t from s_t using policy derived from Q
4: $\mathbf{v} \leftarrow \mathbf{v} + \eta[r_{t-1} + \gamma Q(s_t, a_t) - Q(s_{t-1}, a_{t-1})] \frac{\partial Q}{\partial \mathbf{v}}|_{t-1}$
5: **return** $\mathbf{s}_t.n_t + a_t$;

the MDP time step $t = \lfloor \frac{m}{T} \rfloor$. In this paper, we empirically set $T = 5$, which means the MDP time step is 5 times slower than the number of training iterations. Note that both t and m start from 0.

States $\mathcal{S}$: The MDP state $\mathbf{s} \in \mathcal{S}$ at time step t is denoted as $\mathbf{s}_t = \left[n_t, \overline{\mathbf{L}}_t\right]$, where $n_t \in Z^+$ is the most confident quantization bits at time step t, predicted by the Q function; $\overline{\mathbf{L}}_t$ is calculated as follows: at time step t, the MDP receives T consequent global losses, denoted as $\mathbf{L}_t = \{L_t^1, \cdots, L_t^T\}$, where $L_t^i (i = 1 \cdots T)$ is the global loss calculated based on the local losses received at the training iteration $(t-1) \times T + i$. The T global losses reflect the goodness of the quantization bits used at the last T iterations. These values, however, may vary to a large extent. To make the statistics of these losses stable, following the practices in [3], MQGrad makes use of the moving average technique to smooth these losses:

$$\overline{L}_t^i = \begin{cases} \alpha * L_t^i + (1-\alpha) * \overline{L}_t^{i-1} & 1 < i \leq T \\ \alpha * L_t^1 + (1-\alpha) * \overline{L}_{t-1}^T & i = 1, \end{cases} \quad (1)$$

where α is the parameter. Thus $\overline{\mathbf{L}}$ is a sequence of T values: $\overline{\mathbf{L}}_t = \{\overline{L}_t^1, \cdots, \overline{L}_t^T\}$.

Actions $\mathcal{A}$: MQGrad has two actions at each time step: $\mathcal{A} = \{0, 1\}$, where 0 keeps the current quantization bits and 1 increases the quantization bits by one. Thus given $\mathbf{s}_t$ and the chosen action a_t, the quantization bits for the immediate next T training iterations is $n_t + a_t$, where n_t is the quantization bits in state $\mathbf{s}_t$. Note that MQGrad does not decrease the quantization bits. The configuration is based on the observation that with the machine learning training iteration goes on, more accurate gradients are needed to update the model parameters because the parameters are closer to the optimal solution. Experimental results also showed that the configuration can achieve better results.

Q **function**: The Q function predicts the value of taking action a at the state $\mathbf{s}$ following policy $\boldsymbol{\pi}$. Following the practice in DQN [16], MQGRad configures the Q function as a neural network (parameterized by $\mathbf{v}$). The input to the neural network is the state and the outputs are the confidence values for the available actions. The parameter $\mathbf{v}$ will be updated during the MDP iterations with SARSA algorithm [20].

Policy $\boldsymbol{\pi}$:$\boldsymbol{\pi}$ defines the probability of selecting an action a at state $\mathbf{s}$. We define the policy $\boldsymbol{\pi}$ with the ϵ-greedy criteria for balancing the exploration and exploitation. Specifically, at the MDP time step t, given the state $\mathbf{s}_t$, the probability of selecting an action a_t is denoted as $\pi(a_t|\mathbf{s}_t)$ and defined as:

$$\pi(a_t|\mathbf{s}_t) = \begin{cases} 1-\epsilon & a_t = \arg\max_a Q(\mathbf{s}_t, a) \\ \epsilon & \text{otherwise,} \end{cases}$$

Reward R: MQGrad calculates the reward on the basis of the T consequent losses collected from the last T training iterations and the total time cost for executing the T iterations. Intuitively, small decrease in loss with high time cost makes the reward small, and vise versa. Specifically, suppose that the moving averaged losses are $\overline{\mathbf{L}}_{t+1} = \{\overline{L}_{t+1}^1, \cdots, \overline{L}_{t+1}^T\}$ and the time cost for executing the last T training iterations is $c_{t+1} \in Z^+$ (in milliseconds). MQGrad solves the following linear regression problem for getting the decreasing rate with respect to iteration β:

$$(\beta, b) \leftarrow \arg\min_{\beta, b} \sum_{i=1}^{T} \left(\beta \times i + b - \overline{L}_{t+1}^i\right)^2,$$

where b is the bias. The reward is calculated as:

$$R(s_t, a_t) = -\frac{1}{c_{t+1}} \times \beta \times \gamma, \quad (2)$$

where $\gamma > 0$ is a scaling parameter.

Transition T: The transition function $T : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ defines the transition of the MDP state. The output of T also consists of

two parts: $\mathbf{s}_t = [n_t, \overline{\mathbf{L}}_t]$. These two components are calculated as:

$$\begin{aligned}\mathbf{s}_t = [n_t, \overline{\mathbf{L}}_t] =& T(\mathbf{s}_{t-1} = [n_{t-1}, \overline{\mathbf{L}}_{t-1}], a_{t-1}) \\ =& [n_{t-1} + \arg\max_{a \in \{0,1\}} Q(\mathbf{s}_{t-1}, a), \overline{\mathbf{L}}_t].\end{aligned} \tag{3}$$

After selecting a_{t-1} on the basis of s_{t-1}, the server broadcasts the quantization bits $n_{t-1} + a_{t-1}$ (where n_{t-1} is the quantization bits in s_{t-1}) to all of the worker nodes. MQGrad then monitors and collects the losses of the immediate T training iterations and constructs a sequence of moving averaged losses $\overline{\mathbf{L}}_t = \{\overline{L}_t^1, \cdots, \overline{L}_t^T\}$, on the basis of Equation (1). As for n_t, if $Q(s_{t-1}, 0) \geq Q(s_{t-1}, 1)$, MQGrad keeps $n_t = n_{t-1}$. Otherwise, MQGrad increases the quantization bits in state s_t by 1.

During the running of the MDP, the SARSA algorithm is used for determining the quantization bits and learning the parameters in Q function, as shown in Algorithm 2. The running of the MDP in MQGrad can be described as follows: at each MDP time step $t = 0, 1, \cdots$, the agent(server) receives the state $s_t = [n_t, \overline{\mathbf{L}}_t]$ (line 1 of Alg. 2) and the reward r_{t-1} (line 2 of Alg. 2). Then an action a_t is selected on the basis of the policy $\pi(a_t|\mathbf{s}_t)$ (line 3 of Alg. 2). After that, the system updates the parameter $\mathbf{v}$ of the Q network (line 4 of Alg. 2). Finally the number of bits $s_t.n_t + a_t$ is returned for conducting the gradient quantization (line 5 of Alg. 2).

The source code of MQGrad can be found in the Github https://github.com/cuiguoxin/MQGrad.

4 EXPERIMENTS

4.1 Experimental settings

To test the performances of the proposed MQGrad system, experiments were conducted on two PS clusters. One consists of 12 nodes and the other consists of 18 nodes. Each nodes in the clusters contains 4 cores each of which has a frequency of 2.3GHz and these nodes were connected by a network with 10MB/s bandwidth.

The experiments were conducted on the basis of the CIFAR-10 [11] dataset. The machine learning algorithm tested is a 5-layer neural network: the first two are convolutional layers with each layers' parameter's shape being [5, 5, 3, 64] and [5, 5, 64, 64]. Local response normalization after max-pooling is used [12]. The third and fourth layers are fully connected layers with shapes [3136, 2304] and [2304, 3840], respectively. The last softmax layer is also a fully connected layer with shape [3840, 10]. Cross entropy loss with ℓ_2 norm of the third layer and fourth layer's parameters are used as the loss function. During the training, the batch size is set to 32 and the learning rate is set to 0.2. Considering the third and fourth layers have about 99.2% of the network parameters, gradient quantization is applied to these two layers. Other parameters are communicated without any compression.

The range of quantization bits is set to 2 to 8 bits (7 levels). The Q function has three layers: the first layer contains 5 nodes, representing the 5 average smoothed values in state s. The second layer contains 10 nodes with ReLU activation. The third layer is a linear layer which has 7 nodes, each corresponds to a quantization bits.

MQGrad has some hyper parameters. The variables in SARSA $\epsilon = 0.1$ and $\eta = 0.1$. The moving average parameter $\alpha = 0.01$ and the scaling variable $\gamma = 300$.

We compared MQGrad with several state-of-the-arts baselines in gradient quantization, including the adaptive quantization method [17] (denoted as "Adaptive" in the paper) and the fixed quantization methods. For the fixed bit quantization methods, the numbers of quantization bits were set to 2, 4, and 8 and denoted as "Fix (2-bit)", "Fix (4-bit)", and "Fix (8-bit)", respectively.

4.2 Experimental results

Figure 4 and Figure 5 show the training curves of "MQGrad" as well as the baselines in terms of the neural network loss being optimized, on the 12-node PS cluster and the 18-node PS cluster, respectively. The x-axises indicate the training time (in terms of hours). From the results, we can see that "MQGrad" outperformed all the baseline methods (used less training time to reach smaller loss) on both of these two clusters. For example, compared with the best baseline "Fix (4-bit)", "MQGRad" used less 7.5 hours to reach the same loss on the 18-node cluster. The results indicate the effectiveness of using reinforcement learning for gradient quantization.

From the results, we can also see that the training curve "Fix (2-bit)" decreased the loss function very fast during the first ten hours. However, it did not converge in the remaining training time. The phenomenon indicated that at the early stage of the training low quantization bits helped to minimize the loss function fast. However, with the training goes on, high accurate gradients were necessary and the low quantization bits hurt the convergence. On the other hand, the training curve of "Fix (8-bit)", which used more bits for quantizing the gradients during the training, steadily decreased during all of the training time. However, the decreasing speed was slow because a lot of time was wasted for transiting the gradients. Thus, "Fix (8-bit)" needed longer time to converge. "MQGrad" made a good trade-off: it used low quantization bits at the early training stages for saving the communication volume, and gradually increased the quantization bits for increasing the gradient accuracies. The method of "Adaptive" can also decrease the communication volume at the early training stages. However, the predefined heuristics in "Adaptive" cannot make good decisions to guarantee the gradient accuracy at the later training phases.

We also tested the model accuracies for these methods. Table 1 and Table 2 show the results on the 12-node cluster and 18-node cluster, respectively. "N/A" indicates the result is not available because the model has converged at the time. From the results we can see that the accuracies of MQGrad are higher than the baselines when trained with the same time, indicating the lower loss leads to higher performances. The final converged performances of MQGrad are comparable to "Adaptive" and "Fix (8-bit)", indicating MQGrad can accelerate the training process while keeping model accuracies.

Note that the execution of the quantization/de-quantization (and the MDP) modules in the baselines and MQGrad needs some additional time. We conducted experiments to show the fraction of these additional time among the whole training time. From the results shown in Table 3, we can see that on the 12-node and the 18-node clusters, MQGrad respectively need 7.41% and 4.11% of the time for running the quantization, de-quantization, and the MDP modules. For other baseline methods, most of the fractions are less than 10%. The results indicate that 1) the additional time

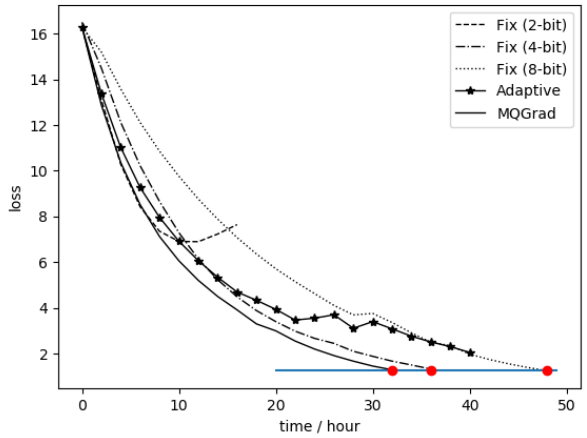

Figure 4: Learning curve on the 12-node cluster.

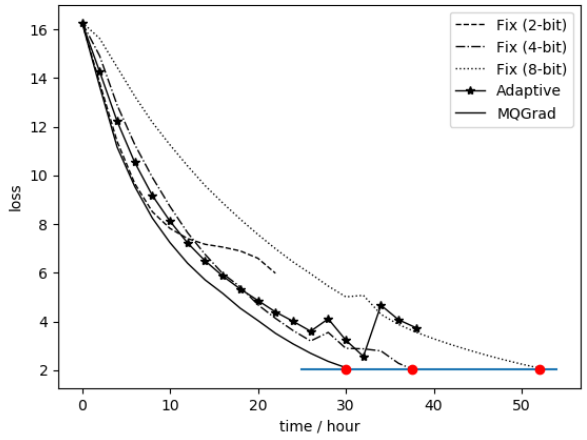

Figure 5: Learning curve on the 18-node cluster.

Table 1: Test accuracies (%) of models trained on 12-node cluster.

method \ time	5 hours	15 hours	30 hours	40 hours
Fix (2-bit)	50.8	55.9	N/A	N/A
Fix (4-bit)	41.9	57.5	66.7	N/A
Fix (8-bit)	35.7	55.6	60.8	68.1
Adaptive	49.4	60.1	65.3	N/A
MQGrad	54.7	65.6	67.7	N/A

costs caused by MDP module in MQGrad is negligible; 2) the time cost for quantizing/de-quantizing gradients is not high, making all of these methods can accelerate the overall training iterations.

Table 2: Test accuracies (%) of models trained on 18-node cluster.

method \ time	5 hours	15 hours	30 hours	40 hours
Fix (2-bit)	51.5	58.5	N/A	N/A
Fix (4-bit)	45.4	57.9	68.4	N/A
Fix (8-bit)	42.1	58.2	65.9	68.2
Adaptive	43.6	57.3	63.4	N/A
MQGrad	51.5	62.0	68.2	N/A

Table 3: Fraction of time cost (%) caused by gradient quantization.

	Fixed (2-bit)	Fixed (4-bit)	Fixed (8-bit)	Adaptive	MQGrad
12-node cluster	13.3	8.0	4.44	8.13	7.41
18-node cluster	11.4	7.27	4.21	5.84	4.11

5 CONCLUSION

In the paper we propose a novel gradient quantization method called MQGrad, for accelerating the distributed machine learning algorithms on parameter server. MQGrad learns to determine the number of bits for gradient quantization with the information collected from the past optimization iterations. MDP is used to formalize the process and the on-policy learning algorithm SARSA is used to learn the quantization bits and update the MDP parameters. Experimental results on a benchmark dataset showed that MQGrad outperformed the state-of-the-arts gradient quantization methods, in terms of accelerate the speeds of learning large scale machine learning models. Analysis showed that MQGrad accelerated the learning speeds through lowering the communication volume at the early stage of training and gradually improving the gradient accuracies with the training went on.

REFERENCES

[1] Dan Alistarh, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2016. QSGD: Randomized Quantization for Communication-Optimal Stochastic Gradient Descent. *arXiv preprint arXiv:1610.02132* (2016).

[2] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. 2016. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*. 3981–3989.

[3] Christian Daniel, Jonathan Taylor, and Sebastian Nowozin. 2016. Learning Step Size Controllers for Robust Neural Network Training.. In *AAAI*. 1519–1525.

[4] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

[5] Jie Fu, Zichuan Lin, Miao Liu, Nicholas Leonard, Jiashi Feng, and Tat-Seng Chua. 2016. Deep Q-Networks for Accelerating the Training of Deep Neural Networks. *arXiv preprint arXiv:1606.01467* (2016).

[6] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 243–254.

[7] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[8] Stephen José Hanson and Lorien Y Pratt. 1989. Comparing biases for minimal network construction with back-propagation. In *Advances in neural information processing systems*. 177–185.

[9] Babak Hassibi and David G Stork. 1993. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*. 164–171.

[10] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*. 1223–1231.

[11] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009).

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[13] Yann LeCun, John S Denker, and Sara A Solla. 1990. Optimal brain damage. In *Advances in neural information processing systems*. 598–605.

[14] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server.. In *OSDI*, Vol. 1. 3.

[15] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. *arXiv preprint arXiv:1706.04972* (2017).

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[17] Anders Øland and Bhiksha Raj. 2015. Reducing communication overhead in distributed learning by an order of magnitude (almost). In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*. IEEE, 2219–2223.

[18] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*.

[19] Nikko Ström. 1997. Phoneme probability estimation with dynamic sparsely connected artificial neural networks. *The Free Speech Journal* 5 (1997), 1–41.

[20] Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge.

[21] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. *arXiv preprint arXiv:1705.07878* (2017).

[22] Chang Xu, Tao Qin, Gang Wang, and Tie-Yan Liu. 2017. Reinforcement Learning for Learning Rate Control. *arXiv preprint arXiv:1705.11159* (2017).

[23] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).