CLUE: Systems Support for Knowledge Transfer in Collaborative Learning With Neural Nets

Harshit Daga, Yiwen Chen, Aastha Agrawal, and Ada Gavrilovska

***Abstract*—For highly distributed environments such as edge computing, collaborative learning approaches eschew the dependence on a global, shared model, in favor of models tailored for each location. Creating tailored models for individual learning contexts reduces the amount of data transfer, while collaboration among peers provides acceptable model performance. Collaboration assumes, however, the availability of knowledge transfer mechanisms, which are not trivial for deep learning models where knowledge isn't easily attributed to precise model slices. We present CLUE – a framework that facilitates knowledge transfer for neural networks. CLUE provides new system support for dynamically extracting significant parameters from a helper node's neural network, and uses this with a multi-model boosting-based approach to improve the predictive performance of the target node. The evaluation of CLUE with different PyTorch and TensorFlow neural network models demonstrates that its knowledge transfer mechanism improves by up to 3.5× how quickly a model adapts to changes, compared to learning in isolation, while affording up to several magnitudes reduction in data movement costs compared to federated learning.**

***Index Terms*—Deep learning, edge computing, edge inference, intelligent edge.**

I. Introduction

TO MANAGE, process and analyze the increasing amount of data generated at the edges of the network, companies have started relying on machine learning techniques to provide automation and to improve customer experience and service delivery. With data being generated in a highly distributed manner, companies build machine learning (ML) models using systems for distributed learning such as Federated Learning (FL) [1]. FL creates a global and generic model through periodic communication among distributed locations using a *push* based approach to transfer raw data or model updates.However, highly distributed environments, particularly such as those observed across nodes in mobile edge computing (MEC) [2], have been shown to benefit from models tailored to the context of specific locations [3], [4], [5]. In addition, distinct locations may not exhibit any change in their localized inputs, thus not benefiting from model updates due to churn in data trends observed at other locations. Abandoning any cross-node coordination in learning in favor of learning in isolation, eliminates data transfer costs. However, if data trends shift across locations (i.e., due to concept or data drift), models learned in isolation suffer from loss of accuracy and need time to adapt to changes, compared to when relying on a global model. In response, distributed machine learning models for *collaborative learning* across edge nodes have been proposed, as a way to create robust tailored models.

Collaborative learning (CL) allows distributed nodes to adapt quickly to input shifts and to retain model accuracy, with low data transfer costs [6], [7]. A key assumption in realizing collaborative learning is the ability to transfer knowledge from one peer who has previously learned a given input class – *a helper node* – to another peer exhibiting a change in its inputs – *a target node*. This approach is particularly suitable for geographically distributed enterprises and services, where different edge sites operate on behalf of a single organization and trust each other to collaboratively exchange model information with one another.

For certain types of machine learning models, such as the online random forest (ORF) and online support vector machines (OSVM) used in [6], realizing knowledge transfer is straightforward, as the association of input classes to model parameters is easily identifiable: to transfer knowledge about a given class from a helper to a target model, the corresponding subtrees in ORF or matrix rows in OSVM models need to be copied over across models. In practice, however, services are increasingly turning to deep learning techniques, where knowledge about a particular input class cannot be trivially attributed to a slice of the model. This raises the question, *how to enable knowledge transfer across nodes that use neural network (NN) models?* Without the support to extract features from a helper node's model relevant for the knowledge needed at the target, and to then integrate them with the target node's model, collaborative learning will not be feasible for deep learning techniques. This will limit the opportunity for the benefits of collaborative distributed learning to be afforded to important application classes relying on deep learning.

In response, we present **CLUE** – a framework that provides systems support for *collaborative deep learning* by facilitating selective knowledge transfer across neural network models. CLUE combines system support for dynamic creation of helper models from an existing NN model, with a multi-model boosting-based approach to use this knowledge and update a target model. The knowledge transfer in CLUE is made possible

Manuscript received 3 November 2022; revised 20 May 2023; accepted 24 June 2023. Date of publication 12 July 2023; date of current version 6 December 2023. This work was supported in part by industry Grants from Cisco and VMware Research, in part by the National Science Foundation under Grants PPoSS-2217070 and CNS-1909769, and in part by using research infrastructure provided by the NSF Chameleon Cloud. Recommended for acceptance by L. Cui. *(Corresponding author: Harshit Daga.)*

The authors are with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: harshitdaga@gatech.edu; yiwen.chen@gatech.edu; aagrawal319@gatech.edu; ada@cc.gatech.edu).

Digital Object Identifier 10.1109/TCC.2023.3294490

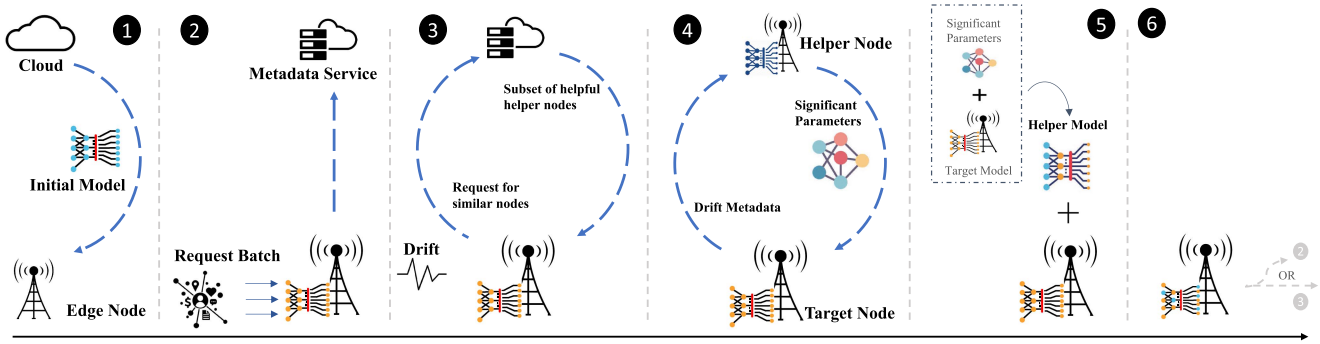


Fig. 1. Collaborative learning in distributed edge computing.

through the use of mechanisms for (1) on-demand gleaning of *significant parameters* of a model, (2) use of *boosted helper models*, and (3) support for *managing a pool of helper models*. The effectiveness of these mechanisms is evaluated for several NN models based on PyTorch and TensorFlow, and the results shows opportunities for significant improvements in model performance and data transfer costs. The evaluation also demonstrates that CLUE exposes new opportunities to exploit tradeoffs among model accuracy, the ability to quickly adapt to changes in a model's predictive performance, data transfer costs and learning overheads. Future policy engines can exploit such tradeoffs to further tailor the behavior of the distributed learning process.

In summary, the contributions of this paper include:

- New mechanisms to facilitate knowledge transfer for deep learning models (Section V);
- The design and implementation of CLUE , the first framework that integrates these mechanisms to provide support for collaborative deep learning with neural nets (Section VI);
- Evaluation with several models, ML frameworks, and workloads, which demonstrates that CLUE enables collaborative deep learning across distributed nodes in a manner that allows models to adapt to change $3.5\times$ faster than isolated learning while giving a similar predictive performance compared to federated learning, but with orders of magnitude lower overall data transfer costs (Section VII).

II. Background and Motivation

Multi-Access Edge Computing: MEC provides compute capabilities at the edge of the network, in close proximity to client devices, such as on wireless gateways, cellular base stations, enterprise on-prem edge server infrastructure, etc. [8]. It offers infrastructure for execution of applications requiring low latency (e.g., AR/VR, automation), reduction in backhaul data demand (e.g., IoT analytics, content delivery), and geo-localized data handling (e.g., for GDPR). These base stations are commonly equipped with servers that typically possess multi-core processors, ample memory capacity, and rapid storage subsystems [9], [10]. To further enhance their performance, they may also be outfitted with hardware accelerators such as GPUs and Edge TPUs [11], as well as FPGAs (Field-Programmable Gate Arrays) that facilitate custom hardware acceleration. One challenge MEC exposes is that of creating an extremely dispersed distributed infrastructure, which differs from datacenter-based distributed environments in its scale and in the fine-grained geo-localization of the inputs observed at each node [3]. In such settings, COLLA [7], a collaborative system to predict user behavior patterns, highlights that a shared global model might not achieve the desired performance for all the users and trains smaller more tailored models. Cellscope [4] also demonstrated the ineffectiveness of using a shared global model for managing the base stations in mobile network; similar observations are confirmed in other research [5], [12].

Collaborative Distributed Learning: Fig. 1 illustrates a typical workflow in a collaborative distributed learning system, similar to what is enabled by our prior work Cartel [6]. Edge nodes use their resident model to perform inference, following which the model is re-trained locally. Upon shift in the data distribution, the model performance at the edge node deteriorates. In such a scenario, the node relies on a *pull* based (on-demand) approach of collaborative learning to find a helper node, and to leverage the helper's model (knowledge) to boost its own model. This is in contrast to FL approaches where a centralized primary/master coordinator periodically aggregates updates and pushes those updates to all other distributed trainer nodes. In CL, in order to find a helper node in a timely manner, a common metadata service (MdS) aggregates information from each node in the system. Each edge node uses drift detection algorithms [13], [14] on the local metadata information to decide when a model at the target edge node needs to be updated, and to query the MdS for candidates for a possible helper. The MdS uses the aggregated metadata to determine helper node candidates using similarity detection algorithms.

Note that, the workload characteristics shared with MdS is a histogram or trend of the characteristics observed by the nodes, which is then used to determine logical neighbor(s) which have seen similar workload patterns. This makes it possible for collaborative learning to proceed without exchange of raw data among edge nodes. We assume that collaboration is done across distributed nodes part of the same service, where the primary goal is reducing data transfer costs, and sharing information

regarding data trends at specific locations is not a major concern.For instance, it would allow edge nodes such as those present at Starbucks locations or at the PoPs of online game engines to share information within the system and one another.

To update the target node's model and improve its predictive performance, a collaborative system uses knowledge transfer [6] or distillation [7]. Knowledge distillation [15], [16] uses a (larger) teacher model to train a (smaller) student model. Knowledge transfer (KT) is a mechanism to transfer the knowledge regarding several (typically not all) classes from one machine learning model to another. In this paper, we use knowledge transfer mechanism to collaborate among edge nodes. We chose KT over distillation because in distillation the teacher model is assumed to be a complete model, whereas, we already point out, a complete global model may not be needed for MEC, and it may be too costly to maintain it at large scale in a highly distributed settings.

Federated Learning: While centralized learning produces a global model trained on diverse data, it suffers from resource inefficiency, requiring data movement to a centralized location and compromising data privacy. In contrast, federated learning (FL) has emerged as the state-of-the-art technique for distributed machine training. FL offers improved training performance, data privacy, and reduced communication and data transfer costs compared to other distributed ML approaches like centralized learning. It enables training of models on decentralized data sources without the need for data aggregation in a central server. The models are trained locally on individual devices or edge nodes, and only aggregated updates are shared. Collaborative learning shares similarities with FL as it also trains models locally without sharing raw data and aims to reduce data transfer costs. However, CL exploits edge locality to further minimize data movement, offering comparable model performance to FL. Both FL and CL are well-suited for scenarios where data cannot be easily centralized or shared due to privacy or network constraints.

Neural Networks for MEC: A growing number of companies have started to deploy deep learning services at the edge of the network for various use cases such as predictive workload analysis, intelligent caching, network security, intrusion detection and data classification [17], [18], [19]. When working with NN, even small amounts of raw data can result in large amounts of information being communicated to the aggregating node [20]. This makes support for collaborative learning for NN an important distributed learning strategy, which in turn, requires support for knowledge transfer.

Knowledge Transfer in Neural Networks: Knowledge transfer for NN is a hard problem that is receiving a lot of attention from the ML community [21], [22], [23]. In such a model information required to classify a class is spread across each layer, requiring specialized algorithms to extract it. If the data observed at target node is very similar to the one observed by the helper, fine-tuning [24] is one methodology that can be used. However, as the workload characteristics can vary across nodes, using this technique can be slow and can involve data movement across nodes. Techniques such as [25], [26] assume the helper model to be an expert teacher model, which in our case might not be always feasible. A different approach is to apply techniques that extract the portion of the model that most contribute to its performance for a given input. [21] suggests removing one neuron at a time to determine if it has impact on the model performance. [27] analyzes the activation, gradients and visualization patterns to determine a critical path. However, as the models are continuously (re)trained on the incoming data stream at each edge node, knowledge transfer needs to be performed dynamically, and over fresh models, requiring time and resource-efficient techniques.

III. Goal and Challenges

Goal: The goal of CLUE is to provide system support for collaborative distributed deep learning by enabling fast knowledge transfer across the NN models of the peer nodes.

System Model: In a collaborative system each edge node creates a customized model and requests for help when there is drift in the model performance. The concept of providing help is the ability to perform knowledge transfer from a helper node to a target node. In such a system there are few challenges that needs to be addressed which can be divided into two categories – system level and ML techniques dependent. System level challenges include determining when to initiate collaboration among the node(s) and dynamically discovering appropriate helper nodes that have observed similar workload as the target node, while performing KT across collaborative nodes depends on the ML technique. By focusing on the missing support for KT for NNs, CLUE provides an extension to our prior work on collaborative learning [6] to enable its use, and its benefits, for distributed learning with neural nets.

Challenges: The fundamental issue in performing KT across collaborative peer nodes lies in the fact that the helper node's knowledge is neither all necessary at the target node, nor is it strictly a superset of the target node's knowledge. Simply using the helper node's model introduces either unnecessary data movement (for portion of the knowledge not relevant for the inputs at the target), or worse, degrades the target node's knowledge (by displacing localized knowledge that was already acquired). This introduces the following main challenges:

C1: *How to extract a relevant portion of a neural network model?* A neural network model consists of multiple hidden layers. It takes an input, passes it through multiple layers of hidden neurons and outputs a prediction representing the combined input of all the neurons. Thus, various parameters spread across different layers help the model in providing a good predictive performance. The challenge is to select the relevant parameters from all the layers, and to do so with low overheads and with low resulting data transfer demands.

C2: *How to utilize the knowledge at the target node?* We assume the same model architecture for all the nodes in the distributed system, similar to other systems such as [28], [29], [30]. The challenge is to incorporate the acquired knowledge from the helper model into or alongside the target node's model quickly and effectively, without degrading its performance on previously known classes.

IV. Overview of CLUE

Approach: To address these challenges, CLUE adopts an approach that combines support to *dynamically extract significant model parameters* (Section V-B1), with use of *multi-model boosting* (Section V-B2). It relies on helper nodes, but only extracts the portion of their knowledge relevant for the target node. It then uses boosting to create custom helper models at the target node and combine them with the target's existing knowledge.

The multi-model approach allows CLUE to decouple the updates that need to be performed to improve a target model's prediction accuracy, from the specific parameters extracted from the helper node's original model. By using a simple boosting technique to create the helper models, CLUE makes it possible to create a helper model that benefits from the helper for recognizing new classes, while retaining its knowledge quality for pre-existing, familiar classes.

Mechanisms: Underpinning CLUE are three key mechanisms designed for *selecting significant parameters* from the helper node's model, *creating a boosted helper model* and *managing model pooling* on the target node. To solve the challenge *C1* of selectively picking the parameters from each layer of the DNN, the framework uses a mechanism for significant parameter selection, ④ in Fig. 1. The task of selecting significant parameters of a DNN model can be rephrased as follows: given a DNN f and input image x, what is the part of $f(x)$ that made the DNN reach the particular decision of classifying the input as class y. In other words, which coefficients at each layer in the model contributed towards the classification of x to an output y?

On the target node, CLUE creates helper models using the significant parameters from the original model ⑤. A helper model with poor performance for some classes can degrade the overall model performance, effectively displacing knowledge that already existed as part of the target model. To prevent this, helper model should be created by using the knowledge obtained from the helper node with the existing knowledge on the target node. The resulting helper model is then used in conjunction with target model.

A target node may exhibit multiple instance of input drift over time. This will require new knowledge transfers, and can lead to a pool of helper models. To prevent a resource bloat on the target, they are ephemeral, and are discarded when no longer needed. The target node continues to learn during the boosting period, so as to make it possible to remove its dependence on the helper model. Once all helper models are discarded, the target continues to operate only with its, now updated, target model ⑥. The use of a boosted helper model and the techniques for managing pools of helper models, solve the challenge *C2*:

Workflow: Large-scale companies learn from their user behavior by collecting and analyzing the geo-distributed trends. Such an environment allows for collaboration and knowing specific trends at various nodes in the system. Our work considers an edge as nodes close to clients and devices, potentially deployed at cellular towers and base stations [31], as edge servers at branch locations, etc. At each edge node, the *resident model* makes predictions on *batches of requests* that are received over a period of time. Fig. 1 presents a workflow in this context, where MdS provides the target node with the list of helper nodes ③ which have observed similar workload as target node. Since within a single company collaboration across distributed sites is not a concern, target node then reaches out to these nodes to obtain the relevant knowledge ④–⑥. The KT is performed using the mechanisms described in (Section V), which allow the target node model to more efficiently adapt its predictive performance to the changes in the workload characteristics.

CLUE incorporates the same approach for drift detection ② and helper node selection ③ as Cartel. In both systems, these tasks require that a node only shares metadata that could potentially distinguish it from other nodes and does not share the raw data received by node. Note that the KT mechanisms are also useful in context when direct communication among edge nodes may not be possible, due to lack of efficient edge-to-edge communication paths, or for regulatory, trust or privacy related reasons. In such settings, knowledge transfer can still be performed by leveraging a trusted and centralized broker (similar to an aggregator node in FL). The broker will gather the relevant knowledge from the helper node(s) and will share it with the target node. The tradeoff here is an addition hop in the transfer process, however since the data size of KT is small (few MBs Section VII), the general observations we make about CLUE will persist.

V. Design of CLUE

The design for collaborative learning system consists of – metadata operations and knowledge transfer support. We first provide a brief overview of the system level support that helps in drift detection and finding the helper node. Next, we describe in greater detail the design of the three mechanisms CLUE uses to realize knowledge transfer from a helper to a target node for neural network models.

A. Metadata Operations

CLUE builds upon our previous work [6], which introduces collaborative support mechanisms. These mechanisms are based on the use of system-level support for working with metadata which refers to any information about a node that has the potential to differentiate it from other nodes. This information may include estimates of class priors (used in our implementations), enabled features, user distribution, geographic information, software configuration, and distribution of active users by segments, among others. The metadata information is regularly shared by all the edge nodes in the system with a centralized Metadata service (MdS).

Drift Detection: A shift in workload characteristics can significantly impact the performance of the model at a given node. To effectively monitor and detect drift in model performance, CLUE employs a threshold-based drift detection mechanism. The proposed system records the performance metrics of the model, such as accuracy or loss, for each batch during the training or inference phase. It calculates the rolling average error rate of the model over the previous W batches, if the

current rolling average of error rate exceeds a certain threshold, it indicates the presence of drift in the model's performance. Here W is user-defined *window length parameter*. Drift detection is a well-explored research area, offering a range of algorithms that provide different trade-offs in terms of detection speed, sensitivity to drift, and adaptability to varying data patterns. Other sophisticated algorithms such as [13], [14], [32] can also be used to detect negative model drift (i.e., an increase in the model's error rate over time).

Helper Node Selection: When drift is detected, a call for *help* is sent to the MdS, which uses a *helper node selection* mechanism that relies on metadata collected from each edge node. In CLUE , where class priors may experience shifts, the comparison of empirical distributions between the target node and other nodes helps identify logical neighbors. The MdS determines the helper nodes by applying a similarity measure that quantifies the resemblance between nodes based on their metadata attributes. CLUE uses cosine similarity, alternative approaches such as [33], [34], [35], [36] can also be employed. Once a list of highly similar logical neighbors is identified, the top-k neighbors are selected and transferred to the target node. The target node then selects the helper node and reaches out for knowledge transfer.

The impact of timely detection of drift and helper node selection on the model performance at the target node is demonstrated in (Section VII-D1).

B. Knowledge Transfer Mechanisms

B.1) Significant Parameters Selection*:* During the learning process each neuron takes a group of weighted inputs, applies an activation function and returns the output. These inputs can be features from the dataset or output of a previous layer's neuron. The weights are adjusted using backpropagation ("backprop"), and the process is repeated until the network error drops below a threshold. Thus, for a given input, a combination of coefficients across multiple layers plays a role in determining the outcome. Our goal is to select from each layer of a model the significant parameters that are relevant to the requested classes. A trivial design choice could be to send the entire model from a helper to a target node and avoid any overhead of selecting a relevant portion of the helper model. However, NN models are large, and can include millions of parameters. Furthermore, when considering collaborative learning, only certain parameters relevant for the missing classes are required by the target node. Thus, we can reduce the data transfer cost by pre-processing the helper's node model to identify and send only the significant parameters for the relevant classes.

Intuition: To address the challenge *C1*, CLUE provides support for extracting the significant parameters of a helper node's model. The fundamentals behind this lie in the gradient descent algorithm. To optimize the loss functions after every step, the changes are applied to the model, while in our case, instead of applying the changes back to the model, we use this information to determine the portion of the parameters that are relevant to a given class.

Parameter Sensitivity: We use the backprop technique to determine the sensitivity by finding the impact that a small change in an input neuron (i) has on an output neuron (o). If drastic change occurs, i is considered to be significant for producing the current activation value of o. Let $f(W, x) = p$ denote the neural network model where W, x and p are set of all the model parameters, inputs and prediction, respectively. For a multi-class classification task, the output of the neural network is an array of size C where each element represents the confidence of the input being class C_i. We determine the sensitivity value of a parameter W_j towards the class output value C_i as

$$sen(W_j, C_i) = \frac{\partial C_i}{\partial W_j} = \lim_{\Delta W_j \to 0} \frac{C_i(W_j + \Delta W_j) - C_i(W_j)}{\Delta W_j} \tag{1}$$

$sen(W_j, C_i) > 0$ means increasing the value of parameter W_j, increases the output value for class C_i which implies the model is more confident that the input belongs to class C_i. On the contrary, $sen(W_j, C_i) < 0$ means increasing W_j makes the model more confident about an input not being class C_i. Thus, $|sen(W_j, C_i)|$ is defined as the measure of sensitivity of the parameter. The larger magnitude indicates higher significance of the parameter, as a small change in the parameter value could change the output.

Determining Sensitivity for a Changing Model: Over time the model at the edge node changes. As a result significant values cannot be calculated in the initial phase and used at a later time when a request from the target node is received. As the model is continuously retrained there are few ways to evaluate these values [21], [37], [38].

The first method is based on a *continuous* update of a sensitivity map for all model parameters and classes after every batch. We constantly make an observation of the parameters that are sensitive and keep track of them. Since the gradient is linear, we combine the neural net output of all data points in a batch to compute the parameters sensitivity value for all classes. For subsequent batches the values are recalculated and are added with the previously stored values. After every batch, this calculation of parameters adds a compute overhead of up to $0.7\times$ the time taken to process a batch. Additionally, since the sensitivity value of a parameter is different from its actual value in the model, this requires an addition memory of $O(CW)$ at each edge node. With large NN model sizes and resource constrained edge nodes, this method might not be a viable option. However, the pre-calculated values help to quickly respond by simply performing a lookup in the sensitivity map when a request for a portion of the model is made.

The second approach involves identifying the sensitivity of parameters *on-demand* when a request is received at the helper node.To keep overhead of the on-demand selection process low we decided to use data reservoir methodology over techniques such as testing each neuron [21] or adding Gaussian noise at different layers [39]. We keep a smaller local database of last B request batches. On request, sensitivity of the parameters is calculated against the stored batches in a similar way as before. The effectiveness and overheads of this method depend on B. In general, the choice depends on the workload dynamics and model. For our datasets and models we experimentally found

that B=1 works well as each request batch consists of sufficient data points.

These two approaches showcase a tradeoff between the memory required to calculate the sensitivity parameters and response time. Section VII-C discusses the overhead involved in each of them. CLUE provides a *pluggable parameter selection* component which enables the use of any technique.

Significant Parameters: Once the helper node determines the sensitivity of the parameters for the requested classes, the system selects the top Z percent of the sensitive parameter (both positive and negative). For a multi-class request we average the sensitivity value for all the classes before selecting the top Z parameters. After selecting the top parameters, CLUE selects the corresponding weights of these parameters from the model. Since these parameters are the most relevant for classifying a class, we refer to them as *significant parameters*. The significant parameters are compressed, which reduces the data size by up to 3-5×, and are sent to the target node. The exact setting of Z depends on the model as well as the tradeoff between the data transfer versus knowledge transfer to improve the target node model. For our dataset and models we experimentally found that Z can vary in the range 20% to 50%. Section VII-D illustrates the impact of different values of Z and we leave the automatic tuning of Z as a topic for future investigation.

B.2) Helper Model: After receiving the significant parameters from the helper node(s), the target node needs a mechanism to apply these updates to improve the predictive performance of the node. A simple approach would be to apply the received knowledge directly on the existing target model. Different from replacing the significant parameters is to update the model by averaging the weights of target model with the received significant parameters, similarly to federated averaging [40].

The outcome of replacing or averaging the existing parameters is dependent on the workload distribution. The workload distribution observed at different nodes in the system can vary. It can be *overlapping* where the workload observed at the target is a subset of the workload observed at another node, or *non-overlapping* where the other classes observed at the nodes are not the same. When the target node workload distribution is a subset of the workload observed at the helper node (*overlapping*), replacing the parameters would provide more benefits. However, when the workloads differ (*non-overlapping*), averaging the parameters would help more when compared to parameter replacement. Our evaluations, done for 100 request batches on the EMNIST dataset using MobileNet convolution neural nets for image classification (more details Section VII-A), showed that for a *non-overlapping* workload replacing the parameters further increases the model error rate since it replaces parameters for classes already learned by the model, as shown in Fig. 2(a). When the target and helper model observe a similar workload, replacing the parameters helps, whereas averaging the parameters would result in the target model taking more time to adapt to the new changes, Fig. 2(b).

Multi-Model Approach: To decouple the dependence on the workload distribution across the target and helper node(s) in the system, we use the significant parameters to create a helper model. The intuition behind this approach is based on the ensemble learning technique where multiple models are strategically generated and combined to improve the overall performance of a model. We create a helper model which acts as a discriminator network to responds to a particular class, in our case the newly introduced class(es), without discriminating the existing classes.

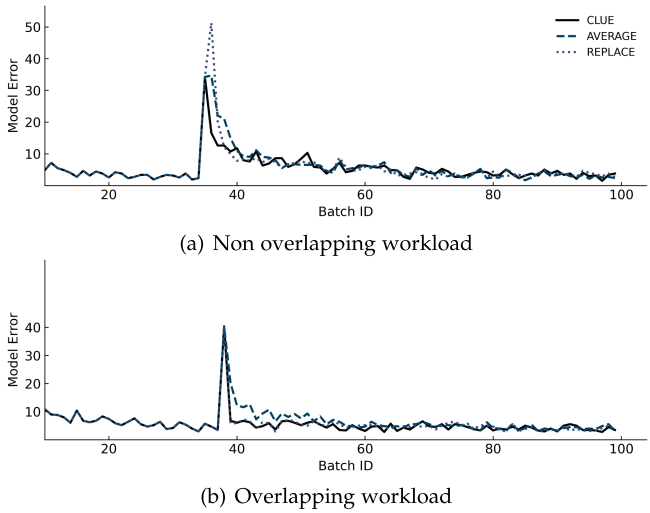


Fig. 2. Multi-model approach compared with averaging or replacing the existing model parameters: CLUE performed well and was consistent in both the scenarios; replacing the parameters worked well with overlapping classes and average performed better when the classes at the two nodes were non-overlapping.

As neither the target model nor the significant parameters can solely make accurate prediction, we use multi-model approach where a helper model is created using boosting technique [41]. The original target model is used for maintaining the accuracy for the existing classes while the helper model provides a boost in accuracy for new classes which were already known at the helper node. The final output is a resultant of combining the output from both models.

Helper Model Creation: As the target node receives only a subset of the parameters from the helper node(s), we consider two ways to create a helper model. In the first method, helper model is created by using the significant parameters, and the remaining parameters of the model are set to 0. By having some weights in the layer as 0 effectively reduces the magnitude of the output of that layer, eventually impacting the overall output. In the worst case, if the parameters in one of the layers are all set to 0, all intermediate values and final class output scores would be the same for any input. In such a case, no useful information could be derived from the helper model output. Therefore, we consider a second method, which creates a helper model by combining the significant parameters from the helper node with the existing parameters from the target node. The resulting *boosted helper model* is represented as shown in (2). Fig. 3 showcases a helper model created using the target model, compared to the one created using no prior information: BOOST experiences a lower spike in the model error rate when a previously known class is observed at the target, and it more quickly adapts and

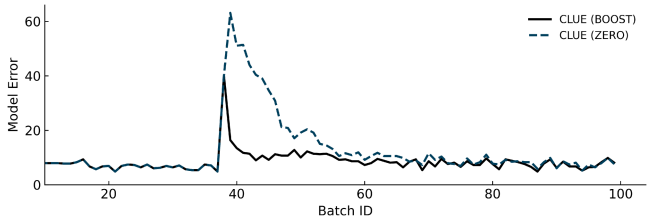


Fig. 3. Boosted model error rate for two types of helper models. Helper model created with boosting from target model performs better than the one created only using the significant parameters.

converges to its original accuracy.

$$W_i^h \leftarrow \begin{cases} W_i^s, & \text{if } W_i^s \text{ is significant} \\ W_i^t, & \text{if } W_i^s \text{ is not significant} \end{cases} \quad (2)$$

where W^t and W^s represents the target node and significant parameters, respectively.

Boosted Model Output: After creating the helper model the prediction function is updated to take into account the classification outputs from the helper model. At this stage we have a target model, good for the existing classes, and a helper model, good for the new ones. We use average of both the models to compute the final output, though other function can also be used.

Managing Pool of Helper Models: The use of helper models improves the overall performance of the model. However, it creates the problem of repeatedly generating new helper models, on-demand, whenever a concept drift is detected. Maintaining an ever lasting pool of helper models would exhaust resources at the edge node. To alleviate this we take two steps. First, since the goal of the helper model is to provide assistance to the target model, we freeze the helper model by not retraining it, while we continue to (re)train the target model. Second, we discard the helper model(s) as the performance of the model stabilizes for the class(es) for which the helper model was created.

A straightforward way would be to discard the model after a user-defined number of batches. Discarding a helper model prematurely could result in a second smaller drift, while discarding it too late would result in unnecessary computation and resource usage. To avoid this uncertainty CLUE uses a dynamic approach where after every request batch the performance of the target model is tracked separately and is compared to the pre-drift performance. When it reaches the earlier performance level helper model is discarded. In presence of multiple helper models forming a pool, CLUE provides an additional check which compares the predictive performance of the new classes for the past L batches to determine which model to discard. The precise value of L depends on the number of samples observed, models and dataset. For our experiments the value was experimentally determined and we leave the online tuning of L as a topic for future work. The system allows the precise discard policy to be configurable by the developer.

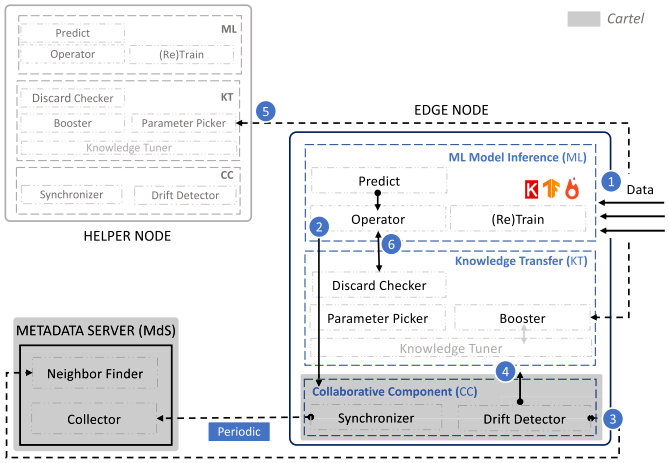


Fig. 4. CLUE system components.

VI. CLUE Runtime

Components: The CLUE runtime is divided into three main blocks – ML Model Interface (ML), Knowledge Transfer (KT), and Collaborative Component (CC), shown in Fig. 4.

The **CC** component controls how a node collaborates with other nodes in the system. It encapsulates the functionality needed to discover a helper node and to determine when to reach out for help. *Synchronizer* is responsible for exchange of metadata that supports helper node discovery. *Drift detector* determines when interactions with a helper node get triggered, i.e., when the local model degrades in performance, based on a *drift detection* algorithm., as in [6]. In [7], synchronization is performed periodically, and is always coupled with a receiving help via model distillation.

The **ML** component provides the interface to the ML framework (e.g., PyTorch or TensorFlow). It uses the *predict* interface to perform the prediction on the incoming request data batch. The output of the prediction is passed to an *operator* component. During normal operation, the prediction result is used directly. In the presence of helper model(s), during periods of knowledge transfer, the *operator* averages the outcome of all the model(s). The cycle is completed with training the resident target model with the new batch of data using the *(re)train* interface.

The **KT** component facilitates the knowledge transfer mechanisms. At each node, *parameter picker* supports the parameter selection mechanism, and is invoked when a node receives a request for help for a specific class(es). The process is performed by either the *continuous* or the *on-demand* mechanism. The resulting parameters are sent to the target node where *booster* creates a new helper model. When helper model(s) are created, the *operator* function is adjusted to perform averaging, and a *discard checker* monitors the convergence process to determine when helpers are to be discarded, dynamically or statically after predefined number of batches, based on a configurable policy.

CLUE exposes APIs to configure the system parameters controlling the knowledge transfer process in terms of the percentage of significant parameters transferred from the helper, the number of request batches to be locally stored for *on-demand*

parameter selection, the discard policy, etc. In this manner, CLUE exposes new *tuning knobs* that control the knowledge transfer process in terms of its effectiveness in improving the target model and the overheads it introduces, which can be used for new types of resource orchestrators for distributed learning.

Implementation: We implemented a prototype of CLUE for the PyTorch and TensorFlow (TF) ML frameworks, using Flask to support the communication among different nodes. The use of the different frameworks exposes the following implementation requirement. The significant parameter selection can be done in different ways, however using the dynamic selection approach would require the ML framework to provide APIs for calculating the gradients on the model. The framework should also provide the capability to dynamically create, use and discard helper model(s), in conjunction with the existing model. Given the available APIs in PyTorch, our implementation of CLUE supports all the mechanisms described in Section V. TF creates a static computation graph before a model is run and any interaction with the model is tightly integrated with the session interfaces and state placeholders. Due to these limitations, the current TF implementation of CLUE cannot support *on-demand* significant parameter selection, and only uses the *continuous* significant parameter mechanism. Since TF does not permit models to be dynamically created, we use a pool of helper models from the initial state of the experiment. The resulting TF implementation of CLUE introduces runtime memory overheads which may be prohibitive for practical settings. We note that the limitations of the current TF APIs may be addressed in future versions of the system, and we still compare the two implementations to illustrate the generality of CLUE for realizing knowledge transfer across models developed with different ML frameworks (Section VII-E).

VII. Evaluation

The goal of the evaluation is to experimentally validate the effectiveness of the mechanisms introduced, in terms of their ability to perform knowledge transfer among neural network models. Our previous work [6] addresses the efficiency and cost associated with the drift detection and helper node selection mechanisms. In this work, we explore the following questions:

1. How effective is the knowledge transfer mechanism enabled by CLUE in allowing neural network models to adapt to change, and what impact does it have on improving the overall model performance and curtailing data transfer costs? (Section VII-B)
2. What are the overheads introduced by the parameter selection in CLUE on the helper nodes, and by multi-model learning on the target node? (Section VII-C)
3. What is the impact of the different system-level parameters and the effectiveness of the collaborative mechanisms for drift detection and helper node selection on the overall benefits CLUE provides? (Section VII-D)

We used multiple workloads, models, and frameworks, as described below, to ensure the generality of the observations.

A. Experimental Methodology

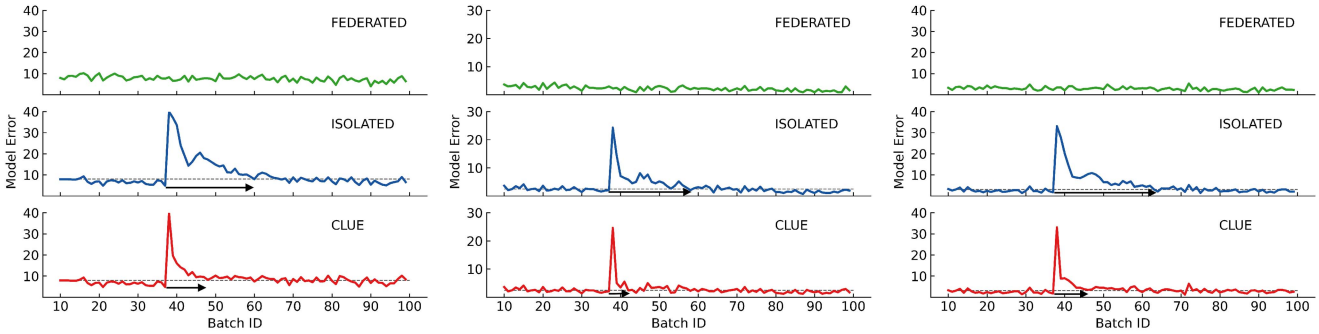
Experimental Setup: As mentioned earlier, edge servers are potentially deployed at cellular towers and base stations. These nodes can be equipped with GPUs and Edge TPUs accelerator [11]. We evaluate CLUE on a cluster of 5 Intel Xeon nodes on the Chameleon Cloud, each with 125GiB RAM with 48 cores running 2 threads per core at 2.30 GHz. The experiments are performed with two versions of CLUE , integrated with two existing machine learning frameworks: PyTorch v1.7.1 and TensorFlow v2.3.1 (Section VII-E).

Applications, Datasets, Workloads: We use two application use cases in the evaluation: image classification and intrusion detection (Section VII-F). For the image classification use case the results are obtained with the EMNIST dataset, consisting of 280 k data samples. We made similar observations with the smaller MNIST dataset. For the intrusion detection use case, we used the NSL-KDD dataset which consists of benign network requests mixed with various malicious attacks consisting of 126 k records.

During the experiments, each edge node processes request batches over time, with each batch consisting of a varied number of data points corresponding to different classes in the dataset. To create a change in class priors that triggers the need for collaboration and KT, we generate workloads with several synthetic request patterns, corresponding to *introduction* of new class(es) at an edge node, *fluctuation*, where new class(es) appears and disappears, with overlapping and non-overlapping classes, as described in Section V-B2. The use of synthetic workload patterns is due to the limited availability of real infrastructure and data with both temporal and spatial metadata on the request distribution. The concrete patterns we use are based on information about distributions used in other research [3], [6], [12].

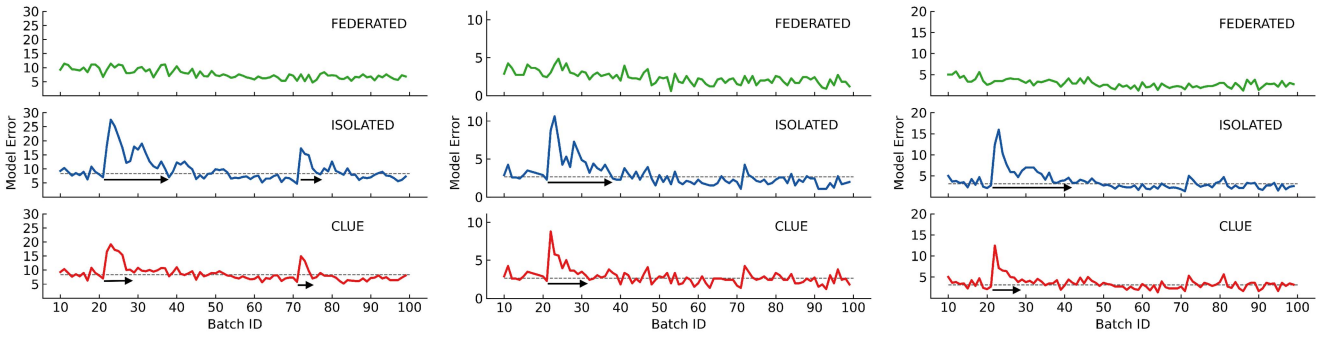
Models: For the image classification use case, the presented results are based on PyTorch and the MobileNet (3.54 M), DenseNet (8.064 M) and ResNet (11.694 M) CNN models. The models are fine tuned using 2 k data points from the EMNIST dataset using Adam optimizer and learning rate 0.00005, and trained for 20 epochs with batch size of 30. We also used the MobileNet, DenseNet and InceptionNet models with TensorFlow v2.3.1, to confirm the effectiveness of CLUE across different machine learning frameworks (Section VII-E). The intrusion detection use case uses a four-layer DNN model with 209 k parameters [42]. For model initialization we used the SGD optimization algorithm with a learning rate is 0.000005 and 25 k data points, and also trained for 20 epochs with batch size of 30.

Baseline: Collaborative learning shares similarities with FL (Section II), making it as an obvious choice for baseline. In our evaluation, we consider FL with FedAvg [40] as one baseline, and a second baseline is isolated learning at each edge node, avoiding wide area data transfers but with slower adaptation to change. The goal is to showcase how the mechanisms in CLUE enable customized edge models with acceptable accuracy similar to a global model while reducing data transfer costs. Additionally, our previous work [6] highlighted the advantages of collaborative learning over centralized learning.



(a) MobileNet (b) DenseNet (c) ResNet

Fig. 5. Performance comparison for introduction workload distribution (lower error rate is better) showcasing overall model error change. Misclassification of introductory class degrades the model performance. CLUE is able to quickly adapt to the change in distribution (given by the horizontal arrow). The horizontal dotted line (in black) defines the lower bound obtained through averaging the federated learning error rate.



(a) MobileNet (b) DenseNet (c) ResNet

Fig. 6. Performance comparison for fluctuation workload. CLUE is able to adapt to the changes in the distribution and behaves close to a federated learning system.

Metrics: When using local, tailored models, change in the workload pattern at a node due to previously unseen classes, results in a temporary drop in the predictive performance of the model, observed as an increase in its error rate. The time taken for the model to reach back the acceptable performance levels is defined as the model's *adaptability to change*. We evaluate CLUE by measuring the improvements in adaptability to change compared to isolated learning. Given the synthetic workloads, we measure time in terms of the number of batches that need to be processed for the model performance to reach the same error rate as what is possible with FL. We also measure the improvements system provides in terms of per-node and aggregate data transfer costs (in bytes). Finally, we measure the runtime overheads of CLUE in terms of additional compute and memory resources.

B. Benefits of CLUE

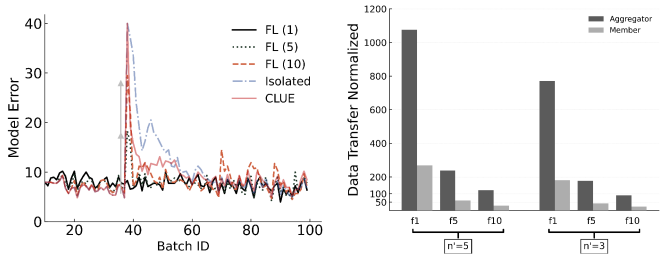
B.1) Adaptability: Figs. 5 and 6 illustrate the adaptability enabled by CLUE for the *introduction* and *fluctuation* workload where the system uses 50%, 20% and 30% of the significant parameters for MobileNet, DenseNet and ResNet, respectively. The parameter Z that controls the percentage of knowledge transferred for each of the three models is experimentally chosen to that it minimizes data transfer while keeping the model adaptability within 10% of the best case. For the testbed configurations in these experiments, the helper model is created within 4 request batches after the drift detection. We observe that with CLUE , the models converge to the performance level equivalent to using a global model up to 2 to $3.5\times$ faster as compared to an isolated system. Also, in Fig. 6(a) we observe a second spike in the model performance as the class reappears in the later part. This is because the other two models are larger and are able to retain more information even when the class was not present for few request batches.

B.2) Reduction in Data Transfer Costs: CLUE exposes a tradeoff in spending cycles to adapt to the change in workload versus the data transfer cost associated in creating and updating a global model, as done in FL.

Cost Associated With Operation: To capture the network usage we divide the operations associated with these systems into *Out* and *In* categories. For FL the delta in the local model since the last batch captures the model update MU_{out} as the *Out* cost and the received aggregated model update MU_{in} is the *In* cost. For CLUE , the metadata sent to the MdS server MdS_{out} is considered as *Out* cost and when data drift is detected on a node, that node incurs the knowledge transfer cost KT_{in} which is captured as *In* cost. Table I highlights the cost for

TABLE I
AVERAGE DATA TRANSFER COST (MB) ASSOCIATED WITH AN *OUT* AND *IN* OPERATIONS FOR TARGET NODE IN CLUE AND PARTICIPATING MEMBER NODE IN FEDERATED LEARNING

Model	Operation	FL	CL	Gains (×)
MobileNet	Out	6.38	0.00058	11000
	In	7.36	4.57	1.61
DenseNet	Out	16.18	0.00058	27896
	In	18.35	8.80	2.08
ResNet	Out	12.19	0.00058	21017
	In	13.75	8.55	1.60



(a) Predictive Performance (b) Data Transfer

Fig. 7. Comparison of federated learning (FL) predictive performance for introduction workload as aggregation frequency is changed from sending the model update after every batch to 5 and 10 batches (left). Total data transfer (in and out) gains in CLUE for a single node when compared against different FL settings. We change the aggregation frequency (f) and the number of member nodes selected for each round (n') (right).

each operation based on the model. At the fundamental level CL uses *pull* (on-demand) approach where *In* cost occurs only when there is a change observed at a target node. While FL uses *push* approach where both *Out* and *In* cost occurs regularly, which when taken into account for the entire experimental duration results in overall larger data transfer when compared to CL.

Data Transfer Breakdown Analysis: Using the above operations the total data transfer cost for the experiment can be represented as follows:

$$DT_{FL} = \left(\sum_{i=1}^{n'} MU_{out} + MU_{in} * n\right) * f_{update} \tag{3a}$$

$$DT_{CL} = MdS_{out} + \sum_{i=0}^{d} h_i * KT_{in} \tag{3b}$$

where for FL n is the total edge nodes in the system, n' is subset of nodes participating in each round and f_{update} is aggregation frequency. For CL MdS_{out} is the metadata send to the MdS service while d represents the total number of drifts observed and h_i the number of the helper node(s) needed for each drift.

Effect of Aggregation Frequency: The aggregation frequency used for FL can impact the model accuracy and the total backhaul data transfer. While sending the model updates less frequently will impact the overall data transfer cost, it however will *not* impact the operation cost highlighted in the Table I. Fig. 7(b) compares the backhaul usage savings from the perspective of the aggregator and member nodes, and illustrates the change in the data transfer gains for a single node as the aggregation frequency for FL changes while keeping the nodes that participate in each round equal to all. We observe that CLUE helps in reducing data movement cost by 270 to 30× for each participating member node, or 1070 to 120× for the aggregator node. Importantly, as illustrated in Fig. 7(a), such a change could impact the model's predictive performance of FL, because of absence/stale knowledge about the new changes in the workload at the member (target) node.

Effect of Member Node Selection: A large scale FL deployment to perform model update in each round selects only a portion of the member nodes (n'). This selection can be performed randomly or based on some eligibility criteria [28], [43] which would impact the data transfer cost and model convergency rate in FL system. In our experiment we randomly selected three out of the five nodes and observed that CL still saves data transfer cost by 270 to 25× for each participating member node, or 1070 to 90× for the aggregator node, as illustrated in Fig. 7(b).

Overall Experimental Savings: FL is data movement intensive, given that even small amounts of raw data can result in large amounts of information being communicated over the network [20]. Different methodologies such as tuning the aggregation frequency, number of member node selection or using sophisticated training algorithms could help in communication costs reduction.

At the system level, involving large number of nodes, to help create a global model FL system requires model updates to be regularly shared by different nodes. This adds to the communication cost in FL when compared to a CL system where data (KT) is only transferred between few nodes only when there is a need. In our evaluation setup, a participating member node observes on average 1240 to 106˜MB of data transfer. The total data transfer for all the nodes ranges from 4920 to 415˜MB for various FL configuration while only 4.85˜MB was transferred in CLUE for same workload distribution, a reduction of up to $10^3\times$ in the overall data transfer costs for all the entire system. Thus, when data transfer cost are combined across the nodes in the system, the total combined data savings for CLUE would be much higher than a federated learning system.

C. Cost of CLUE Mechanisms

Extracting Significant Parameters: The *continuous* method for parameter selection stores the sensitivity value of all parameters for all of the classes, and its overheads depend on the model. For our dataset and given models, this introduces additional memory overhead of 140 - 470˜MB, depending on the model. The *on-demand* approach creates a local datastore which is only used when a request from a target node is received. For our dataset, storing only the most recent request batch was sufficient and required an addition of 6.8˜MB of memory. We performed the same experiments for different batch sizes, and, across workloads, observed similar gains.

In terms of the compute overhead, the *on demand* approach adds to the response time 0.9 to 1.35× time required to process a request batch. The *continuous* approach, by pre-calculating these values, avoids such delays in the critical path of response,

TABLE II
OVERHEAD IN PERFORMING MECHANISMS DESCRIBED IN SECTION V NORMALIZED WITH THE TIME REQUIRED TO PROCESS A REQUEST BATCH

Model	Continuous (×) per batch	On-Demand (×) per request	Helper Model (×)
MobileNet	0.70	0.98	0.08
DenseNet	0.65	1.35	0.11
ResNet	0.70	0.95	0.14

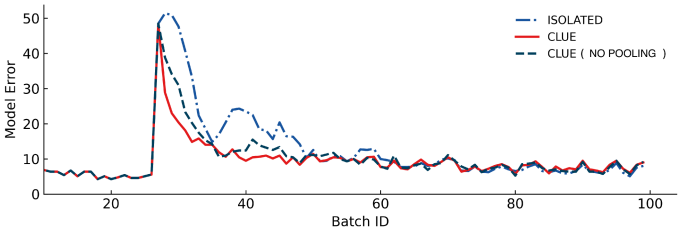


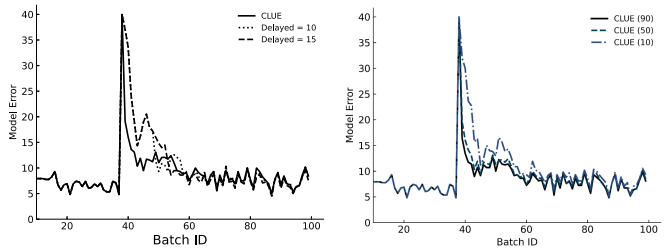
Fig. 8. Pooling multiple helper models improves the predictive performance and helps the system adapt to the new changes faster.

however, to maintain the sensitivity map it adds an overhead of up to $0.7\times$ time required to process a request batch, for every request, as shown in Table II.

Helper Model Creation: The time taken to create the model depends on the size of the portion of model transferred from the helper node (KT_{in}), as shown in Table I, and the time taken to create a model at the target node. Upon receiving the significant parameters creating a helper model takes only up to $0.14\times$ the time required to process a request batch.

Management of Helper Models: A helper model in CLUE uses similar memory as the resident node model however since the model is frozen, no additional resources are required after its creation. Using multiple helper models in a pool adds to the memory overheads. Thus, if the knowledge about the introductory classes at the target node is dispersed among multiple nodes, in such a case CLUE exposes a tradeoff of using more memory for a short duration of time by deploying a pool of helper models versus spending more time in adapting to the changes. Fig. 8 shows the collaborative performance of using help from only one of the nodes at a time (no pooling) compared to creating a pool of models from multiple nodes and averaging across all at the same time. The ability to combine multiple knowledge sources provides the lower bound on adaptability to the new changes in the system.

Effect of Number of New Classes: The change in the system in the above experiments is caused by the introduction of two new classes at the edge node. The overhead on the collaborative process depends on the number of helper nodes involved. To provide better understanding we can divide this into two scenarios where knowledge relevant to the new class(es) is: (i) available at one helper node, (ii) spread across multiple helper nodes. The significant parameters extracted from a single model depend only on the user defined value Z and is independent on the number of requested classes, because for a multi-class request the sensitivity value for all the classes is averaged before selecting the top Z parameters. Thus, capping the total number of



(a) Delayed Drift Detection (b) Significant parameter selection

Fig. 9. Impact of various drift detection policies on the overall model and introductory class error rates where "Delayed by X" refers to drift detection being postponed by X batches (left). Comparison of impact of percent of significant parameter Z (right).

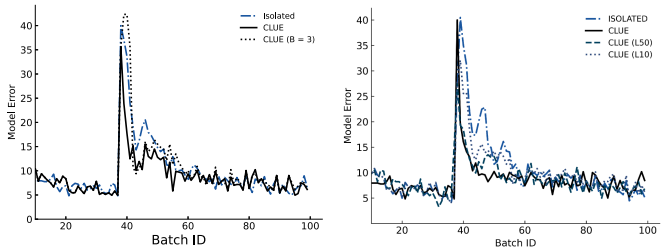
parameters sent from the helper node to Z parameters. However, if the new classes introduced at the target node are spread across different helper nodes, in such a scenario the total KT_{in} data transfer cost would be the aggregation of knowledge from all the helper nodes, as shown in (3b).

D. Sensitivity Analysis

D.1) Drift Detection Timeliness: In CLUE , it is important to detect concept drift quickly, because a delay in detection can lead to a decrease in the model's predictive performance. To show the impact of slow concept drift detection, we modified the drift policy to delay the request to the MdS for logical neighbors by a variable number of batches. The results in Fig. 9(a) show the impact of drift detection delay on the overall model's performance and the misclassification rate of the introduced class for the MobileNet model. If the delay is too large (e.g., 15 batches in the figure), model transfer does not provide any collaboration benefits, because online retraining would have eventually improved the model.

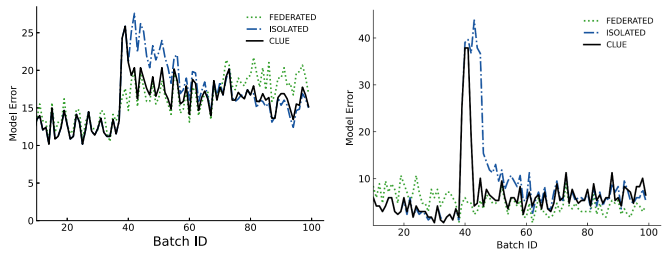
D.2) Impact of Quantity of Knowledge Transferred: CLUE controls the portions of the significant parameters that are transferred via the parameter Z. Fig. 9(b) shows the impact of different values of Z on the adaptability and data transfer costs for *introductory* workload for MobileNet; the results with the remaining models had a lower sensitivity to Z than for MobileNet. The results illustrate first that an effective helper model can be created using only a portion of the relevant parameters. Second, CLUE introduces new ability to control a tradeoff among learning performance and data transfer costs, by using the amount of knowledge that should be transferred across nodes as a configuration knob. This can be leveraged by future learning management policies. For instance, the higher value of Z (90) may provide improvements in error rate, when compared to Z (10), specifically for this setup 25% reduction in average model error rate while using the helper model. However, it would result in $5.53\times$ more data transfer.

D.3) Helper Node Selection: In a distributed environment with a large number of edge nodes, various methodologies exist for selecting a helper node. This selection process can be random or based on similarity measures. The core concept of collaboration is to leverage the existing knowledge pertaining to



(a) Incorrect Helper Node (b) Helper node quality

Fig. 10. Comparison of consequences of incorrect helper node selection, with the correct helper node being identified 10 batches later (left) and impact of quality of helper node model (right).



(a) (b)

Fig. 11. Comparison of model performance using *introductory* workload for (a) network attack dataset (b) smaller model on EMNIST dataset.

specific class(es). To emphasize the significance of this aspect, we introduced modifications to the selection algorithm. Specifically, for the initial request, the algorithm randomly selects the least similar node, whereas for subsequent requests, it chooses the most similar node. Fig. 10(a) illustrates the impact of these modifications on the overall model performance. In this scenario, the first helper node lacks the knowledge associated with the newly observed classes at the target node. As a result, the Knowledge Transfer (KT) process does not contribute to an improvement in the overall error rate; instead, it exacerbates the situation. Given the persistently high model error rate, the system initiates another request for help. This time, KT is performed from the most similar node, which leads to an enhancement in the model's performance compared to isolated learning. Multiple collaborations also impacts the overall data transfer by increasing it by $2\times$.

Impact of Quality of Helper Node: While selecting similar nodes is crucial, it is equally important to consider the quality of the workload observed by each node. This factor can significantly influence the adaptability of the model performance at the target node during collaboration. To understand this, in Fig. 10(b), we compare the execution of CLUE and Z(50) with MobileNet and the same helper node as used in the previous experiment, to that of two other scenarios where the helper nodes have observed only 50% (L50) and 10% (L10) of the data observed in the baseline. We observe that the quality of the helper node does impact the realized benefits from CLUE , however, using even a weaker helper node (L10) can still be useful, compared to just learning in isolation. Note that, simply increasing Z for a weak helper node would not be helpful, as the quality of the model parameters is not good for the required knowledge.

E. Extensibility to Different ML Frameworks

To demonstrate the generality of the CLUE , we implement it for PyTorch and TensorFlow (TF). Using the same experiments as for PyTorch, and the MobileNet, DenseNet and InceptionNet CNNs, we measure that TF CLUE adapts to change up to $3\times$ faster than isolated learning, while using only 30-60% of the model. This results in overall data transfer reduction of 700 to $1200\times$ compared to FL.

F. Use Case

F.1) Smaller Model*:* The standard models mentioned earlier may be too large for the EMNIST dataset, which raises concerns. To solve this problem, we used a smaller neural network model with only four layers, containing 470 k parameters and Z(50). Using the same introductory workload, Fig. 11(a) demonstrates that CLUE is able to adapt to the changes $2.5\times$ faster than isolates system, while achieving a similar model performance as FL with an overall $860\times$ less data transferred.

F.2) Intrusion Detection*:* The IDS use case is based on the NSL-KDD dataset [44] which consists of four types of attacks. The testbed consists of 5 edge nodes, where initially the target node observes only the Remote-to-Local (R2L) and Probe attacks, while the other nodes experiences all of the attacks. After some time the target node starts to experience the DoS attack. We use a four layer DNN model as described in Section VII-A with *introduction* workload consisting of 100 request batches with 400 data points in each batch and $Z(40)$.

Results: Fig. 11(b) illustrates the model performance for FL, CLUE and isolated. FL allows the target node to classify the attack immediately, since the attack was observed by other nodes earlier, and is captured in the global model updates. In isolated learning mode, the target node experiences both a large spike in its model error rate, and takes substantial time ($\approx$20 batches) to retrain its model to the original performance (error rate). CLUE helps the target node adapt to the new attack $1.75\times$ faster than isolated system and to provide similar average model error rate compared to the FL system with overall $600\times$ less data transferred.

VIII. Discussion and Future Work

CLUE enabled and demonstrated the potential of collaborative learning with neural networks, while achieving similar predictive performance compared to federated learning, but with orders of magnitude lower overall data transfer cost.

Collaboration Tipping Point: The evaluation demonstrates that the KT mechanisms in CLUE make it possible to perform collaborative learning for NN models in a way that maintains good accuracy while reducing data transfer by orders of magnitude. The precise data transfer gains of CLUE over FL depend not only on the evaluated configuration parameters of FL, but also on the properties of the workload, particularly the frequency

and distribution of input drift occurrences. More frequent drifts will reduce the observed benefits. However, we show that at the level of a participating member node, the gains are substantial, therefore the number of drifts needs to increase in a major way for, the data transfer gains to completely disappear. With CLUE we offer new mechanisms that will allow ML operators to tune the configuration of the ML application deployments, but we leave the support for policy engines that will evaluate the tradeoffs and make (dynamic) configuration recommendations to future work.

Privacy: CLUE does not share the raw data received at the edge, however, it assumes the nodes can act in collaboration. This means knowing specific trends at various nodes in the system is not a concern. Fig. 1 is one design that would allow the trends to get shared among the nodes. To provide better privacy guarantees, an alternative design point is to use CLUE while avoiding sharing trends directly among edge nodes. In such a case, to provide on-demand relevant portion of the knowledge from the helper node, CLUE , could use a trusted centralized broker as described in (Section IV). This alternative design still retain the overall benefits of data reduction due to *on-demand* nature of collaborative learning, while obfuscating the helper node from other peers. The significant parameter selection only sends the weights of the model. These weights are similar to the summary of the changes, also called model update, sent in federated learning. Thus, any privacy concerns would be similar to the one observed in approaches such as federated learning. Additional techniques such as differential privacy, reputation-based helper node selection, etc., could be integrated with the mechanisms for helper node selection and helper model pool management, however this is beyond the scope of this paper.

Model Heterogeneity: CLUE assumes the same model structure at all the nodes in the system. The focus with this work was to create a system support for collaborative learning for NN. New support for "once-for-all" networks [45] create a path to leverage a same model in heterogeneous settings, and we plan to explore support for collaborative learning in such settings in the future.

IX. Related Work

Distributed Machine Learning: Federated learning creates a generic model that is used by all nodes in the distributed system. Improvements have been made to its accuracy and convergence rate [1], [28], [46], [47], [48]. To address data transfer overheads in geo-distributed systems, Gaia [30] distinguishes among communication over local versus wide area networks. It uses a threshold-based approach to determine the importance of model update while CLUE is concerned with identifying a portion of the model that most contributes to the model quality with respect to a specific class. For settings where a global model is not required, distributed collaborative learning system are developed [6], [7], [49], [50]. CLUE contributes to such approaches by proposing a generic system support for such system to use neural networks.

Knowledge Transfer Technique: CLUE uses significant parameters of an ML model to perform knowledge transfer. [21], [51], [52] are different approaches to determine significant parameters from a model. CLUE uses similar but simplified techniques to provide for fast calculations and enable their online use. Unlike maintaining multiple models and using ensemble techniques to improve the prediction [53] CLUE uses ensemble approach on the dynamically created helper models which are destroyed in the future. CLUE is intended to be a general framework that can be used with different techniques for significant parameter selection, with a corresponding impact on the benefits or costs of learning.

X. Conclusion

We introduce CLUE , a system that enables collaborative learning for neural networks. CLUE contributes new support for knowledge transfer across neural networks, which makes it possible to dynamically extract helpful knowledge from one node – in the form of significant model parameters – and to apply it to another – using dynamically created helper models and multi-model boosting. The specific methodologies used in the current prototype are one way to perform this, however the APIs provided by the system allow easy substitution with other algorithms in the future. Furthermore, it provides new interfaces to exercise the tradeoffs among accuracy and overheads in the learning process that can be exploited in future resource management policies. CLUE is prototyped and evaluated with several NN models for the PyTorch and TensorFlow ML frameworks. The experimental results demonstrate that CLUE can enable knowledge transfer across NNs in a manner that improves the model accuracy when a drift occurs, compared to learning in isolation, while reducing data transfer costs by orders of magnitude, compared to federated learning.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable feedback.

References

[1] J. Konečnỳ, B. McMahan, and D. Ramage, "Federated optimization: Distributed optimization beyond the datacenter," 2015, *arXiv:1511.03575*.
[2] "Multi-access Edge Computing (MEC)," European Telecommunications Standards Institute (ETSI). Accessed: Jul. 18, 2023. [Online]. Available: https://www.etsi.org/technologies/multi-access-edge-computing
[3] M. Polese, R. Jana, V. Kounev, K. Zhang, S. Deb, and M. Zorzi, "Machine learning at the edge: A data-driven architecture with applications to 5G cellular networks," 2018, *arXiv: 1808.07647*.
[4] A. Padmanabha Iyer, L. Erran Li, M. Chowdhury, and I. Stoica, "Mitigating the latency-accuracy trade-off in mobile data analytics systems," in *Proc. 24th Annu. Int. Conf. Mobile Comput. Netw.*, 2018, pp. 513–528.
[5] X. Zhou, Z. Zhao, R. Li, Y. Zhou, and H. Zhang, "The predictability of cellular networks traffic," in *Proc. IEEE Int. Symp. Commun. Inf. Technol.*, 2012, pp. 973–978.
[6] H. Daga, P. K. Nicholson, A. Gavrilovska, and D. Lugones, "Cartel: A system for collaborative transfer learning at the edge," in *Proc. ACM Symp. Cloud Comput.*, 2019, pp. 25–37.
[7] Y. Lu et al., "Collaborative learning between cloud and end devices: an empirical study on location prediction," in *Proc. IEEE/ACM 4th Symp. Edge Comput.*, 2019, pp. 139–151.
[8] M. Satyanarayanan, W. Gao, and B. Lucia, "The computing landscape of the 21st century," in *Proc. 20th Int. Workshop Mobile Comput. Syst. Appl.*, 2019, pp. 45–50.

[9] "Poweredge C servers. Proven performance and hyper-efficiency at scale," Dell Technologies, Round Rock, TX, USA. Accessed: Jul. 18, 2023. [Online]. Available: https://i.dell.com/sites/csdocuments/Shared-Content_data-Sheets_Documents/ar/dz/ESG-PowerEdge-C-Portfolio-Brochure.pdf
[10] M. Trifio and J. Smith, "State of the edge 2021: A market and ecosystem report for edge computing,". Accessed: Jul. 18, 2023. [Online]. Available: https://www.lfedge.org/wp-content/uploads/1166 2021/08/StateoftheEdgeReport_2021_r3.11.pdf
[11] "Edge TPU: Google's purpose-built ASIC designed to run inference at the edge," Google Cloud, Mountain View, CA, USA. Accessed: Jul. 18, 2023. [Online]. Available: https://cloud.google.com/ edge-tpu
[12] U. Paul, A. P. Subramanian, M. M. Buddhikot, and S. R. Das, "Understanding traffic dynamics in cellular data networks," in *Proc. IEEE INFOCOM*, 2011, pp. 882–890.
[13] J. Gama, I. Zliobaite, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 1–37, 2014, doi: 10.1145/2523813.
[14] D. Kifer, S. Ben-David, and J. Gehrke, "Detecting change in data streams," in *Proc. Very Large Data Bases*, 2004, vol. 4, pp. 180–191.
[15] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015, *arXiv:1503.02531*.
[16] C. Tan and J. Liu, "Online knowledge distillation with elastic peer," *Inf. Sci.*, vol. 583, pp. 1–13, 2022.
[17] D. Xu et al., "Edge intelligence: Architectures, challenges, and applications," 2020, *arXiv: 2003.12172*.
[18] R. Boutaba et al., "A comprehensive survey on machine learning for networking: Evolution, applications and research opportunities," *J. Internet Serv. Appl.*, vol. 9, no. 1, pp. 1–99, 2018.
[19] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," *Heliyon*, vol. 4, no. 11, Nov. 2018, doi: 10.1016/j.heliyon.2018.e00938.
[20] K. Bonawitz et al., "Towards federated learning at scale: System design," 2019, *arXiv: 1902.01046*.
[21] B. Zhou, Y. Sun, D. Bau, and A. Torralba, "Revisiting the importance of individual units in CNNs via ablation," 2018, *arXiv: 1806.02891*.
[22] A. Bau, Y. Belinkov, H. Sajjad, N. Durrani, F. Dalvi, and J. Glass, "Identifying and controlling important neurons in neural machine translation," 2018, *arXiv: 1811.01157*.
[23] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?," 2014, *arXiv:1411.1792*.
[24] N. Tajbakhsh et al., "Convolutional neural networks for medical image analysis: Full training or fine tuning?," *IEEE Trans. Med. Imag.*, vol. 35, no. 5, pp. 1299–1312, May 2016.
[25] E. Parisotto, J. L. Ba, and R. Salakhutdinov, "Actor-mimic: Deep multitask and transfer reinforcement learning," 2015, *arXiv:1511.06342*.
[26] Z. Xu, K. Wu, Z. Che, J. Tang, and J. Ye, "Knowledge transfer in multi-task deep reinforcement learning for continuous control," in *Proc. 34th Int. Conf. Neural Inf. Process. Syst.*, Red Hook, NY, USA, 2020, pp. 15146–15155, Art. no. 1270.
[27] F. Yu, Z. Qin, and X. Chen, "Distilling critical paths in convolutional neural networks," 2018, *arXiv: 1811.02643*.
[28] F. Lai, X. Zhu, H. V. Madhyastha, and M. Chowdhury, "OORT: Informed participant selection for scalable federated learning," 2010, *arXiv: 2010.06081*.
[29] D. Li, T. Salonidis, N. V. Desai, and M. C. Chuah, "DeepCham: Collaborative edge-mediated adaptive deep learning for mobile object recognition," in *Proc. IEEE/ACM Symp. Edge Comput.*, 2016, pp. 64–76.
[30] K. Hsiehet al., "Gaia: Geo-distributed machine learning approaching {LAN } speeds," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 629–647.
[31] C. Bravo and H. Bäckströ, "Edge computing and deployment strategies for communication service providers," White Paper, Ericsson, 2020. [Online]. Available: https://cloud.report/whitepapers/edge-computing-and-deployment-strategies-for-communication-service-providers
[32] A. Bifet and R. Gavalda, "Learning from time-changing data with adaptive windowing," in *Proc. SIAM Int. Conf. Data Mining*, 2007, pp. 443–448.
[33] J. M. Joyce, "Kullback-leibler divergence," in *International Encyclopedia of Statistical Science*. Berlin, Germany: Springer, 2011.
[34] J. N. Rao and A. J. Scott, "The analysis of categorical data from complex sample surveys: Chi-squared tests for goodness of fit and independence in two-way tables," *J. Amer. Statist. Assoc.*, vol. 76, no. 374, pp. 221–230, 1981.
[35] J. Lin, "Divergence measures based on the Shannon entropy," *IEEE Trans. Inf. Theory*, vol. 37, no. 1, pp. 145–151, Jan. 1991, doi: 10.1109/18.61115.
[36] R. Beran et al., "Minimum hellinger distance estimates for parametric models," *Ann. Statist.*, pp. 445–463, 1977.
[37] X. Zeng and D. S. Yeung, "Sensitivity analysis of multilayer perceptron to input and weight perturbations," *IEEE Trans. Neural Netw.*, vol. 12, no. 6, pp. 1358–1366, Nov. 2001.
[38] S.-I. Amari, "Natural gradient works efficiently in learning," *Neural Comput.*, vol. 10, no. 2, pp. 251–276, 1998.
[39] A. A. Rusu et al., "Progressive neural networks," 2016, *arXiv:1606.04671*.
[40] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. Y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proc. Artif. Intell. Statist.*, 2017, pp. 1273–1282.
[41] W. Dai, Q. Yang, G.-R. Xue, and Y. Yu, "Boosting for transfer learning," in *Proc. 24th Int. Conf. Mach. Learn.*, 2007, pp. 193–20.
[42] M. Gao, L. Ma, H. Liu, Z. Zhang, Z. Ning, and J. Xu, "Malicious network traffic detection based on deep neural networks and association analysis," *Sensors*, vol. 20, no. 5, 2020, Art. no. 1452.
[43] S. Caldas, J. Konečny, H. B. McMahan, and A. Talwalkar, "Expanding the reach of federated learning by reducing client resource requirements," 2018, *arXiv: 1812.07210*.
[44] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the KDD CUP 99 data set," in *Proc. IEEE Symp. Comput. Intell. Secur. Defense Appl.*, 2009, pp. 1–6.
[45] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once for all: Train one network and specialize it for efficient deployment," 2019, *arXiv:1908.09791*.
[46] A. Li et al., "LotteryFL: Empower edge intelligence with personalized and communication-efficient federated learning," in *Proc. IEEE/ACM Symp. Edge Comput.*, 2021, pp. 68–79.
[47] S. Cai, Y. Zhao, Z. Liu, C. Qiu, X. Wang, and Q. Hu, "Multi-granularity weighted federated learning in heterogeneous mobile edge computing systems," in *Proc. IEEE 42nd Int. Conf. Distrib. Comput. Syst.*, 2022, pp. 436–446.
[48] A. M. Abdelmoniem, A. N. Sahu, M. Canini, and S. A. Fahmy, "Resource-efficient federated learning," 2021, *arXiv:2111.01108*.
[49] R. Hong and A. Chandra, "DLion: Decentralized distributed deep learning in micro-clouds," in *Proc. 30th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2021, pp. 227–238.
[50] J. Yao, F. Wang, K. Jia, B. Han, J. Zhou, and H. Yang, "Device-cloud collaborative learning for recommendation," in *Proc. 27th ACM SIGKDD Conf. Knowl. Discov. Data Mining*, 2021, pp. 3865–3874.
[51] I. Rafegas, M. Vanrell, L. A. Alexandre, and G. Arias, "Understanding trained CNNs by indexing neuron selectivity," *Pattern Recognit. Lett.*, vol. 136, pp. 318–325, 2020.
[52] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," 2013, *arXiv:1312.6034*.
[53] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *Proc. 14th {USENIX} Symp. Netw. Syst. Des. Implementation*, 2017, pp. 613–627.

Harshit Daga is currently working toward the PhD degree in the School of Computer Science, Georgia Tech, specializing in systems support for distributed machine learning with the edge-cloud continuum.

Yiwen Chen received the MSCS degree with Georgia Tech and joined Amazon.

Aastha Agrawal received the MSCS degree with Georgia Tech and joined Salesforce.

Ada Gavrilovska is an associate professor with the School of Computer Science at Georgia Tech. Her research is supported by the National Science Foundation, the US Department of Energy, the JUMP programs by the Semiconductor Research Corporation and DARPA, and several industry grants. In 2024, she will be program co-chair for the USENIX Symposium on Operating Systems Design and Implementation (OSDI'24).