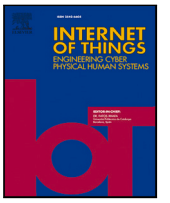
Internet of Things 27 (2024) 101263

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot



Research article

Resource-aware in-edge distributed real-time deep learning

Amin Yoosefi [a], Mehdi Kargahi [a,b,*]

[a] School of Electrical and Computer Engineering, College of Engineering, University of Tehran, Tehran, Iran
[b] School of Computer Sciences, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran

ARTICLE INFO

Keywords:
Distributed deep learning
Edge computing
Real-time embedded systems
Resource constraints

ABSTRACT

Deep neural networks (DNNs) are widely used in IoT devices for applications like pattern recognition. However, slight variations in the input data may cause considerable accuracy loss, while capturing all data variations to provide a rich training dataset is almost unrealistic. Online learning can assist by offering to continue adapting the model to the data variations even during inference, however at the expense of higher resource demands, namely a challenging requirement for resource-constrained IoT devices. Furthermore, training on a data sample must be concluded in a timely manner, to have the model updated for subsequent data inferences, compelling the data inter-arrival time as a time constraint. Distributed learning can mitigate the per-device resource demand by splitting the model and placing the partitions on the IoT devices. However, the previous distributed learning studies primarily aim to improve the throughput (through accelerating the training by large-scale CPU or GPU clusters), with less attention to the timeliness constraints. This paper, however, pays attention to some application-specific constraints of timeliness and accuracy under IoT device resource limitations using modular neural networks (MNNs). The MNN clusters the input space using a proposed online approach, where a module is specialized to each of the dynamic data clusters to perform inference. The MNN adjusts its computational complexity adaptively by adding, removing, and tuning the module clusters as new data arrives. The simulation results show that the proposed method effectively adheres to the application constraints and the device resource limitations.

1. Introduction

Machine learning (ML) algorithms are widely requested by IoT end devices at the network edge for data inference in various pattern recognition applications. As a type of ML algorithms, *deep neural networks (DNNs)* have proven superior accuracy compared to the other variants [1,2]. However, this superior performance of DNNs comes at the expense of high resource demands in terms of memory, computing power, and energy. On the other hand, IoT end devices requesting data analysis are usually subject to strict resource limitations. The edge resource-constrained devices range from battery-operated smartphones (with mobile processors) to much weaker devices such as small IoT sensor nodes (with low-power processors).

Although numerous research studies have focused on reducing inference resource demands to enable DNN inference at the edge [3–9], edge-level DNN training is more challenging as it involves substantially higher resource demands [10]. The usual approach is thus to first train the DNN on powerful resource-rich Cloud servers, and then deploy it on IoT resource-constrained end devices for data inference.

By separating the training and inference phases, a model trained in the cloud can no longer benefit from parameter adaptation once deployed at the edge for inference. Such a deployed DNN may experience accuracy degradation due to the DNNs' sensitivity to

* Corresponding author at: School of Electrical and Computer Engineering, College of Engineering, University of Tehran, Tehran, Iran.
E-mail addresses: a.yoosefi@ut.ac.ir (A. Yoosefi), kargahi@ut.ac.ir (M. Kargahi).

https://doi.org/10.1016/j.iot.2024.101263
Received 18 April 2024; Received in revised form 6 June 2024; Accepted 23 June 2024
Available online 27 June 2024

slight perturbations in the input data caused by natural variations when capturing the input data [11–14]. For example, changes in weather conditions reflected in an input image may confuse the DNN employed in an autonomous vehicle [15]. Further, capturing all possible variations to provide some rich training dataset is almost impractical. This highlights the benefits of employing models capable of adapting to new data while being used for inference, representing the online learning paradigm.

Online learning performs inference and training in an interleaved fashion. Moreover, training on a data sample must have been concluded to have the model updated for the subsequent data inferences. This requirement signifies a timeliness constraint: Inference and training for each data sample must be completed before receiving the next data. However, it should be noted that online training of DNNs is resource-intensive.

Cloud computing [16] is a solution to deal with the high resource demands of DNNs in online learning by outsourcing the computations to the cloud. However, data transmission to the cloud may raise concerns like (i) delayed responses, threatening the timeliness requirement of the online learning application, and (ii) privacy and security breaches to sensitive data, such as face images and speech, due to exposure to potential unauthorized actors during the computation offloading.

Edge computing [17] addresses the above concerns associated with cloud computing by eliminating the long-distance data offloading and performing the training and inference near data sources instead. However, the intense resource demand of the DNN training is still a barrier when a single resource-limited end device is responsible for the computations. Distributed learning [18–34] and approximation [35–40] are two main courses of the literature to address the high resource demands of DNNs for deployment at the edge.

Approximation leverages the high inherent architectural redundancy in DNNs to mitigate their computational complexity. However, the model architecture is usually pre-determined by the designer and fixed during the training, signifying two drawbacks: (i) The designer may make a sub-optimal architectural design which may still be over-parameterized for a given problem, and (ii) in an online learning application (where new data continually comes), the model does not adjust its architecture to adapt to new data.

Distributed learning exploits the cumulative resource capacities of end devices at the network edge for learning when the resource capacity of a single end node is not adequate. This implies distributing the DNN model across end nodes and performing the training in a distributed manner. However, the previous studies target distributed deployment over large-scale CPU or GPU clusters, mainly to accelerate the training and increase the throughput. Moreover, timeliness, as the primary contributor to an online learning application, is not considered. This paper is tailored towards model-based distributed learning to benefit from resources collectively available at the edge.

We use *modular neural networks (MNNs)*, which follow the *divide and conquer principle*, to offer self-adaptive model-based distributed learning. In these neural networks, the main problem is decomposed into a set of simpler *sub-problems*, where each sub-problem is handled by a separate *sub-network* or *module*. With the problem decomposition, a form of clustering is applied to the input space of the main problem, and each module from the MNN is specialized to learn the inputs belonging to a resulting *input sub-space*. Therefore, when an input is fed into the MNN, only those modules that are specialized to the input are activated to make the inference. The final output of the MNN is obtained by applying an integration logic to the outputs of the activated modules. For training, in a similar fashion, only the activated modules learn the input data in parallel.

Modular learning allows the decomposition of a large resource-intensive learning workload among a set of smaller manageable modules tailored towards meeting the resource limitations of individual end nodes. However, when it comes to an online learning application in the presence of resource limitations, to ensure that the timeliness constraint of each round of training is met, the number of activated modules in each round must be controlled with regard to the resource constraints (the available end nodes and their internal resources), which in turn necessitates appropriate input space clustering. Some works [41,42] use MNNs to learn the non-linear behavior of a system adaptively. These studies, however, do not take care of the concerns of timeliness or resource constraint. Thus, they suppose no limitation on the employed MNN size, i.e. on the number of modules and the internal resource demand of each module.

This study addresses high resource demands in online training of DNNs at the network edge by proposing online model-based distributed learning using an MNN. It is supposed that the training data is sampled online at a specific rate, introducing the sampling interval as a timeliness constraint for each round of training. The number of modules activated within the MNN is limited by the number of end devices available to the online learning application and the timeliness constraint corresponding to the online data sampling rate. The MNN can grow and shrink by adding and removing the modules as well as by adjusting the input space cluster on which each module is activated. This enables the MNN to adaptively adjust its computational complexity as new data arrives.

Adjusting the input space cluster corresponding to each module affects the MNN's accuracy. A module with too many training samples with respect to its complexity is not flexible enough to learn patterns hidden in its data. On the other hand, a module with too few training samples may be overly sensitive to noise and unable to generalize well to new data. A lack of balance between a module's architecture complexity and the number of samples it learns from may lead to accuracy degradation, which may be intolerable by the application. Therefore, the input space clustering policy employed in our MNN adjusts the modules' clusters to strike a good balance between the module complexity and the number of training samples inside the corresponding cluster. This way, the constraints of accuracy, timeliness, and number of end nodes are considered together.

In summary, the contributions of this paper are as follows:

- Presenting an edge-level real-time training method of DNNs based on MNNs, capable of adaptively managing its computational complexity by adding, removing, and adjusting the module clusters as new data arrives;
- subjecting the online training to constraints both at the resource level (computational power and memory) and at the application level (accuracy and timeliness);

- presenting an online clustering algorithm that meets the accuracy constraint by adjusting the module cluster radii to strike a good balance between the module's sample size and computational complexity; and
- providing a cluster monitoring and management approach to meet the timeliness and resource constraints by ensuring that the number of activated modules does not exceed the pre-determined number of end devices.

The rest of this paper is organized as follows. Section 2 describes our target application, i.e. an online multi-class classification problem. It also presents preliminaries regarding modular learning, online training of an MNN, and the definition of accuracy. Section 3 describes the platform and application models, representing the supposed edge model, and demonstrates the mapping between the models. It then presents the formal problem statement. Section 4 discusses the proposed approach for input space clustering. The experimental results are discussed in Section 5. Section 6 reviews the related work, and Section 7 concludes the paper.

2. Background

2.1. Online multi-class classification

We study a *multi-class classification* problem that uses *artificial neural networks (ANNs)* to estimate some unknown *mapping function* f from an *input space* $\mathcal{X}$ to an *output space* $\mathcal{Y}$. The function is represented as $f : \mathcal{X} \rightarrow \mathcal{Y}$, in which $\mathcal{X}$ is of an l-dimensional real-valued space $\mathbb{R}^l$ and $\mathcal{Y}$ is a discrete set of M distinct *class labels*, formulated as $\mathcal{Y} = \{1, 2, \dots, M\}$. An example of f is an object recognition application which maps an input image of an object to the corresponding object category label.

The employed ANN estimates f by $\hat{f}$, as:

$$\hat{Y} = \hat{f}(X; \theta), \tag{1}$$

where θ refers to the *parameter set* within the ANN; $X = [X^1, X^2, \dots, X^l] \in \mathcal{X}$ is an l-dimensional *input vector*; and $\hat{Y} \in \mathcal{Y}$ represents the output corresponding to the input vector X, predicted by the ANN.

Training the ANN is supervised by a *sample set* $S^{onl} = \{(x_k, y_k) | k = 1, 2, \dots, K\}$ of size K, with the samples arriving sequentially and periodically. The training is defined as an optimization problem that adjusts the network parameter set to minimize the *cost function* $Err(\theta)$, i.e.:

$$\underset{\theta}{\text{minimize}}\ Err(\theta), \tag{2}$$

in which $Err(\theta)$ is defined as the average number of *misclassifications* over the sample set S^{onl}, known as the *error rate*. The error rate is formulated as follows:

$$Err(\theta) = \frac{1}{K} \sum_{k=1}^{K} I(y_k \neq \hat{y}_k), \tag{3}$$

wherein y_k and $\hat{y}_k$ are the *actual* and *predicted outputs*, respectively; and $I(y_k \neq \hat{y}_k)$ is an indicator variable that returns 1 if $y_k \neq \hat{y}_k$ and returns 0 otherwise.

2.2. Modular learning

Conventionally, an existing DNN architecture (like AlexNet or VGG) is chosen to learn a given mapping function (like the one involved in 100-class and 1000-class object recognition problems in CIFAR-100 and ImageNet datasets, respectively). However, the huge fixed number of parameters within the pre-determined architecture may be much larger than what is needed for the mapping function, increasing the network resource requirements, emphasizing the benefits of a neural network capable of managing its computational complexity at run-time. To this aim, we focus on the modularity concept in MNNs.

An MNN uses sub-networks or modules to estimate the complex mapping function f of the main problem based on the divide and conquer principle. According to this principle, with "*divide*", the main problem is decomposed into simpler sub-problems, with the *sub-function* inherent in each sub-problem estimated by a separate module. In fact, each module is only specialized in a specific sub-space of the f's input space. This implies some clustering on the input space. With "*conquer*", the outputs from all the modules are integrated to form the final output. Overall, by employing this principle, a complex problem is reduced into a set of much less-complicated sub-problems.

Fig. 1 illustrates the structure of an MNN, composed of three parts: (i) The *input space clustering part*, (ii) the *deep learning part*, employing a number of parallel DNN modules, and (iii) the *output integrating part*. Each part is described below.

Input Space Clustering Part. This part performs the "divide" phase by clustering the input space, where each *cluster* corresponds to a specific sub-problem of the main problem. To apply the clustering, we employ a *radial basis function (RBF)* layer.

The RBF layer is composed of *RBF neurons*. The number of existing RBF neurons is denoted by n. We associate each neuron with a unique integer identifier (ID) to refer to it in our formulation. Let C denote the ID set of all existing RBF neurons, i.e. $|C| = n$. Each neuron $i \in C$ comprises: (i) An l-dimensional *center vector* $c_i \in \mathbb{R}^l$ and (ii) a *radius* $r_i \in \mathbb{R}$. An RBF neuron differs from an ordinary neuron in that instead of computing the weighted sum of its inputs, it computes the *Euclidean distance* between its center vector and the input vector. By applying an *activation function* to the computed distance, the output of the neuron is generated. The activation

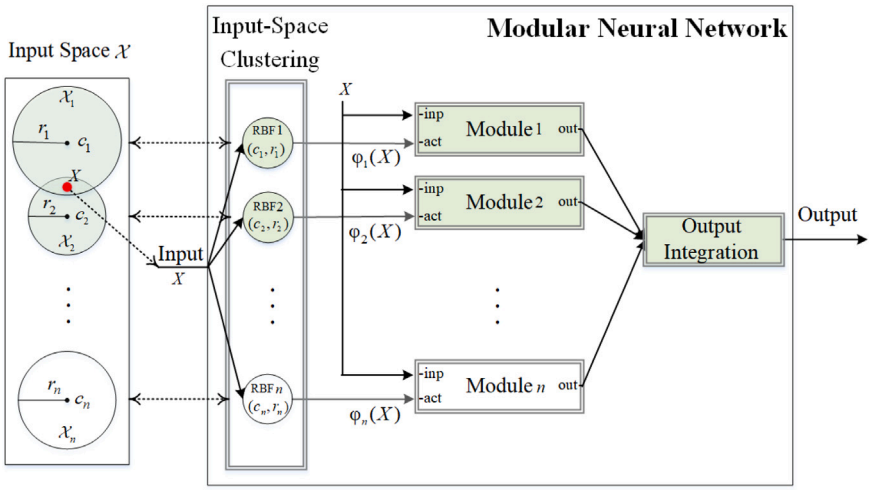


Fig. 1. Illustrating different parts of a modular neural network: (i) The RBF neuron set clusters the input space $\mathcal{X}$ into spherical cluster areas $\mathcal{X}_1, \ldots, \mathcal{X}_n$, (ii) the module set performs the deep learning in parallel, and (iii) the output integration part generates the final output. Elements activated by an input X are highlighted in green.

function is usually in the form of a *Gaussian function*. With $\varphi_i(X)$ denoting the output of the neuron given the input vector X, we have:

$$\varphi_i(X) = \exp(-\frac{\|X - c_i\|^2}{r_i^2}), \tag{4}$$

wherein $\|.\|$ denotes *Euclidean norm*.

As (4) implies, $\varphi_i(X)$ associates positive near-zero values to the input vectors of a great distance from the center and associates values near to one to the input vectors of close vicinity to the center. Further, $\varphi_i(X)$ can be rewritten as a hard-limited function, as demonstrated below:

$$\varphi_i(X) = \begin{cases} 1, & \text{if } \|X - c_i\| \leq r_i \\ 0, & \text{otherwise.} \end{cases} \tag{5}$$

Eq. (5) introduces $\varphi_i(X)$ as a membership function, determining whether a given input X falls within a hyperspherical cluster with center c_i and radius r_i or not. The set of points within this cluster is denoted by $\mathcal{X}_i \subseteq \mathcal{X}$ and formulated as:

$$\mathcal{X}_i = \{X \in \mathcal{X} \mid \|X - c_i\| \leq r_i\}, \quad i \in C. \tag{6}$$

Given an input vector X, the RBF neurons having X within their corresponding hyperspherical clusters (i.e. $X \in \mathcal{X}_i$) fire by generating 1 as the output. We use the words RBF neuron and cluster interchangeably throughout the paper.

Deep Learning Part. This part carries out the learning process required to estimate f by employing DNN modules. There is a one-to-one association between the modules and the RBF neurons. As a result, the corresponding module and RBF neuron have the same ID values. Each module is responsible for learning the input vector belonging to the cluster of its corresponding RBF neuron when the neuron fires. Therefore, this part uses n DNN modules to generate the estimate $\hat{f}$ in terms of n sub-functions, i.e. $\hat{f}_i$, $i \in C$, in which $\hat{f}_i$ refers to the sub-function estimated by module i. Each module is associated with a distinct set of parameters, denoted by θ_i, $i \in C$, in which $|\theta_i|$ represents the number of parameters for that module.

For an input vector X fed into the RBF layer, the fired RBF neurons activate their corresponding modules in the deep learning part for parallel training on X. The number of activated modules represents the amount of processing involved with the received input vector X. If the input vector does not belong to any existing cluster, the MNN must assign one cluster to include the input vector by deciding on whether expanding one of the existing clusters or adding a new one.

Output Integrating Part. This part performs the "conquer" phase of the divide and conquer principle. It applies an integration logic to the outputs coming from the activated modules of the previous part, and this way, generating the final output:

$$\hat{Y} = \hat{f}_{integ}(\{\varphi_i(X)\hat{f}_i(X;\theta_i)\}_{i \in C}), \tag{7}$$

in which $\hat{f}_{integ}$ represents the integration function.

2.3. Online training of a modular neural network

Online learning of an MNN involves adjusting two sets of parameters: (i) The centers and radii of the RBF neurons within the RBF layer and (ii) the weights and biases within the modules and the integration part. Clustering algorithms are usually used to

determine the parameter values of the RBF layer. The weights and biases within the rest of the MNN can be adjusted using a backpropagation algorithm. The following paragraphs demonstrate training an MNN in two subsequent steps: (i) Training the RBF layer and (ii) training the modules and the integration part.

Adjusting Centers and Radii of the RBF Neurons. Clustering algorithms are used to organize a collection of *data points* $x_k \in \mathcal{S}^{onl}$ into clusters by identifying some points in the collection's domain $Z \in \mathcal{X}$ as cluster centers. Selecting points as cluster centers can be decided based on their *potential* values. Potential is a measure of a point's spatial proximity to all points in the input data collection. Since the data points in online learning are provided incrementally over time, the employed clustering algorithm must facilitate the *recursive* calculation of potential values, as demonstrated in [43]. According to [43], the potential of a particular point is defined using a simple monotonic function inversely proportional to the average distance of that point to all the data points within the data collection. As the selection of cluster centers is not generally restricted to the data collection points, the monotonic function proposed in [43] is rewritten as follows:

$$P_k(Z) = \frac{1}{1 + \frac{1}{k}\sum_{i=1}^{k}(Z - x_i)^\top (Z - x_i)}, \quad Z \in \mathcal{X}, \quad k = 1, 2, 3, \dots, \tag{8}$$

in which $P_k(Z) \in (0, 1]$ refers to the potential value of a particular point $Z \in \mathcal{X}$, calculated at the kth round; x_i is the data point already received at the ith round, $i \leq k$; and $(Z - x_i)^\top (Z - x_i)$ is the squared distance between points Z and x_i, with $\top$ representing the transpose operator.

The function in (8) effortlessly allows recursive calculation of the potential value for a particular point Z. The function can be rewritten in two different recursive forms depending on whether the potential value of Z at the previous round is available or not. When the previous potential value of Z is not available, the recursive form is as below:

$$P_k(Z) = \frac{k}{k(Z^\top Z + 1) - 2\eta_k Z + \sigma_k}, \quad k = 1, 2, 3, \dots, \tag{9}$$

wherein η_k and σ_k are computed as:

$$\eta_k = \eta_{k-1} + x_k^\top, \tag{10}$$

$$\sigma_k = \sigma_{k-1} + x_k^\top x_k, \tag{11}$$

in which η_0 is a zero-valued vector and $\sigma_0 = 0$. However, when the previous potential value is available, the recursive form is:

$$P_k(Z) = \frac{k P_{k-1}(Z)}{(k-1) + (\zeta_k(Z) + 1) P_{k-1}(Z)}, \quad k = 2, 3, \dots, \tag{12}$$

where $\zeta_k(Z)$ is computed as below:

$$\zeta_k(Z) = (Z - x_k)^\top (Z - x_k), \tag{13}$$

and we use (9) for $P_1(Z)$.

Thus, the algorithm given in [43] starts by assuming the potential value of 1 for the first data point and considering it as the first cluster center. For each new data point x_k received in the upcoming rounds, the recursive form in (9) is used to calculate its potential value. Each new data point affects the potential values of the existing cluster centers because, by definition, the potential value of a point depends on its distance to all data points. As a result, upon receiving a new data point x_k, the recursive form in (12) is used to update the potential values of the existing cluster centers. The updated potential values are saved for the upcoming rounds.

As (8) implies, a point surrounded by many close data points has a higher potential value. Such a point is a candidate cluster center as it is more descriptive and has higher summarization power than the other points with lower potential values. Therefore, the potential values can be used to evaluate the appropriateness of a newly arrived data point x_k to be a cluster center. Each x_k can be a center point in two ways: (i) The center of a new cluster, i.e. incrementing the number of clusters, or (ii) the center of an existing cluster, i.e. shifting that cluster. Thus, an online clustering algorithm must effectively decide on the appropriate operation choice upon receiving a new data point.

Adjusting Parameters of the Modules and the Integration Network. The deep learning and the output integrating parts form a single ANN that can be trained using a backpropagation algorithm like an ordinary ANN, with backpropagation only applied to the activated modules of the MNN. This implies that only the parameters of the activated modules are updated through backpropagation, while the parameters in other modules remain intact.

For an ANN, the backpropagation generally consists of two phases: (i) The *forward pass* and (ii) the *backward pass*. In the forward pass phase, an input to the ANN passes through the network, and the corresponding output is generated. The backward pass follows two steps: (i) The *error backpropagation* and (ii) the *gradient computation*. The first step computes the errors corresponding to the neurons or *activations* by backpropagating the error between the actual and predicted outputs through the network. The second step uses the computed errors to calculate the gradients and parameter changes, and then the parameter values are updated accordingly. With this demonstration, the backpropagation algorithm for an MNN is as follows:

- In the forward pass phase, an input X is forward passed in parallel through all the activated modules within the MNN. The resulting outputs from those modules are then forward passed through the integration network, and the predicted class label $\hat{Y}$ is generated.

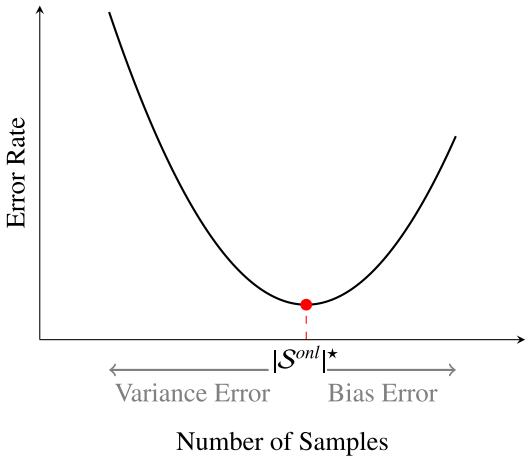


Fig. 2. General form of a module learning curve.

- In the backward pass phase, the set of error values corresponding to the output neurons, denoted by $\mathcal{L}$, is computed based on the actual output (Y) and the predicted output ($\hat{Y}$). Next, the error backpropagation and gradient computation steps are performed for both the integration network and the activated modules, followed by updating their parameters. The error set corresponding to module i is denoted by $\mathcal{L}_i$, obtained after $\mathcal{L}$ is backpropagated through the integration network.

2.4. Accuracy

The goal of training an ANN is to minimize the error rate over the sample set, as demonstrated in (2). The resulting estimate $\hat{f}$ leads to an error rate, which is driven by two competing factors: The *variance error* and the *bias error*.

The variance error is originated from the sample set S^{onl}, in that using a different sample set may result in a distinct $\hat{f}$. When the complexity of the ANN increases, the resulting network flexibility in learning may become excessively high with respect to the size of the sample set. Then, the ANN inspects the training data very closely, and thus, the estimate $\hat{f}$ provided by the ANN is highly tailored towards that specific sample set. In such a situation, the ANN has a high variance error, and it is said to be *overfitted*.

The bias error, on the other hand, relates to the insufficient complexity of the ANN, namely when approximating the mapping function f is done with an ANN of complexity much lower than the complexity of f. In such a situation, the ANN is said to be *underfitted*.

Increasing the complexity of the ANN makes the ANN more flexible, and thus, the bias error is mitigated. However, when the flexibility is exceptionally high compared to the size of the sample set S^{onl}, the variance error increases. As a result, adjusting the sample set size with respect to the ANN's complexity introduces some trade-off between the bias and variance errors, known as the *bias–variance trade-off* [44].

This paper inspects the bias–variance trade-off at the module level. With a given module architecture, the number of samples learned by a module, i.e. the module's sample set size, is leveraged to adjust the trade-off. A module's sample set size is determined based on the module's learning curve. *Learning curves* depict the dependence of the error rate on the sample set size. Fig. 2 shows the general form of a learning curve for a module. When training the MNN, modules other than the target module are frozen, i.e. their parameter sets remain intact. Small sample set sizes lead to high variance errors. The variance error, and thus, the error rate decreases by providing more samples. However, from some size onwards, i.e. $|S^{onl}|^{\star}$, the error rate increases as the sample set size increases. This behavior is because the module architecture is not sufficiently complex to learn the enlarged sample set. Fig. 2 introduces $|S^{onl}|^{\star}$ as an optimal sample size, empirically shown to minimize the error rate by balancing the bias and variance errors.

3. Edge model and problem statement

This section presents the edge model, including the platform and application models. It then discusses the stages of operation within the edge, and finally, it provides the problem statement.

3.1. Edge model

This subsection discusses our edge model from the following aspects: (i) The platform model, (ii) the application model, and (iii) the mapping model. The platform model describes the employed computation and communication hardware and its resource capacities. Our MNN workload and its associated constraints, i.e. accuracy and timeliness, are demonstrated in the application model. Finally, the mapping model clarifies how our MNN workload is mapped onto the platform model considering the running stages involved during one round of training.

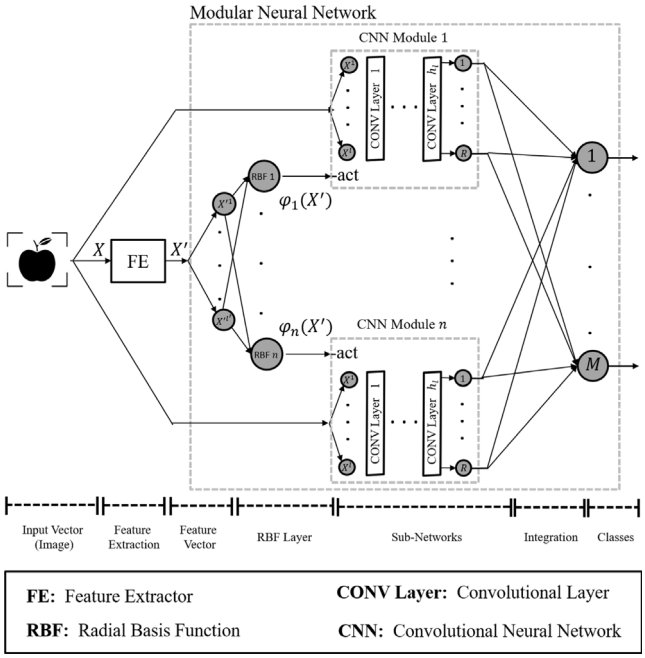


Fig. 3. Illustration of the structure of our MNN.

3.1.1. Platform model

The platform model employs N_{node} end nodes to accommodate the application's workload. The end nodes are assumed to be homogeneous in terms of resources. Each end node is equipped with four types of resources: (i) Computation resource with capacity of ρ GFLOPS (giga floating-point operations per second), (ii) flash memory with capacity of W^{flash} bits, (iii) RAM with capacity of W^{RAM} bits, and (iv) communication resource with bandwidth capacity of β bps. It is assumed that the communication links and the end nodes are reliable. The theoretical peak GFLOPS for an end node's processing core is computed as follows [45]:

$$\rho\ (GFLOPS) = \frac{flops}{cycle} \times \frac{cycles}{second}\ (\text{GHz}), \tag{14}$$

in which $^{flops}/_{cycle}$, denoted by ρ_{flop}, refers to the number of floating-point operations performed per cycle, which is a constant for a core of a given microarchitecture [45,46]; and $^{cycles}/_{second}$, denoted by ρ_{freq}, represents the number of cycles performed per second, i.e. the processing core's frequency.

3.1.2. Application model

The application model includes the MNN architecture, as the workload, and its constraints. At the application level, two constraints exist: (i) Accuracy and (ii) timeliness.

The MNN Architecture. The structure of our MNN is depicted in Fig. 3. It comprises the RBF layer, the integration network, and the module set. The platform model can accommodate at most N_{mod} modules, i.e.:

$$|C| \leq N_{mod}, \tag{15}$$

and N_{mod} is determined by the flash memory constraints.

A module is a *convolutional neural network (CNN)*, composed of a sequence of *convolutional (CONV)* layers. Each layer is defined by a set of hyperparameters, as described in Appendix A. The number of layers and the number of kernels in each layer are known as the *depth* (h_l) and *width* (h_p) of the CNN, respectively [47]. The arrangement of the CONV layers with their hyperparameter values determines the module's architecture. All modules are assumed homogeneous in terms of architecture. Thus, they all have the same number of output neurons, R.

As illustrated in Fig. 3, the RBF layer is not fed directly with a received input vector $X \in \mathcal{X}$. Instead, it is fed with the corresponding feature vector $X' = [X'^1, X'^2, \ldots, X'^{l'}] \in \mathcal{X}'$, in which $\mathcal{X}'$ refers to the l'-dimensional real-valued feature space $\mathbb{R}^{l'}$, $l' < l$, extracted using a feature extractor. A feature extractor allows replacing the redundant, noisy, and large-scale input space with the more informative compact feature space. In other words, the RBF layer applies clustering to the feature space, and thus, the clusters associated with the modules belong to the feature space, as demonstrated below:

$$\mathcal{X}'_i = \{X' \in \mathcal{X}' \mid \|X' - c_i\| \leq r_i\}, \quad i \in C, \tag{16}$$

in which $\mathcal{X}'_i \subset \mathcal{X}'$ refers to the cluster associated with module i; thus, we redefine c_i as $c_i \in \mathbb{R}^{l'}$.

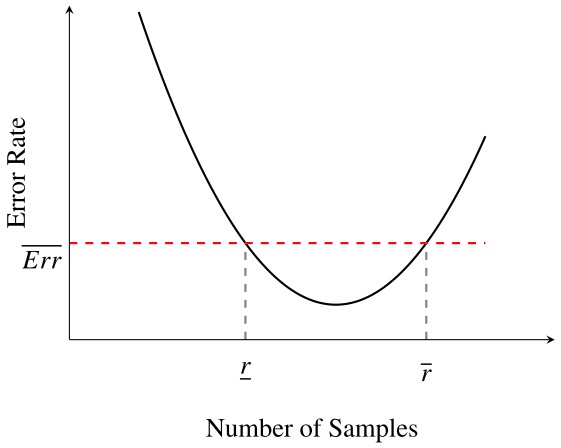


Fig. 4. Determining the lower and upper bounds on the module cluster radius according to the learning curve.

Therefore, given an input vector X, a subset of modules whose clusters contain the corresponding X' is activated for parallel training on X. Let $\mathcal{C}_{act}(X') \subseteq \mathcal{C}$ represent the ID set of the RBF neurons activated by X', as formulated below:

$$\mathcal{C}_{act}(X') = \{i \in \mathcal{C} \mid \|X' - c_i\| \leq r_i\}. \tag{17}$$

Since the RBF layer learns the problem decomposition, the rationale behind each extracted sub-problem, and thus, the rationale behind the outputs of each module is not necessarily known. Therefore, an ANN is used to learn the integration logic. The integration ANN is a fully-connected (FC) layer with $|\mathcal{C}|R$ input neurons and M output neurons.

The Accuracy Constraint. The application is supposed to respect some given maximum error rate of $\overline{Err} \in [0, 1]$, i.e. a constraint on the cost function of (3). The MNN's error rate is affected by the number of samples each module learns, as shown by their learning curves. Further, the module's cluster radius controls the number of training samples it receives. Therefore, the learning curve can be transformed to directly depict how the MNN's accuracy varies with increasing the module cluster radius.

Algorithm 10 in Appendix B estimates the learning curve for each module of the MNN. The modules' learning curves are then aggregated to obtain a single curve. To account for the uncertainty in estimating a module learning curve, we use Algorithm 11 in Appendix B to form a confidence interval for the curve resulting from Algorithm 10. The confidence interval is composed of upper- and lower-bound learning curves. The resulting upper-bound curve is used to transform the upper bound $\overline{Err}$ on the cost function into lower and upper bounds on a module's radius, denoted by $\underline{r}$ and $\bar{r}$, respectively, as illustrated in Fig. 4. Therefore, the accuracy constraint is formulated as:

$$\underline{r} \leq r_i \leq \bar{r}, \quad i \in \mathcal{C}. \tag{18}$$

The Timeliness Constraint. The training data is assumed to be sampled online periodically at a certain rate, and the learning process for each data sample is expected to be completed before the start of the next sampling period. Thus, the sampling rate determines the inter-arrival of learning samples, which is denoted by T. The timeliness constraint is defined by introducing a relative deadline of T for each training round.

3.1.3. The running stages

The end nodes are organized in a *single-master multiple-slave* topology, as shown in Fig. 5. With this topology, among the N_{node} nodes, one end node is employed as a single master node to supervise the learning process. This node has the following responsibilities: (i) Hosting the MNN's parameter set, including the parameter sets of all modules, the RBF layer, and the integration network, (ii) providing training samples, (iii) feature extracting a received input data, (iv) training the RBF layer by adjusting centers and radii of the RBF neurons employing an online clustering approach, (v) supervising distributed training of the modules on slave nodes, and (vi) training the integration network. The remaining $N_{node} - 1$ nodes are used to act as slave nodes, which contribute to training the MNN by performing local training of their assigned module. The slave nodes communicate with the master node in a centralized manner, meaning that the data communication flow is only allowed between the master node and the slave nodes, and none of the slave nodes can directly communicate with each other.

Without loss of generality, we assume that each slave node can train only one module within the inter-arrival time of T. Thus, the module's architecture must be designed such that the timing and resource requirements calculated based on this restriction align with the corresponding constraints. Therefore, the maximum number of modules activated must not exceed the total number of slave nodes, i.e.:

$$\forall X' \in \mathcal{X}' : |\mathcal{C}_{act}(X')| \leq N_{node} - 1. \tag{19}$$

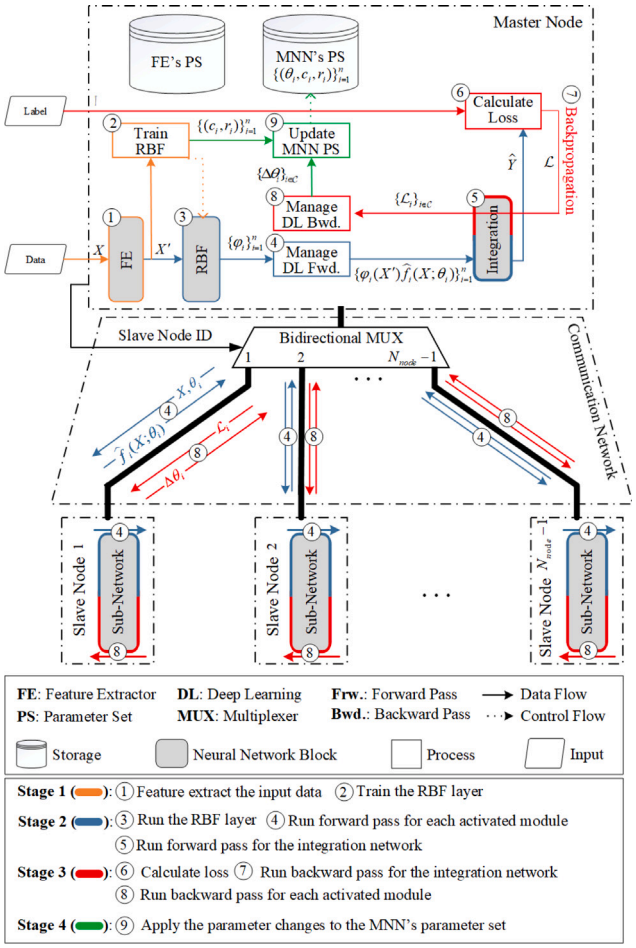


Fig. 5. Single-maser multiple-slave network topology of the edge model. The master node hosts the MNN's and FE's parameter sets and supervises the online learning process, while the slave nodes perform local training of their assigned modules. The communication flow is only allowed between the master node and individual slave nodes.

Algorithm 1 (Learn_Online): Conduct the MNN online training.

Input: Pre-trained feature extractor FE, pre-trained modular neural network MNN^{init}, minimum and maximum cluster radii $\underline{r}$ and $\bar{r}$, training sample set S^{onl}

for $k \in \{1, 2, ..., |S^{onl}|\}$ **do**

Stage 1:

- $s_k = (x_k, y_k) \leftarrow$ select next data sample from S^{onl};
- $x'_k \leftarrow$ feature extract x_k using FE;
- Train the RBF layer using the proposed clustering algorithm;
- Reflect clustering changes to the MNN architecture;

Stages 2-4:

- Perform distributed training of MNN^{init} for (x_k, y_k);

Algorithm 1 demonstrates our MNN's online training. The inputs to the algorithm are a feature extractor FE, a pre-trained MNN, and the minimum and maximum radius values resulting from the bias–variance trade-off analysis. FE is used to extract the corresponding feature vector from a received input vector. The pre-trained MNN's parameter set is used to initialize the parameters of new modules instantiated at run-time. In other words, when we have a new cluster and need to add a new module to our MNN, we initialize the new module's parameter set with the parameter values from the module within the pre-trained MNN whose cluster is closest to the new cluster. Initializing a new module's parameter set with pre-trained parameter values rather than raw parameters

helps improve the convergence and accuracy of online learning. The running stages for one round of the MNN training on a received input vector X is described as follows (see Algorithm 1):

Stage 1. The master node provides the next training sample (X, Y) (Line 2). It applies feature extraction to the input vector X to obtain the corresponding feature vector X' (Line 3). Based on the extracted feature, it adjusts the centers and radii of the RBF neurons using the proposed online clustering algorithm (Line 4), as described in Section 4. If a cluster is added or removed, the corresponding module must be added or removed accordingly (Line 5). Steps 1 and 2 in Fig. 5 depict this stage.

Stage 2. The master node initiates distributed training of the MNN (Line 6). In this stage, it performs the forward pass phase. It feeds the RBF layer with the feature vector X' to determine the modules that are specialized to the training sample. For each activated module, it selects a slave node and sends the module's parameter set (θ_i) and the input vector (X) to it. Next, the slave nodes assigned with a module load their received parameter set into the module architecture already instantiated, as the module architecture is apriori known by all the slave nodes. They feed their module with the received input vector and initiate a forward pass phase. The resulting output $(\hat{f}_i(X;\theta_i))$ is sent back to the master node. With the predicted outputs received from all the slave nodes, the master node performs a forward pass phase on the integration network and generates the final output $(\hat{Y})$. Steps 3 to 5 in Fig. 5 illustrate this stage.

Stage 3. In this stage, the backward pass phase is performed. The error set $(\mathcal{L})$ is calculated considering the predicted $(\hat{Y})$ and actual (Y) values for the final output. The master node performs a backward pass phase on the integration network and computes the error set for each activated module $(\mathcal{L}_i)$. Next, the computed error sets are delivered to the corresponding slave nodes . Based on the error value received, each slave node performs a backward pass phase on its assigned module, and sends the parameter changes $(\Delta\theta_i)$ to the master node. Steps 6 to 8 in Fig. 5 correspond to this stage.

Stage 4. The master node applies the parameter changes received from the slave nodes, as illustrated by Step 9 in Fig. 5.

With the running steps presented, the RAM and flash memory requirements and the completion time of one round of training are modeled in Appendix A. The amount of flash memory required for the master node is denoted by $Flash_{master}$ and computed using (A.14). We assume the input vector and parameter set are loaded into the slave node's RAM upon receiving them, i.e. it is not needed to store them on the node's flash memory. The amounts of RAM that an individual slave node and the master node require for online training of the MNN are denoted by RAM_{slave} and RAM_{master}, respectively, and computed using (A.15) and (A.16), respectively. Further, t_{total} denotes the total execution time of our online training and is computed using (A.17).

Feasibility Check. Given an MNN architecture, both the following conditions must hold to call our MNN training feasible under the given resource and timeliness constraints.

First, each end node of the platform must have adequate memory to accommodate its assigned modules when the worst-case scenario occurs, i.e. the MNN employs the maximum number of modules (i.e. $|C| = N_{mod}$). In other words, we must have:

$$Flash_{master}\Big|_{|C|=N_{mod}} \leq W^{flash}, \tag{20}$$

$$\max(RAM_{slave}, RAM_{master}\Big|_{|C|=N_{mod}}) \leq W^{RAM}. \tag{21}$$

Second, the calculated time requirement must meet the timeliness constraint, i.e.:

$$t_{total}\Big|_{|C|=N_{mod}} \leq T. \tag{22}$$

In other words, given an MNN architecture, training our MNN under the given timeliness and platform resource constraints is marked as infeasible if at least one of the conditions (20), (21), or (22) is not satisfied. In that case, the module's complexity may be leveraged to make training feasible.

3.2. Problem statement

The problem we consider is an online in-edge deep learning that is to manage its computational complexity considering the resource limitations at the network edge while satisfying some application-specific accuracy and timeliness constraints. With the modular learning model presented, the problem reduces to controlling the number of modules activated for a received input instance X, i.e. $C_{act}(X')$, where $C_{act}(X')$ is controlled by the radii and centers of the clusters associated with the modules of the neural network. This signifies the radii and centers of the clusters (i.e. r_is and c_is) as the actual decision variables for the resource-constrained computational complexity management of the learning model. Our research problem is formulated as follows.

Problem Definition. Suppose an unknown mapping function f pertaining to a given M-class classification problem, and assume we have N_{node} homogeneous resource-constrained end nodes, with one node acting as the single master-node and the others acting as the slave nodes. Moreover, the training samples of the set S^{onl} are provided one by one (online) with the inter-arrival time of T. We aim at providing an estimation $\hat{f}$ by incrementally training the MNN over S^{onl} and across the set of end nodes in a distributed manner. The training is subject to the following constraints:

- **Constraint 1**. The limit $N_{node} - 1$ on the maximum number of activated modules, as described in (19);
- **Constraint 2**. The accuracy constraint through the lower and upper bounds on each cluster radius, as formulated in (18);
- **Constraint 3**. The memory constraint, as formulated in (20) and (21); and
- **Constraint 4**. The timeliness constraint of relative deadline T, as formulated in (22).

To meet the constraints, r_i and c_i, $i \in C$, act as our decision variables.

4. The proposed approach

This section presents the proposed online clustering approach, which adjusts the centers and radii of the RBF neurons.

The clustering algorithm decides how a newly received feature data point x'_k should affect the MNN. Affection may appear as *growing*, *shrinking*, or *modifying* the MNN. Growing the MNN can manifest itself in two ways: (i) Growing in the number of employed modules by adding a new cluster at x'_k and (ii) growing in the sub-domain areas covered by modules by extending the radius of an existing cluster to include x'_k. Shrinking and modifying the MNN occur by removing redundant clusters and shifting the center of an existing cluster to x'_k, respectively. Therefore, the clustering algorithm must decide on an appropriate clustering operation upon receiving each feature data, with the decided operation denoted by $op \in \{add, shift, extend, remove\}$. If no operation is required, then $op = None$.

Since clusters may overlap, a new feature data point belonging to an overlapping area activates more than one module for training. On the other hand, we must comply with Constraint 1, which limits the number of activated modules. Thus, the clustering algorithm must continuously monitor the overlapping areas and identify and resolve any group of overlapping clusters whose number exceeds the number of slave nodes. In other words, the clustering algorithm must ensure that the number of clusters within all groups of overlapping clusters does not exceed the number of slave nodes.

Therefore, the clustering algorithm has two main responsibilities when receiving a new feature data point:

1. Deciding on an appropriate clustering operation; and
2. managing the cluster overlaps.

4.1. Deciding on a clustering operation

In this subsection, we discuss the policy for deciding on the clustering operation op upon receiving a new feature data point, which leads the MNN to grow, shrink, or be modified.

4.1.1. Growing and modifying the MNN

Algorithm 2 is used to decide whether to grow or modify the MNN. The choice depends on whether the new feature data point x'_k belongs to any existing cluster (i.e. $C_{act}(x'_k) \neq \emptyset$). If it does, the clustering algorithm looks for any activated cluster that must shift its center to the new feature data point as it may be a more well-suited center for that cluster. The rationale behind this shift is clarified as follows. Since the feature data points are not provided in advance, cluster centers are selected based on the potential value of the data points that have thus far been received. However, it is likely to receive a feature data point with a potential value higher than that of an existing cluster center. As such a feature data point is more descriptive than the existing cluster center, it is better suited to serve as the center. Thus, incrementally receiving feature data points causes the clusters to shift at run-time.

Therefore, upon receiving a feature data point x'_k that belongs to at least one existing cluster, the clustering algorithm checks for the possibility of a *shift operation* (Lines 3–6). It targets the closest activated cluster to x'_k as a candidate for shifting, which is found as follows:

$$\upsilon_{act} = \underset{i \in C_{act}(x'_k)}{\operatorname{argmin}} \{\|x'_k - c_i\|\}. \tag{23}$$

The candidate cluster is then moved to x'_k only if $P_k(x'_k) > P_k(c_{\upsilon_{act}})$.

However, when a received feature data point x'_k does not belong to any existing cluster (i.e. $C_{act}(x'_k) = \emptyset$), no module is activated to learn it. As a result, for such feature data points, the clustering algorithm grows the current MNN structure to include them for learning by choosing one of the two cluster operations: (i) *Add operation* or (ii) *extend operation* (Lines 7–21). First, we introduce these operations and then talk about the policy for choosing between them.

With the add operation, a new cluster is created at center x'_k, with its radius set to the minimum value, i.e. $\underline{r}$. The extend operation targets the closest cluster to x'_k for extending, which is found as follows:

$$\upsilon_{cl} = \underset{i \in C}{\operatorname{argmin}} \{\|x'_k - c_i\|\}. \tag{24}$$

The radius of this cluster must be increased not only to include x'_k but also to cover an area around x'_k, as we may receive more data points like it in the future. On the other hand, the potential value of x'_k may be higher than the cluster's center potential, introducing the x'_k's vicinity as an area with a higher density. This also suggests moving the cluster center $c_{\upsilon_{cl}}$ towards the higher density area around x'_k. The higher the potential value of x'_k is, the closer the cluster's new center and x'_k will be. As a result, upon an extend operation, cluster υ_{cl} is moved and its radius is increased to cover a fraction of the area around x'_k as well as retaining a fraction of the area previously covered by the cluster. The fractions of the areas around x'_k and $c_{\upsilon_{cl}}$ covered by the extended cluster are proportional to their relative potential values, π_k and $\pi_{\upsilon_{cl}}$, calculated as follows:

$$\pi_k = \frac{P_k(x'_k)}{P_k(x'_k) + P_k(c_{\upsilon_{cl}})}, \tag{25}$$

$$\pi_{\upsilon_{cl}} = 1 - \pi_k. \tag{26}$$

Therefore, the extend operation expands the line segment connecting x'_k and $c_{v_{cl}}$ by $\pi_k \underline{r}$ and $\pi_{v_{cl}} r_{v_{cl}}$ from the end points x'_k and $c_{v_{cl}}$, respectively, as shown in Fig. 6. Then, the cluster's center is moved to the point in the middle of the expanded line segment, i.e. the cluster's new center is computed as:

$$c^+_{v_{cl}} = c_{v_{cl}} + \frac{\|c^+_{v_{cl}} - c_{v_{cl}}\|}{\|x'_k - c_{v_{cl}}\|}(x'_k - c_{v_{cl}}), \tag{27}$$

in which:

$$\|c^+_{v_{cl}} - c_{v_{cl}}\| = \frac{1}{2}\left(\|x'_k - c_{v_{cl}}\| + \pi_k \underline{r} - \pi_{v_{cl}} r_{v_{cl}}\right); \tag{28}$$

and half the length of the expanded line segment is set as the new radius, i.e.:

$$r^+_{v_{cl}} = \max\left\{\frac{1}{2}(\|x'_k - c_{v_{cl}}\| + \pi_k \underline{r} + \pi_{v_{cl}} r_{v_{cl}}), r_{v_{cl}}\right\} \tag{29}$$

With the add and extend operations introduced, we now discuss the decision policy for these operations. The decision is based on how probable it is for an add or extend operation to increase the number of cluster overlaps, leading to the violation of Constraint 1. If there is no overlap chance by adding a cluster at x'_k with radius $\underline{r}$, add operation is approved. Thus, the algorithm first obtains the set of existing clusters that may overlap with a new cluster at x'_k (Line 8). In general, let $\mathcal{C}_{ovr}(X')$ denote the set of clusters overlapping with a new cluster at $X' \in \mathcal{X}'$, and is formulated as:

$$\mathcal{C}_{ovr}(X') = \{i \in \mathcal{C} \mid \|X' - c_i\| < \underline{r} + r_i\}. \tag{30}$$

If $\mathcal{C}_{ovr}(x'_k) = \emptyset$, the add operation is issued (Lines 19–21). Otherwise, the clustering algorithm goes for extending or adding a cluster to include x'_k (Lines 9–18). First, it checks if these operations are applicable. If only one of these operations is applicable, it goes for the applicable one. If both are applicable, the algorithm selects the add operation for dispatching only when the following condition holds:

$$P_k(x'_k) > \frac{d_{thr} - \|x'_k - c_{v_{cl}}\|}{d_{thr} - r_{v_{cl}}} \exp((1-\lambda_k)(\|x'_k - c_{v_{cl}}\| - r_{v_{cl}}))\overline{P_k^{ovr}}, \tag{31}$$

Algorithm 2 (Cluster_OnGrow): Decide if a cluster needs to be added, extended, or shifted when a new feature data point is received.

Input: Feature data point x'_k and its potential value $P_k(x'_k)$
Output: Clustering operation $op \in \{add, shift, extend, None\}$, target cluster that receives the clustering operation v, new center c and new radius r of the target cluster

1: $op, c, r, v \leftarrow None, None, None, None$;
2: Obtain $\mathcal{C}_{act}(x'_k)$ using (17);
3: **if** $\mathcal{C}_{act}(x'_k) \neq \emptyset$ **then**
4: Obtain v_{act} using (23);
5: **if** $P_k(x'_k) > P_k(c_{v_{act}})$ **then**
6: **Shift Cluster**: $op \leftarrow shift$; $c \leftarrow x'_k$; $r \leftarrow r_{v_{act}}$; $v \leftarrow v_{act}$;
7: **else**
8: Obtain $\mathcal{C}_{ovr}(x'_k)$ using (30);
9: **if** $\mathcal{C}_{ovr}(x'_k) \neq \emptyset$ **then**
10: Calculate v_{cl} and $r^+_{v_{cl}}$ by (24) and (29), respectively;
11: **if** $|\mathcal{C}| < N_{mod}$ *or* $r^+_{v_{cl}} \leq \bar{r}$ **then**
12: Calculate $\overline{P_k^{ovr}}$ by (33);
13: **if** $|\mathcal{C}| < N_{mod}$ and ($r^+_{v_{cl}} > \bar{r}$ or (31) satisfied) **then**
14: **Add Cluster**: $op \leftarrow add$; $c \leftarrow x'_k$; $r \leftarrow \underline{r}$; $v \leftarrow$ new ID;
15: **else**
16: Calculate $c^+_{v_{cl}}$ by (27);
17: **Extend Cluster**: $op \leftarrow extend$; $c \leftarrow c^+_{v_{cl}}$; $r \leftarrow r^+_{v_{cl}}$; $v \leftarrow v_{cl}$;
18: Update d_{thr} by (32);
19: **else**
20: **if** $|\mathcal{C}| < N_{mod}$ **then**
21: **Add Cluster**: $op \leftarrow add$; $c \leftarrow x'_k$; $r \leftarrow \underline{r}$; $v \leftarrow$ new ID;
22: **return** *op, c, r, v*

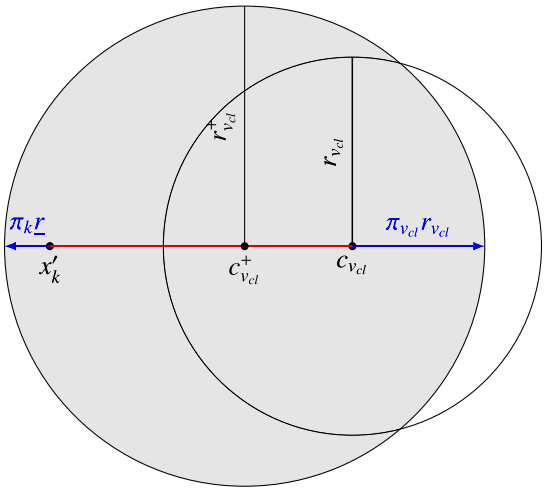


Fig. 6. Illustrating how the new center and radius are calculated when extending cluster v_{cl} to include the received feature data point x'_k.

in which $\lambda_k \in \mathbb{R}$ is a parameter to be set at each training round k; d_{thr} represents the maximum distance a new cluster center can be from an existing cluster center while still overlapping, formulated as follows:

$$d_{thr} = \underline{r} + \max_{i \in C}\{r_i\}; \tag{32}$$

and $\overline{P_k^{ovr}}$ is the maximum potential value among the potential values of the centers whose clusters overlap with the new cluster at x'_k, as formulated below:

$$\overline{P_k^{ovr}} = \max_{i \in C_{ovr}(x'_k)}\{P_k(c_i)\}. \tag{33}$$

The coefficient term of $\overline{P_k^{ovr}}$ in (31) has two parts: The linear part $(d_{thr} - \|x'_k - c_{v_{cl}}\|)/(d_{thr} - r_{v_{cl}})$ and the exponential part $\exp((1-\lambda_k)(\|x'_k - c_{v_{cl}}\| - r_{v_{cl}}))$. To explain the rationale behind the linear part, consider (31) without the exponential part by setting $\lambda_k = 1$ for $k \geq 1$. In such an inequality, having a linear coefficient term is intuitively derived from the behavior expected in two extreme cases: (i) When x'_k is very close to cluster v_{cl}'s boundary, i.e. $\|x'_k - c_{v_{cl}}\| = r_{v_{cl}}$, and (ii) when x'_k is far from it, i.e. $\|x'_k - c_{v_{cl}}\| \geq d_{thr}$. In the first case, adding a new cluster at x'_k will result in intense overlap with cluster v_{cl} as x'_k is very close to it. Thus, extending cluster v_{cl}'s boundary to include x'_k is preferable. Inequality (31) applies this preference by setting a strong condition, i.e. $P_k(x'_k) > \overline{P_k^{ovr}}$, on add operation for such an x'_k. In other words, for an x'_k very close to cluster v_{cl}'s boundary, the extend operation is the default operation unless it is well suited to serve as a cluster center. Such x'_k can be the one with $P_k(x'_k) > \overline{P_k^{ovr}}$ as it is more descriptive and has more summarization power than its overlapping cluster centers. In the second case, the add operation is preferable since no overlapping is caused by a new cluster at x'_k. This preference is applied by setting an add operation's condition which is simply met for $\|x'_k - c_{v_{cl}}\| \geq d_{thr}$. For a feature point x'_k with $r_{v_{cl}} < \|x'_k - c_{v_{cl}}\| < d_{thr}$, (31) considers a linear relation between $P_k(x'_k)$ and $\overline{P_k^{ovr}}$.

However, assuming a linear coefficient term may not balance the frequencies of the add and extend operations fairly. To demonstrate, if feature data points mostly appear near cluster boundaries, the extend operation is prioritized over the add operation more often, and the MNN grows mainly by extending clusters rather than adding new ones. To address this issue, we include an exponential part to create a fair balance between the add and extend operations' frequencies based on run-time conditions. To enhance the chances of adding more clusters when there have been frequent cluster radius rises, we set a more significant value for λ_k and vice versa. We approximate λ_k as a linear function of two parameters: The normalized cluster radius and the normalized number of clusters, as formulated below:

$$\lambda_k = 1 + \gamma \frac{r_{v_{cl}} - \underline{r}}{\overline{r} - \underline{r}} - \frac{|C| - 1}{N_{mod} - 1}, \gamma > 0, \tag{34}$$

in which γ is a constant coefficient determined empirically, as described in Section 5.

According to (34), we have $\lambda_1 = 1$, which relates $P_k(x'_k)$ and $\overline{P_k^{ovr}}$ linearly in (31). Extending cluster v_{cl} increases the chances of adding new clusters in the future by increasing λ_k. On the other hand, adding new clusters increases the chances of extending existing clusters by reducing λ_k.

4.1.2. Shrinking the MNN

In online learning, clusters may shift as data points are provided one by one. Shifting a cluster may cause a small cluster to fall entirely inside a large one, marking the small one as redundant. Thus, removing the small cluster can result in a more compact cluster set.

Algorithm 3 (Cluster_OnShrink): Decide if a cluster needs to be removed.

Input: Target operation $op \in \{add, shift, extend, None\}$, cluster added/shifted/extended v, current remove queue q
Output: Updated remove queue q, cluster to remove u

```
1:  u ← None;
2:  if op ≠ None then
3:      for i ∈ C, i ≠ v do
4:          Calculate status_{v,i} using (35);
5:          if status_{v,i} ≤ 0 then
6:              if {v, i} already in q then
7:                  Update its priority to −status_{v,i};
8:              else
9:                  Add {v, i} to the queue with priority −status_{v,i};
10:         else
11:             if {v, i} already in q then
12:                 Remove {v, i} from q;
13: if q is not empty and |C| close to N_mod then
14:     i_1, i_2 ← pop a pair from the front of the queue;
15:     if r_{i_1} == r_{i_2} then u ← argmin_{i∈{i_1,i_2}} {P_k(c_i)} else u ← argmin_{i∈{i_1,i_2}} {r_i};
16:     Remove all {u, i} pairs, i ∈ C, from q;
17: return q, u;
```

Two clusters i and j, $i, j \in C$, lie inside one another if:

$$status_{i,j} = \|c_i - c_j\| - |r_i - r_j| \leq 0. \tag{35}$$

We use a priority queue denoted by q to keep track of cluster pairs $\{i, j\}$ with $status_{i,j} \leq 0$, and we set $-status_{i,j}$ as their priority value. Algorithm 3 demonstrates the MNN shrinking policy, as described below.

Whenever a cluster v is added, shifted, or extended, q needs an update as the clustering operation might cause the cluster v to fall inside another cluster or leave one (Lines 2–12). As a result, the algorithm calculates $status_{v,i}$ for the cluster v and any other existing cluster i (Line 4). For a cluster i with $status_{v,i} \leq 0$, if the cluster pair $\{v, i\}$ is already in the queue, its priority value is updated to $-status_{v,i}$ (Line 7); otherwise, the cluster pair is pushed into the queue with $-status_{v,i}$ as its priority value (Line 9). On the other hand, for a cluster i with $status_{v,i} > 0$, if the cluster pair $\{v, i\}$ is already in the queue, it is removed from the queue, as they no longer fall inside each other (Line 12); otherwise, no update on the queue is required.

The algorithm starts removing when the maximum number of modules is almost reached. It pops the cluster pair with the highest priority from the queue and removes the cluster with the smaller radius. If the radius values are the same, the cluster with the lower potential is removed (Lines 13–16).

4.2. Managing cluster overlaps

The clustering algorithm must also ensure that the change to the clustering upon receiving an input complies with the constraint on the maximum number of overlaps allowed between clusters (i.e. Constraint 1). To facilitate this, we model clustering using an undirected graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = C$ is the set of vertices, with each vertex representing a cluster ID, and $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$ is the set of edges, with each edge indicating an overlap between the involved clusters. Two clusters i and j overlap if:

$$\|c_i - c_j\| < r_i + r_j. \tag{36}$$

A subset $\mathcal{Q}$ from $\mathcal{V}$ is called a *clique* $\mathcal{Q}$ of $\mathcal{G}$ if every two vertices in $\mathcal{Q}$ are adjacent, i.e. $\forall u, v \in \mathcal{Q} : \{u, v\} \in \mathcal{E}$. A clique with a cardinality of ω is called an ω-clique. A clique is *maximum* if its cardinality is the largest among all the graph's cliques. The cardinality of a maximum clique of $\mathcal{G}$ is called the *clique number* of $\mathcal{G}$, denoted by $\overline{\omega}$. In our work, $\overline{\omega}$ represents the maximum number of overlaps in the clustering. Thus, Constraint 1 is satisfied if we always have $\overline{\omega} \leq N_{node} - 1$.

To manage the cluster overlaps, the clustering algorithm is responsible for:

- Monitoring the clique number by keeping track of the maximum cliques using Algorithm 4 when a cluster is added, shifted, or extended or using Algorithm 6 when a cluster is removed; and
- eliminating the maximum cliques that have caused $\overline{\omega} > N_{node} - 1$ using Algorithm 7.

Algorithm 4. This algorithm updates the set of maximum cliques, denoted by $\mathscr{Q}$, whenever a new vertex v is added to $\mathcal{G}$ because of an add operation or when a vertex v in $\mathcal{G}$ is targeted for a shift or extend operation. Let $\mathscr{Q}^+$, $\mathcal{G}^+$, and $\overline{\omega}^+$ represent the updated

Algorithm 4 (UpdMC_OnGrow): Update the current set of maximum cliques upon an add, shift, or extend operation.

Input: Cluster targeted by the clustering operation v, new graph after clustering $\mathcal{G}^+(\mathcal{V}^+, \mathcal{E}^+)$, current clique number $\overline{\omega}$, set $\mathcal{Q}_{exc}$ consists of all the existing maximum cliques in $\mathcal{G}$ that exclude v
Output: Set $\mathcal{Q}^+$ consists of all the maximum cliques in the new graph $\mathcal{G}^+$ after the clustering operation, new clique number $\overline{\omega}^+$

1: $\mathcal{Q}^+ \leftarrow \emptyset$;
2: **if** $|\mathcal{A}_{\mathcal{G}^+}(v)| \geq \overline{\omega}$ **then**
3: **for** $\mathcal{Q} \in \mathcal{Q}_{exc}$ **do**
4: **if** $\mathcal{Q} \subseteq \mathcal{A}_{\mathcal{G}^+}(v)$ **then**
5: $\mathcal{Q}^+ \leftarrow \mathcal{Q}^+ \cup \{\mathcal{Q} \cup \{v\}\}$;
6: $\overline{\omega}^+ \leftarrow \overline{\omega} + 1$;
7: **if** $\mathcal{Q}^+ == \emptyset$ **then**
8: $\mathcal{Q}^+_{inc} \leftarrow$ **Clique**$(\mathcal{G}^+, \overline{\omega}, \{v\}, \mathcal{A}_{\mathcal{G}^+}(v))$;
9: $\mathcal{Q}^+ \leftarrow \mathcal{Q}^+_{inc} \cup \mathcal{Q}_{exc}$;
10: $\overline{\omega}^+ \leftarrow \overline{\omega}$;
11: **if** $\mathcal{Q}^+ == \emptyset$ **then**
12: $\mathcal{Q}^+ \leftarrow$ **Clique**$(\mathcal{G}^+, \overline{\omega} - 1, \emptyset, \mathcal{V}^+)$;
13: $\overline{\omega}^+ \leftarrow \overline{\omega} - 1$;
14: **return** $\mathcal{Q}^+$, $\overline{\omega}^+$;

Algorithm 5 (Clique): Find all ω-cliques, with each containing the clique $\mathcal{Q}$ and constructed from the vertex set $\mathcal{Z}$.

Input: Undirected graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, target clique cardinality ω, clique $\mathcal{Q}$, candidate set $\mathcal{Z}$
Output: Set of all target cliques $\mathcal{Q}$

1: $\mathcal{Q} \leftarrow \emptyset$;
2: **if** $|\mathcal{Q}| == \omega$ **then**
3: $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{\mathcal{Q}\}$;
4: **if** $|\mathcal{Q}| < \omega$ **then**
5: **if** $|\mathcal{Q}| + |\mathcal{Z}| \geq \omega$ **then**
6: **while** $\mathcal{Z} \neq \emptyset$ **do**
7: $z \leftarrow$ select a vertex from $\mathcal{Z}$;
8: $\mathcal{Z} \leftarrow \mathcal{Z} \backslash \{z\}$;
9: $\mathcal{Q}^+ \leftarrow \mathcal{Q} \cup \{z\}$;
10: $\mathcal{Z}^+ \leftarrow \mathcal{Z} \cap \mathcal{A}_{\mathcal{G}}(z)$;
11: $\mathcal{Q} \leftarrow \mathcal{Q} \cup$ **Clique**$(\mathcal{G}, \omega, \mathcal{Q}^+, \mathcal{Z}^+)$;
12: **return** $\mathcal{Q}$;

maximum clique set, graph, and clique number, respectively. The vertex v receiving the clustering operation can increment, preserve, or decrement $\overline{\omega}$, with each case described in the following paragraphs.

The clique number gets incremented if at least one existing maximum clique excluding v has all its vertices adjacent to v after clustering (Lines 2–6). If an existing maximum clique builds a maximum clique of a larger size with v, we say that v has *promoted* that maximum clique. The algorithm takes the set of existing maximum cliques excluding v, denoted by $\mathcal{Q}_{exc}$, as input to check for any promotion. Similarly, the set of existing maximum cliques including v is denoted by $\mathcal{Q}_{inc}$, and we have $\mathcal{Q}_{exc} \cup \mathcal{Q}_{inc} = \mathcal{Q}$ and $\mathcal{Q}_{exc} \cap \mathcal{Q}_{inc} = \emptyset$. The required condition for the target vertex v to promote at least one existing maximum clique in $\mathcal{Q}_{exc}$ is that the number of its adjacent vertices in $\mathcal{G}^+$ not be less than the current clique number $\overline{\omega}$ (Line 2). In Algorithm 4, $\mathcal{A}_{\mathcal{G}^+}(v)$ returns the set of vertices in $\mathcal{G}^+$ adjacent to v.

If v does not promote any clique in $\mathcal{Q}_{exc} \neq \emptyset$, this implies that the clique number is preserved, and the algorithm uses Algorithm 5 to update $\mathcal{Q}_{inc}$ by finding all maximum cliques of size $\overline{\omega}$ (Lines 7–10). Algorithm 5 is a recursive algorithm that receives three inputs at each iteration: (i) A clique $\mathcal{Q}$, (ii) a candidate set $\mathcal{Z}$, and (iii) a target cardinality ω. It is then expected to find all ω-cliques by starting from $\mathcal{Q}$ and adding vertices from $\mathcal{Z}$. Every vertex in $\mathcal{Z}$ must be adjacent to all vertices in $\mathcal{Q}$. In Algorithm 4, to find the maximum cliques of size $\overline{\omega}$ including v, Algorithm 5 is initially invoked with $\mathcal{Q} = \{v\}$ and $\mathcal{Z} = \mathcal{A}_{\mathcal{G}^+}(v)$ (Line 8).

When $\mathcal{Q}_{exc} = \emptyset$ and there is no maximum clique of size $\overline{\omega}$ after clustering (i.e. $\mathcal{Q}^+_{inc} = \emptyset$), the algorithm looks for maximum cliques of size $\overline{\omega} - 1$ using Algorithm 5, implying that the clique number is decremented (Lines 11–13).

Algorithm 6. This algorithm updates the maximum clique set $\mathcal{Q}$ when a vertex v is removed from $\mathcal{G}$ due to a remove operation. Only are the maximum cliques including v affected by the operation, having their cardinality decremented. As a result, the algorithm

iterates through $\mathcal{Q}$ and keeps only those excluding v (Lines 2–4). If all the existing maximum cliques include v, the remove operation decrements the clique number. In this case, the algorithm looks for maximum cliques of size $\overline{\omega}-1$ using Algorithm 5 (Lines 5–7).

Algorithm 7. These considerations imply that only adding, shifting, or extending clusters can increment the clique number. Now, consider a case wherein $\overline{\omega}=N_{node}-1$ and the center c and radius r decided by Algorithm 2 for cluster v promote some maximum cliques in $\mathcal{Q}_{exc}$, resulting in $\overline{\omega}^{+}>N_{node}-1$. In this case, Algorithm 7 determines an alternative center point c^{+} to prevent such promotion and keep $\overline{\omega}^{+}$ at $N_{node}-1$. Let $\mathcal{Q}_{pr}$ and $\mathcal{Q}_{\bar{pr}}$ denote the sets of maximum cliques in $\mathcal{Q}_{exc}$ promoted and those not promoted by cluster v, respectively. Thus, we have $\mathcal{Q}_{pr}\cup\mathcal{Q}_{\bar{pr}}=\mathcal{Q}_{exc}$ and $\mathcal{Q}_{pr}\cap\mathcal{Q}_{\bar{pr}}=\emptyset$. The alternative center point must satisfy the following requirements:

- **Requirement 1**. It must be far enough away from the centers of the cliques in $\mathcal{Q}_{pr}$ to eliminate any chance of forming N_{node}-cliques.
- **Requirement 2**. It must keep its distance from cliques in $\mathcal{Q}_{\bar{pr}}$ as before to avoid forming N_{node}-cliques.
- **Requirement 3**. Cluster v with the alternative center point must still include x'_k in its cluster boundary.

Too many points in the l'-dimensional feature space may satisfy all the requirements. Algorithm 7 looks for a subset of them, denoted by $\mathcal{Z}_{intr}$, as the candidate points for the alternative center point c^{+}. Among them, the closest to c is selected as c^{+}.

To find $\mathcal{Z}_{intr}$, the algorithm starts by targeting each clique in $\mathcal{Q}_{pr}$, finding the cluster with the maximum distance from c within the clique, and adding its ID to C_{md} (Line 3). To avoid the alternative point joining a clique in $\mathcal{Q}_{pr}$, it must be far enough from the cluster $i\in C_{md}$ corresponding to that clique. One can find such a point at the intersection points of circles (c_i, r_i+r) and (c, r). Other points that could be the alternative center are those at the intersection points of circles (c_i, r_i+r) and (c_j, r_j+r), with j being another cluster from C_{md}. The intersection points containing x'_k in their cluster boundary are added to $\mathcal{Z}_{intr}$ (Lines 4–7).

Nevertheless, the points in $\mathcal{Z}_{intr}$ found so far may not fully adhere to Requirement 1, as a point in $\mathcal{Z}_{intr}$ that maintains a safe distance from a center $c_i, i\in C_{md}$, to avoid forming an N_{node}-clique may still be close to another center $c_j, j\in C_{md}$, and forms such a clique. Moreover, Requirement 2 needs to be checked for points in $\mathcal{Z}_{intr}$. To check requirements 1 and 2 for the points in $\mathcal{Z}_{intr}$, the algorithm first extends C_{md} by including the cluster with the maximum distance from c for each clique in $\mathcal{Q}_{\bar{pr}}$ (Line 8). Next, it visits the points in $\mathcal{Z}_{intr}$ in ascending order of their distance to c, and checks for any overlapping between a cluster with center $z\in\mathcal{Z}_{intr}$ and radius r and a cluster with center c_i and radius r_i, $i\in C_{md}$. if no such overlapping is found, the point z is returned as c^{+} (Lines 9 and 10).

4.3. Adjusting the centers and radii

This subsection discusses the proposed online clustering approach, which is described in Algorithm 8. The algorithm adjusts the centers and radii of the RBF neurons upon receiving a feature data point x'_k and maintains the number of cluster overlaps at $N_{node}-1$. It calls algorithms 2 and 3 to check for any growing and shrinking required in the RBF neurons. Adding or removing an RBF neuron requires a corresponding module within the MNN to be added or removed accordingly. However, shifting or extending does not require any architectural change to be reflected in the MNN. As a result, the algorithm returns the IDs of added and removed RBF neurons, denoted by u_{add} and u_{rem}, respectively, to be used to reflect a similar change in the MNN. The algorithm uses algorithms 4, 6, and 7 to manage the cluster overlaps. The following paragraphs demonstrate Algorithm 8 in more detail.

Initially, the RBF layer has no RBF neurons. Upon receiving the first data sample, an RBF neuron is added. The algorithm then returns the neuron's ID to indicate that the first module needs to be instantiated within the MNN. Thus, the first round commences with some initializations (Lines 2 and 3), followed by adding the first RBF neuron (Line 4).

For each feature data point x'_k coming in the following rounds, the algorithm calculates $P_k(x'_k)$ and updates the potential values of existing centers (Lines 6 and 7). It then invokes Algorithm 2 to decide on an add, shift, or extend operation (Line 8). If a clustering operation is decided (i.e. op is not $None$), we need to obtain the new set of maximum cliques. Therefore, the algorithm obtains the new graph and splits the current set of maximum cliques $\mathcal{Q}$ into the disjoint subsets $\mathcal{Q}_{inc}$ and $\mathcal{Q}_{exc}$ based on cluster v (Lines 11

Algorithm 6 (UpdMC_OnShrink): Update the current set of maximum cliques upon a remove operation.

Input: Cluster targeted for removing v, new graph after removing $\mathcal{G}^{+}(\mathcal{V}^{+},\mathcal{E}^{+})$, set of maximum cliques before removing $\mathcal{Q}$, clique number before removing $\overline{\omega}$

Output: Set consists of all the maximum cliques in the new graph $\mathcal{Q}^{+}$, new clique number $\overline{\omega}^{+}$

1: $\mathcal{Q}^{+}\leftarrow\emptyset$; $\overline{\omega}^{+}\leftarrow\overline{\omega}$;
2: **for** $Q\in\mathcal{Q}$ **do**
3: **if** $v\notin Q$ **then**
4: $\mathcal{Q}^{+}\leftarrow\mathcal{Q}^{+}\cup\{Q\}$;
5: **if** $\mathcal{Q}^{+}==\emptyset$ **then**
6: $\mathcal{Q}^{+}\leftarrow$ **Clique**$(\mathcal{G}^{+},\overline{\omega}-1,\emptyset,\mathcal{V}^{+})$;
7: $\overline{\omega}^{+}\leftarrow\overline{\omega}-1$;
8: **return** $\mathcal{Q}^{+}$, $\overline{\omega}^{+}$;

Algorithm 7 (Eliminate_Clique): For a cluster that has promoted some existing maximum cliques, find an alternative center that avoids promoting any existing maximum clique.

Input: Feature data point x'_k, cluster that has promoted some existing maximum cliques v, center c and radius r of cluster v, set $\mathcal{Q}_{exc}$ consists of existing maximum cliques excluding v
Output: Alternative center point c^+

$C_{md} \leftarrow \emptyset$; $\mathcal{Z}_{intr} \leftarrow \emptyset$;
Split $\mathcal{Q}_{exc}$ into $\mathcal{Q}_{pr}$ and $\mathcal{Q}_{\bar{pr}}$ based on cluster v;
Obtain the cluster with the maximum distance from c for each $\mathcal{Q} \in \mathcal{Q}_{pr}$, and add its ID to C_{md};
for $i \in C_{md}$ **do**
    Find two intersection points of hyperspheres $(c_i, r_i + r)$ and (c, r) that are not outside the hypersphere (x'_k, r), and add them to $\mathcal{Z}_{intr}$;
    for $j \in C_{md}$ with $\{i, j\}$ not visited before **do**
        Find two intersection points of hyperspheres $(c_i, r_i + r)$ and $(c_j, r_j + r)$ that are not outside the hypersphere (x'_k, r), and add them to $\mathcal{Z}_{intr}$;
Obtain the cluster with the maximum distance from c for each $\mathcal{Q} \in \mathcal{Q}_{\bar{pr}}$, and add its ID to C_{md};
for $z \in \mathcal{Z}_{intr}$ visited in ascending order of $\|z - c\|$ **do**
    if $\|z - c_i\| \geq r_i + r$ for all $i \in C_{md}$ **then return** z;
return $None$;

and 12). These subsets and the new graph are used by Algorithm 4 to obtain $\mathcal{Q}^+$ and $\overline{\omega}^+$ (Line 13). Now, it checks if the clustering operation has caused more than $N_{node} - 1$ clusters to overlap.

If $\overline{\omega}^+ < N_{node}$, the algorithm proceeds by calculating $P_k(c)$ (Lines 25–28). For add and shift operations, the feature data point x'_k serves as the added or shifted cluster's center, then $P_k(c) = P_k(x'_k)$. However, for an extend operation, the extended cluster's center can be any point within the feature space, and thus, its potential is calculated using (9). In case $\overline{\omega}^+ \geq N_{node}$, Algorithm 7 is invoked to suggest an alternative center point that keeps $\overline{\omega}^+$ at $N_{node} - 1$ (Line 15). If such a point is found, the algorithm calculates its potential and obtains $\mathcal{G}^+$ and $\mathcal{Q}^+$ accordingly (Lines 16–20). Since cluster v has already maintained its safe distance from the cliques in $\mathcal{Q}_{exc}$ by Algorithm 7, we just need to obtain $\mathcal{Q}^+_{inc}$. In case no alternative center point is found, the algorithm ignores the feature data point by setting op to $None$ (Line 22).

After managing the cluster overlaps, the RBF neuron with $ID = v$ is added in case of an add operation (Lines 29–31). Finally, the changes to $\mathcal{Q}$, $\mathcal{G}$, and $\overline{\omega}$ are applied, and the cluster v targeted by the clustering operation receives its center, radius, and potential updates (Lines 32–34).

Moreover, the algorithm checks for any cluster removal in each round by calling Algorithm 3 (Line 35). If a cluster u_{rem} needs to be removed, the algorithm updates $\mathcal{G}$ accordingly, removes the RBF neuron with $ID = u_{rem}$, and updates $\mathcal{Q}$ by calling Algorithm 6 (Lines 36–39).

5. Experimental results

In this section, we first introduce the design flow of our experiments. Next, we present the parameter values selected for conducting our experiments. Finally, the obtained results and discussions are presented.

5.1. Experiment design flow

In our experiments, we first conduct the module-level bias–variance trade-off analysis to establish the upper and lower bounds for a module cluster radius. Then, we proceed with the online training of our MNN. Algorithm 9 outlines the steps to set up our experiments. The algorithm needs the following inputs: The dataset, the module architecture, and the platform model. The module architecture can be tuned for a given application using two hyperparameters: The number of layers h_l and the width multiplier h_p. The following paragraph briefly summarizes the steps.

We first set up the given dataset for the bias–variance trade-off analysis and online learning experiments (Line 1). Since modules' clusters are formed over the feature space, a feature extractor is trained to extract the feature space (Line 2). Next, we determine the application model by specifying the MNN architecture and the timeliness and accuracy constraints (Lines 3–5). Finally, we conduct our bias–variance trade-off analysis and online learning experiments (Lines 6 and 7). The following subsections provide a more detailed demonstration of each step.

Step 1: Dataset Setup. A given dataset $\mathcal{S}$ is randomly divided into two parts: $\mathcal{S}^{ana}$ and $\mathcal{S}^{onl}$. $\mathcal{S}^{ana}$ simulates data available offline in a batch prior to the online learning experiment, and it is used for the bias–variance trade–off analysis. $\mathcal{S}^{onl}$ represents data not seen by the analysis and is provided incrementally during online learning. Moreover, in the bias–variance trade–off analysis, we must train an MNN for different sample sizes and evaluate its accuracy. Therefore, the data used for the analysis (i.e. $\mathcal{S}^{ana}$) is then split into the training and validation parts, denoted by $\mathcal{S}^{ana}_{trn}$ and $\mathcal{S}^{ana}_{val}$, respectively. $\mathcal{S}^{ana}_{trn}$ is used for training the MNN, while $\mathcal{S}^{ana}_{val}$

Algorithm 8 (Train_RBF): Adjust the centers and radii of the RBF layer.

Input: Minimum and maximum cluster radii $\underline{r}$ and $\overline{r}$
Output: Module to be added u_{add}, module to be removed u_{rem}

if $k == 1$ **then**
  $P_k(x'_k) \leftarrow 1; c_1 \leftarrow x'_k; r_1 \leftarrow \underline{r}$; $P_k(c_1) \leftarrow P_k(x'_k)$;
  Initialize $\mathcal{Q}$, $\mathcal{G}$, and C; $\overline{\omega} \leftarrow 1; d_{thr} \leftarrow 2\underline{r}$;
  Add the first RBF neuron with $ID = 1$;
  return $1, None$
Calculate $P_k(x'_k)$ *recursively* by (9);
Update $P_k(c_i)$ *recursively* by (12), $i \in C$;
$op, c, r, v \leftarrow$ **Cluster_OnGrow**$(x'_k, P_k(x'_k))$;
$u_{add} \leftarrow None$;
if $op \neq None$ **then**
  $\mathcal{G}^+ \leftarrow$ update $\mathcal{G}$ based on (op, c, r, v);
  Split $\mathcal{Q}$ into $\mathcal{Q}_{inc}$ and $\mathcal{Q}_{exc}$ based on cluster v;
  $\mathcal{Q}^+, \overline{\omega}^+ \leftarrow$ **UpdMC_OnGrow**$(v, \mathcal{G}^+, \overline{\omega}, \mathcal{Q}_{exc})$;
  if $\overline{\omega}^+ \geq N_{node}$ **then**
    $c \leftarrow$ **Eliminate_Clique**$(x'_k, v, c, r, \mathcal{Q}_{exc})$;
    if $c \neq None$ **then**
      Calculate $P_k(c)$ by (9);
      $\mathcal{G}^+ \leftarrow$ update $\mathcal{G}$ based on c;
      $\mathcal{Q}^+_{inc} \leftarrow$ **Clique**$(\mathcal{G}^+, \overline{\omega}, \{v\}, \mathcal{A}_{\mathcal{G}^+}(v))$;
      $\mathcal{Q}^+ \leftarrow \mathcal{Q}^+_{inc} \cup \mathcal{Q}_{exc}$;
    else
      $op \leftarrow None$;
    $\overline{\omega}^+ \leftarrow \overline{\omega}$;
  else
    if $op == extend$ **then**
      Calculate $P_k(c)$ by (9);
    else
      $P_k(c) \leftarrow P_k(x')$;
  if $op == add$ **then**
    Add an RBF neuron with $ID = v$;
    $u_{add} \leftarrow v$; $C \leftarrow C \cup \{v\}$;
  if $op \neq None$ **then**
    $\mathcal{Q} \leftarrow \mathcal{Q}^+; \mathcal{G} \leftarrow \mathcal{G}^+; \overline{\omega} \leftarrow \overline{\omega}^+$;
    $c_v \leftarrow c; r_v \leftarrow r; P_k(c_v) \leftarrow P_k(c)$;
$q, u_{rem} \leftarrow$ **Cluster_OnShrink**(op, v, q);
if $u_{rem} \neq None$ **then**
  $\mathcal{G} \leftarrow$ remove u_{rem} from $\mathcal{G}$; $C \leftarrow C \backslash \{u_{rem}\}$;
  Remove the RBF neuron with $ID = u_{rem}$;
  $\mathcal{Q}, \overline{\omega} \leftarrow$ **UpdMC_OnShrink**$(u_{rem}, \mathcal{G}, \mathcal{Q}, \overline{\omega})$;
return u_{add}, u_{rem};

represents data not seen by the MNN during training and is used to evaluate the accuracy of the trained MNN. In summary, the dataset S is split into three parts: S^{ana}_{trn}, S^{ana}_{val}, and S^{onl}.

Step 2: Feature Extractor Setup. In this step, a feature extractor is trained on S^{ana}_{trn} with S^{ana}_{val} as the validation dataset. Clustering works on the feature space, and its quality is thus affected by the feature extraction quality. Hence, we select sophisticated CNN models as our feature extractor. However, complex models are resource-intensive during training. Fortunately, this is not a concern since, unlike the MNN, the feature extractor is frozen at run-time and will be used solely in the inference phase, which requires fewer resources than the training phase. Therefore, we used MobileNetV2 [48] as our feature extractor, whose architecture is tailored towards resource-efficient DNN inference in mobile and embedded applications.

Step 3: Application Model Setup. To set up our application model, we need to determine the workload, i.e. our MNN architecture, and the accuracy and timeliness constraints. The MNN architecture is determined by specifying three hyperparameters: The maximum number of modules N_{mod} and the module's hyperparameters h_l and h_p. On the other hand, the MNN architecture

Algorithm 9 (Setup_Exp): Set up the experiments.

Input: Dataset $\mathcal{S}$, module architecture with hyperparameters h_l and h_p, platform model $(N_{node}, \rho, \beta, W^{flash}, W^{RAM})$

1: Split $\mathcal{S}$ into three parts: $\mathcal{S}^{ana}_{trn}$, $\mathcal{S}^{ana}_{val}$, and $\mathcal{S}^{onl}$;
2: $FE \leftarrow$ train a feature extractor on $(\mathcal{S}^{ana}_{trn}, \mathcal{S}^{ana}_{val})$;
3: Use Bayesian search to determine N_{mod}, h_l, and h_p leading to the MNN architecture MNN^{init} with clustering $\{(c^{init}_i, r^{init}_i)\}_{i=1}^{N^{init}_{mod}}$ that gives the minimum execution time and error rate $\underline{t_{total}}$ and $\underline{Err}$ subject to Constraint 3;
4: Define the target accuracy constraint $\overline{Err}$ according to $\underline{Err}$;
5: Set $\underline{t_{total}}$ as the timeliness constraint T;
6: Call **Gen_CI**$(\mathcal{S}^{ana}_{trn}, \mathcal{S}^{ana}_{val}, MNN^{init}, \{(c^{init}_i, r^{init}_i)\}_{i=1}^{N^{init}_{mod}})$ to obtain the upper-bound learning curve, and use the curve to obtain the minimum and maximum radius values $\underline{r}$ and $\overline{r}$ given $\overline{Err}$;
7: Call **Learn_Online**$(FE, MNN^{init}, \underline{r}, \overline{r}, \mathcal{S}^{onl})$ to initiate online learning;

affects the resource requirements, total execution time, and accuracy, which in turn affects the range of accuracy and timeliness constraints (i.e. $\overline{Err}$ and T) feasible under the given platform model and module architecture. In this step, we seek the feasible accuracy and timeliness constraints for a given platform model and module architecture.

We use Bayesian search to tune our MNN's hyperparameters to minimize error rate and execution time subject to Constraint 3. The Bayesian search takes the following steps for each trial, examining a hyperparameter combination (N_{mod}, h_l, h_p) that complies with Constraint 3. First, the feature space is clustered into N_{mod} clusters. We used k-means for feature space clustering. The radius of each cluster is defined as the maximum distance between the cluster center and a point within the cluster. Next, an MNN with the hyperparameter values of N_{mod}, h_l, and h_p is instantiated, and each module is associated with a cluster. The instantiated MNN is batch-trained on $\mathcal{S}^{ana}_{trn}$, and its error rate on $\mathcal{S}^{ana}_{val}$ is reported. Additionally, the training execution time is calculated using (A.17) and reported.

The output of the Bayesian search is a pre-trained MNN with the minimum error rate and execution time found, $\underline{Err}$ and $\underline{t_{total}}$, respectively. We denote this MNN by MNN^{init}, which will be used in the bias–variance trade–off analysis and the online learning experiment latter. The number of modules and the clustering associated with MNN^{init} are denoted by N^{init}_{mod} and $\{(c^{init}_i, r^{init}_i)\}_{i=1}^{N^{init}_{mod}}$, respectively. With the application workload determined, we set $T = \underline{t_{total}}$, and the target accuracy constraint $(\overline{Err})$ is set so that $\overline{Err} \geq \underline{Err}$.

Step 4: Bias-Variance Trade-off Analysis. The pre-trained modular neural network MNN^{init} as well as the training and validation datasets $\mathcal{S}^{ana}_{trn}$ and $\mathcal{S}^{ana}_{val}$ are then fed into Algorithm 11 to generate the upper-bound learning curve. The curve is then used to compute the upper and lower bounds for the modules' cluster radii based on the accuracy constraint $\overline{Err}$.

Step 5: Online Training of the MNN. In our online learning, the batch size is set to 1 to provide data incrementally over time. Before initiating our online learning experiment, we use a grid search to determine γ in (34). γ is determined based on the validation dataset $\mathcal{S}^{ana}_{val}$ to maximize the number of time steps that the error rate is below $\overline{Err}$. Next, the pre-trained modular neural network MNN^{init}, the feature extractor FE, and the online learning dataset $\mathcal{S}^{onl}$ are fed into Algorithm 1 to conduct the online learning experiment. The algorithm also needs $\underline{r}$ and $\overline{r}$ to adjust the modules' cluster radii at run-time.

5.2. Experiment parameter values

Dataset. We used CIFAR-100 [49] as our dataset $\mathcal{S}$. We split it into parts $\mathcal{S}^{ana}_{trn}$, $\mathcal{S}^{ana}_{val}$, and $\mathcal{S}^{onl}$ with ratios of 70%, 15%, and 15%, respectively. To prevent overfitting our MNN, we enlarged $\mathcal{S}^{ana}_{trn}$ by two times using augmentation. Finally, the number of images in $\mathcal{S}^{ana}_{trn}$, $\mathcal{S}^{ana}_{val}$, and $\mathcal{S}^{onl}$ was 80 000, 10 000, and 10 000, respectively.

Module Architecture. We used MobileNetV2 as our module architecture. The architecture consists of a standard CONV layer followed by seven groups of bottleneck layers. We introduced the hyperparameter h_l as the number of bottleneck groups used in the module architecture, i.e. $0 \leq h_l \leq 7$. Moreover, according to the architecture, we have $R = 1280 \times h_p$.

Platform Model. Our online learning application was assumed to use two slave nodes, i.e. $N_{node} = 3$. We aimed for nodes with a CPU frequency of around 1 GHz and a RAM capacity of tens to hundreds of MB. To set the resource constraints for a node, we chose a standard MobileNetV2 (i.e. $h_l = 7$ and $h_p = 1.0$), a DNN model commonly used at the edge, as our baseline model. We then defined the constraints as a fraction of its resource requirements. We set the RAM resource constraint for the platform model at 50% of the RAM amount required by the baseline model, i.e. $W^{RAM} = 50\% \times 25$ (MB) $= 12.5$ (MB). Since embedded devices are usually equipped with larger amounts of flash memory compared to RAM, we set the flash memory constraint two times the RAM constraint, i.e. $W^{flash} = 2 \times 12.5$ (MB) $= 25$ (MB). For our targeted nodes, $\rho_{freq} = 1$ GHz, and according to [45], we have $\rho_{flop} = 4$ and $MAC_{flop} = 2$. Hence, based on (14), our nodes have a computational power of $\rho = 4\ GFLOPS$. For a node provided with a Wi-Fi module using 802.11g protocol, the maximum theoretical data rate is $\beta = 54$ Mbps.

Feature Extractor. We used a MobileNetV2 with $h_l = 7$ and $h_p = 0.8$ as our feature extractor. Thus, we have $Par_{fe} = 1.64 \times 10^6$ and $Act_{fe} = 0.04 \times 10^6$. Then, assuming 32-bit scalar numbers (i.e. $b = 32$), (A.12) gives $RAM_{fe} = 6.72$ (MB).

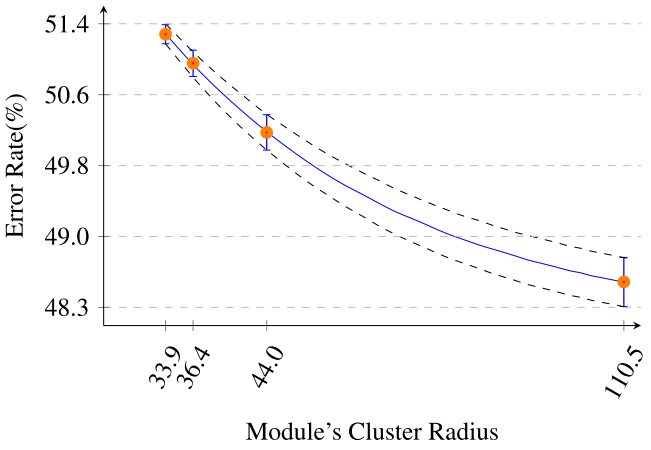


Fig. 7. Confidence interval of a module's learning curve after running the bias–variance trade-off analysis.

We trained our feature extractor on training and validation datasets S^{ana}_{trn} and S^{ana}_{val} using a batch size of 256 for 120 epochs with a momentum of 0.9, weight decay of 0.0001, and an initial learning rate of 0.1. The learning rate was decayed by a factor of 10 every 30 epochs. The feature extractor achieved a top-1 accuracy of 69 %. Moreover, we used *principal component analysis (PCA)* to reduce the feature space's dimensionality, resulting in $l' = 2$.

Application Model. We conducted a Bayesian search using Optuna [50] for 22 trials. In the search space, the hyperparameters N_{mod} and h_l were set as integers ranging from 3 to 5 and 0 to 7, respectively, while the hyperparameter h_p was a floating-point variable ranging from 0.6 to 1.4 with a step size of 0.2. In each trial, we trained an MNN for 40 epochs with a batch size of 32, momentum of 0.9, weight decay of 0.0001, and an initial learning rate of 0.01. The learning rate was decreased by a factor of 10 every 10 epochs. The best hyperparameter values found by the search were $N_{mod} = 4$, $h_l = 4$, and $h_p = 0.6$. We then instantiated an MNN using these hyperparameter values and performed more intensive training with 100 epochs and a step size of 25. Our pre-trained MNN's error rate was found to be 45.6%. We set the target error rate ($\overline{Err}$) to be 5% worse than the pre-trained MNN's best error rate, i.e. $\overline{Err} = 50.6\%$. The training time for the pre-trained MNN was computed using (A.17) to be 410 (ms), thus $T = 410$ (ms).

Bias-Variance Trade-off Analysis and Online Learning. We performed 40 experiments, i.e. $N_{exp} = 40$. To set N^{init}_{smp}, we had to calculate r_{cvr}. Considering $\overline{d_{p2p}} = 212$, (B.1) calculates $r_{cvr} = 26.5$. Thus, $N^{init}_{smp} = 36000$, and we set $N^{base}_{smp} = 4000$ to have four sample sizes in total. For each sample size, an MNN was batch-trained for 40 epochs with a batch size of 256, momentum of 0.9, weight decay of 0.0001, and an initial learning rate of 0.01. The learning rate was reduced by a factor of 10 every 10 epochs. To generate the corresponding confidence interval, $\alpha = 80\%$. To determine γ for our online learning experiment, we used a grid search with γ ranging from 1 to 10. The result of our search was $\gamma = 4$.

5.3. Results and discussion

After running the bias–variance trade-off analysis described in Algorithm 11, the resulting lower- and upper-bound module learning curves are depicted in Fig. 7.

With $\overline{Err} = 50.6\%$, Fig. 7 introduces $\underline{r} = 42$. The radius upper bound was set based on the maximum distance, i.e. $\bar{r} = \overline{d_{p2p}}/2 = 106$. The results of our online learning experiment are reported in Fig. 8. Fig. 8(a) reports the error rate CDF. The number of activated modules and the total number of modules in each time step are depicted in Fig. 8(b) and Fig. 8(c), respectively. Fig. 8(d) reports the clique number before applying any maximum clique elimination of Algorithm 7. The following observations are made from Fig. 8:

Observation 1. Fig. 8(a) shows that the error rate of our online training was below the accuracy constraint in all time steps.

Observation 2. Fig. 8(b) reports that the maximum number of modules activated in each time step was equal to the number of available slave nodes.

Observation 3. Fig. 8(c) signifies that the MNN experienced variations in the number of employed modules during online learning.

Observation 4. Fig. 8(d) illustrates some time steps at which maximum cliques of sizes exceeding the number of slave nodes were formed.

Observations 1 and 2 demonstrate how our distributed online learning effectively adheres to the accuracy constraint while keeping the maximum number of activated modules at the number of slave nodes. By maintaining $|C_{act}(x'_k)| \leq N_{node} - 1$, it is guaranteed that the timeliness constraint is met.

Based on Observation 3, it is concluded that our MNN effectively utilizes all its modules instead of just relying on a few. Additionally, it is inferred that our MNN can adaptively adjust its size by growing and shrinking as data arrives.

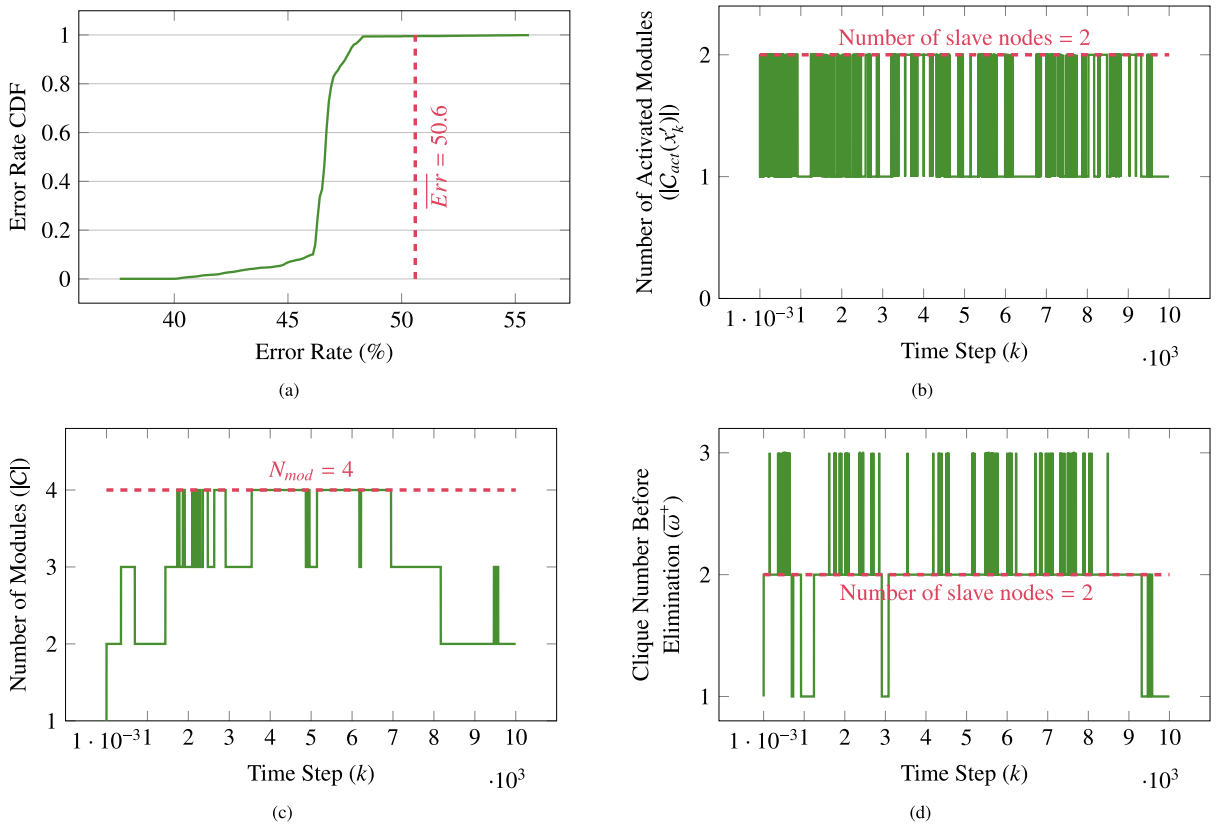


Fig. 8. The online learning experiment's result with regard to (a) the error rate CDF, (b) the activated number of modules, (c) the total number of modules, and (d) the clique number before applying any maximum clique elimination of Algorithm 7.

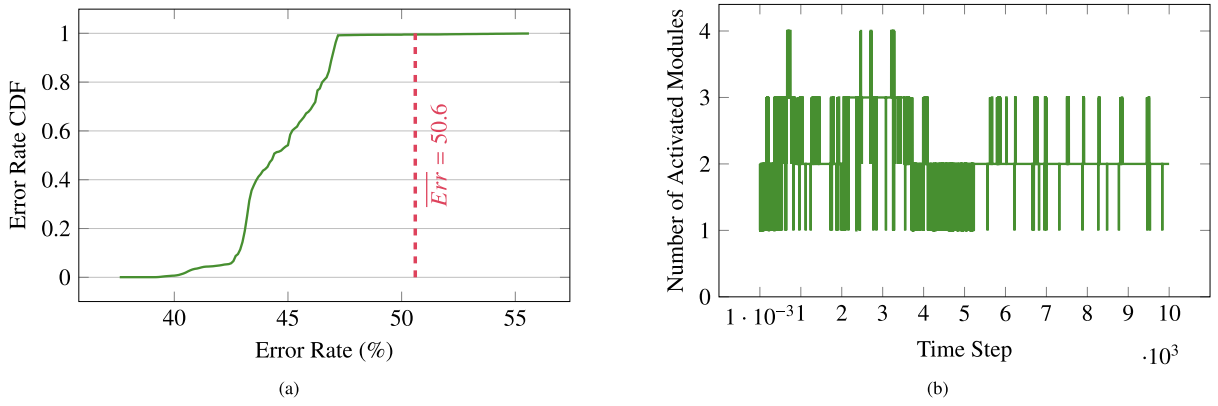


Fig. 9. The result of online learning experiment without using Algorithm 7. The result includes (a) the error rate CDF and (b) the number of activated modules.

Observation 4 highlights the need for Algorithm 7 to keep the maximum number of activated modules at the number of slave nodes. As shown, there were cases where an add, extend, or shift operation caused the number of clusters overlapping with each other to exceed the number of slave nodes. In these scenarios, the proposed clustering algorithm is expected to suggest an alternative center point to avoid the formation of such a group of overlapping clusters. Observation 2 shows that our proposed clustering algorithm has succeeded in this task.

To further highlight the role of Algorithm 7 in keeping the number of activated modules at the number of slave nodes, we conducted our online learning experiment again, this time excluding Algorithm 7. Fig. 9 shows the result. The following observation is made:

Observation 5. With Algorithm 7 excluded, the accuracy constraint is met while the number of activated modules has exceeded the available number of slave nodes.

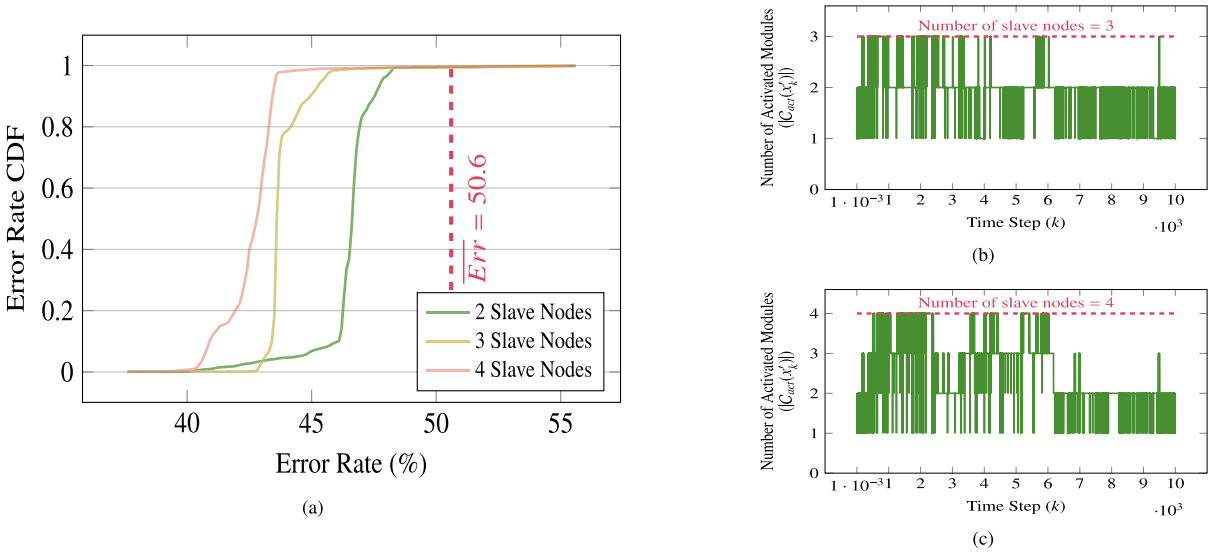


Fig. 10. Reporting the result of conducing the online learning experiment with 3 and 4 slave nodes. The result includes (a) the error rate CDF, and the number of activated modules over time for (b) 3 and (c) 4 slave nodes.

Table 1
Overview of the literature on the resource-intensive training of DNNs.

Category		Studies	Highlights
Distributed learning	Data parallelism	[51]	Primarily focused on preserving privacy, mitigating the network bandwidth demands, and accelerating training (**Shortcoming**: Paying less attention to edge-level resource limitations)
	Model parallelism Pipeline parallelism Hybrid parallelism	[18–34]	Focused on increasing throughput through accelerating training by large-scale CPU/GPU clusters (**Shortcoming**: Paying less attention to edge-level resource limitations and the timeliness constraint inherent in online learning)
Approximation	Network compression Network expansion	[35–40]	Providing real-time on-device online training (**Shortcoming**: Suggesting sub-optimal pre-determined architectures, lacking any self-adaptivity feature)
Modular design	TreeNets and DSNs	[52–54]	Focused on increasing throughput through accelerating training by large-scale CPU/GPU clusters (**Shortcoming**: Paying less attention to edge-level resource limitations and the timeliness constraint inherent in online learning)
	MNNs	[41,42,55]	Providing self-adaptive architecture designs (**Shortcoming:** Paying less attention to edge-level resource limitations and the timeliness and accuracy constraints)
Computation offloading		[56,57]	Focused on the resource allocation and scheduling aspects (**Shortcomings**: Offering limited innovation in model partitioning and still suffering from privacy issues)

As demonstrated by Observation 5, eliminating Algorithm 7 has caused the number of activated modules to reach the maximum number of modules. By comparing Figs. 8(a) and 9(a), it is inferred that using Algorithm 7 effectively manages cluster overlaps while meeting the accuracy constraint.

We also examined the MNN's online training with different numbers of slave nodes. Since the MNN has a maximum number of four modules and an individual slave node is responsible for the local training of only one module, we repeated our online learning experiment with three and four slave nodes. For three and four slave nodes, we obtained $\gamma = 4$ and $\gamma = 7$, respectively. The results are depicted in Fig. 10. Fig. 10(a) reports the error rate CDF, and Figs. 10(b) and 10(c) depict the number of activated modules over time when employing three and four slave nodes, respectively. The following observation is made:

Observation 6. Fig. 10(a) shows that increasing the number of slave nodes has shifted the CDF curve to the left, satisfying the accuracy constraint. Additionally, Figs. 10(b) and 10(c) demonstrate that the constraint on the maximum number of activated modules was fulfilled when employing various numbers of slave nodes.

A key finding from Observation 6 is that as the number of slave nodes increases, we allow more modules to be activated by an input. As a result, more modules contribute to the online training corresponding to the input, improving the overall error rate. Moreover, the observation confirms that our proposed cluster overlap management algorithm (i.e. Algorithm 7) has effectively kept the maximum number of activated modules at the number of slave nodes.

6. Related work

This section reviews the literature on the resource-intensive training of DNNs. The training has three main aspects: (i) The data, (ii) the model, and (iii) the resource platform on which the model is trained. The literature leverages these aspects to address

the high resource demands of training DNNs. We classify the related literature into four main categories: Distributed Learning, approximation, modular architecture designs, and computation offloading. Distributed learning applies a form of partitioning to the data and model. Approximation mainly targets the model aspect and reduces the model's computational complexity. Some studies propose modular architecture designs to target the model aspect. Further, extensive research studies leverage the resource aspect and provide resource allocation and task scheduling algorithms to collaboratively offload computations across a broad set of resource platforms, ranging from resource-constrained end devices to resource-rich servers located at the edge or in the cloud.

Our work targets the model aspect to enable distributed deep learning across resource-constrained end nodes; however, it is enriched with the following distinctive properties: (i) The consideration of application-specific timeliness and accuracy constraints in addition to the device resource limitations, and (ii) architecture self-adaptivity. Table 1 provides a brief overview of the related literature. The following subsections review the literature and demonstrate where our work stands from the perspective of these properties.

6.1. Distributed learning

Distributed learning involves applying partitioning to the data (*data parallelism*), model (*model parallelism*), or both (*hybrid parallelism*).

Data parallelism allows several users to collaborate in training a single model using their local datasets, offering two benefits: (i) Improving privacy through local training and (ii) accelerating training through parallelism. *Federated learning* is an example. Data parallelism primarily assumes each processing node has sufficient computational and memory resources to train an entire model locally. However, our work concerns stringent resource limitations at the network edge that hinder training an entire DNN model on a single edge node. As a result, data parallelism does not apply to our work. For a survey on a data-parallelism approach like federated learning, see [51].

Model parallelism mitigates the memory and computing demands per node by partitioning the stack of 3D tensors within a DNN model and then allocating the partitions to separate data processing nodes to learn the same training data in a distributed manner. The partitioning occurs in two forms: (i) Horizontally at the tensor level, called *tensor-wise partitioning* [18–21], or (ii) vertically at the layer level, called *layer-wise partitioning* [22]. In tensor-wise partitioning, each partition is a slice from a tensor of a layer. The slicing can occur at different tensor dimensions, i.e. the channel, height, or width. Layer-wise partitioning considers one or multiple layers from the stack as a partition, with the tensors of the layers unaffected.

Pipeline parallelism [22–27] addresses resource under-utilization of layer-wise partitioning scheme as only one end node is active at a time with this scheme. It achieves this by pipelining data samples, offering parallelism at both sample and layer levels.

Hybrid parallelism attempts to uncover higher degrees of reductions in resource consumption and training time by enabling parallel training across various dimensions, i.e. sample, layer, channel, height, and width. The literature introduces hybrid parallelism in two general forms: (i) *Expert-designed* [28–30] and (ii) *automated* [31–34]. The expert-designed form manually determines a hybrid parallelization for a specific DNN based on an expert's domain knowledge and experience. However, the automated form aims to find an optimized hybrid parallelization in a search space defined by different dimensions.

Current research studies in model, pipeline, and hybrid parallelism categories mainly aim to accelerate training large-scale DNN models and increase throughput by distributing the model over large-scale CPU- or GPU-based clusters. In these works, model partitioning is tailored more towards accelerating offline batch learning of DNNs than enabling real-time on-device training with strict edge-level limitations on the resource capacity and number of processing nodes. Although acceleration can aid timeliness, these studies have not considered this factor, which is the main contributor to an online learning application.

6.2. Approximation

Some works employ approximation techniques like *model compression* and *model expansion* to offer on-device training of DNNs.

Model compression makes existing DNN model designs resource-efficient by reducing the number of parameters and MAC operations. *Pruning* and *quantization* are two common compression techniques. However, most works targeting model compression techniques are envisioned to enable resource-efficient inference. Only recently have compression techniques been considered for on-device training.

Pruning discards parts of a DNN model that have a non-essential effect on prediction accuracy. These parts may include parameters and activations during the forward pass or gradients during the backward pass. [35] targets training on a microcontroller and reduces the memory footprint of the full backward computation by skipping the gradient computation for the less important layers and sub-tensors. [36] targets training on more resource-rich devices but provides a dynamic sparse computation graph to prune activations during training according to the input sample instead of applying permanent pruning.

Quantization involves approximating the floating-point parameters, activations, and gradients with low-bit-width numbers, thus enabling DNN computations in a reduced-precision format. [35] reduces training resource demands by enabling real quantized training. To stabilize the quantized training and improve accuracy, the authors propose the *quantization-aware scaling (QAS)* method to automatically scale the gradients of tensors having different bit-precisions. [37] offers DNN training with 8-bit floating-point numbers.

Model expansion techniques handle the high resource demands of training a large DNN model by freezing it and incorporating a small trainable neural network. The training resource demand is decreased as the parameter updates only occur for the embedded small network. Furthermore, this approach enables a pre-trained DNN model to adapt to new data. TinyOL [38] enables online

learning on a microcontroller for an inference model by adding an extra trainable layer on top of it. TinyTL [39] and Rep-Net [40] target training on more resource-rich devices and introduce a lightweight module that is incorporated alongside the main model. All the parameters inside the module are updated, while the main model only has its biases updated. Rep-Net also introduces activation connectors that allow feature exchange between the main and side models.

Similar to approximation-based research studies, our work follows the fact that DNN architecture designs are not inherently resource-efficient due to their high architectural redundancies. These redundancies can be leveraged to enhance resource efficiency. However, the model architecture is usually pre-determined by the designer and fixed during training. Our work differs from these studies by allowing the model to adapt its architectural design according to the problem's complexity and available resources. This adaptive architectural design management is beneficial in two ways. First, the designer may make a sub-optimal design of the architecture, and the fixed pre-determined architecture may still be over-parameterized for a given problem, even if the architecture has been approximated. Second, in scenarios facing online data, new aspects of the problem's complexity emerge as new data arrives, and it is beneficial to let the model adjust its complexity by adjusting its architectural design to adapt to new data.

6.3. Modular architecture designs

Some works focus on modular designs for neural networks. TreeNets [52] facilitate distributed training for an ensemble of DNNs (i.e. groups of DNNs trained for the same task, with their outputs averaged to generate more accurate predictions). *Deep stack networks (DSNs)* [53,54] consist of modules stacked to build deep architectures, allowing for efficient parallelized training. However, the main objective of these works is to accelerate large-scale training of DNNs over large-scale GPU-based clusters, which is not our focus.

Some works use MNNs and fuzzy logic to capture the non-linear behavior of a system [41,42]. Our study is similar to these works in that they enable adaptive incremental learning of a non-linear behavior. However, these works assume no timeliness or resource constraints, and thus, there is no constraint on the MNN size (i.e. the number of modules and the architectural complexity of the modules). The authors of [55] propose a modular architecture for a points-of-interest (POI) recommendation system, where two modules extract local and global features separately. However, the system architecture is designed manually, lacking the self-adaptivity feature.

6.4. Computation offloading

Some studies explore resource allocation and task scheduling algorithms for offloading the DNN computations to more resource-rich entities like edge servers. There are two offloading schemes: (i) *Binary offloading* and (ii) *partial offloading*. Binary offloading, extensively researched for the inference phase, entails deciding either to outsource the DNN computation to an edge server or to perform a less-complex version of the computation on an end device. Partial offloading [56,57], on the other hand, partitions and distributes the resource-intensive DNN computation across end devices and an edge server. The algorithm proposed in [56] also leverages the size of the computation shares allocated to an end device and an edge server to accelerate the execution time further.

In these studies, the partitioning mostly takes a naive data-parallelism or layer-wise model-parallelism approach. Instead, they mainly focus on the resource allocation and scheduling aspects of the offloading. Moreover, although using an edge server instead of a cloud server mitigates the latency, security, and privacy concerns considerably, it does not necessarily eliminate them [58].

7. Conclusion

Online learning allows for adapting DNNs deployed at the edge for inference to variations in data caused by the environmental changes. However, the high computational complexity of DNNs leads to high resource demands for the training, which is in contrast to the edge resource limitations. This paper addresses this issue through distributed in-edge deep learning using the module set within an MNN. The MNN employs an online clustering algorithm that allows it to manage its computational complexity adaptively by adding, removing, and adjusting module clusters as input data comes. The proposed online clustering is subject to the application-specific accuracy and timeliness constraints. The simulation results show that the proposed online clustering effectively adheres to the application and resource constraints.

CRediT authorship contribution statement

Amin Yoosefi: Writing – review & editing, Writing – original draft, Software, Resources, Methodology, Formal analysis, Conceptualization. **Mehdi Kargahi:** Writing – review & editing, Validation, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Appendix A. Time and resource requirements of our online learning

This appendix presents: (i) The resource requirement of each part in our MNN as well as the feature extractor, (ii) the resource requirement of the master and individual slave nodes, and (iii) the total execution time of the MNN's online training. Our formulation considers a batch size of 1 since an online learning application is involved.

Resource Requirement of the MNN Parts. Our MNN consists of the module set, the integration network, and the RBF layer. The resource requirement of each part is calculated as follows.

A module consists of several CONV layers, with each defined by the following set of hyperparameters: The width (h_w) and height (h_h) of the input to the layer, the number of input channels (h_c), the number of kernels (h_m), the kernel size (h_k), the stride size (h_s), and the number of groups (h_g), as described in [59]. Assuming a homogeneous architecture design for all modules, they all have the same number of activations, Act_{mod}, and the same number of parameters, Par_{mod}, i.e. $Par_{mod} = |\theta_i| = |\theta_j|$, $i \neq j$. Act_{mod} and Par_{mod} are obtained by summing the number of activations and parameters over the constituent CONV layers. The number of parameters and the number of activations for a CONV layer are respectively computed using (A.1) and (A.2):

$$Par_{conv} = \frac{h_m \times h_c}{h_g} \times h_k^2, \tag{A.1}$$

$$Act_{conv} = h_h \times h_w \times h_c. \tag{A.2}$$

Training a module requires enough RAM space to accommodate: (i) The module's parameters and their changes and (ii) the values of activations and their corresponding error values. As a result, the amount of RAM required for online training of a module is denoted by RAM_{mod}, and is calculated as follows:

$$RAM_{mod} = 2(Par_{mod} + Act_{mod})b, \tag{A.3}$$

in which b represents the number of bits of a single scalar value. The amount of flash memory to hold the parameter set of a module is denoted by $Flash_{mod}$ and computed as:

$$Flash_{mod} = Par_{mod} b. \tag{A.4}$$

For the integration network, the number of parameters and the number of activations are respectively computed using (A.5) and (A.6):

$$Par_{integ} = |C|MR, \tag{A.5}$$

$$Act_{integ} = |C|R + M. \tag{A.6}$$

The RAM and flash memory requirements are denoted by RAM_{integ} and $Flash_{integ}$, respectively, and computed as:

$$RAM_{integ} = 2(Par_{integ} + Act_{integ})b, \tag{A.7}$$

$$Flash_{integ} = Par_{integ} b. \tag{A.8}$$

The set of all the l'-dimensional centers and all the real-valued radii form the parameter set of the RBF layer. Thus, the number of parameters and the flash memory requirement for this layer are respectively computed using (A.9) and (A.10):

$$Par_{rbf} = |C|(l' + 1), \tag{A.9}$$

$$Flash_{rbf} = Par_{rbf} b. \tag{A.10}$$

The required amount of RAM for the RBF layer is computed based on the proposed clustering algorithm (i.e. Algorithm 8), and is formulated as:

$$RAM_{rbf} = N_{scalar} b, \tag{A.11}$$

in which N_{scalar} denotes the number of scalars required.

Table A.2 outlines the primary sources of memory consumption. Our formulation disregards the sources of memory consumption with $N_{scalar} \in \mathcal{O}(1)$. We need RAM to accommodate some computed distances (Rows 1–3) and the values of radii, centers, and potentials (Row 4). The graph $\mathcal{G}$ requires $N_{scalar} = |C|^2$ (Row 5). The amount of RAM required by $\mathcal{Z}_{intr}$ depends on the maximum number of intersection points. The maximum number happens when (i) $|C_{md}|$ is maximum, i.e. $C_{md} = C$, (ii) all the hyperspheres in Algorithm 7 intersect at two points, and (iii) all the intersection points are qualified. Hence, $\max(|\mathcal{Z}_{intr}|) = 2(|C| + \binom{|C|}{2}) = |C|(|C|+1)$, and we need $N_{scalar} = |C|(|C|+1)l'$ for $\mathcal{Z}_{intr}$ (Row 6). The priority queue q has a maximum number of $\binom{|C|}{2}$ elements, each containing two cluster IDs and one priority value; hence, $N_{scalar} = 3/2|C|(|C| - 1)$ (Row 7).

To calculate N_{scalar} for $\mathcal{Q}$, we know that $\mathcal{Q}$ is a set of maximum cliques, where each maximum clique is a set of at most N_{node} cluster IDs. Therefore, we need the maximum number of these cliques in our graph. According to [60], a graph with $|C|$ vertices can have at most $(|C|/\omega)^\omega$ maximum cliques of size ω. Assuming $(|C|/\omega)^\omega$ as a function of $\omega \in \mathbb{R} \geq 1$, the extreme value happens at

Table A.2
Description of RAM requirements for the various variables used in the proposed online clustering algorithm (i.e. Algorithm 8).

Row	Variables	RAM requirement
1	$\{\|\|x'_k - c_i\|\|\}_{i\in C}$	$\|C\|b$
2	$\{\|\|c_v - c_i\|\|\}_{i\in C}$	$\|C\|b$
3	$\{\|\|c_v - z\|\|\}_{z\in \mathcal{Z}_{intr}}$	$\|C\|(\|C\|+1)b$
4	$\{(c_i, r_i, P_k(c_i))\}_{i\in C}$	$\|C\|(l'+2)b$
5	$\mathcal{G}$	$\|C\|^2 b$
6	$\mathcal{Z}_{intr}$	$\|C\|(\|C\|+1)l'b$
7	q	$^3/_2\|C\|(\|C\|-1)b$
8	$\mathcal{Q}$	$N_{node}\lceil\exp(^{\|C\|}/_e)\rceil b$

$\omega = {}^{|C|}/_e$. Hence, the maximum size of $\mathcal{Q}$ is $\exp({}^{|C|}/_e)$, and we have $N_{node}\lceil\exp({}^{|C|}/_e)\rceil$ as an upper bound for N_{scalar} (Row 8). Summing the RAM requirements over the table rows approximates the amount of RAM required by our online clustering algorithm.

Resource Requirements of the Feature Extractor. Since the feature extractor is also a CNN, the number of its parameters, Par_{fe}, is the sum of the number of the parameters within each constituent CONV layer. However, since the feature extractor is exclusively utilized for inference during online learning, we need sufficient RAM to accommodate the model's parameter set and a layer's input and output activations. Therefore, with Act_{fe} denoting the maximum number of input and output activations for a layer within the feature extractor model, the RAM and flash memory requirements are respectively formulated using (A.12) and (A.13):

$$RAM_{fe} = (Par_{fe} + Act_{fe})b, \tag{A.12}$$

$$Flash_{fe} = Par_{fe}b. \tag{A.13}$$

Resource Requirements of the End Nodes. The amount of flash memory that the master node requires to accommodate the parameter sets associated with the pre-trained MNN, the modules within our run-time MNN, and the RBF and integration layers, denoted by $Flash_{master}$, is calculated as:

$$Flash_{master} = N_{mod}(Par_{mod} + Par_{rbf})b + |C|Flash_{mod} + Flash_{fe} + Flash_{rbf} + Flash_{integ}. \tag{A.14}$$

As (A.14) implies, the maximum number of modules N_{mod} is limited by the master node's flash memory capacity.

The amount of RAM that a slave node requires for online training of its assigned module is denoted by RAM_{slave}, and is calculated as follows:

$$RAM_{slave} = RAM_{mod} + lb. \tag{A.15}$$

For the master node, we have:

$$RAM_{master} = RAM_{fe} + RAM_{rbf} + RAM_{integ} + (l + l')b. \tag{A.16}$$

Training Execution Time. The time required for one round of training of the MNN comprises two parts: (i) Computation time, t_{proc}, and (ii) communication time, t_{comm}, as formulated below:

$$t_{total} = t_{proc} + t_{comm}. \tag{A.17}$$

The *communication time* has four parts: (i) Sending the input vector and parameter sets to the slave nodes in Stage 2, with maximum communication overhead of $(N_{node}-1)(l + Par_{mod})$, (ii) receiving the forward pass outputs from the slave nodes in Stage 2, with a maximum communication overhead of $(N_{node}-1)R$, (iii) sending the error values to the slave nodes in Stage 3, with a maximum communication overhead of $(N_{node}-1)R$, and (iv) receiving the parameter changes from the slave nodes in Stage 3, with a maximum communication overhead of $(N_{node}-1)Par_{mod}$. Thus, we have:

$$t_{comm} = (N_{node}-1)(2Par_{mod} + 2R + l)\frac{b}{\beta}. \tag{A.18}$$

The *computation time* is formulated based on the number of *multiply-and-accumulate (MAC)* operations. Therefore, we first formulate the number of MAC operations for each part in our MNN. The number of MAC operations required by the feature extractor and a module, denoted by MAC_{fe} and MAC_{mod}, respectively, is the sum of the number of MAC operations involved within their constituent CONV layers. The number of MAC operations involved in training a CONV layer is denoted by MAC_{conv} and computed as:

$$MAC_{conv} = 3(h_h \times h_w \times h_s^{-2} \times Par_{conv}). \tag{A.19}$$

Eq. (A.19) also applies to the inference mode but with a coefficient of 1 instead of 3. This is because the computations involved within the error backpropagation and gradient computation steps of the backward pass phase are symmetric to the forward pass phase's computations. Hence, the number of operations required for the backward pass phase is approximately two times the number of

Table A.3
Number of MAC operations involved in the proposed online clustering algorithm (i.e. Algorithm 8).

Row	Computation	Number of MAC operations
1	$P_k(x'_k)$	$3l'$
2	$P_k(c_v)$	$3l'$
3	$\{P_k(c_i)\}_{i\in C}$	$\|C\|l'$
4	$\{\|x'_k - c_i\|\}_{i\in C}$	$\|C\|l'$
5	$\{\|c_v - c_i\|\}_{i\in C}$	$2\|C\|l'$
6	$\mathcal{Z}_{intr}$	$3\|C\|(\|C\|+1)l'$
7	$\{\|c_v - z\|\}_{z\in\mathcal{Z}_{intr}}$	$\|C\|(\|C\|+1)l'$
8	$\{\|c_i - z\|\}_{\substack{z\in\mathcal{Z}_{intr}\\ i\in C_{md}}}$	$\|C\|^2(\|C\|+1)l'$

operations for the forward pass phase [61,62]. Therefore, we use a coefficient of 1 when calculating the number of MAC operations for the feature extractor, as it is used only for inference.

The number of MAC operations for training the integration network is denoted by MAC_{integ} and computed as follows:

$$MAC_{integ} = 3Par_{integ}, \tag{A.20}$$

which (A.20) has a coefficient of 3, one for the forward pass and two for the backward pass.

The number of MAC operations for training the RBF layer is computed based on the proposed online clustering algorithm. This algorithm relies on feature points' distances to define the cluster potential values, decide on the clustering operation, and manage the cluster overlaps. This fact introduces distance computation as the primary operation in our online clustering. As a result, we approximate MAC_{rbf} by counting the number of involved distance computations in the l'-dimensional feature space. A distance computation between two feature points x'_1 and x'_2, i.e. $\|x'_1 - x'_2\|$, can be considered a sequence of l' MAC operations, each preceded by a subtract operation. Since a MAC operation is more resource-intensive than a subtract operation, we approximate the computational overhead of a distance calculation by l' MAC operations. Therefore, we formulate MAC_{rbf} as follows:

$$MAC_{rbf} = N_{dist}l', \tag{A.21}$$

in which N_{dist} determines the number of distance computations.

Table A.3 outlines the primary sources of distance computations. Having a new feature data point x'_k received or a new center c_v suggested by the clustering algorithm, calculating its potential value using (9) requires three distance computations for $Z^\top Z$, σ_k, and η_k; thus, $N_{dist} = 3$ (Rows 1 and 2). Updating the potential value of each existing center using (12) costs one distance computation for $\zeta_k(.)$, i.e. $N_{dist} = |C|$ (Row 3). Calculating the distance between x'_k and all existing centers costs $N_{dist} = |C|$ (Row 4). Once computed, these distances can be utilized in Eqs. (23), (24), and (30) in Algorithm 2. Similarly, calculating the distance between c_v and all centers requires $N_{dist} = |C|$. This calculation is required once for lines 3 and 8 in Algorithm 7 and once for Line 4 in Algorithm 3. Therefore, we have $N_{dist} = 2|C|$ (Row 5). We use an algorithm that requires three distance computations to find each intersection point of two hyperspheres. Considering the maximum number of the intersection points, we have $N_{dist} = 3|C|(|C|+1)$ for obtaining $\mathcal{Z}_{intr}$ (Row 6). Sorting intersection points in Line 9 of Algorithm 7 costs $N_{dist} = |C|(|C|+1)$ (Row 7). Moreover, Line 10 of Algorithm 7 calculates the distance between the points in $\mathcal{Z}_{intr}$ and C_{md} with cost $N_{dist} = |C|^2(|C|+1)$ (Row 8). Summing the computational overheads outlined in Table A.3 gives an approximate for MAC_{rbf}.

With the number of MAC operations computed, the computation time for our online training is calculated as:

$$t_{proc} = t_{fe} + t_{rbf} + t_{mod} + t_{integ}, \tag{A.22}$$

in which t_{fe} represents the time for feature extracting an input data; and t_{rbf}, t_{mod}, and t_{integ} denote the training execution time of the RBF layer, a module, and the integration network, each computed as follows:

$$t_{fe} = \frac{MAC_{fe} \times MAC_{flop}}{\rho}, \tag{A.23}$$

$$t_{mod} = \frac{MAC_{mod} \times MAC_{flop}}{\rho}, \tag{A.24}$$

$$t_{rbf} = \frac{MAC_{rbf} \times MAC_{flop}}{\rho}, \tag{A.25}$$

$$t_{integ} = \frac{MAC_{integ} \times MAC_{flop}}{\rho}, \tag{A.26}$$

wherein MAC_{flop} represents the number of flops performed per MAC operation. MAC_{flop} is a constant for a processing core of a given microarchitecture [45].

Algorithm 10 (Gen_ModLC): Generate the module learning curve.

Input: Training and validation datasets S_{trn}^{ana} and S_{val}^{ana}, modular neural network MNN^{init} pre-trained on S_{trn}^{ana} and S_{val}^{ana}, pre-trained MNN's clustering $\{(c_i^{init}, r_i^{init})\}_{i=1}^{N_{mod}}$

Require: Total number of samples sizes N_{smp}^{iter}, initial number of samples N_{smp}^{init}, the first cluster increase made to the initial number of samples N_{smp}^{base}

Output: Learning curve's point set $\{(r_i, Err_i)\}_{i=1}^{N_{smp}^{iter}}$

1: **for** $i \in \{1, 2, \dots, N_{mod}\}$ **do**
2: $MNN \leftarrow$ clone MNN^{init};
3: $S_{trn}^{ana(clone)} \leftarrow$ clone S_{trn}^{ana}; $S_{trn}^{clust} \leftarrow \emptyset$; $S_{val}^{clust} \leftarrow \emptyset$;
4: Set S_{trn}^{clust} as the module i's cluster and let the other modules' clusters be defined by $(c_l^{init}, r_l^{init}), l \neq i$;
5: Freeze MNN except for module i and the integration part;
6: **for** $j \in \{1, 2, \dots, N_{smp}^{iter}\}$ **do**
7: Initialize module i with raw parameters;
8: Calculate $N_{smp,j}^{step}$ using (B.2);
9: Select points from $S_{trn}^{ana(clone)}$ whose features are the $N_{smp,j}^{step}$ nearest to c_i^{init}, and move them to S_{trn}^{clust};
10: $r \leftarrow$ calculate the maximum distance between the feature of a point in S_{trn}^{clust} and c_i^{init};
11: $S_{val}^{clust} \leftarrow$ obtain all points from S_{val}^{ana} whose features are inside a circle with center c_i^{init} and radius r;
12: $Err \leftarrow$ batch-train MNN on S_{trn}^{clust} with S_{val}^{clust} as the validation dataset;
13: Correspond (r, Err) with $|S_{trn}^{clust}|$ in the module i's learning curve point set;
14: Aggregate radius values to associate each sample size with a single radius value;
15: Aggregate error rates to generate a single learning curve;
16: **return** aggregated learning curve point set;

Appendix B. Confidence interval of the module learning curve

Algorithm 10 targets each module within the MNN and conducts a similar analysis to estimate its learning curve. The analysis involves incrementally increasing the cluster radius of a target module and calculating the corresponding MNN's accuracy. The inputs to the algorithm are training and validation datasets S_{trn}^{ana} and S_{val}^{ana}. The analysis taken for each target module is described in the following paragraphs.

First, to capture the accuracy variation caused when the target module receives more training samples due to an increase in its cluster radius, it is crucial to eliminate the impact of cluster radius changes and poor training performance for the other modules. Thus, the algorithm uses a clone of an MNN already pre-trained on S_{trn}^{ana}. Then, for each increase in the module's cluster radius, the cloned MNN is set up by initializing the target module with raw parameters, while the other modules keep their pre-trained parameters. In other words, the target module is the only module that receives training and cluster radius changes during its analysis, whereas the remaining modules contribute only to inference and preserve their initial parameter sets and cluster radii from the pre-trained MNN (Lines 2–5 and 7).

Next, the module's cluster size increases gradually, starting from including an initial number of samples and growing incrementally by having more training data samples from the training dataset. The initial number of samples is denoted by N_{smp}^{init} and determined by a minimum radius $\underline{r_{cvr}}$ that ensures the feature space is covered by employing at most N_{mod} number of clusters, calculated as follows:

$$\underline{r_{cvr}} = \frac{\overline{d_{p2p}}}{2N_{mod}}, \tag{B.1}$$

in which $\overline{d_{p2p}}$ is the maximum distance between two feature points in the feature space.

Starting with an empty cluster, the number of samples added to the target module's cluster at the jth growth, denoted by $N_{smp,j}^{step}$, is as follows:

$$N_{smp,j}^{step} = \begin{cases} N_{smp}^{init}, & \text{if } j = 1 \\ (2^{j-1} - 1)N_{smp}^{base}, & \text{if } j > 1, \end{cases} \tag{B.2}$$

in which N_{smp}^{base} is the first cluster size increase made to the initial number of samples. The cluster size after the jth growth is computed as follows:

$$N_{smp,j} = N_{smp}^{init} + (2^j - j - 1)N_{smp}^{base}, \quad j \geq 1, \tag{B.3}$$

and the maximum value of j for which we still have $N_{smp,j} \leq |S_{trn}^{ana}|$ is denoted by N_{smp}^{iter}.

Algorithm 11 (Gen_CI): Generate a confidence interval for the module learning curve resulting from Algorithm 10.

Input: Training and validation datasets $\mathcal{S}_{trn}^{ana}$ and $\mathcal{S}_{val}^{ana}$, modular neural network MNN^{init} pre-trained on $\mathcal{S}_{trn}^{ana}$ and $\mathcal{S}_{val}^{ana}$, pre-trained MNN's clustering $\{(c_i^{init}, r_i^{init})\}_{i=1}^{N_{mod}}$
Require: Number of experiments N_{exp}, confidence level α%
Output: Lower- and upper-bound learning curves

1: **for** $j \in \{1, 2, \ldots, N_{exp}\}$ **do**
2: $\quad \mathcal{S}_{trn}^{ana(bstrp)} \leftarrow$ generate a bootstrap dataset from $\mathcal{S}_{trn}^{ana}$;
3: $\quad MNN \leftarrow$ fine-tune a clone of MNN^{init} on $\mathcal{S}_{trn}^{ana(bstrp)}$;
4: $\quad$ Call **Gen_ModLC**$(\mathcal{S}_{trn}^{ana(bstrp)}, \mathcal{S}_{val}^{ana}, MNN, \{(c_i^{init}, r_i^{init})\}_{i=1}^{N_{mod}})$ to generate the j-th learning curve;
5: Aggregate corresponding radius values from all experiments;
6: Sort the experiments' error rates in ascending order for each radius;
7: Set the $(N_{exp} \times (1 \pm \alpha/100)/2)$-th error rates for each radius as the corresponding lower- and upper-bound points;
8: Form the lower- and upper-bound learning curves by fitting one curve to all the lower-bound points and another to all the upper-bound points;
9: **return** lower- and upper-bound learning curves;

Each time, the algorithm expands the module's cluster by targeting the training samples that have not yet been included in the cluster and adding $N_{smp,j}^{step}$ of those whose corresponding features are closest to the target module's cluster center (Lines 8 and 9). The radius of the expanded cluster is calculated as the maximum distance between the cluster center and a feature data point inside the cluster (Line 10). The data samples inside the expanded cluster, denoted by $\mathcal{S}_{trn}^{clust}$, will be used as training data for the ongoing training. The corresponding validation dataset, denoted by $\mathcal{S}_{val}^{clust}$, is formed from the points in $\mathcal{S}_{val}^{ana}$ whose features fall within the target module's cluster boundary (Line 11). With the training and validation datasets $\mathcal{S}_{trn}^{clust}$ and $\mathcal{S}_{val}^{clust}$ obtained for a sample size under analysis, the algorithm initiates batch-training of the cloned MNN on the datasets, and the error rate obtained from the batch-training is reported as the error rate corresponding to that sample size (Lines 12 and 13).

With the modules' learning curves obtained, each sample size corresponds to different radius values coming from different learning curves. Algorithm 10 associates each sample size with a single radius value by aggregating the corresponding radius values coming from the modules' learning curves (Line 14). Furthermore, the error rates corresponding to a radius value resulting from different curves are aggregated to generate a single learning curve (Line 15). We use average as our aggregation function.

As explained, Algorithm 10 estimates the modules' learning curves and generates an aggregated curve. However, to account for the uncertainty in estimating a module learning curve, we use Algorithm 11 to form a confidence interval for the curve resulting from Algorithm 10. The upper-bound curve is then used to determine $\underline{r}$ and $\overline{r}$.

Algorithm 11 uses the *Bootstrapping* method [63] and conducts multiple experiments to generate the confidence interval, and then it returns the corresponding upper- and lower-bound curves. Each experiment creates a bootstrap training dataset by resampling from the original training dataset $\mathcal{S}_{trn}^{ana}$ (Line 2). The sampling is done with replacement, and the bootstrap dataset's size equals the size of the original one. The same validation dataset $\mathcal{S}_{val}^{ana}$ is used for all experiments. With the bootstrap training dataset available, an MNN must be trained on it. To avoid training an MNN from scratch for each experiment and accelerate the process, we train an MNN on the original training dataset, and then fine-tune it on the bootstrap training dataset for each experiment (Line 3). The validation dataset $\mathcal{S}_{val}^{ana}$, the bootstrap training dataset, and the fine-tuned MNN are fed into Algorithm 10 to generate a learning curve corresponding to that experiment (Line 4). Once all the experiments are concluded, the radius values corresponding to a sample size resulting from different experiments are aggregated to associate each sample size with a single radius value (Line 5). Next, for each radius value, the error rate estimates from different experiments are sorted in ascending order (Line 6). Assuming a confidence level of α% and with N_{exp} denoting the number of experiments, the $(N_{exp} \times (1-\alpha/100)/2)$-th and $(N_{exp} \times (1+\alpha/100)/2)$-th estimate values represent the lower and upper bounds for the error rate corresponding to that radius (Line 7). Finally, the confidence interval is established by fitting two curves, one on the upper-bound points and another on the lower-bound points, representing the upper- and lower-bound learning curves, respectively (Line 8).

References

[1] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, in: Advances in Neural Information Processing Systems, NeurIPS, 2012.
[2] J. Pan, et al., AI-driven blind signature classification for IoT connectivity: A deep learning approach, IEEE Trans. Wirel. Commun. 21 (8) (2022) 6033–6047.
[3] J. Lin, et al., MCUNet: Tiny deep learning on IoT devices, in: Advances in Neural Information Processing Systems, NeurIPS, Vol. 33, 2020, pp. 11711–11722.
[4] J. Lin, et al., MCUNetV2: Memory-efficient patch-based inference for tiny deep learning, 2021, CoRR abs/2110.15352. arXiv:2110.15352. URL https://arxiv.org/abs/2110.15352.
[5] C. Banbury, et al., MicroNets: Neural network architectures for deploying TinyML applications on commodity microcontrollers, in: Proceedings of Machine Learning and Systems, MLSys, Vol. 3, 2021, pp. 1–16.
[6] A. Burrello, et al., Dory: Automatic end-to-end deployment of real-world DNNs on low-cost IoT MCUs, IEEE Trans. Comput. 70 (8) (2021) 1253–1268.
[7] I. Fedorov, R.P. Adams, M. Mattina, P. Whatmough, Sparse: Sparse architecture search for CNNs on resource-constrained microcontrollers, in: Advances in Neural Information Processing Systems, NeurIPS, Vol. 32, 2019.

[8] E. Liberis, N.D. Lane, Neural networks on microcontrollers: Saving memory at inference via operator reordering, 2019, CoRR abs/1910.05110. arXiv:1910.05110. URL http://arxiv.org/abs/1910.05110.
[9] M. Rusci, A. Capotondi, L. Benini, Memory-driven mixed low-precision quantization for enabling deep network inference on microcontrollers, in: Proceedings of Machine Learning and Systems, MLSys, Vol. 2, 2020, pp. 326–335.
[10] T. Chen, B. Xu, C. Zhang, C. Guestrin, Training deep nets with sublinear memory cost, 2016, CoRR abs/1604.06174. arXiv:1604.06174. URL http://arxiv.org/abs/1604.06174.
[11] C. Zhang, K. Zhang, Y. Li, A causal view on robustness of neural networks, in: Advances in Neural Information Processing Systems, NeurIPS, Vol. 33, 2020, pp. 289–301.
[12] X. Gao, R.K. Saha, M.R. Prasad, A. Roychoudhury, Fuzz testing based data augmentation to improve robustness of deep neural networks, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 1147–1158.
[13] H. Yokoyama, S. Onoue, S. Kikuchi, Towards building robust DNN applications: An industrial case study of evolutionary data augmentation, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 1184–1188.
[14] K. Pei, Y. Cao, J. Yang, S. Jana, DeepXplore: Automated whitebox testing of deep learning systems, in: Proceedings of the 26th Symposium on Operating Systems Principles, 2017, pp. 1–18.
[15] F. Lambert, Understanding the fatal Tesla accident on autopilot and the Nhtsa Probe, Electrek 1 (2016).
[16] H.T. Dinh, C. Lee, D. Niyato, P. Wang, A survey of mobile cloud computing: Architecture, applications, and approaches, Wirel. Commun. Mob. Comput. 13 (18) (2013) 1587–1611.
[17] W. Shi, et al., Edge computing: Vision and challenges, IEEE Internet Things J. 3 (5) (2016) 637–646.
[18] J. Dean, et al., Large scale distributed deep networks, in: Advances in Neural Information Processing Systems, NeurIPS, Vol. 25, 2012.
[19] M. Wang, et al., Minerva: A scalable and highly efficient training platform for deep learning, in: NIPS Workshop, Distributed Machine Learning and Matrix Computations, 2014.
[20] T. Chilimbi, Y. Suzue, J. Apacible, K. Kalyanaraman, Project Adam: Building an efficient and scalable deep learning training system, in: 11th USENIX Symposium on Operating Systems Design and Implementation, 2014, pp. 571–582.
[21] J.K. Kim, et al., STRADS: A distributed framework for scheduled model parallel machine learning, in: Proceedings of the Eleventh European Conference on Computer Systems, ACM, 2016, pp. 1–16.
[22] A. Paszke, et al., PyTorch: An imperative style, high-performance deep learning library, in: Advances in Neural Information Processing Systems, NeurIPS, Vol. 32, 2019.
[23] Y. Huang, et al., Gpipe: Efficient training of giant neural networks using pipeline parallelism, in: Advances in Neural Information Processing Systems, NeurIPS, Vol. 32, 2019.
[24] C. Kim, et al., TorchGpipe: On-the-fly pipeline parallelism for training giant models, 2020, CoRR abs/2004.09910. arXiv:2004.09910. URL https://arxiv.org/abs/2004.09910.
[25] A. Harlap, et al., PipeDream: Fast and efficient pipeline parallel DNN training, 2018, CoRR abs/1806.03377. arXiv:1806.03377. URL http://arxiv.org/abs/1806.03377.
[26] L. Guan, W. Yin, D. Li, X. Lu, XPipe: Efficient pipeline model parallelism for multi-GPU DNN training, 2019, CoRR abs/1911.04610. arXiv:1911.04610. URL http://arxiv.org/abs/1911.04610.
[27] C.C. Chen, C.L. Yang, H.Y. Cheng, Efficient and robust parallel DNN training through model parallelism on multi-GPU platform, 2018, CoRR abs/1809.02839. arXiv:1809.02839. URL http://arxiv.org/abs/1809.02839. Withdrawn..
[28] A. Krizhevsky, One weird trick for parallelizing convolutional neural networks, 2014, CoRR abs/1404.5997. arXiv:1404.5997. URL http://arxiv.org/abs/1404.5997.
[29] N. Shazeer, et al., Mesh-TensorFlow: Deep learning for supercomputers, in: Advances in Neural Information Processing Systems, NeurIPS, Vol. 31, 2018, pp. 10414–10423.
[30] M. Shoeybi, et al., Megatron-LM: Training multi-billion parameter language models using model parallelism, 2019, CoRR abs/1909.08053. arXiv:1909.08053. URL http://arxiv.org/abs/1909.08053.
[31] Z. Jia, S. Lin, C.R. Qi, A. Aiken, Exploring hidden dimensions in parallelizing convolutional neural networks, in: International Conference on Machine Learning, ICML, PMLR, 2018, pp. 2279–2288.
[32] Z. Jia, M. Zaharia, A. Aiken, Beyond data and model parallelism for deep neural networks, in: Proceedings of Machine Learning and Systems, MLSys, Vol. 1, 2019, pp. 1–13.
[33] A. Mirhoseini, et al., Device placement optimization with reinforcement learning, in: International Conference on Machine Learning, ICML, PMLR, 2017, pp. 2430–2439.
[34] M. Wang, C. Huang, J. Li, Supporting very large models using automatic dataflow graph partitioning, in: Proceedings of the Fourteenth EuroSys Conference 2019, 2019, pp. 1–17.
[35] J. Lin, et al., On-device training under 256kb memory, in: Advances in Neural Information Processing Systems, NeurIPS, Vol. 35, 2022, pp. 22941–22954.
[36] L. Liu, et al., Dynamic sparse graph for efficient deep learning, 2018, CoRR abs/1810.00859. arXiv:1810.00859. URL http://arxiv.org/abs/1810.00859.
[37] N. Wang, et al., Training deep neural networks with 8-bit floating-point numbers, in: Advances in Neural Information Processing Systems, NeurIPS, Vol. 31, 2018.
[38] H. Ren, D. Anicic, T.A. Runkler, TinyOL: TinyML with online-learning on microcontrollers, in: 2021 IEEE International Joint Conference on Neural Networks, IJCNN, IEEE, 2021, pp. 1–8.
[39] H. Cai, C. Gan, L. Zhu, S. Han, TinyTL: Reduce memory, not parameters for efficient on-device learning, in: Advances in Neural Information Processing Systems, NeurIPS, Vol. 33, 2020, pp. 11285–11297.
[40] L. Yang, A.S. Rakin, D. Fan, Rep-Net: Efficient on-device learning via feature reprogramming, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2022, pp. 12277–12286.
[41] J. Qiao, Z. Zhang, Y. Bo, An online self-adaptive modular neural network for time-varying systems, Neurocomputing 125 (2014) 7–16.
[42] W. Li, M. Li, J. Qiao, X. Guo, A feature clustering-based adaptive modular neural network for nonlinear system modeling, ISA Trans. 100 (2020) 185–197.
[43] P.P. Angelov, D.P. Filev, An approach to online identification of Takagi-Sugeno fuzzy models, IEEE Trans. Syst. Man Cybern. B (Cybernetics) 34 (1) (2004) 484–498.
[44] G. James, et al., An Introduction to Statistical Learning, Vol. 112, Springer, 2013.
[45] R. Dolbeau, Theoretical peak FLOPS per instruction set: A tutorial, J. Supercomput. 74 (3) (2018) 1341–1377.
[46] [link] URL https://en.wikichip.org/wiki/flops.
[47] A.G. Howard, et al., MobileNets: Efficient convolutional neural networks for mobile vision applications, 2017, CoRR abs/1704.04861. arXiv:1704.04861. URL http://arxiv.org/abs/1704.04861.
[48] M. Sandler, et al., MobileNetV2: Inverted residuals and linear bottlenecks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2018, pp. 4510–4520.
[49] A. Krizhevsky, G. Hinton, Learning multiple layers of features from tiny images, 2009.
[50] T. Akiba, et al., Optuna: A next-generation hyperparameter optimization framework, in: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2019, pp. 2623–2631.

[51] C. Zhang, et al., A survey on federated learning, Knowl.-Based Syst. 216 (106775) (2021).
[52] S. Lee, et al., Why m heads are better than one: Training a diverse ensemble of deep networks, 2015, CoRR abs/1511.06314. arXiv:1511.06314. URL http://arxiv.org/abs/1511.06314.
[53] L. Deng, D. Yu, J. Platt, Scalable stacking and learning for building deep architectures, in: 2012 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP, IEEE, 2012, pp. 2133–2136.
[54] B. Hutchinson, L. Deng, D. Yu, Tensor deep stacking networks, IEEE Trans. Pattern Anal. Mach. Intell. 35 (8) (2012) 1944–1957.
[55] Z. Guo, et al., Deep-distributed-learning-based POI recommendation under mobile-edge networks, IEEE Internet Things J. 10 (1) (2023) 303–317.
[56] A. Mahmood, Y. Hong, M.K. Ehsan, S. Mumtaz, Optimal resource allocation and task segmentation in IoT enabled mobile edge cloud, IEEE Trans. Veh. Technol. 70 (12) (2021) 13294–13303.
[57] Y. Liu, L. Zhang, Y. Wei, Z. Wang, Energy efficient training task assignment scheme for mobile distributed deep learning scenario using DQN, in: 2019 IEEE 7th International Conference on Computer Science and Network Technology, ICCSNT, IEEE, 2019, pp. 442–446.
[58] A. Alwarafy, et al., A survey on security and privacy issues in edge-computing-assisted internet of things, IEEE Internet Things J. 8 (6) (2020) 4004–4022.
[59] [link]. URL https://pytorch.org/docs/stable/nn.html.
[60] D.R. Wood, On the maximum number of cliques in a graph, Graphs Combin. 23 (3) (2007) 337–352.
[61] K. He, J. Sun, Convolutional neural networks at constrained time cost, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2015, pp. 5353–5360.
[62] H. Kim, H. Nam, W. Jung, J. Lee, Performance analysis of CNN frameworks for GPUs, in: 2017 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, IEEE, 2017, pp. 55–64.
[63] B. Efron, Bootstrap methods: Another look at the jackknife, in: Breakthroughs in Statistics: Methodology and Distribution, Springer, 1992, pp. 569–593.