DEEPCON: Improving Distributed Deep Learning Model Consistency in Edge-Cloud Environments via Distillation

Bin Qian, *Member IEEE*, Jiaxu Qian, Zhenyu Wen*, *Senior Member, IEEE*, Di Wu, *Member IEEE*, Shibo He, *Senior Member IEEE*, Jiming Chen, *Fellow IEEE*, Rajiv Ranjan, *Fellow IEEE*

Abstract—In a typical distributed Deep Learning (DL) based application, models are configured differently to meet the requirements of resource constraints. For instance, a large ResNet56 model is deployed on the cloud server while a small lightweight MobileNet model is more suitable for the end-user device with fewer computation resources. However, the heterogeneity of the model architectures and configurations may bring a systemic problem - models may produce different outputs when given the same input. This inconsistency problem may cause severe system failure of prediction agreement inside the application. Current research has not studied the systemic design for efficiently detecting and reducing the inconsistency among models in distributed DL applications. With the increasing scale of distributed DL applications, the challenges of inconsistency mitigation should consider both algorithm and system design. To this end, we design and implement DEEPCON, an adaptive deployment system across the edge-cloud layer with over-the-air model updates. We implement ASRS sampling for efficiently sampling data to reveal the real data distribution as well as model prediction inconsistency. Then, we implement DMML-Par, an asynchronous parallel training algorithm for quickly updating the models and reducing inconsistency. DEEPCON implements over-the-air updates with a set of APIS to enable seamless inconsistency detection and reduction in such deep learning applications. Our experiment results on both vision and language tasks demonstrate that DMML could improve the model consistency up to 4%, 7%, and 13% at CIFAR10/100 and IMDB datasets without sacrificing the accuracy of individual models. We also show that the ASRS sampling can save 90% network bandwidth of data transmission and that DMML-Par is up to 60% faster compared to simple synchronous parallel training.

Index Terms—Model Consistency, Distributed Applications, Knowledge Distillation, Edge Computing, Internet of Things

1 INTRODUCTION

Modern deep learning applications are distributed across various hardware platforms such as cloud or edge GPUs, end-user mobile phones, and IoT devices, as emphasized in the work [1], [2]. However, because these devices vary in their hardware capabilities, there arises a need to deploy several deep learning models dynamically. These models should have similar functionalities while differing in their architectures and parameters, to address various quality of service (QoS) demands. This adaptive approach is crucial for efficiently maximizing the use of available resources.

The adaptive deployment strategies, however, could lead to a significant issue in distributed deep learning applications. When deploying multiple models within a single application, these modes may yield inconsistent inference outcomes in production. For example, a cylinder object may be classified as either a cylinder or a cube by two different models. This inconsistency could potentially disrupt the functioning of the distributed deep learning application, which depends heavily on the coordinated outputs of various models, as detailed in Section 2.

To address this issue, it is necessary for the models to cooperate and jointly identify the inconsistent instances. They should engage in interactions, gaining knowledge from one another, in order to reach an agreement. The inconsistency problem has been studied in various related works. Works on model "irreproducibility/disagreement" [3], [4], [5], [6] identify several factors that may lead to inconsistent predictions during model re-training, including activation functions [3], model randomness [4], model architectures [5], optimizers [6]. Techniques such as co-distillation [7], [8] and label smoothing [9] are proposed to reduce such model inconsistency as well. Nevertheless, these techniques solely tackle the inconsistency problem in situations where a single model is consistently updated throughout training. Their methodology cannot be directly extended to encompass multiple models with distinct parameters and architectures. Thus, it becomes imperative to develop innovative communication structures and learning mechanisms to accommodate the emerging challenges.

To reduce model inconsistency in distributed deep learning applications need to resolve the following challenges: **(1). How to detect the inconsistency among the distributed models?** A distributed DL application can have M versions (types) of models which are deployed on N edge nodes. This brings the challenge of how to efficiently interact with the different outputs of the models to provide a model consistency measurement. Unlike the model accuracy that can

- *Bin Qian, Shibo He, and Jiming Chen are with the State Key Laboratory of Industrial Control Technology, Zhejiang University, Hangzhou, China. (E-mail: bin.qian@zju.edu.cn, s18he@zju.edu.cn, cjm@zju.edu.cn).*
- *Jiaxu Qian and Zhenyu Wen are with the Institute of Cyberspace Security and College of Information Engineering, Zhejiang University of Technology, Hangzhou, China. (E-mail: q1anjiaxu001@gmail.com, zhenyuwen@zjut.edu.cn)*
- *Di Wu is with the School of Computer Science, University of St Andrews, UK. (E-mail: dw217@st-andrews.ac.uk)*
- *Rajiv Ranjan is with the Newcastle University, UK. (E-mail: raj.ranjan@newcastle.ac.uk)*

be measured locally, measuring the inconsistency requires comparing the outputs of various models. A very intuitive idea is to offload the data from N nodes to the cloud and then perform the consistency measurement. This may cause a waste of the network bandwidth. Hence, in this work, we seek to design a data acquisition method that efficiently samples the data from various edge nodes to perform the consistency measurement in the cloud. **(2) How to reduce the inconsistency among the heterogeneous models?** There are a few works [3], [4], [5], [6] that have studied the model inconsistency issues, but there are not applicable to the real-world distributed DL applications in the following reasons. First, the multiple models' inconsistency problem presents a unique challenge in that topologies and parameters of the models are various, and therefore how to cooperate with the models to mitigate the gaps is a research question. In addition, each model needs to fine-tune with the rest of the models. For example, we assume a distributed DL application consists of M models, and we have to perform the fine-tuning at least M times. Hence, designing an effective fine-tuning mechanism is a challenge for this paper [10].

To tackle these challenges, we design and implement DEEPCON to realize our goal of quickly *detecting* and *reducing* the model inconsistency of an edge-based application. We first motivate the importance of tackling inconsistent issues, formally define the inconsistent metrics, and conduct comprehensive experiments to show the difference between the model prediction accuracy and inconsistency. Then we develop the core technology Deep Mixup Mutual Learning (DMML) in DEEPCON, a learning algorithm - We employ knowledge distillation (KD) [11] to encourage the model to produce more consistent outputs by devising mixup labels as the distillation target across different models. Furthermore, we extend DMML and implement asynchronous parallel DMML (DMML-Par) for accelerating the model training in multiple GPU workers. Finally, we design an over-the-air update framework of DEEPCON with Adaptive stratified reservoir sampling (ASRS), and implement with a set of APIs to support seamless communications between edge and cloud for data sampling (Edge), model consistency validating (Cloud), and model updating (GPU cluster).

Overall, this paper makes the following contributions:

- **Definition and quantification of model inconsistency (§2).** We show with an example the severity of the inconsistent prediction in distributed deep learning applications. Then we define <Acc-cc> gap, a metric that differentiates model inconsistency and accuracy. Extensive benchmark has been made to show that the <Acc-cc> gap is ubiquitous in such applications.
- **DMML for reducing model inconsistency (§3).** We illustrate the importance of consistency in evaluating the performance of geo-distributed DL applications and define a new consistency metric (CC) for measurement. Then we propose DMML, a KD-based learning algorithm for cross-model learning, improving consistency among models. To improve the DMML's scalability, we develop the DMML-Par that can scale DMML to multiple GPU nodes.
- **Design and implementation of DEEPCON (§4).** DEEPCON provides over-the-air updates to reduce the model inconsistency of distributed DL applications. We design an ASRS sampling method for fast and precise detection of model inconsistency. Moreover, DEEPCON offers an algorithm and system co-design solution to maintain a distributed DL application deployment life-cycle.
- **Comprehensive evaluation of DEEPCON (§5).** We evaluate DMML on vision and language classification tasks: CIFAR-10, CIFAR-100, and IMDB. We also evaluate the training speed of DEEPCON on the same dataset. Our results show that DMML achieves 34.1% to 56.8% CC improvement on pre-trained models. DEEPCON can detect model inconsistency with less than 10% data samples and also achieve up to 60% speedup compared to the simple model parallel algorithm.

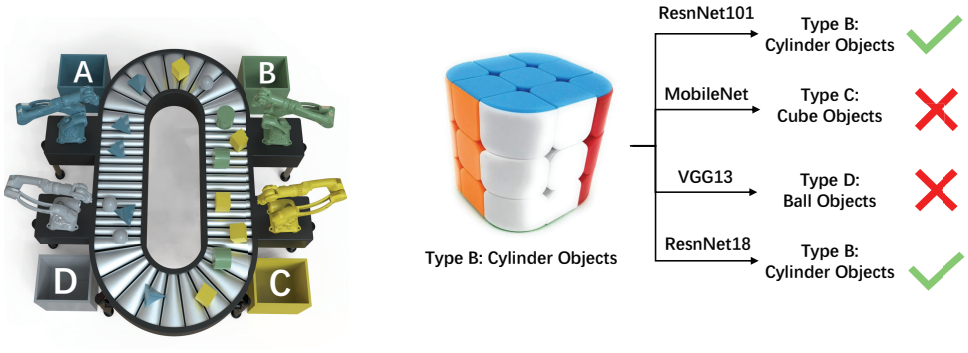
2 IMPACT OF MODEL INCONSISTENCY FOR DISTRIBUTED DEEP LEARNING APPLICATION

2.1 Model Inconsistencies Lead to System Failures

Multiple models within the application may produce different outputs for the same input, as they are usually trained individually or calibrated to different configurations after development. Figure 1(a) shows an application of 4 robot arms that classify items and picks them into different boxes from the conveyors. Due to the various model configurations, a cylinder object as shown in Figure 1(b) belongs to type B and is correctly classified by ResNet101 and ResNet18, but incorrectly by MobileNet and VGG13. In such an application [12], [13], inconsistent model prediction behaviors lead to even worse system failures, which we reveal via a detailed example shown in Table 1(a) and 1(b).

Table 1(a) shows an example prediction results for both cases: consistent and inconsistent prediction results among all models. In case A consistent prediction, M_A, M_B, M_C, M_D are designed to pick item A, B, C and D respectively. The models are making correct predictions on example $X_1 - X_{16}$ while incorrect predictions on example $X_{17} - X_{20}$. All models produce the same results for all the examples, achieving 80% accuracy. The setting is the same for case B as well, with one difference that, in case B, M_{A1}, M_{B1}, M_{C1}, M_{D1} may generate different classification results for the same item: $X_{13} - X_{20}$. In this case, models make correct predictions on example $X_1 - X_{12}$. M_{A1} and M_{B1} make incorrect predictions on example $X_{13} - X_{16}$, while M_{C1} and M_{D1} make incorrect predictions on example $X_{17} - X_{20}$. The 4 models also achieve 80% accuracy.

In Table 1(b), we show the system performance when models make consistent and inconsistent results. Note that we assume that all models have achieved their best classification accuracy after training, where failures are inevitable in model inference. Thus our focus is mainly on the different system failures that arise even when models have the same prediction accuracy. Generally, when models make different/inconsistent prediction results, the systems may encounter much worse failure. 1) incorrect picks on previous correct picks, i.e., example $X_{13} - X_{14}$. 2) None picks on previous correct picks, i.e., example $X_{15} - X_{16}$. 3) None picks on previous incorrect picks, i.e., example $X_{17} - X_{20}$. This inconsistency reduces the ability of the system to correctly pick objects (from 80% to 60%) and potentially collapses the whole system and causes much energy waste.



(a) Classify and Pick (b) Classification Error

Fig. 1: An Example of classification application with 4 models collaborating and picking up the right items from the conveyor. The robot arms are equipped with specifically designed models but serve the same task.

Correct / Incorrect		Predictions							
		Case A: Consistent				Case B: Inconsistent			
Examples	Class	M_A	M_B	M_C	M_D	M_{A1}	M_{B1}	M_{C1}	M_{D1}
X_1 - X_{12}	...	Correct				Correct			
X_{13}	A	A	A	A	A	B	B	A	A
X_{14}	A	A	A	A	A	B	B	A	A
X_{15}	B	B	B	B	B	D	D	B	B
X_{16}	B	B	B	B	B	D	D	B	B
X_{17}	C	D	D	D	D	C	C	B	B
X_{18}	C	D	D	D	D	C	C	B	B
X_{19}	D	C	C	C	C	D	D	A	A
X_{20}	D	C	C	C	C	D	D	A	A
Acc		16/20=80%				16/20=80%			

(a) Prediction Results: consistent vs inconsistent

		Predictions			
		Case A: Consistent		Case B: Inconsistent	
Examples	Class	Picked by	Pick result	Picked by	Pick result
$X_1 - X_{12}$	...	correct		correct	
X_{13}	A	M_A	Correct	M_B	Wrong
X_{14}	A	M_A	Correct	M_B	Wrong
X_{15}	B	M_B	Correct	None	Not picked
X_{16}	B	M_B	Correct	None	Not picked
X_{17}	C	M_D	Wrong	None	Not picked
X_{18}	C	M_D	Wrong	None	Not picked
X_{19}	D	M_C	Wrong	None	Not picked
X_{20}	D	M_C	Wrong	None	Not picked
System success		16/20=80%		12/20=60%	

(b) System Failure: Consistent vs Inconsistent

TABLE 1: An illustration of how inconsistent predictions cause worse system failures than incorrect ones.

2.2 Relationship between Accuracy and Consistency

We have unveiled the detrimental consequences stemming from inconsistent predictions among multiple models. Given that pre-trained models have already achieved their highest possible accuracy through individual training, addressing this inconsistency necessitates the application of appropriate optimization techniques. To effectively investigate this issue, it's imperative to initially differentiate between model accuracy and consistency and to pinpoint the performance gap between them. In this context, we provide precise definitions for accuracy, consistency, and the quantifiable disparity existing between these two metrics.

Accuracy (Acc). Accuracy measures the probability of a model correctly predicting the ground truth labels. For any given task, a set of N models $\{M_1 ... M_N\}$, Dataset $D(X, Y)$, we have the following definition:

$$Acc(M_n) = \mathop{\mathbb{E}}_{(x,y)\sim D}[1\{M_n(x) = y\}] \quad (1)$$

For any model M_n and data $\{x, y\} \in D(X, Y)$, $Acc(M_n)$ measures the probability of M_n correctly predicting y. $\mathbb{E}[.]$ denotes the expected accuracy Acc for dataset D.

Consistency (C). Consistency measures the probability of multiple models producing the same result given the same input. Given $\{x, y\} \in D$ and N models $\{M_1 ... M_N\}$, we have the following definition:

$$C(M_1...M_N) = \mathop{\mathbb{E}}_{(x,y)\sim D}[1\{M_1(x) = ... = M_N(x)\}] \quad (2)$$

C measures the probability of models $\{M_1...M_N\}$ predicting the same result at a given point x, i.e., $\mathbb{E}[.]$ stands for the expected value of model consistency C for dataset D.

Correct Consistency (CC): the Intersection between Accuracy (Acc) and Consistency (C). Consistency (C) only measures if all models produce the same results, no matter if they are correct or not. In real-world applications, we are more interested in *consistent and correct predictions* for a set of models within the same application. Thus, we use correct consistency (CC) to define this as shown in Eq. 3.

$$CC(M_1...M_N) = \mathop{\mathbb{E}}_{(x,y)\sim D}[1\{M_1(x) = ... = M_N(x) = y\}] \quad (3)$$

CC measures the probability of multiple models $\{M_1 ... M_N\}$ correctly predicting the target y at the same input point x. i.e., $\mathbb{E}[.]$ stands for the expected value of correct consistency CC for dataset D.

CC is at the intersection between the Consistent (C) and correct outputs (Acc) produced by all the models, as formulated in Eq. 5.

$$CC(M_1...M_N) = C(M_1...M_N) \cap Acc(M_1)... \cap ...Acc(M_N) \quad (4)$$

<Acc-CC> Gap: Theoretical Gap of Improvement between Accuracy and Correct Consistency. For a set of reported model accuracy $Acc(M_1)$ ... $Acc(M_N)$, the maximum ideal CC is $\min\{Acc(M_1), ..., Acc(M_N)\}$: the accuracy of the smallest/worst performed model.

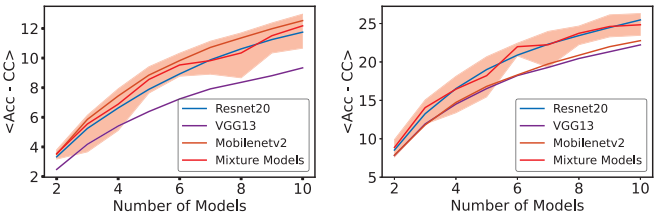
Thus for multiple models $\{M_1 ... M_N\}$, the gap for improvement is defined as:

$$< Acc{-}CC >= \min\{Acc(M_1)...Acc(M_N)\} - CC(M_1...M_N). \quad (5)$$

Accuracy (Acc) and correct consistency (CC) are correlated and serve different purposes in assessing model

	Metrics				
	Acc_1	Acc_2	C	CC	<Acc-CC>
CIFAR-10					
ResNet20+ResNet20	91.73	91.78	92.65	88.56	3.17
ResNet20+ResNet56	91.73	93.27	92.51	89.30	2.43
ResNet20+VGG13	91.73	93.26	92.34	89.15	2.58
CIFAR-100					
ResNet20+ResNet20	66.86	67.47	70.00	58.28	8.58
ResNet20+ResNet56	66.86	70.47	69.89	59.47	7.39
ResNet20+VGG13	66.86	71.57	68.75	59.74	7.12

TABLE 2: Different Metrics Reported on CIFAR10/100



(a) CIFAR-10 (b) CIFAR-100

Fig. 2: Acc and CC Gap (Eq. 6) for Different Models.

performance. An effective system should minimize the <Acc-CC> gap, indicating both high accuracy and correct consistency to be deemed reliable. However, achieving this balance is often challenging in practice. This motivated the paper to build a system that can efficiently detect and reduce model inconsistency within a distributed DL application.

2.3 Factors that impact <Acc-CC> Gap

Model heterogeneity. Table 2 compares the Top-1 accuracy and consistency metrics (C, CC, and <Acc-CC>) of CIFAR-10 and CIFAR-100 dataset obtained from our pre-trained models. Three pairs of model parameters are generated as follows: (i) The same model architecture but trained twice with different initialization parameters (ResNet20 and ResNet20). (ii) Models with the same backbone module but with different depths (ResNet20 and ResNet56). (iii) Models with totally different architectures (ResNet20 and VGG13).

Experimental results show that the <Acc-CC> gap is ubiquitous for models of the same and different architectures. <Acc-CC> gap is around 3% in CIFAR10 to around 8% in CIFAR100. The loss primarily arises from factors such as the random initialization of models [14], the ordering of mini-batches [15], data augmentation and processing [16], and hardware variations [7]. Performing the training multiple times with the same setting can result in models that produce a surprising number of conflicting predictions, even when all models achieve very high accuracies [9].

Model architecture and number. Fig. 2 compares the <Acc-CC> gap among a different number of models. We randomly initialize parameters for Resnet20, VGG13, MobileNetv2_x0_5 and repeat model training 20 times on CIFAR10 and CIFAR100 respectively. Then, for each model, we randomly select n models from all 20 models, and report the <Acc-CC> gap among these models. We report the mean, upper, and lower bound of the <Acc-CC> gap as well.

From the experimental results, we observe that *model number* is positively correlated with the <Acc-CC> gap. When more models join the evaluation, the gap increase is almost linear to the model numbers, for both CIFAR10 and CIFAR100. *Model architecture* also accounts for large <Acc-CC> gap. For the same model, MobileNetv2 and ResNet20

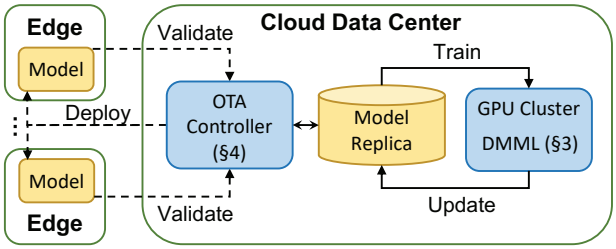


Fig. 3: DEEPCON Overview

report the highest gap on the two datasets. For the mixed model scenario with 3 different models, the gap is the highest with large variance as well, compared to the case where models have the same architecture.

task complexity. *Task complexity* also accounts for large <Acc-CC> gaps: Table 2 and Fig. 2 show that CIFAR100 has 3-2 times the gap compared of CIFAR10 when increasing the model number from 2 to 10.

These findings necessitate the need to improve model consistency, which is paramount for distributed DL applications with multiple models and varying configurations.

3 DEEPCON: IMPROVING DISTRIBUTED MODEL CONSISTENCY VIA DISTILLATION

DEEPCON aims to develop an adaptive deep learning model deployment system that combines the computing resources from the cloud server, GPU cluster, and edge devices, which enables the models to be updated in OTA (over-the-air) manner. To this end, we build DEEPCON that comprises two main components: *Deep Mixup Mutual Learning (DMML)* algorithm and *OTA controller*.

The DMML algorithm is running on a cloud, which is developed to optimize the parameters of the models to mitigate the inconsistency when the inconsistency is detected or exceeded a threshold (see §3.2). In order to maximize the resource usage in distributed cloud GPU servers, we also propose an asynchronous parallel DMML in §3.3. The OTA controller orchestrates the model update and re-deployment across cloud and edge devices (see §4). It maintains a deep learning application deployment life-cycle through the following operations, i.e., Deploy → Validate → Train → Update → Deploy. The high-level system overview of DEEPCON is shown in Fig 3.

The system achieves the following three goals.

- **Seamless OTA.** DEEPCON utilizes the computing resources both from edge devices and the cloud to achieve seamless OTA. This algorithm and system co-design solution ensures non-stop performance validation and model updates on the host devices.
- **Generability.** We develop a generic deep learning model updating algorithm for easy fine-tuning of models without being affected by the number of models and the model architectures.
- **Scalability.** The proposed model updating algorithm is able to fine-tune the models in parallel and automatically scale to multiple computing nodes. Therefore, if the application developers want to reduce update latency, he/she only needs to add more computing resources and require zero code changes.

In this section, we describe the design of DMML. Following the definition in Equation 5, we first formulate the system optimization goal in § 3.1, and transforms the problem into a form that can be solved via knowledge distillation technology. Then, we develop an online knowledge distillation-based method to solve the problem in § 3.2. We also accelerate the training of DMML via parallel training as discissued in § 3.3.

3.1 System Optimization Goal

To ensure the consistency of various models, we have the following optimization goal.

$$\begin{aligned} \arg\min M_1..M_N \quad & Gap < Acc - CC > (M_1, ..., M_N) \\ \text{s.t.} \quad & Acc(M_n; D) - Acc(M_n^{'}; D) <= \xi \\ & \forall M_n \in \{M_1..M_N\} \\ & \xi \geq 0 \end{aligned} \tag{6}$$

For any dataset D, we aim to maximize the CC for all models $\{M_1 \ ... \ M_N\}$. Denote M_n and $M_n^{'}$ as models before/after re-parameterization, we also add a constraint for model accuracy. The slack factor ξ enables the retrained model $M_n^{'}$ to tolerate maximum accuracy decrease ξ, compared to the original model M_n. Here, ξ is a positive integer determined empirically during the retraining process, providing flexibility in balancing model performance. In this paper, we set $\xi = 0$, representing a specific case where the model is constrained to operate without any reduction in accuracy. For all experiments, we only report results where the accuracy after retraining is greater or equal to the original accuracy.

Solving the optimization problem with Knowledge Distillation. Knowledge distillation (KG) [11] allows teaching or "distilling" the knowledge from one or a set of models to other models, which naturally meets the optimization goal illustrated in Eq. 6. To transfer the optimization problem to a KD task, we first have the following goals:

$$\arg\min_{M_S} {}_{\{x,y\}\in D}\mathcal{L}(M_S(x), y) + \mathcal{L}(M_S(x), M_T(x)) \tag{7}$$

Given a teacher model M_T and labeled dataset $D(X, Y)$, the goal of KD is to generate a student model M_S by learning from both M_T and $D(X, Y)$ [11]. Two loss functions $\mathcal{L}(M_S(x), y)$ and $\mathcal{L}(M_S(x), M_T(x))$ measures the difference between M_S and D, M_T respectively. By minimizing both loss functions at the same time, the learned M_S retains the knowledge from M_T and the dataset D.

We note the ground truth label y could also be regarded as the output of a perfect model. In order to agree with the output of all models and the ground truth label, our optimization goal could be formulated as:

$$\arg\min_{M_1..M_N} {}_{\{x,y\}\in D}\mathcal{L}(y, M_1(x)...M_N(x)) \tag{8}$$

Whereby the loss function measures the difference between all models and the ground truth label. However, it is infeasible to optimize all the models at the same time. More practically, we can recursively reparameterize a model to minimize its difference against all other models and the ground truth label:

$$\arg\min_{M_n} {}_{\{x,y\}\in D}\mathcal{L}(y, M_1(x)...M_N(x)), \forall M_n \in \{M_1..M_N\} \tag{9}$$

Algorithm 1: Deep Mixup Mutual Learning

Input: Training data D (X,Y), learning rate $\gamma(t)$, Mixup Ratio α, N pre-trained models $M = \{M_1, M_2 \ ... \ M_N\}$
Initialize: $t \leftarrow 0$
while *Not Convergence* **do**
  for *randomly sample batch data* $\{x,y\} \in D$ **do**
    $\tilde{M} \leftarrow new\ list()$
    for $n \in N$ **do**
      // Compute $\tilde{M}_n(x, y, M_n, \alpha)$ with E.q. 10
      $\tilde{M}_n \leftarrow \tilde{M}_n(x, y, M_n, \alpha)$
    end
    for $n \in N$ **do**
      $\mathcal{L}_n \leftarrow 0$
      for $k \in N$ **do**
        if $k \neq n$ **then**
          $\mathcal{L}_n \leftarrow \mathcal{L}_n + \mathcal{L}(M_n, \tilde{M}_k)$
        end
      end
      $\mathcal{L}_n \leftarrow \mathcal{L}_n \ / \ (N-1)$
      $M_n \leftarrow M_n + \gamma_t \frac{\partial \mathcal{L}_n}{\partial M_n}$
    end
  end
  t = t + 1
  update learning rate $\gamma(t)$
end

The remaining problem is to construct the loss function $\mathcal{L}(M_n; D)$ of each model M_n on dataset D, which will be discussed in the following section.

3.2 Basic Deep Mixup Mutual Learning (DMML)

In order to minimize the loss function defined in E.q 9, we design a DMML algorithm that consists of two components *Deep Mixup Label* and *Multi-model Distillation* (see Fig. 4).

3.2.1 *Deep Mixup Label*

Deep Mixup Label is a learning target that allows DMML to improve consistency among models while ensuring that individual models still maintain or even improve original accuracy after training. The Deep Mixup Label is a weighted average between the model's output and ground truth label as defined in Eq. 10.

$$\tilde{M}(x, y, M_n, \alpha) = \alpha M_n(x) + (1 - \alpha)y \tag{10}$$

At point $\{x, y\}$, $\tilde{M}$ averages the ground truth label y and pseudo label (output) generated from model $M_n(x)$, controlled by the weighting parameter α. The Deep Mixup Label can be applied to most deep learning tasks, $M_n(x)$ can be the output of any form generated by the models, i.e., bounding box coordinates for detection tasks [17]. For the classification task we evaluate in this paper, $M_n(x)$ can be the outputs after softmax indicating the probabilities of one sample x belonging to any class $y \in Y$.

3.2.2 *Multi-model Distillation*

Fig. 4 shows the details of how to use multiple generated Deep Mixup Labels to train a target model through DMML. Assume we have N models in total, and we would like to update the k-th model M_k. Then, for each of other $N-1$ models and data points $\{x, y\}$, we extract the computed Deep Mixup Label $\tilde{M}(x, y, M_n, \alpha), \forall n \in N, n \neq k$. For each Deep Mixup Label, we compute the difference of $M_k(x)$

This article has been accepted for publication in IEEE Transactions on Cognitive Communications and Networking. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCCN.2025.3547049

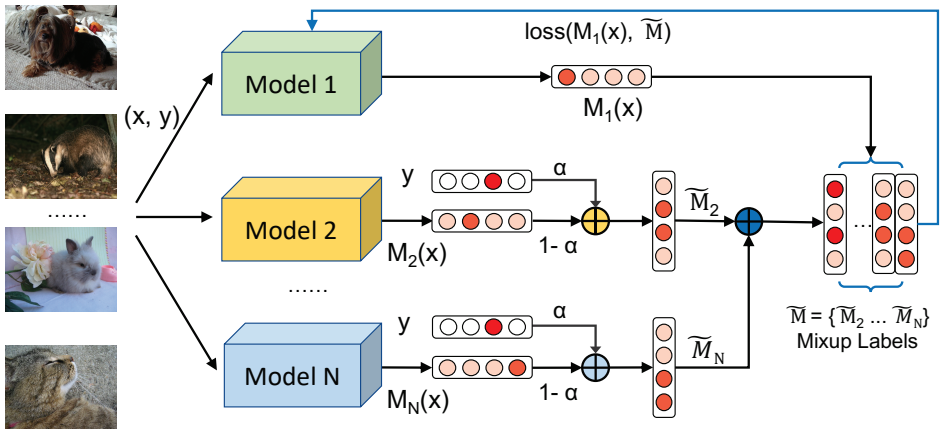


Fig. 4: The overview of **Basic Computation of Deep Mixup Mutual Learning (DMML)**. The same inputs are fed to all models and get results M_1 ... M_N. Then, each M_n and true label y jointly generate a mixup label $\tilde{M}_n$ controlled by a weighting parameter α. For each model, the loss function is computed by comparing its output M_x against all other mixup labels $\tilde{M}$. For example, L_1 is computed by M_1 and $\tilde{M} = \{\tilde{M}_2 ... \tilde{M}_N\}$.

against the Deep Mixup Label $\tilde{M}(x, y, M_n, \alpha)$ with a loss function. All loss functions are summed and averaged to get the final loss function for model M_k, as shown in E.q 11:

$$\mathcal{L}(M_k, \{x, y\}) = \frac{1}{N-1} \sum_{n=1, n\neq k}^{N} \mathcal{L}(M_k(x), \tilde{M}(x, y, M_n, \alpha)) \tag{11}$$

Finally, the computed loss is sent back to the model M_k for computing the gradients and updating the model parameters. (see example M_1 in Fig. 4). In this paper, we use cross entropy for computing each loss in E.q 11.

The design of mixup labels and the loss function in multi-model distillation forces the model to update parameters to produce consistent outputs compared to other models. The accuracy of the models, though, remains stable during training due to the incorporation of ground truth labels in the *Deep Mixup Labels*. We carefully tuned α such that the consistency is maximized while the accuracy of individual models is preserved. We validate the effects of mixup ratio α in Fig. 9 as well.

3.2.3 *DMML Algorithm*

Based on the Deep Mixup label and multi-model distillation, we develop DMML, an algorithm that optimizes all models' parameters simultaneously. Alg. 1 shows details of DMML.

For a given model $M_n \in M$ and the sampled data $\{x, y\}$, we compute the Deep Mixup Label $\tilde{M}_n$ (see line 7). Then, we use E.q. 11 to compute the average of the difference between the output of M_n and other deep mixup labels (from line 10 to 19). Next, we update $\mathcal{L}_n$ and M_n as shown in line 17 and 18. In each iteration, the learning rate $\gamma(t)$ will be updated based on convergence curves to speed up algorithm convergence [18]. Obviously, DMML is not very scaled with the increasing number of models. The next subsection introduces a parallel DMML algorithm.

3.3 Parallel Training of DMML (DMML-Par)

In this subsection, we propose DMML-Par, an approach that parallelizes the DMML, which can be scaled up to multiple GPU nodes [19] in a distributed manner [20], [21]. Alg. 2 illustrates the details of DMML-Par. We develop an asynchronous mechanism that allows DMML-Par to offer a lock-free, non-wait model parameter exchange solution during the fine-tuning phase [22].

We first initialize two lists *run_model* and *nodes_idle* for maintaining the states of the models and workers, where their index represents the corresponding worker and model. All models are not yet allocated and all workers are not occupied (see line 4). Then for all the workers $c_n \in C$ in sequence, we first check if worker c_n is idle (line 7), and if not we will move to the next idle worker. If the worker is idle, we check the model status in *run_model* and allocate the selected model $M_n \in M$ to an idle worker c_n. Then, M_n and worker c_n is marked as running and occupied (line 12). At the same time, the training process is distributed to the target worker c_n via **Train()**.

For training M_n (from line 26 to 39), we first pull the latest model replica from the global model replica M^S. Then, we compute the deep mixup labels and loss against $M_n(x)$, calculate gradients, and update M_n. We iterate through the dataset D until all data has been trained once (from line 28 to 37). Finally, **Update()** function pushes the M_n to the server and release the worker c (from line 22 to 24).

Discussion. The asynchronous design of DMML-Par ensures maximum utilization of available computation resources across the GPU clusters. We ensure that all models get the same rounds of training, such to mitigate the performance drop brought by the asynchronous parallel training (see Fig. 8). DMML-Par is also easily scalable and adaptive to a dynamic number of worker nodes, to meet the latency requirements of the users.

Example of DMML-Par with 5 models on 4 workers. Fig. 5 shows how DMML-Par allocates the models to various workers for parallel training. In the first round, we train models 1 to 4 on workers 1 to 4 respectively. When worker 4 finishes its process, only model 5 (M_5) is not trained. Thus, M_5 is allocated to worker 4. Next, when worker 3 completes the training process for M_3, two models i.e., M_3, and M_4 are able to be allocated to worker 3 (the other three models are still doing first-round training). Since M_3 was trained on worker 3 already, we directly run the second-round training for M_3 on worker 3. Following the same scheduling policy, we allocate M_2, M_1, and M_4 to workers

Algorithm 2: Parallel Deep Mixup Mutual Learning

Input: Training data D (X,Y), learning rate $\gamma(t)$, Mixup Ratio α, N pre-trained models $M = \{M_1, M_2 \ldots M_N\}$, GPU workers $c \in C$, server S

Initialize: t ← 0, Server model replica: $M^S \leftarrow M$
run_model ← new list([False for n in N]) , nodes_idle ← new list([True for c in C])

while *Not Convergence* **do**
  for $c \in C$ **do**
    if *nodes_idle[c] == False* **then**
      continue
    else
      for $n \in N$ **do**
        if *run_model[n] == False* **then**
          nodes_idle[c] ← False, run_model[n] ← True
          Train(M_n, α, t, c, n) // Remote procedure call to train M_n on worker c
          break
        end
      end
    end
  end
  t = t + 1
  update learning rate $\gamma(t)$
end

Function `Update` (c, n, M_n)**:**
  $M_n^S \leftarrow M_n$ // Update M_n in S
  nodes_idle[c] ← True, run_model[n] ← False
return

Function `Train` (M_n, α, t, c, n)**:**
  $M \leftarrow M^S$ // Worker c pulls latest model from S
  for *randomly sample batch data* $\{x,y\} \in D$ **do**
    $\mathcal{L}_n \leftarrow 0$
    for $k \in N$ **do**
      if $k \neq n$ **then**
        $\tilde{M}_k \leftarrow \tilde{M}_k(x, y, M_k, \alpha)$ // Compute $\tilde{M}_k(x, y, M_k, \alpha)$ with E.q. 10
        $\mathcal{L}_n \leftarrow \mathcal{L}_n + \mathcal{L}(M_n, \tilde{M}_k)$
      end
    end
    $\mathcal{L}_n \leftarrow \mathcal{L}_n$ / $(N-1)$
    $M_n \leftarrow M_n + \gamma_t \frac{\partial \mathcal{L}_n}{\partial M_n}$
  end
  Update(c , n, M_n) // Remote procedure call to update model n and release the worker c
return

1, 2, and 3, respectively. Notably, before starting a training process on a worker and a training process finishes, DMML-Par will pull the latest version of all models from the master node (see Alg. 2 line 27) and push the trained model to the master node (see Alg. 2 line 23), this operation allows the models to be updated asynchronously maximizing the utility of computing resources.

4 Over-the-Air Update in DeepCon

Over-the-air (OTA) updates play a crucial role in distributed deep learning applications, ensuring seamless integration of the latest advancements and improvements, and empowering the models to continuously evolve and adapt to emerging system failures [23], [24]. By integrating computation and communication across the edge and the cloud [25], OTA updates can accomplish the fine-tuning of the deployed models via efficient data sampling and parallel computation, thereby significantly improving the update efficiency.

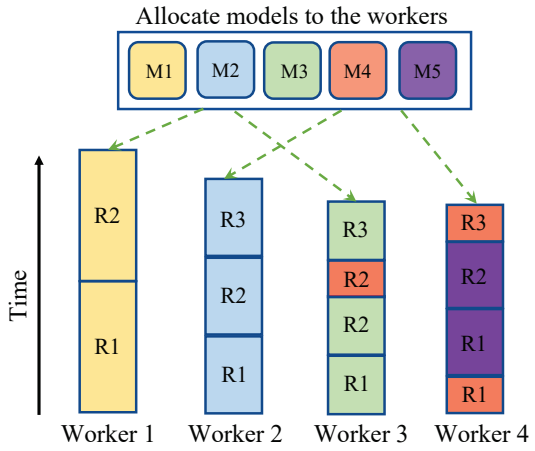


Fig. 5: Example of DMML-Par on 5 Models and 4 workers

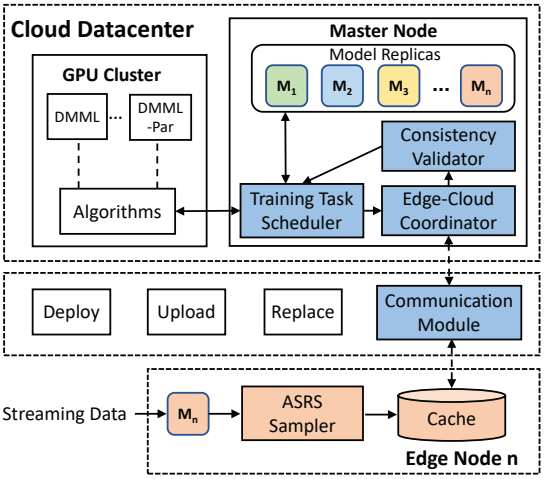


Fig. 6: The High-level Implementation of DeepCon

4.1 Over-the-air update in DeepCon

This section provides an overview of the OTA update implementation in Fig.6, with its high-level API call in table 3.

Cloud. The cloud holds replicas of all models within the system and implements the *OTA Controller* (see Fig. 3) via multiple modules for coordinating with the GPU cluster and edge nodes, updating and deploying models. When the *Consistency Validator* detects the inconsistency of models, the *Training Task Scheduler* will be triggered and perform the model fine-tuning process over the GPU cluster. The updated models are then pushed to the corresponding edge nodes for deployment. The *Edge-Cloud Coordinator* communicates with the GPU cluster, CPU server (Master node), and edge nodes through the *Communication Module* for operations like model deployment, data uploading, model training, model validation, and model update.

Edge node. Each edge node deploys a pre-trained model to make predictions on incoming streaming data. The cloud server remains unaware of the model's performance. To gain an overall perspective of the model's performance, data must be transmitted to the cloud server for validation and

subsequent model updates. Nevertheless, data transmission could result in excessive bandwidth consumption. Thus, a sampling mechanism is essential to *precisely represent real-time data distributions while minimizing bandwidth usage*. The *ASRS Sampler* utilizes an adaptive stratified reservoir sampling method that periodically samples the data and shares them with the cloud in a pre-defined time interval. These data will be used to check the consistency of models deployed on various edge nodes. *Consistency Validator* notifies the edge nodes to upload full training data when the <Acc-cc> gap of the sampled data is too large.

I: Training Task Scheduler. The scheduler is designed to orchestrate training algorithms to update the models in the GPU cluster. We, therefore, define two main APIs: $trainModels(w, m, t)$ and $getModels(t)$. $trainModels(w, m, t, a)$ defines that in task t a set of models m are trained on w workers (i.e., GPU nodes) via algorithm a. $getModels(t)$ returns the updated models in task t.

II: ASRS Sampler & Consistency Validator. During each model-update window, i.e., every T seconds, each edge node collects input streams and samples a subset of the frames via $ASRS(images, sampleSize)$ (see §4.2). The sampled frames are sent to the *Consistency Validator*. Based on the collected samples and the newest cloud replicas, $getValid(m, r, l)$ checks the <Acc-cc> gap with Equation 5 of a set of models m based on the inference results in r and true labels l of the collected sample data.

III: Communication Module. This module interacts the Edge-Cloud Coordinator in Cloud with the edge node to provide seamless OTA solution. To this end, in Edge-Cloud Coordinator the $getDeploy(m, e)$ specifies the pre-trained model m to be deployed on the edge node e. Next, the $getReplace(m, e)$ is designed to replace the running model on the edge node e with the updated model m. To provide a "no stop" model replacement operation, the three high-level APIs i.e., $install(m)$, $migrate(m_1, m_2)$ and $stop(m)$ in edge node agent are designed. $install(m)$ builds a instance of model m, and $migrate(m_1, m_2)$ moves the input streams from model m_1 to model m_2 for inference, and $stop(m)$ is to stop and remove the instance of model m. Finally, the $getUpload(d, r)$ uploads the sampled data stream d and their corresponding inference results r to the cloud for further validation. In this paper, we assume that both cloud and edge nodes have the right to access the raw data, and privacy preserving will be considered in future work.

4.2 Adaptive stratified reservoir sampling

Reservoir sampling. Reservoir sampling draws data from a stream and keeps a sample in a storage area termed the reservoir [26]. Initially, this method fills the reservoir with the stream's first N items. Once past those initial N items, for every subsequent i-th item (where i > N), each of the reservoir's N items has a $\frac{1}{i}$ chance of being substituted with the i-th item. This means the i-th item has an $\frac{N}{i}$ likelihood of being accepted and then randomly replaces an item already in the reservoir. The beauty of this approach is that it doesn't require knowing the stream's total item count, and it guarantees every item in the stream has an equal chance of making it into the reservoir.

Stratified sampling. While reservoir sampling is a prevalent method in stream processing, it might compromise the statistical integrity of the sampled data when the input stream has multiple sub-streams with varying distributions [27]. This stems from the fact that reservoir sampling could potentially miss out on sub-streams that consist of only a few data items. Specifically, it doesn't ensure that every sub-stream is given an equitable chance for its data items to be chosen for the sample. To address this, stratified sampling was introduced. This method first divides the input data stream into distinct sub-streams and then conducts sampling, such as simple random sampling, on each sub-stream separately. This ensures each sub-stream's data items have an equal chance of selection, ensuring none are missed. However, a limitation of stratified sampling is that it's effective only when it has prior knowledge of the statistics of all sub-streams, like the length of each one.

Adaptive stratified reservoir sampling. To reduce the bandwidth cost while transmitting data from edges to the cloud for consistency validation, we develop a novel sampling method, i.e., Adaptive Stratified Reservoir Sampling (ASRS). It accomplishes the benefits of both stratified and reservoir sampling without inheriting their limitations. Notably, ASRS ensures no sub-streams are missed regardless of how popular they are, requires no prior knowledge of the sub-stream statistics for sampling, and operates efficiently in real-time across a distributed system [28], [29].

Algorithm 3 summarizes the workflow of applying ASRS to sample the streaming data (e.g., images or texts) in an edge node. We stratify the input stream into sub-streams according to the inference results of the model deployed on the edge node (line 9). Thus, each sub-stream is sampled independently (see line 8-12). To this end, we determine the sample size N_i of each substream based on the previous stream interval's input rate and then perform without replacing reservoir sampling to select the data items without exceeding its sample size N_i.

5 Evaluation

5.1 Experiment Setup

Datasets. CIFAR-10 and CIFAR-100 [30] datasets contain 10 and 100 classes respectively, with 50,000 images and 10,000 test images, each 32 x 32 colored pixels. IMDB [31] is a binary text classification task with 25,000 data samples for both the train and test sets.

Models. For the image classification task, we use three network architectures and vary their architecture depth when generating models: ResNet-20, ResNet-56 [32] and MobileNetv2_x0_5, MobileNetv2_x1_4 [33] and VGG13 [34]. For text classification task, 5 different structures are implemented for the experiments: TextRNN, BiLSTM, TextCNN, TextRCNN and Self-Attention. All models use pre-trained glove embeddings for feature representation.

Cluster and Training Setup. We implement all neural networks and training processes with Pytorch and conduct experiments on an Ubuntu 16.04 server with 5 Nvidia Tesla V100 GPUs, 40 Intel Gold 5218 CPUs, and 100GB memory.

All pre-trained models are generated by ourselves serving as base models for further fine-tuning. For pre-training CIFAR models, we use an SGD optimizer with an initial

TABLE 3: DEEPCON APIs

Cloud APIs	Description
$trainModels(w, m, t, a)$	In task t a set of models m are trained on w workers via algorithm a.
$getModels(t)$	Returns the updated models in task t.
$getValid(m, r, l)$	Returns the CC of a set of models m based on the inference results r and true labels l of the collected sample data.
$getDeploy(m, e)$	Specifies the pre-trained model m to be deployed on edge node e.
$getReplace(m, e)$	Replace the running model on the edge node e with the updated model m.
Edge node APIs	**Description**
$install(m)$	Builds a instance of model m.
$migrate(m_1, m_2)$	Moves the input streams from model m_1 to model m_2 for inference.
$stop(m)$	Stop the instance of model m.
$getUpload(d, r)$	Upload the sampled data stream d and their corresponding inference results r to the cloud.
$ASRS(images, sampleSize)$	get $sampleSize$ number of samples from the collected $images$.

Algorithm 3: Adaptive Stratified Reservoir Sampl.

Input: images, sampleSize
Output: sample
Function ASRS $(images, sampleSize)$**:**
  sample ← [] // Set of items sampled within the time interval
  Sub ← [] // Set of sub-streams seen so far within the time interval
  N_i ← getSampleSize(sampleSize, Sub_i) // Determine the sample size for each substream
  for $Sub_i \in Sub$ **do**
    $C_i \leftarrow 0$ // Initial counter to measure items in each sub-stream
    // arriving items in each interval
    for $item_j \in Item$ **do**
      $item_j^i \leftarrow getSubstream(item_i)$ // inference and determine belonging to which sub-stream
      $sample_i \leftarrow RS(item_j^i, N_i)$ // Reservoir sampling
      sample ← $sample_i$ // add the selected sample to global sample
    end
    return $sample$
  end
return

learning rate of 0.1. The training lasts 100 epochs, and we decrease the learning rate to 0.01, 0.001, and 0.0001 at 30, 60, and 90 epoches. For pre-training on IMDB, ADAM optimizer is used and the learning rate is set to 0.0001 for 200 epochs of training.

During re-training, we fine-tune the model learning rate, mixup ratio, and the training epochs to get the best performance, i.e., we perform training on CIFAR for 50 epochs, set the learning rate to 0.01, 0.001, 0.0001 at 0, 20, and 40 epochs. We perform model re-training on IMDB with a fixed learning rate of 0.0001, ADAM optimizer for 100 epochs.

We perform data pre-processing and augmentation for vision and language dataset: For vision CIFAR dataset, we augment the data set with random crops of 4 padding sizes and horizontal flips. For language IMDB dataset, we perform data augmentation by randomly replacing words with default tokens. We use Glove embeddings [1] with 100 dimensions for converting texts into word embeddings. The input of the Glove embedding is the word token while the output is the vector representations. It works as a dictionary for converting words into vector representations. We fix word embedding during all experiments.

Note that for pre-training we do not aim to reach the best-benchmarked accuracy for respective model architectures and datasets. For all our settings, the pre-trained models have converged to a stable point and we show the benefits of different techniques for improving the CC and reducing the <Acc-CC> gap.

Baselines. We compare our method against the following baselines. 1) *Vanilla-KD* [11] assume all students learn from the teachers and true labels with distillation loss and cross-entropy loss. KD is widely studied for training small networks that mimics the behaviors of large networks. 2) *Label Smooth* [35], [8], [9] regulates the true label and is capable of reducing model churn during training. We extend KD with label smoothing for comparison. 3) *Mixup Distillation* [8] proposes distillation based training for reducing model churn. We extend the method to multiple models where all models but the best-performed model learn from the best models. 4) *Ensemble Distillation*: Ensemble methods are not directly applicable to models with different architectures. We implement ensemble distill where each model learns from the ground truth label and the average ensemble logits from all models in the group. 5) *KDCL* [36] is similar to ensemble distillation but implements random augmentation to different models during model training. The training loss consists of both entropy loss with true label and distillation loss with ensemble digits. 6) *Deep Mutual Learning* [37] is an online-KD method that jointly learns multiple models at the same time. The loss consists of entropy loss with true label and pair-wise distillation loss against all other models. 7) *Co-Distillation* [38] is another online-KD method that enables parallel training of multiple models. The training loss includes entropy loss and distillation loss against average logits over all other models. For a fair comparison, all baseline results and DMML use the same optimizer and training epochs. We vary the weighting parameters and report results with the best *Correct Consistency* (CC) metric.

Metrics. The evaluation metrics used in the paper include: 1) Correct Consistency (E.q 3 CC): the percentage of all models to all produce the positive results given the same input. 2) Performance gap between ACC and CC: <Acc-CC> = $\min\{Acc(M_1), ..., Acc(M_N)\} - CC(M_1...M_N)$ (E.q 5).

All experiments are repeated 5 times with mean value, and standard deviation reported in the results. We report results that maintain or increase individual model accuracy, making sure no accuracy loss during training.

1. https://nlp.stanford.edu/projects/glove/

	CIFAR10		CIFAR100		IMDB	
	CC ↑	<Acc-CC> ↓	CC ↑	<Acc-CC> ↓	CC ↑	<Acc-CC> ↓
	ResNet20 + VGG13		ResNet20 + VGG13		TextRCNN + SelfAtten	
Pretrained	89.15	2.58	59.74	7.12	62.76	11.89
vanilla kd	89.34±0.05	2.69±0.07	60.02±0.07	7.36±0.18	66.58±0.43	7.79±0.38
ls	89.45±0.06	2.63±0.06	59.79±0.02	7.35±0.08	67.12±0.23	7.53±0.13
mixup	89.48±0.18	2.64±0.13	60.31±0.11	7.19±0.13	67.21±0.32	7.44±0.41
ensemble-distill	90.00±0.21	2.21±0.16	61.81±0.07	5.69±0.15	67.44±0.18	6.66±0.26
kdcl	89.87±0.14	2.21±0.12	61.84±0.11	5.95±0.05	66.89±0.22	6.74±0.17
dml	90.44±0.12	1.88±0.12	62.59±0.24	5.06±0.23	67.35±1.15	6.29±0.23
co-distill	90.40±0.05	1.96±0.09	**62.91±0.07**	5.01±0.18	67.30±0.74	6.25±0.45
dmml	**90.74±0.07**	**1.7±0.02**	62.83±0.06	**4.57±0.18**	**69.75±0.13**	**5.14±0.33**
	5 Models		5 Models		5 Models	
Pretrained	84.27	8.34±0.78	50.35	19.69±2.13	46.86	26.61±1.97
vanilla kd	85.00±0.10	7.85±0.61	50.82±0.15	19.4±1.90	52.67±0.10	21.88±1.16
ls	85.29±0.09	7.64±0.62	50.77±0.05	19.57±2.09	53.01±0.26	21.62±1.10
Mixup	85.61±0.11	7.36±0.63	52.35±0.20	18.32±2.08	53.3±0.03	21.32±1.09
ensemble-distill	86.85±0.17	6.39±0.73	55.22±0.20	16.45±2.27	51.41±2.61	21.96±0.79
kdcl	86.51±0.20	6.66±0.70	54.34±0.15	17.26±2.25	51.05±3.21	22.18±0.91
dml	87.50±0.17	5.80±0.67	56.11±0.18	15.74±2.20	57.21±0.24	18.2±0.66
co-distill	87.23±0.10	6.05±0.72	55.64±0.09	16.15±2.29	57.21±0.07	18.31±0.63
dmml	**88.17±0.07**	**4.98±0.52**	**57.38±0.07**	**13.93±2.11**	**59.45±0.19**	**16.31±0.59**

TABLE 4: Correct Consistency (%) and <Acc-CC> Gap on the CIFAR10/100 and IMDB Dataset with 2 and 5 Models.

5.2 DMML Performance on Vision and Language Tasks

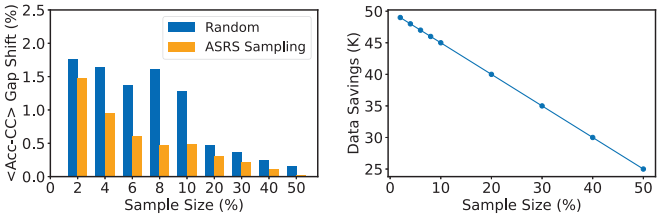
Table 4 compares the performance of DMML to other baselines on three popular datasets: CIFAR10, CIFAR100 and IMDB. We conduct 2 sets of experiments with 2 models and 5 models respectively.

Generally, in all experiments, the implemented baselines and DMML are capable of improving the consistency CC among models, indicating that cross-model learning is effective in generating more similar models, regardless of the model architectures and parameters. Online KD methods (e.g., dml, codistill, dmml, kdcl and ensembledistill) are more effective as compared to offline-KD (e.g., vanilla kd, label smooth, mixup) in overall performance.

In experiments with 2 models, DMML is able to reduce about 34.1%, 35.8%, and 56.8% of <Acc-CC> for 3 datasets respectively. DMML achieves higher CC than baseline solutions in most cases. However, DMML reports slightly worse CC than codistillation on ResNet20 + VGG13, CIFAR100 experiment. This is because the codistillation improves all models' Acc thereby resulting the better CC. However, this method only works in rare cases and very randomly.

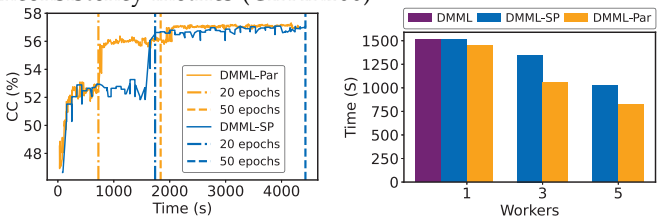
On the IMDB dataset (NLP), we observe a higher inconsistency in pre-trained TextRCNN and SelfAttention models due to the bigger difference between their model architectures. After applying our methods, the CC can be greatly improved, meaning that DMML is more effective for models of greater difference. KDCL [36] as reported in the paper can improve model invariance by adding image distortion for each model but is less effective in improving model consistency. In the following, we conduct experiments with 5 models of different architectures to show the scalability of DMML in improving model consistency.

Comparing the experiments of 5 models and 2 models, we see a larger difference of <Acc-CC>: 8.34%, 19.69%, and 26.61% for the three datasets. After applying DMML, the <Acc-CC> has reduced to 4.98%, 13.93%, and 16.31%, indicating 40.28%, 29.25% and 38.70% reduction to this gap. Comparing the computer vision tasks and NLP tasks, we also see a larger gap in the latter, despite that all models use the same glove embeddings at the first layer. In the future, we will investigate the performance of DMML for other basic language tasks, i.e., semantic parsing [39].

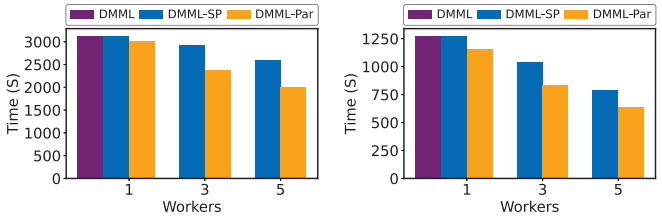


(a) <Acc-CC> Gap Shift Comparison (b) Data Transmission Savings with Various Sample Size

Fig. 7: Evaluation of sampling methods in revealing system inconsistency metrics (CIFAR100)



(a) Convergence Curve on CIFAR100 validation set, 5 GPU workers (b) CIFAR10



(c) CIFAR100 (d) IMDB

Fig. 8: Evaluation of DMML-Par with 5 models on 3 datasets

5.3 Performance of ASRS Sampling method

In this subsection, we validate the effectiveness of the proposed ASRS sampling methods in precisely revealing the performance gap during application run-time. We sample various proportions of data from the CIFAR100 dataset and test the <Acc-CC> gap with the sampled data. Then we show the performance gap shift as compared to inference

with the whole dataset. For comparison purposes, we establish the baseline with random sampling methods. Both results are shown in Figure 7(a).

We can see from the figure that when sampling 2% of data from the dataset, the performance shift is around 1.45%. When more data is sampled, the shift is generally becoming less, with 50% of the data revealing barely any shift. When we shift 6% - 10% percentage data from the dataset, the performance shift is small enough, around 0.5%, which shows the effectiveness of our method in precisely revealing the real data distributions during the application running process. As a comparison, the random sampling methods barely show any performance improvement when less than 10% of the data is sampled from the whole dataset, indicating a gap shift of around 1.5%. When sampling 4% to 10% data samples, the random sampling methods show 1.6X, 2.8X, 4X, and 3.2X performance shift than ASRS sampling method. Thus, the stratified sampling in our ASRS sampling is capable of selecting the most valuable data samples that reveal the true data distributions from the real data, capable of fast detecting the system performance variance. As Figure 7(b), when making predictions for data, the ASRS sampling can substantially save transmission bandwidth when preparing data for consistency check. When making predictions for stream data of around 50,000 samples (CIFAR100), 45,000 samples are excluded to transmit to the cloud when we sample 10% of the data during the application run-time. This is especially valuable when the models are distributed at different locations with limited bandwidth resources.

5.4 Performance of DMML-Par

Fig. 8 plots the performance of DMML-Par implemented in our DEEPCON, trained on 1, 3, and 5 workers (GPU nodes) with three datasets. As a comparison, we extend DMML to a simple synchronous parallel training algorithm namely DMML-SP. We iteratively allocate model training tasks to available workers and wait for all the models to be trained once. Then we update the whole server model replica and push them to the next round of training. We also report the performance of DMML as a baseline. All training setup and parameter setting are the same as the DMML, including the Mixup ratio, the learning rate, and optimizers.

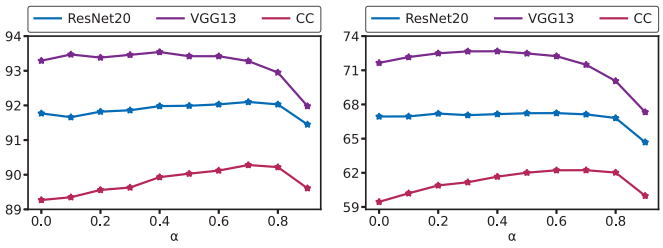
Fig. 8(a) shows the convergence curve of DMML-SP and DMML-Par with 5 models on the CIFAR100 validation set. Generally, we see that the training speed of DMML-Par is much faster than DMML-SP. It takes 725 and 1838 seconds for DMML-Par to complete 20 and 50 rounds of model training, nearly 60% reduction as compared to DMML-SP (i.e., 1739 and 4427 seconds respectively.) As far as the CC metric, DMML-Par also reports faster and stable growth as compared to the DMML-SP. After finishing 50 rounds of training, the highest CC reported is 56.45, slightly worse than the best CC (57.38) reported in Tab. 4. However, when we continue training for a few rounds, we can still guarantee the best CC as compared to synchronous DMML-SP.

Fig. 8(b), 8(c), and 8(d) report latency of three algorithms. We report the minimum time it takes for each algorithm to reach the best CC as marked by Tab. 4. Overall, on three datasets, DMML-Par can reduce 20%, 14%, and 20% training time compared to DMML-SP and DMML on average, while reaching the same CC. In reality, we can opt to sacrifice few CC for much less training time, e.g., in Fig. 8(a), DMML-Par already achieves 56.12% CC (1% less than best CC) when training for only 1100s, much faster than 1950s when reaching the best CC as reported in Fig. 8(c).

Also, we can observe that the training time of both DMML-SP and DMML-Par decreases with the increase in the number of workers. However, DMML-SP has to wait for all models to finish their round of training and then update the models, which may cause some nodes to be idle.

5.5 The impact of parameter α

We analyze how the weight parameter α affects the model accuracy and correct consistency (CC) metric. In Fig. 9 we report the hyper-parameter tuning results of α on the two datasets, with varying the α from 0 to 0.9.



(a) CIFAR-10 (b) CIFAR-100

Fig. 9: Metrics (%) with α, Resnet20 + VGG13

As α increases (more model output in the label mixup), the CC metric is getting better, while the accuracy remains stable, and slightly outperforms the accuracy of the pre-trained models respectively. The maximum point is reached around 0.7/0.8 and then starts to drop when further increasing α. This indicates that around 20% of information from the true label is already a strong indicator, enough to keep the model stable from the accuracy perspective.

6 RELATED WORK

Model Consistency. Model consistency measures the ability of models to produce identical outputs when given the same/similar inputs. It is different from accuracy in that the latter measures the prediction of each model, without considering the relationship between different models.

Model consistency has been studied in several related areas by different terminologies. [5], [6] attempt to quantify the disagreement level between two networks trained on the same dataset. They reveal that model disagreement could arise from different architectures and initializations, training samples, and optimizers. [3], [4] study the reproducibility problem during model re-training and conclude that factors like activation functions or data order could lead to drastic prediction differences between the fine-tuned model and the base model. [40], [41] report model instability which measures the output variation of a given model when slight perturbation is added to the input sources. A stability loss is added to the training loss to mitigate the model instability. *Prediction Churn* [8], [7], [9] propose churn, another definition to measure model consistency during the training process. By using techniques such as mixup label [8], adaptive label smoothing [9], models at different phases are forced to produce same results during the training process. [42] is most related to the definition in our paper, which apply ensemble-based techniques during model training to improve model consistency. However, the

ensemble technique is limited in applicability when models are different in sizes or architectures.

In summary, model inconsistency is ubiquitous during both training and deployment. While some have aimed to improve model consistency, but are limited to the model training phase with the same model architecture. There is no study on the severity of model consistency with the growth of system complexity, i.e., the growing model size, and different model architectures. Also, a general method is required to guarantee consistency between the models in an ML system that needs consistency guarantee.

Model Distillation. knowledge distillation [43], [11] trains a shallow network that mimics the behavior of a deep network. The training process leverages knowledge from both the ground true label and the teacher network. Knowledge representation from the teacher model could be from the model output at the final layer [11] or feature maps from middle layers [44], with loss computed by KL divergence loss, or MSE loss. When there is a lack of pre-trained teacher models, however, online knowledge distillation [37] enable simultaneous training of both the teacher and student models at the same time. In Deep Mutual Learning (DML) [37], the model update is implemented via averaged pair-wise distillation loss for the current model against all other models. Co-distillation [36] is similar to DML and is directly applicable to training large-cohort models in a collaborative way. It is also studied that co-distillation in effective in reducing model inconsistency. Many other works extend the training paradigm of online KD by by using ensembles of model outputs and the true label [36], adding diverse peers [45] or multi-branch architecture [46].

We compare DMML with many of the above baselines on online-KD and techniques on improving model consistency. We show that DMML is easy to implement, effective in improving model consistency and generic on different data sources, i.e., vision and language datsets.

7 Conclusion

In this paper, we propose DeepCon, an adaptive deployment framework for quickly detecting and reducing the model inconsistency via over-the-air parallel training. We design a whole pipeline for quickly detecting the inconsistency within the systems, and propose an efficient learning algorithm (DMML) based on knowledge distillation for improving the consistency between the models. In order to further accelerate the training process, we implement DMML-Par, asynchronous parallel training of DMML, a high-scalable algorithm that is easily adapted to various numbers of computation resources. We prototype DeepCon and implement a set of APIs for seamless communication between the edge and cloud layers. The evaluation results show the effectiveness of DMML in improving model consistency. We also evaluate the training speed up of DMML-Par, which can guarantee the best consistency improvement while greatly reducing the training time.

In the future, we plan to explore cross-domain applications by evaluating how our inconsistency measurement performs across tasks like human activity recognition, and person re-identification, providing valuable insights into its adaptability and effectiveness. Additionally, exploring the impact of contextual factors on model inconsistency will be essential: we plan to analyze how different input conditions or environments affect predictions, and develop strategies to mitigate inconsistency in real-world applications.

Acknowledgment

This work was supported by the National Key Research and Development Program of China (2022YFE0196000), National Nature Science Foundation of China under Grant 62472387, Key Research and Development Program of Zhejiang Province (Grant No: 2025C01064).

References

[1] B. Qian, J. Su, Z. Wen, D. N. Jha, Y. Li, Y. Guan, D. Puthal, P. James *et al.*, "Orchestrating the development lifecycle of machine learning-based iot applications: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–47, 2020.

[2] B. Qian, Z. Wen, J. Tang, Y. Yuan, A. Y. Zomaya, and R. Ranjan, "Osmoticgate: Adaptive edge-based real-time video analytics for the internet of things," *IEEE Transactions on Computers*, 2022.

[3] G. I. Shamir, D. Lin, and L. Coviello, "Smooth activations and reproducibility in deep networks," *arXiv:2010.09931*, 2020.

[4] R. R. Snapp and G. I. Shamir, "Synthesizing irreproducibility in deep networks," *arXiv preprint arXiv:2102.10696*, 2021.

[5] P. Nakkiran and Y. Bansal, "Distributional generalization: A new kind of generalization," *arXiv preprint arXiv:2009.08092*, 2020.

[6] Y. Jiang, V. Nagarajan, C. Baek, and J. Z. Kolter, "Assessing generalization of sgd via disagreement," in *ICLR*, 2021.

[7] S. Bhojanapalli, K. Wilber, A. Veit, A. S. Rawat, S. Kim, A. Menon, and S. Kumar, "On the reproducibility of neural network predictions," *arXiv preprint arXiv:2102.03349*, 2021.

[8] H. Jiang, H. Narasimhan, D. Bahri, A. Cotter, and A. Rostamizadeh, "Churn reduction via distillation," in *ICLR*, 2021.

[9] D. Bahri and H. Jiang, "Locally adaptive label smoothing for predictive churn," *arXiv preprint:2102.05140*, 2021.

[10] D. Yu, Z. Xie, Y. Yuan, S. Chen, J. Qiao *et al.*, "Trustworthy decentralized collaborative learning for edge intelligence: A survey," *High-Confidence Computing*, p. 100150, 2023.

[11] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[12] S. Han, X. Liu, X. Han, G. Wang, and S. Wu, "Visual sorting of express parcels based on multi-task deep learning," *Sensors*, vol. 20, no. 23, p. 6785, 2020.

[13] W. Xiao, J. Yang, H. Fang, J. Zhuang, Y. Ku, and X. Zhang, "Development of an automatic sorting robot for construction and demolition waste," *Clean Technologies and Environmental Policy*, vol. 22, pp. 1829–1841, 2020.

[14] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[15] I. Loshchilov and F. Hutter, "Online batch selection for faster training of neural networks," *arXiv preprint arXiv:1511.06343*, 2015.

[16] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of big data*, vol. 6, no. 1, pp. 1–48, 2019.

[17] Y. Wu, P. Tian, Y. Cao, L. Ge, and W. Yu, "Edge computing-based mobile object tracking in internet of things," *High-Confidence Computing*, vol. 2, no. 1, p. 100045, 2022.

[18] G. Montavon, G. Orr, and K.-R. Müller, *Neural networks: tricks of the trade*. springer, 2012, vol. 7700.

[19] J. A. Stuart and J. D. Owens, "Multi-gpu mapreduce on gpu clusters," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 1068–1079.

[20] H. Mostafaei, M. U. Chowdhury, and M. S. Obaidat, "Border surveillance with wsn systems in a distributed manner," *IEEE Systems Journal*, vol. 12, no. 4, pp. 3703–3712, 2018.

[21] P. K. Varshney, *Distributed detection and data fusion*. Springer Science & Business Media, 2012.

[22] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski *et al.*, "Scaling distributed machine learning with the parameter server," in *(OSDI 14)*, 2014, pp. 583–598.

[23] S. Dörner, S. Cammerer, J. Hoydis, and S. Ten Brink, "Deep learning based communication over the air," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 132–143, 2017.

[24] K. Yang, T. Jiang, Y. Shi, and Z. Ding, "Federated learning via over-the-air computation," *IEEE transactions on wireless communications*, vol. 19, no. 3, pp. 2022–2035, 2020.

[25] Y. Yuan, S. Chen, D. Yu, Z. Zhao, Y. Zou, L. Cui, and X. Cheng, "Distributed learning for large-scale models at edge with privacy protection," *IEEE Transactions on Computers*, 2024.

[26] M. Al-Kateb, B. S. Lee, and X. S. Wang, "Adaptive-size reservoir sampling over data streams," in *19th International Conference on Scientific and Statistical Database Management (SSDBM 2007)*. IEEE, 2007, pp. 22–22.

[27] M. Al-Kateb and B. S. Lee, "Stratified reservoir sampling over heterogeneous data streams," in *International Conference on Scientific and Statistical Database Management*. Springer, 2010, pp. 621–639.

[28] ——, "Adaptive stratified reservoir sampling over heterogeneous data streams," *Information Systems*, vol. 39, pp. 199–216, 2014.

[29] Z. Wen, D. L. Quoc, P. Bhatotia, R. Chen, and M. Lee, "Approxiot: Approximate analytics for edge computing," 2018.

[30] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 cifar-100," *(Canadian Institute for Advanced Research)*, 2009. [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[31] A. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, 2011, pp. 142–150.

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[33] A. G. Howard, M. Zhu, B. Chen *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv:1704.04861*, 2017.

[34] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[35] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[36] Q. Guo, X. Wang, Y. Wu, Z. Yu, D. Liang, X. Hu, and P. Luo, "Online knowledge distillation via collaborative learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 11 020–11 029.

[37] Y. Zhang, T. Xiang, T. M. Hospedales, and H. Lu, "Deep mutual learning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4320–4328.

[38] R. Anil, G. Pereyra, A. Passos, R. Ormandi, G. E. Dahl, and G. E. Hinton, "Large scale distributed neural network training through online distillation," in *International Conference on Learning Representations*, 2018.

[39] C. Hidey, F. Liu, and R. Goel, "Reducing model jitter: Stable re-training of semantic parsers in production environments," *arXiv preprint arXiv:2204.04735*, 2022.

[40] S. Zheng, Y. Song, T. Leung, and I. Goodfellow, "Improving the robustness of deep neural networks via stability training," in *Proceedings of the ieee conference on computer vision and pattern recognition*, 2016, pp. 4480–4488.

[41] E. Cidon, E. Pergament, Z. Asgar, A. Cidon, and S. Katti, "Characterizing and taming model instability across edge devices," *Proceedings of Machine Learning and Systems*, vol. 3, 2021.

[42] L. Wang, D. Ghosh *et al.*, "Wisdom of the ensemble: Improving consistency of deep learning models," *arXiv:2011.06796*, 2020.

[43] J. Ba and R. Caruana, "Do deep nets really need to be deep?" *Advances in Neural Information Processing Systems*, vol. 27, 2014.

[44] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "Fitnets: Hints for thin deep nets," *arXiv preprint arXiv:1412.6550*, 2014.

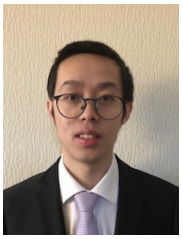
[45] D. Chen, J.-P. Mei, C. Wang, Y. Feng, and C. Chen, "Online knowledge distillation with diverse peers," in *AAAI*, vol. 34, no. 04, 2020, pp. 3430–3437.

[46] X. Lan, X. Zhu, and S. Gong, "Knowledge distillation by on-the-fly native ensemble," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 2018, pp. 7528–7538.

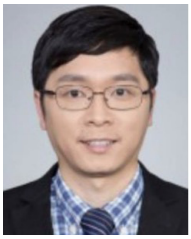
Bin Qian (Member, IEEE) received the Ph.D. degree in computer science from Newcastle University, Newcastle Upon Tyne, U.K, in 2024; M.Sc. degree in Data Science from University of Southampton, U.K, in 2018. He is currently a postdoc researcher with State Key Laboratory of Industrial Control Technology, Zhejiang University, Hangzhou, China. His research interests include IoT, deep learning.

Jiaxu Qian received the B.E. degree from Hangzhou Dianzi University, China, in 2021. He is currently pursuing the M.Sc. degree in information engineering from Zhejiang University of Technology, China. His research interests include distributed computing, machine learning, and graph nerual network.

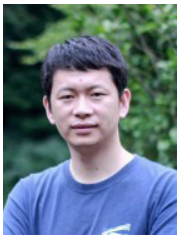
Zhenyu Wen (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in computer science from Newcastle University, Newcastle Upon Tyne, U.K., in 2011 and 2016, respectively. He is currently a Professor with the Institute of Cyberspace Security and college of Information Engineering, Zhejiang University of Technology, China. His current research interests include IoT, crowd sources, AI system, and cloud computing.



Di Wu (Member, IEEE) received the PhD degree in computer science at University of St Andrews, UK. He received a B.S. degree in Information System and Information Management from Northeast Forestry University, China in 2015, and an M.S. degree in Data Science from University of Southampton, UK in 2018. His major interests are in the areas of federated learning, distributed machine learning, edge computing, model compression, and Internet-of-Things.



Shibo He (Senior Member, IEEE) received the Ph.D. degree in control science and engineering from Zhejiang University, Hangzhou, China, in 2012. He is a Professor with Zhejiang University. He was an Associate Research Scientist from March 2014 to May 2014, and a Postdoctoral Scholar from May 2012 to February 2014, with Arizona State University, Tempe, AZ, USA. From November 2010 to November 2011, he was a Visiting Scholar with the University of Waterloo, Waterloo, ON, Canada. His research interests include Internet of Things, crowdsensing, big data analysis, etc.



Jiming Chen (Fellow, IEEE) received the Ph.D. degree in control science and engineering from the Zhejiang University, Hangzhou, China, in 2005. He is currently a Professor at the Department of Control Science and Engineering, at Zhejiang University. His research interests include IoT, networked control, and wireless networks. He serves on the editorial boards of multiple IEEE Transactions. He is an IEEE VTS Distinguished Lecturer and a fellow of CAA.

Rajiv Ranjan (Fellow, IEEE) is a Full professor in Computing Science at Newcastle University, United Kingdom. He was Julius Fellow (2013-2015), Senior Research Scientist and Project Leader in the Digital Productivity and Services Flagship of Commonwealth Scientific and Industrial Research Organization. He was a Senior Research Associate in the School of Computer Science and Engineering, University of New South Wales (UNSW). Prof. Ranjan has a PhD (2009) from the department of Computer Science and Software Engineering, the University of Melbourne.