Reusing Convolutional Neural Network Models through Modularization and Composition

BINHANG QI, SKLSDE, School of Computer Science and Engineering, Beihang University, China
HAILONG SUN*, SKLSDE, School of Software, Beihang University, China
HONGYU ZHANG, Chongqing University, China
XIANG GAO, School of Software, Beihang University, China

With the widespread success of deep learning technologies, many trained deep neural network (DNN) models are now publicly available. However, directly reusing the public DNN models for new tasks often fails due to mismatching functionality or performance. Inspired by the notion of modularization and composition in software reuse, we investigate the possibility of improving the reusability of DNN models in a more fine-grained manner. Specifically, we propose two modularization approaches named CNNSplitter and GradSplitter, which can decompose a trained convolutional neural network (CNN) model for N-class classification into N small reusable modules. Each module recognizes one of the N classes and contains a part of the convolution kernels of the trained CNN model. Then, the resulting modules can be reused to patch existing CNN models or build new CNN models through composition. The main difference between CNNSplitter and GradSplitter lies in their search methods: the former relies on a genetic algorithm to explore search space, while the latter utilizes a gradient-based search method. Our experiments with three representative CNNs on three widely-used public datasets demonstrate the effectiveness of the proposed approaches. Compared with CNNSplitter, GradSplitter incurs less accuracy loss, produces much smaller modules (19.88% fewer kernels), and achieves better results on patching weak models. In particular, experiments on GradSplitter show that (1) by patching weak models, the average improvement in terms of precision, recall, and F1-score is 17.13%, 4.95%, and 11.47%, respectively, and (2) for a new task, compared with the models trained from scratch, reusing modules achieves similar accuracy (the average loss of accuracy is only 2.46%) without a costly training process. Our approaches provide a viable solution to the rapid development and improvement of CNN models.

CCS Concepts: • **Software and its engineering → Reusability**.

Additional Key Words and Phrases: model reuse, convolutional neural network, CNN modularization, module composition

1 INTRODUCTION

Modularization and composition are fundamental concepts in software engineering, which facilitate software development, reuse, and maintenance by dividing an entire software system into a set of smaller modules. Each module is capable of carrying out a certain task and can be composed with other modules [46–48]. For instance, when debugging a buggy program, testing and patching the module that contains the bug will be much easier than analysing the entire program.

*Corresponding author. Hailong Sun is also with Hangzhou Innovation Institute, Beihang University, China

Authors' addresses: Binhang Qi, SKLSDE, School of Computer Science and Engineering, Beihang University, China, binhangqi@buaa.edu.cn; Hailong Sun, SKLSDE, School of Software, Beihang University, China, sunhl@buaa.edu.cn; Hongyu Zhang, Chongqing University, China, hyzhang@cqu.edu.cn; Xiang Gao, School of Software, Beihang University, China, xiang_gao@buaa.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1049-331X/2023/11-ART $15.00
https://doi.org/10.1145/3632744

Just Accepted

We highlight that modularization and composition are also important for deep neural networks (DNNs), such as the convolutional neural network (CNN) that is one of the most effective DNNs for processing a variety of tasks [17, 27, 36]. Due to the widespread application of CNNs, many trained CNN models are now publicly available, and reusing existing trained models has gained increasing attention recently [1, 43, 44]. However, reusing existing models has two main challenges: (1) existing CNN models from old projects may perform unsatisfactorily in the target task, and (2) models that can solve the target tasks may not exist. To improve the accuracy of weak CNN models, developers often retrain the models using new data, model structures, training strategies, or hyperparameter values. Additionally, to obtain a new CNN model for a new project, developers can evaluate the accuracy of public CNN models on their own test data and choose the model with the highest accuracy for reuse. However, current practice in model development and improvement has the following limitations: (1) as the neural networks are getting deeper and the numbers of parameters and convolution operations are getting larger, the time and computational cost required for training the CNN models are rapidly growing. (2) even if existing models satisfy developers' requirements, directly reusing the model with the highest overall accuracy may not always be the best solution. For instance, the model with the highest overall accuracy may be less accurate in recognizing a certain class than other models.

At a conceptual level, a CNN model is analogous to a program [38, 49, 77]. Inspired by the application of modularization and composition in software development and debugging, it is natural to ask: *can the concepts of modularization and composition be applied to CNN models for facilitating the development and improvement of CNN models?* Through modularization and composition, the weak modules in a weak CNN model can be identified and patched separately; thus, the weak model can be improved without costly retraining the entire model. Moreover, some modules can be reused to create a new CNN model without costly retraining. Also, a module is much smaller (*i.e.* has fewer weights) than the entire model, which is essential for reducing the overhead of model reuse.

However, decomposing a CNN model into modules faces two main challenges: (1) CNN models are constructed with uninterpretable weight matrices, unlike software programs, which are composed of readable statements. Decomposing CNN models into distinct modules is challenging without fully comprehending the effect of each weight. (2) identifying the relations between neurons and prediction tasks is difficult as the connections between neurons in a CNN are complex and dense. To this end, Pan *et al.* [43] proposed decomposing a fully connected neural network (FCNN) model for N-class classification into N modules, one for each class in the original model. They achieved model decomposition through *uncompressed modularization*, which removes individual weights from a trained FCNN model and results in modules with sparse weight matrices (to be discussed in Section 7.1). However, this approach [43] cannot be applied to other advanced DNN models like CNN models due to the weight sharing [29, 78] in CNNs. That is, different from FCNNs where the relationship between weights and neurons is many-to-one, the relationship in CNNs is many-to-many. Removing weights for one neuron in CNNs will also affect all other neurons. Although the follow-up work [44] can be applied to decompose CNN models, it is still an uncompressed modularization approach. In uncompressed modularization, a module with a sparse weight matrix has the same size as the trained model, resulting in a significant overhead of module reuse.

To address the above challenges, we propose the first *compressed modularization* approach called $CNNSplitter$, which applies a genetic algorithm to decompose a CNN model into smaller and separate modules. By compressed modularization, we mean removing convolution kernels instead of individual weights, resulting in modules with smaller weight matrices. Inspired by the finding that different convolution kernels learn to extract different features from the data [78], we generate modules by selecting different convolution kernels in a CNN model. Therefore, different from the existing work [43], CNNSplitter decomposes a trained N-class CNN model (TM for short) into N CNN modules by removing unwanted convolution kernels. To decompose a trained CNN model for N-class classification, we formulate the modularization problem as a search problem. Search-based algorithms have been proven to be very successful in solving software engineering problems [20, 33]. Given a space of

Just Accepted

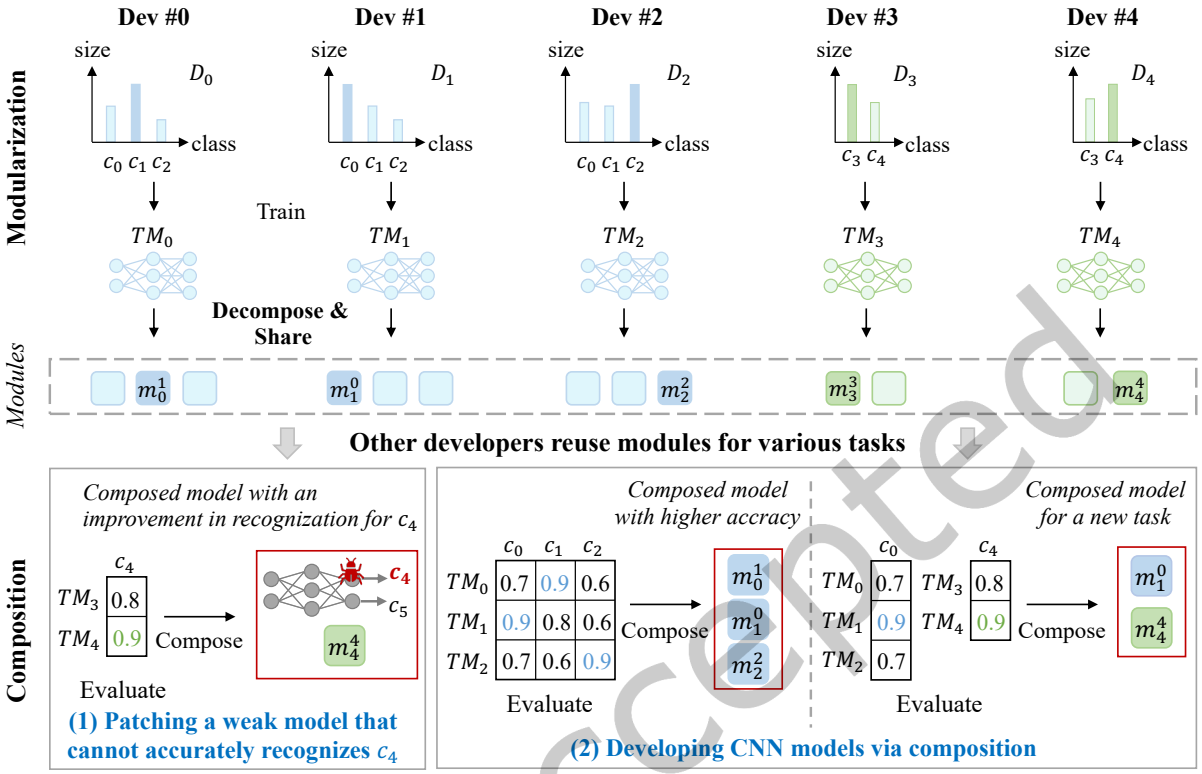

Fig. 1. An illustration of model modularization and composition.

candidate solutions, search-based approaches usually search for the optimal solution over the search space according to a user-defined objective function. In the context of model decomposition, the candidate solution is defined as a set of sub-models containing a part of the CNN model's kernels, while the search objective is to search N sub-models (as N modules) with each of them recognizing one class. To search for optimal modules, CNNSplitter employs a genetic algorithm that utilizes a combination of modules' accuracy and the difference between modules as the objective function. In this way, a trained CNN model is decomposed into N modules. Each module corresponds to one of the N classes and evaluates whether or not an input belongs to the corresponding class.

Due to the huge search space, the traditional search approach (*e.g.* genetic algorithm) could incur a large time cost. To improve the efficiency of modularization, we further propose a gradient-based compressed modularization approach named *GradSplitter*, which applies a gradient-based search method to explore the search space. To decompose a *TM* into N modules, GradSplitter initializes a *mask* and a *head* for each module. The *mask* consists of an array of 0s and 1s, where 0 (or 1) indicates that the corresponding convolution kernels in the *TM* are removed (or retained). The *head*, consisting of fully connected (FC) layers, is appended after the output layer of the masked *TM* to convert N-classification to binary classification, *i.e.* whether an input belongs to the class of the corresponding module. Then, a module M is created by $M = mask \odot TM \oplus head$, where $\odot$ denotes the removal of the corresponding kernels from *TM* according to the 0s in the *mask* and $\oplus$ denotes the appending of the *head* after the masked *TM*. GradSplitter combines the outputs of N modules and optimizes the masks and

Just Accepted

heads of N modules jointly on the N-class classification task. For the optimization, a gradient descent approach is used to minimize the number of retained kernels and the cross-entropy between the predicted class and the actual class. In this way, GradSplitter can search for N modules with each of them containing only relevant kernels.

As illustrated in Figure 1, third-party developers, labeled as "Dev #0-4", generally train models for various tasks, such as distinct tasks (e.g., trained models TM_2 and TM_3) or similar tasks but using datasets with different distributions (e.g., TM_0, TM_1, and TM_2). With the modularization technique, developers not only release their trained CNN models (*TMs* for short) but also share a set of smaller and reusable modules decomposed from *TMs*. With the shared modules, similar to reusing complete models, other developers can evaluate and reuse the suitable modules according to their demands without costly training. For instance, to improve the recognition of a weak CNN model on a target class, the module with the best performance (*e.g.* F1 score) in classifying the target class is reused as a patch to be combined with the weak CNN model. Additionally, developers can build new CNN models entirely by combining optimal modules. Consequently, composed models (*CMs* for short) are constructed through module reuse, which can address new tasks or achieve better performance than existing trained models.

We evaluate CNNSplitter and GradSplitter using three representative CNNs with different structures on three widely-used datasets (CIFAR-10 [26], CIFAR-100 [26], and SVHN [41]). The experimental results show that by decomposing a *TM* into modules with GradSplitter and then combining the modules to build a *CM* that is functionally equivalent to the *TM*, only a negligible loss of accuracy (0.58% on average) is incurred. In addition, each module retains only 36.88% of the convolution kernel of the *TM* on average. To validate the effectiveness of module reuse, we apply modules as patches to improve three common types of weak CNN models, *i.e.* overly simple model, underfitting model, and overfitting model. Overall, after patching, the averaged improvements in terms of precision, recall, and F1-score are 17.13%, 4.95%, and 11.47%, respectively. Also, for a new task, we develop a *CM* entirely by reusing the modules with the best performance in the corresponding class. Compared with the models retrained from scratch, the *CM* achieves similar accuracy with a loss of only 2.46%. Even though there may exist *TMs* that can be directly reused, the *CM* outperforms the best *TM* and the average improvement in accuracy is 5.18%. Although modularization and composition incur additional time and GPU memory overhead, the experimental results demonstrate that the overhead is affordable. In particular, *CMs* can make prediction faster than *TMs* by executing modules in parallel and incur 28.6% less time overhead than *TMs*.

The main contributions of this work are as follows:

- We propose compressed modularization approaches including CNNSplitter and GradSplitter, which can decompose a CNN model into a set of reusable modules. We also apply CNNSplitter and GradSplitter to improve CNN models and build CNN models for new tasks through module reuse. To our best knowledge, CNNSplitter is the first *compressed modularization* approach that can decompose trained CNN models into CNN modules and reduce the overhead of module reuse.
- We formulate the modularization of CNNs as a search problem and design a genetic algorithm and a gradient descent-based search method to solve it. Especially, we propose three heuristic methods to alleviate the problem of excessive search space and time complexity in CNNSplitter and design a module evaluation method to recommend the optimal module for module reuse.
- We conduct extensive experiments using three representative CNNs on three widely-used datasets. The results show that CNNSplitter and GradSplitter can decompose a trained CNN model into modules with negligible loss of model accuracy. Also, the experiments demonstrate the effectiveness of developing accurate CNN models by reusing modules.

This work is an extension of our early work published as a conference paper [51], in which we proposed a compressed modularization approach, CNNSplitter, by applying a genetic algorithm to decompose CNN models into smaller CNN modules. The experiments demonstrated that CNNSplitter could be applied to patch weak CNN models through reusing modules obtained from strong CNN models, thus improving the recognition ability

Just Accepted

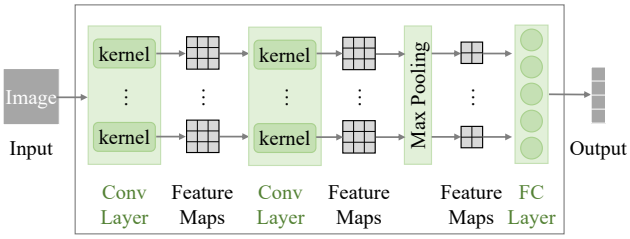

Fig. 2. The architecture of a typical CNN model.

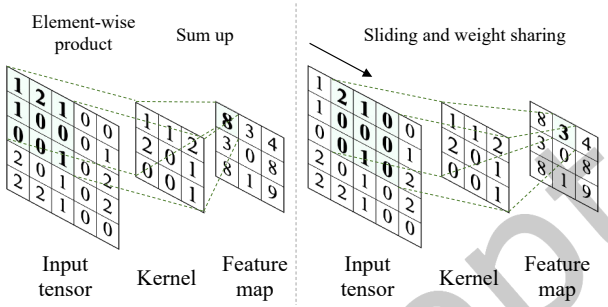

Fig. 3. An example of convolution operation.

of the weak CNN models. Compared to [51], this work (1) proposes a novel compressed modularization approach, named GradSplitter, which applies a gradient-based search method to decompose CNN models and outperforms CNNSplitter in both efficiency and effectiveness (see Sec. 4), (2) verifies the effectiveness of CNN modularization and composition in a new application (see Sec. 5.2), and (3) conducts more comprehensive experiments to evaluate the effectiveness and efficiency of CNN modularization and composition (see RQ3 to RQ5 in Sec. 6.2).

Replication Package: Our source code and experimental data are available at [1] and [2].

2 BACKGROUND

This section briefly introduces some preliminary information about this study, including the convolutional neural network and genetic algorithm.

2.1 Convolutional Neural Network

A CNN model typically contains convolutional layers, pooling layers, and FC layers, of which the convolutional layers are the core of a CNN [64, 78]. For instance, Figure 2 shows the architecture of a typical CNN model. A convolutional layer contains many convolution kernels, each of which learns to extract a local feature of an input tensor [29, 78]. An input tensor can be an input image or a feature map produced by the previous convolutional layer or pooling layer. A pooling layer provides a downsampling operation [78]. For instance, max pooling is the most popular form of pooling operation, which reduces the dimensionality of the feature maps by extracting the maximum value and discarding all the other values. FC layers are usually at the end of CNNs and are used to make predictions based on the features extracted from the convolutional and pooling layers.

Figure 3 shows an example of convolution operation. By sliding over the input tensor, a convolution kernel calculates how well the local features on the input tensor match the feature that the convolution kernel learns to extract. The more similar the local features are to the features extracted by the convolution kernel, the larger the

Just Accepted

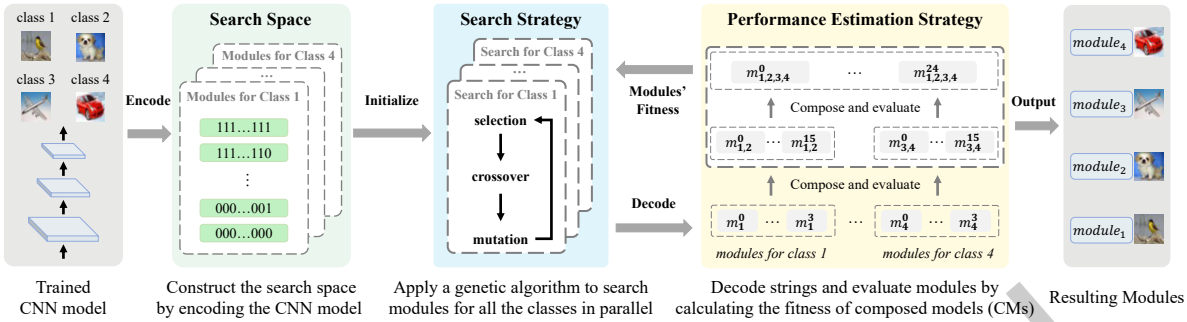

Fig. 4. The overall workflow of CNNSplitter.

output value of the convolution kernel at the corresponding position, and vice versa. Since a convolution kernel slides over the input tensor to match features and produce a feature map, all values in the feature map share the same convolution kernel. For instance, in the feature map of Figure 3, the values in the top-left (8) and top-middle (3) share the same kernel, *i.e.* weights. Weight sharing [29, 78] is one of the key features of a convolutional layer. The values in a feature map reflect the degree of matching between the kernel and the input tensor. For instance, compared to the position in the input tensor corresponding to the top-middle (3) in the feature map, the position corresponding to the top-left (8) is more similar to the kernel.

2.2 Genetic Algorithm

Inspired by the natural selection process, the genetic algorithm performs *selection*, *crossover*, and *mutation* for several *generations* (*i.e.* rounds) to generate solutions for a search problem [22, 58]. A standard genetic algorithm has two prerequisites, *i.e.* the representation of an *individual* and the calculation of an individual's *fitness*. For instance, the genetic algorithm is used to search for high-quality CNN architectures [56, 76]. An *individual* is a bit vector representing a NN architecture [76], where each bit corresponds to a convolution layer. The *fitness* of an individual is the classification accuracy of a trained CNN model with the architecture represented by the individual. During the search, in each generation, the *selection* operator compares the fitness of individuals and preserves the strong ones as parents that obtain high accuracy. The *crossover* operator swaps part of two parents. The *mutation* operator randomly changes several bit values in the parents to enable or disable these convolution layers corresponding to the changed bit values. After the three operations, a new *population* (*i.e.* a set of individuals) is generated. And the process continues with the new generation iteratively until it reaches a fixed number of generations or an individual with the target accuracy is obtained.

3 CNNSPLITTER: GENETIC ALGORITHM-BASED COMPRESSED MODULARIZATION

Figure 4 shows the overall workflow of CNNSplitter. For a given trained N-class CNN model $\mathcal{M}=\{k_0, k_1, \ldots, k_{L-1}\}$ with L convolution kernels, the modularization process is summarized as follows:

(1) *Construction of Search Space*: CNNSplitter encodes each candidate module into a fixed-length bit vector, where each bit represents whether the corresponding kernels are kept or not. The bit vectors of all candidate modules constitute the search space.

(2) *Search Strategy*: From the search space, the search strategy employs a genetic algorithm to find modules for N classes.

(3) *Performance Estimation*: The performance estimation strategy measures the performance (*i.e.* fitness) of the searched candidate and guides the search process.

Just Accepted

3.1 Search Space

As shown in Figure 4, the search space is represented using a set of bit vectors. For a CNN model with a lot of kernels, the size of vector could be very long, resulting in an excessively large search space, which could seriously impair the search efficiency. For instance, 10-class VGGNet-16 [64] includes 4,224 kernels, so the number of candidate modules for each class is 2^{4224}. In total, the size of the search space will be 10×2^{4224}. To reduce the search space, the kernels in a convolutional layer are divided into groups. A simple way is to randomly group kernels; however, this could result in a group containing both kernels necessary for a module to recognize a specific class and those that are unnecessary. The randomness introduced by random grouping cannot be eliminated by subsequent searches, resulting in unnecessary kernels in the searched modules.

To avoid unnecessary kernels as much as possible, an *importance-based grouping scheme* is proposed to group kernels based on their importance. As introduced in Section 2, the values in a feature map can reflect the degree of matching between a convolution kernel and an input tensor. The kernels producing feature maps with weak activations are likely to be unimportant, as the values in the feature map with weak activations are generally small (and even zero) and have little effect on the subsequent calculations of the model [32]. Inspired by this, CNNSplitter measures the importance of kernels for each class based on the feature maps. Specifically, given m samples labeled class n from the training dataset, a kernel outputs m feature maps. We calculate the sum of all values in each feature map and use the average of m sums to measure the importance of the kernel for class n. Then L kernels are divided into G groups following the importance order. Consequently, a module is encoded into a bit vector $[0, 1]^G$, where each bit represents whether the corresponding group of kernels is removed. The number of candidate modules for the nth class is 2^G, and for N classes, the search space size is reduced to $N \times 2^G$.

For simplicity, if the number of kernels in a convolutional layer is less than 256, the kernels are divided into 10 groups; otherwise, they are divided into 100 groups. In this way, each kernel group has a moderate number of kernels (*e.g.* about 10), and groups in the same convolutional layer have approximately the same number of kernels.

3.2 Search Strategy

A genetic algorithm [76] is used to search CNN modules, which has been widely used in search-based software engineering [16, 67]. The search process starts by initializing a population of N_I individuals for each of N classes. Then, CNNSplitter performs T generations, each of which consists of three operations (*i.e.* selection, crossover, and mutation) and produces N_I new individuals for each class. The fitness of individuals is evaluated via a performance estimation strategy that will be introduced in Section 3.3.

3.2.1 Sensitivity-based Initialization. In the 0th generation, a set of modules $M_n^0 = \{m_{n,i}^0\}_{i=0}^{N_I-1}$ are initialized for class n, where $n = 0, 1, \ldots, N-1$ and $m_{n,i}^0$ is a bit vector $[0, 1]^G$. Two schemes are used to set the bits in each individual (*i.e.* module): random initialization and *sensitivity-based initialization*. Random initialization is a common scheme [12, 76]. Each bit in an individual is independently sampled from a Bernoulli distribution. However, random initialization causes the search process to be slow or even fail. We observed a phenomenon that some convolutional layers are sensitive to the removal of kernels, which has been also observed in network pruning [32]. That is, the accuracy of a CNN model dramatically drops when some particular kernels are dropped from a sensitive convolutional layer, while the loss of accuracy is not more than 0.01 when many other kernels (*e.g.* 90% of kernels) are dropped from an insensitive layer.

To evaluate the sensitivity of each convolutional layer, we drop out 10% to 90% kernels in each layer incrementally and evaluate the accuracy of the resulting model on the validation dataset. If the loss of accuracy is small (*e.g.* within 0.05) when 90% kernels in a convolutional layer are dropped, the layer is insensitive, otherwise, it is sensitive. When initializing $m_{n,i}^0$ using sensitivity-based initialization, fewer kernel groups are dropped from the

Just Accepted

sensitive layers while more kernel groups from the insensitive layers. More specifically, a drop ratio is randomly selected from 10% to 50% for a sensitive layer (*i.e.* 10% to 50% bit values are randomly set to 0). In contrast, a drop ratio is randomly selected from 50% to 90% for an insensitive layer.

3.2.2 Selection, Crossover, and Mutation. For class n, to generate the population (*i.e.* modules) of the tth generation, CNNSplitter performs selection, crossover, and mutation operations on the $(t-1)$th generation's population M_n^{t-1}. First, the selection operation selects N_P individuals from M_n^{t-1} as parents according to individuals' fitness. Then, the single-point crossover operation generates two new individuals by exchanging part of two randomly chosen parents from N_P parents. Next, the crossover operation iterates until N_I new individuals are produced. Finally, the mutation operation on the N_I new individuals involves flipping each bit independently with a probability p_M. For N classes, selection, crossover, and mutation operations are performed in parallel, resulting in a total of $N \times N_I$ modules.

3.3 Performance Estimation Strategy

A module with high fitness should have the same good identification ability as the trained model $\mathcal{M}$ and only recognize the features of one specific class. Two evaluation metrics are used to evaluate the fitness of modules: the *accuracy* and the *difference* of modules. The higher the *accuracy*, the stronger the ability of the module to recognize features of the specific class. On the other hand, the greater the *difference*, the more a module focuses on the specific class. In addition, the *difference* can be used as a regularization to prevent the search from overfitting the *accuracy*, as the simplest way to improve *accuracy* is to allow each module to retain all the convolution kernels of $\mathcal{M}$. Consequently, the fitness of a module is the weighted sum of the *accuracy* and the *difference*. Furthermore, when calculating the fitness, a pruning strategy is used to improve the evaluation efficiency, making the performance estimation strategy computationally feasible.

3.3.1 Evaluation Metrics. Since a module focuses on a specific class and is equivalent to a single-class classifier, we combine modules into a composed model (CM) to evaluate them. That is, one module is selected from each class's N_I modules, and the N modules are combined into a $CM^{(N)}$ for N-class classification. The $CM^{(N)}$ is evaluated on the same classification task as $\mathcal{M}$ using the dataset D. The accuracy of $CM^{(N)}$ and the difference between the modules within $CM^{(N)}$ are assigned to each module. Specifically, the accuracy and difference of each module are calculated as follows:

Accuracy (Acc). To calculate the *Acc* of $CM^{(N)}$, the N modules are executed in parallel, and the output of $CM^{(N)}$ is obtained by combining modules' outputs. Specifically, given a $CM^{(N)}=\{m_n\}_{n=0}^{N-1}$, the output of module m_n for the jth input sample labeled $class_j$ is a vector $O_{n,j}=[o_{n,j}^0, o_{n,j}^1, \ldots, o_{n,j}^{N-1}]$, where each value corresponds to a class. Since m_n is used to recognize class n, the nth value $o_{n,j}^n$ is retained. Consequently, the output of $CM^{(N)}$ is $O_j=[o_{0,j}^0, o_{1,j}^1 \ldots, o_{N-1,j}^{N-1}]$, and the *Acc* of $CM^{(N)}$ is calculated as follows:

$$Acc = \frac{1}{|D|}\sum_{j}^{|D|} pred(j), \tag{1}$$

$$pred(j) = \begin{cases} 1, & \text{if } \underset{n=0,1,\ldots,N-1}{\arg\max}\ O_j = class_j \\ 0, & \text{if } \underset{n=0,1,\ldots,N-1}{\arg\max}\ O_j \neq class_j. \end{cases} \tag{2}$$

Difference (Diff). Since a module can be regarded as a set of convolution kernels, the difference between two modules can be measured by the Jaccard Distance ($\mathcal{JD}$) that measures the dissimilarity between two sets. The $\mathcal{JD}$ between set A and set B is obtained by dividing the difference of the sizes of the union and the intersection of

Just Accepted

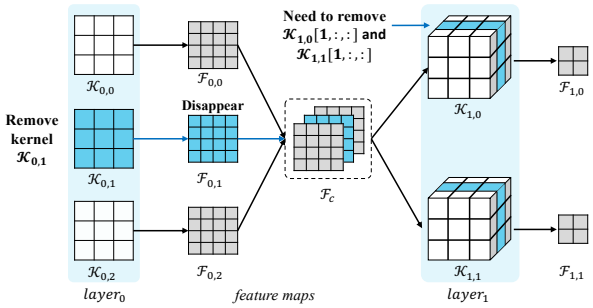

Fig. 5. The process of removing convolution kernels.

two sets by the size of the union:

$$JD(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}. \tag{3}$$

If the $JD(A,B) = 1$, there is no commonality between set A and set B, and if it is 0, then they are exactly the same. Based on $\mathcal{JD}$, the *Diff* value of $CM^{(N)}$ is the average value of $\mathcal{JD}$ between all modules:

$$Diff = \frac{2}{N \times (N-1)} \times \sum_{0 \le i < j \le N-1} JD(m_i, m_j). \tag{4}$$

Based on *Acc* and *Diff*, the fitness value of $CM^{(N)}$ is calculated via:

$$fitness = \alpha \times Acc + (1-\alpha) \times Diff, \tag{5}$$

where α is a weighting factor and $0 < \alpha < 1$. In practice, α is set to a high value (*e.g.* 0.9) because high accuracy is a prerequisite for the availability of modules. The fitness value of $CM^{(N)}$ is then assigned to the N modules within $CM^{(N)}$. Since each module is used in multiple CMs, a set of fitness values is assigned to each module. The maximum value of the set is a module's final fitness.

3.3.2 Decode. To evaluate modules, each bit vector is transformed into a runnable module by removing the kernel groups from $\mathcal{M}$ corresponding to the bits of value 0. Since removing kernels from a convolutional layer affects the convolutional operation in the later convolutional layer, the kernels in the latter convolutional layer need to be modified to ensure that the module is runnable.

Figure 5 shows the process of removing convolution kernels. During the convolution, the three kernels $k_{0,*} \in \mathbb{R}^{3\times3}$ in $layer_0$ output three feature maps $F_{0,*} \in \mathbb{R}^{4\times4}$ that are then combined in a feature map $F_c \in \mathbb{R}^{3\times4\times4}$ and fed to $layer_1$. By sliding on F_c, kernels $k_{1,*} \in \mathbb{R}^{3\times3\times3}$ in $layer_1$ perform the convolution and output two feature maps $F_{1,*} \in \mathbb{R}^{2\times2}$. If $k_{0,1}$ in $layer_0$ is removed, the feature map $F_{0,1}$ generated by $k_{0,1}$ is also removed. The input of $layer_1$ becomes a different feature map $F_c^{'} \in \mathbb{R}^{2\times4\times4}$, the dimension of which does not match that of $k_{1,*} \in \mathbb{R}^{3\times3\times3}$ in $layer_1$, causing the convolution to fail.

To solve the dimension mismatch problem, we remove the part of $k_{1,*}$ that corresponds to $F_{0,1}$, ensuring the first dimension of $k_{1,*}$ to match with that of $F_c^{'}$. For instance, since $F_{0,1}$ is removed, $k_{1,0}[1,:,:]$, which performs convolution on $F_{0,1}$, becomes redundant and causes the dimension mismatch. We remove $k_{1,0}[1,:,:]$ and the transformed kernel $k_{1,0}^{'} \in \mathbb{R}^{2\times3\times3}$ can perform convolution on $F_c^{'}$.

In addition, since the residual connection adds up the feature maps output by two convolutional layers, the number of kernels removed from the two convolutional layers must be the same to ensure that the output feature

Just Accepted

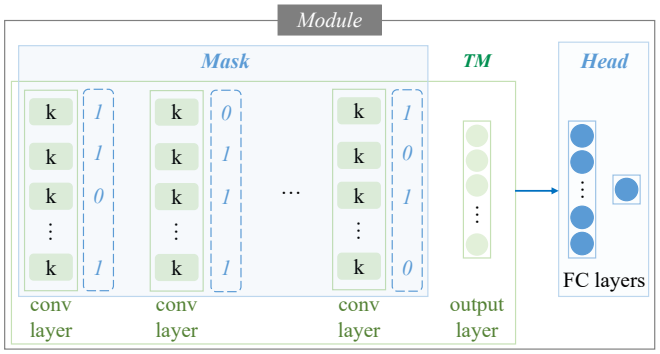

Fig. 6. An illustration of creating a module through $mask \odot TM \oplus head$.

maps match in dimension. When constructing a bit vector, we treat the two convolutional layers as one layer and use the same segment to represent the two layers so that they always remove the same number of kernels.

3.3.3 Pruning-based Evaluation. Since the fitness of each module comes from the one with the highest fitness among the CMs the module participates in, the number of $CM^{(N)}$ that CNNSplitter needs to evaluate is $(N_I)^N$. The time complexity is $O(n^N)$, which could be too high to finish the evaluation in a limited time. To reduce the overhead, a pruning strategy is designed, which is based on the following fact: if the accuracy of $CM^{(N)}$ is high, the accuracy of the $CM^{(n)}$ (*e.g.* $CM^{(2)}$ for the binary classification) composed of the modules within the $CM^{(N)}$ is also high. If the accuracy of a module is low, the accuracy of the $CM^{(n)}$ containing the module is also lower than the $CM^{(n)}$ containing modules with high accuracy. In addition, the number of $CM^{(n)}$ is much smaller than that of $CM^{(N)}$. For instance, the N-class classification task can be decomposed into $N/2$ binary classification subtasks, resulting in $N/2 \times (N_I)^2$ $CM^{(2)}$.

Therefore, the N-class classification task is decomposed into several subtasks. The accuracy of $CM^{(n)}$ is evaluated, and the top N_{top} $CM^{(n)}$ with high accuracy are selected to be combined into $CM^{(N)}$. Through continuous evaluation, selection, and composition, a total of $(N_{top})^2$ $CM^{(N)}$ are composed. The time complexity is $O(n^2)$, which is lower than the original time complexity $O(n^N)$.

4 GRADSPLITTER: GRADIENT-BASED COMPRESSED MODULARIZATION

We propose a gradient-based compressed modularization approach, named GradSplitter, to decompose a trained CNN model *TM* into smaller modules. To achieve the modularization of a *TM*, a *mask* and a *head* are used to create a module through $mask \odot TM \oplus head$. Figure 6 illustrates the creation of a module. A *mask* consists of an array of 0s and 1s, where 0 (or 1) indicates that the corresponding convolution kernels in the *TM* are removed (or retained). The operation $mask \odot TM$ removes kernels from *TM*, resulting in a *masked TM*. A *head* consists of FC layers and converts an N-dimensional input into a 1-dimensional output. The operation $\oplus head$ appends a *head* after the *masked TM*, resulting in a module. As a result, the constructed module is a binary classifier. The output value of a module greater than 0.5 indicates that the input belongs to the target class, and vice versa.

Subsequently, the decomposition is a training process for *masks* and *heads*. As shown in Algorithm 1, the training process mainly consists of the forward propagation (Line 6) and the backward propagation (Line 7). The forward propagation computes the prediction of the composed model (*CM*) constructed using N modules. The N modules are created based on the current *masks* and *heads*. The backward propagation optimizes *masks* and *heads* using the gradient descent based on the current prediction. Section 4.1 and Section 4.2 provide detailed

Just Accepted

Algorithm 1: Overall GradSplitter Algorithm

Input: Training dataset D and trained CNN model TM.
Output: N binarized *masks* and N *heads*.

1 $S_m = \{mask_i\}_{i=1}^{N}$, where $mask_i[:] > 0$;
2 $S_h = \{head_i\}_{i=1}^{N}$, where $head_i$ consists of FC layers;
3 $results = []$;
4 **for** $e = 1, 2, ..., E$ **do**
5 **for** $input, label$ *in* D **do**
 `// Forward() is shown in Alg.2`
6 $pred$ = `Forward(`$S_m, S_h, input$`)` ;
 `// Backward() is shown in Alg.3`
7 S_m, S_h= `Backward(`$S_m, S_h, pred, label, e$`)` ;
8 $results.append($[`Bin(`S_m`)`, S_h]);
9 select S_m and S_h from $results$;
10 **return** S_m, S_h

Algorithm 2: Forward propagation

Input: N masks S_m, N heads S_h, and input data $input$.
Output: prediction $predict$.

1 $modules\_pred = []$;
2 **for** $i = 1, 2, ..., N$ **do**
 `// compute each module's prediction`
3 $mask_i^b$ = `Bin(`$S_m[i]$`)`;
4 $out = mask_i^b \odot TM(input)$;
5 $head_i = S_h[i]$;
6 $pred = head_i(out)$;
7 $modules\_pred.append(pred)$;
8 $predict$ = `Concat(`$modules\_pred$`)`;
9 **return** $predict$

descriptions of the *masks* and *heads*, respectively. Section 4.3 explains how to optimize *masks* and *heads* to obtain modules and achieve CNN model decomposition.

4.1 Mask

A *mask* should consist of binarized values (*i.e.* 0s and 1s) to indicate which convolution kernels in the *TM* are removed (or retained). On the other hand, during the training process, the values in a *mask* should be continuous numerical values, instead of binarized values, to enable gradient descent. Therefore, the *mask* is initialized with random positive numbers during the training process (Line 1 in Algorithm 1). GradSplitter uses a binarization function to transform the *mask*, resulting in a binarized mask as an intermediate value (Line 3 in Algorithm 2). In order to achieve the transformation, the binarization function *Bin* is defined as follows:

$$x_b = Bin(x) = \begin{cases} 1, & \text{if } x > 0, \\ 0, & \text{otherwise,} \end{cases} \tag{6}$$

Just Accepted

Algorithm 3: Backward propagation

Input: N masks S_m, N heads S_h, prediction $pred$, data labels $label$, and the current epoch e.
Output: Updated masks S_m and heads S_h.

```
1  actions = [0, 1]^E;
2  if actions[e] == 0 then
3      update_obj = [S_h];
4      weight = 0;
5  else
6      update_obj = [S_m, S_h];
7      weight = β;
8  loss_1 = CrossEntropy(pred, label);
9  loss_2 = PercentKernels(S_m);
10 loss = loss_1 + weight × loss_2;
11 GradientDescent(loss, update_obj);
12 return S_m, S_h
```

where x_b is the binarized variable and x is the real-valued variable.

To achieve the effect of removing convolution kernels according to the binarized mask in forward propagation, we multiply the output of each convolution kernel by the corresponding value in the binarized mask (Line 4 in Algorithm 2). For instance, as shown in Figure 5, the feature map $F_{0,1}$ generated by the convolution kernel $k_{0,1}$ is multiplied by the corresponding value 0 in the binarized mask, resulting in all values in $F_{0,1}$ being 0. After multiplying the outputs of the previous convolutional layer's kernels by the corresponding values in the binarized mask, the outputs of the convolution kernels that should be removed are set to 0s. In this way, the convolution kernels that should be removed will not affect the subsequent prediction, thus enabling the simulation of removing convolution kernels. Note that, according to the trained mask, the unwanted convolution kernels will be removed from the modules during module reuse.

4.2 Head

A masked *TM* cannot be used as a module directly, as the output of the masked *TM* is still N-dimensional. Although the ith value in the output could indicate the probability that the input belongs to the ith class, using the ith value in the output as the prediction of a module is problematic. The output layer of the *TM* predicts based on the features extracted by all convolution kernels. There could be significant bias in the prediction of masked *TM*, as the output layer predicts based on only retained convolution kernels. For instance, regardless of which class the input belongs to, the ith value in the output of a masked *TM* that recognizes the target class i is always larger than other values in the output. As a result, a masked *TM* that recognizes target class i always classifies the inputs to the target class i even if the inputs actually belong to other classes.

Therefore, an additional output layer (*i.e. head*) consisting of two FC layers and a *sigmoid* activation function is appended after the masked *TM* (see Figure 6). The numbers of neurons in the two FC layers are N and 1, respectively. The *head* transforms the N-class prediction of masked *TM* to the binary classification prediction, resulting in the prediction of a module (Line 6 of Algorithm 2). As a result, each module recognizes a target class and estimates the probability of an input belonging to the target class. Since each module recognizes a target class and outputs a probability value between 0 and 1, N modules can be aggregated as a composed model for the N-class classification task (Line 8 of Algorithm 2).

Just Accepted

4.3 Optimization by Gradient Descent

The goal of modularization is to obtain N *masks* and N *heads* to decompose a *TM* into N modules. Each module can recognize a target class and should retain only the convolution kernels necessary for recognizing the target class. To achieve the goal, GradSplitter needs to optimize *masks* to remove the redundant convolution kernels as many as possible and optimize *heads* to predict based on the retained convolution kernels. The optimization is achieved by minimizing weighted loss through gradient descent-based backward propagation, which is shown in Algorithm 3.

Specifically, to ensure that each module can recognize the target class well, $loss_1$ is defined as the cross entropy between the prediction of *CM* and the actual label (Line 8). By minimizing $loss_1$, the prediction performance of the *CM* is improved, while the improvement of *CM* is essentially owing to the improvement of modules in recognizing target classes. The *masks* tend to make more values greater than zero to retain more convolution kernels, and the *heads* are trained to predict based on the retained convolution kernels. To constraint modules to retain only the necessary kernels, $loss_2$ is defined as the percentage of retained convolution kernels (Line 9). By minimizing $loss_2$, the *masks* tend to have fewer values that are greater than zero, resulting in fewer convolution kernels.

We define *loss* as the weighted sum of $loss_1$ and $loss_2$ (Line 10). By minimizing *loss*, the *masks* are optimized to have as few necessary values greater than zero as possible, *i.e.* retain only the necessary convolution kernels. Meanwhile, the *heads* are trained to predict based on the retained convolution kernels. The larger the value of *weight*, the more the convolution kernels GradSplitter tends to remove. In our experiments, the value of *weight* is usually small (*e.g.* 0.1), allowing GradSplitter to remove convolution kernels carefully, thus avoiding much impact on the recognition ability of modules.

When minimizing *loss*, one problem is that the poor predictions of modules cannot guide the optimization well in the early stages of optimization, as the *heads* are initialized randomly. Moreover, $loss_2$ could affect the optimization of *heads*, leading to the increased loss of accuracy. Therefore, a strategy is designed for the optimization, which is shown in Line 1 to 7 in Algorithm 3. *actions* is a bit vector $[0, 1]^E$ (Line 1), which is used to select the optimization object in an epoch e. When the value of $actions[e]$ is 0, *weight* is set to zero, and GradSplitter minimizes only $loss_1$ to optimize *heads*. When the value of $actions[e]$ is 1, *weight* is set to β, and GradSplitter minimizes *loss* to optimize *masks* and *heads* jointly. With the strategy, GradSplitter can optimize only *heads* in the first few epochs to recover the loss of accuracy caused by the randomly initialized *heads*. On the other hand, GradSplitter can minimize only $loss_1$ after every several epochs of minimizing *loss* to recover the loss of accuracy caused by the removal of convolution kernels.

The gradient descent-based optimization is applied to minimize *loss* (Line 11). When minimizing *loss* by gradient descent, it is important to note that the common backward propagation based on gradient descent cannot be directly applied to update *masks*, as the derivative of the *Bin* function is zero almost everywhere. Fortunately, the technique called straight-through estimator (STE) [4] has been proposed to address the gradient problem occurring when training neuron networks with binarization function (*e.g. sign*) [4, 23, 80]. The function of STE is defined as follows:

$$clip(x, -1, 1) = max(-1, min(1, x)), \tag{7}$$

where x is the gradient value calculated in the previous layer and used to estimate the gradient in current layer. In our work, based on STE, the gradient of *Bin* (defined in Equation 6) is calculated as follows:

$$\frac{\partial loss}{\partial x} = clip(\frac{\partial loss}{\partial x_b}, -1, 1). \tag{8}$$

Just Accepted

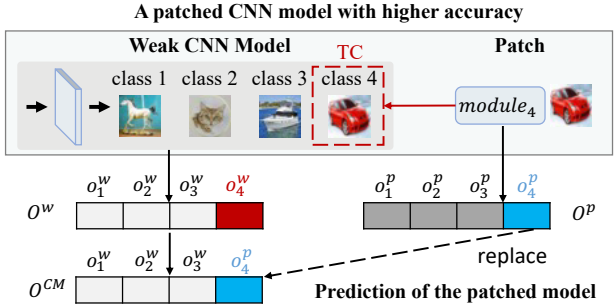

Fig. 7. Patching a weak CNN model.

5 APPLICATIONS OF CNN MODULARIZATION

In this section, we present two applications of CNNSplitter and GradSplitter: 1) patching weak CNN models and 2) developing new models via composition.

5.1 Application 1: Patching Weak CNN Models through Modularization and Composition

The weak CNN model can be improved by patching the target class (TC). To identify the TC of a weak CNN model, developers can use test data to evaluate the weak CNN model's classification performance (*e.g.* precision and recall) of each class. The class in which the weak CNN model achieves poor classification performance is regarded as TC. As illustrated in Figure 7, the TC is replaced with the corresponding module from a strong model. To find the corresponding module, a developer can evaluate the accuracy of a candidate model on TC. If the candidate model's accuracy exceeds that of the weak model, its module can be used as a patch.

Formally, given a weak CNN model $\mathcal{M}_w$, suppose there exists a strong CNN model $\mathcal{M}_s$ whose classification task intersects with that of $\mathcal{M}_w$. For instance, both $\mathcal{M}_w$ and $\mathcal{M}_s$ can recognize TC n. Then, the corresponding module m_n from $\mathcal{M}_s$ can be used as a patch to improve the ability of $\mathcal{M}_w$ to recognize TC n. Specifically, $\mathcal{M}_w$ and m_n are composed into a CM that is the patched CNN model. Given an input, $\mathcal{M}_w$ and m_n run in parallel and the outputs of them are $O^w = [o_0^w, o_1^w, \ldots, o_{N-1}^w]$ and $O^p = [o_0^p, o_1^p, \ldots, o_{N-1}^p]$, respectively. Then, the output O^{CM} of CM is obtained by replacing the prediction corresponding to TC n of O^w with that of O^p.

A straightforward way is to directly replace o_n^w with o_n^p, resulting in $O^{CM} = [o_0^w, \ldots, o_n^p, \ldots, o_{N-1}^w]$. The index of the maximum value in O^{CM} is the predicted class. However, the comparison between o_n^p and the other values in O^{CM} is problematic: since $\mathcal{M}_w$ and $\mathcal{M}_s$ are different models that are trained on the different datasets or have different network structures, there could be significant differences in the distribution between the outputs of $\mathcal{M}_w$ and $\mathcal{M}_s$. For instance, we have observed that the output values of a model could be always greater than that of the other one, resulting in the outputs of a module decomposed from the strong model being always larger/smaller than the outputs of a weak model. This problem could cause error prediction when calculating the prediction of CM; thus, O^w and O^p are normalized before the replacement. Specifically, since the outputs on the training set can reflect the output distribution of a module, we collect the outputs of m_n on the training data with the class label n. Then, the minimum and maximum values of the output's distribution can be estimated using the collected outputs. For instance, (min, max) are the minimum and maximum values of the collected outputs of m_n. The normalized o_n^p is $\frac{o_n^p - min}{max - min}$. In addition, the $softmax$ is used over O^w to scale the values in O^w between 0 and 1. Finally, the prediction of CM is obtained by replacing o_n^w in normalized O^w with normalized o_n^p.

Just Accepted

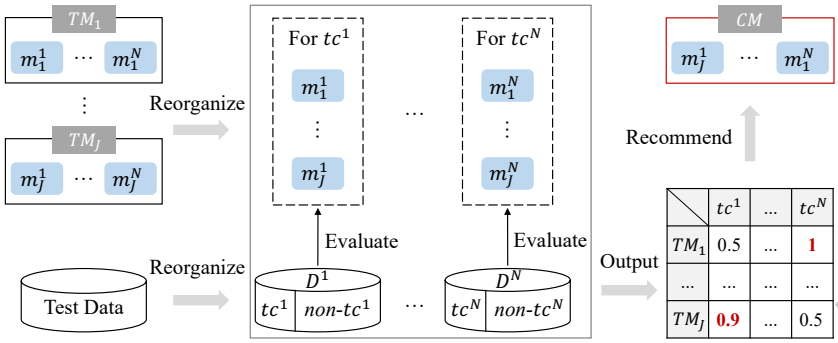

Fig. 8. The process of evaluating modules.

5.2 Application 2: Developing CNN Models via Composition

When a developer needs an N-class classification CNN model, the existing trained CNN models (*TMs*) shared by third-party developers could be decomposed and reused. Developing a new CNN model (*CM*) by composing reusable modules consists of three steps: *module creation*, *module evaluation*, and *module reuse*. *Module creation* mainly involves the removal of irrelevant convolution kernels from models according to masks, which is similar to the "Decode" operation in CNNSplitter introduced in Section 3.3.2. The following sections will introduce *module evaluation* and *module reuse*.

5.2.1 Module Evaluation. To obtain a more accurate *CM* than *TMs* or a new *CM* that can achieve comparable performance to the model trained from scratch, module evaluation is a key step, which can identify the module with the best recognition ability for each target class.

As shown in Figure 8, given a set of trained CNN models $\{TM_j\}_{j=1}^{J}$ for N-class classification and test data for N-class classification, GradSplitter first reorganizes them according to target classes. Specifically, for each target class tc^n, J corresponding modules that can recognize the target class tc^n are put together to form a set of candidate modules $\{m_j^n\}_{j=1}^{J}$. As a result, there are N sets of candidate modules. For each set, the module with the best recognition ability is recommended to the developer. Since each module m_j^n is a binary classification model that recognizes whether an input belongs to tc^n, the candidate modules from the same set can be tested and compared on the same binary classification task. As N sets of candidate modules correspond to N binary classification tasks, the test data for N-class classification needs to be reorganized to form N data sets $\{D^n\}_{n=1}^{N}$ for binary classification, each with the target class tc^n as the positive class and the other $N-1$ non-target classes $non\text{-}tc^n$ as the negative class. Considering that D^n could be imbalanced due to the disproportion among the number of samples of positive class and negative class, F1-score is used to measure the recognition ability of m_j^n for tc^n. For each target class tc^n, the module with the highest F1-score among the candidate modules $\{m_j^n\}_{j=1}^{J}$ is recommended to the developer.

5.2.2 Module Reuse. In the module composition, N recommended modules are combined into a *CM* for N-class classification. The composition is simple, and the *CM* runs like a common CNN model. Specifically, given an input, N modules run in parallel, and their outputs are concatenated as the output of the *CM*. The index of the maximum value in the output is the final prediction. Since each module in the *CM* has the best recognition ability for the corresponding target class, the *CM* can outperform any of the J trained CNN models or achieve competitive performance in accuracy compared to the model trained from scratch.

Just Accepted

6 EXPERIMENTS

To evaluate the effectiveness of the proposed approaches, in this section, we first introduce the benchmarks and experimental setup and then discuss the experimental results. Specifically, we evaluate CNNSplitter and GradSplitter by answering the following research questions:

- RQ1: How effective are the proposed techniques in modularizing CNN models?
- RQ2: Can the recognition ability of a weak model for a target class be improved by patching?
- RQ3: Can a composed model, built entirely by combining modules, outperform the best trained model?
- RQ4: Can a CNN model for a new task be built through modularization and composition while maintaining an acceptable level of accuracy?
- RQ5: How efficient is GradSplitter in modularizing CNN models and how efficient is the composed CNN model in prediction?

6.1 Benchmarks

1) Datasets. We evaluate the proposed techniques on the following three datasets, which are widely used for evaluation in related work [13, 39, 44].

CIFAR-10. The CIFAR-10 dataset [26] contains natural images with resolution 32×32, which are drawn from 10 classes including airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The initial training dataset and testing dataset contain 50,000 and 10,000 images, respectively.

CIFAR-100. CIFAR-100 [26] consists of 32×32 natural images in 100 classes, with 500 training images and 100 testing images per class.

SVHN. The Street View House Number (SVHN) dataset [41] contains colored digit images 0 to 9 with resolution 32×32. The training and testing datasets contain 604,388 and 26,032 images, respectively.

2) Models. We evaluate the proposed techniques on the following three typical CNN structures, which are widely used in popular networks [21, 25, 27, 29, 64, 70].

SimCNN. SimCNN represents a class of CNN models with a basic structure, such as LeNet [29], AlexNet [27], and VGGNet [64], essentially constructed by stacking convolutional layers. The output of each convolutional layer can only flow through each layer in sequential order. Without loss of generality, the SimCNN in our experiments is set to contain 13 convolutional layers and 3 FC layers, totaling 4,224 convolution kernels.

ResCNN. ResCNN represents a class of CNN models with a complex structure, such as ResNet [21], WRN [82], and MobileNetV2 [60], constructed by convolutional layers and residual connections. A residual connection can go across one or more convolutional layers, allowing the output of a layer not only to flow through each layer in sequential order but also to be able to connect with any following layer. Without loss of generality, the ResCNN in our experiments is set to have 12 convolutional layers, 1 FC layer, and 3 residual connections, totaling 4,288 convolution kernels.

InceCNN. InceCNN represents a class of CNN models with a complex structure, such as GoogLeNet [70] and Inception-V3 [71], constructed by branched convolutional layers. The branched convolutional layers mean that the outputs of several convolutional layers are concatenated as one input to be fed into the next branched convolutional layers. Without loss of generality, the InceCNN in our experiments is set to have 12 convolutional layers, 1 FC layer, and 3 branched layers, totaling 3,200 convolution kernels.

All the experiments are conducted on Ubuntu 20.04 server with 64 cores of 2.3GHz CPU, 128GB RAM, and NVIDIA Ampere A100 GPUs with 40 GB memory.

6.2 Experimental Results

RQ1: How effective are the proposed techniques in modularizing CNN models?

Just Accepted

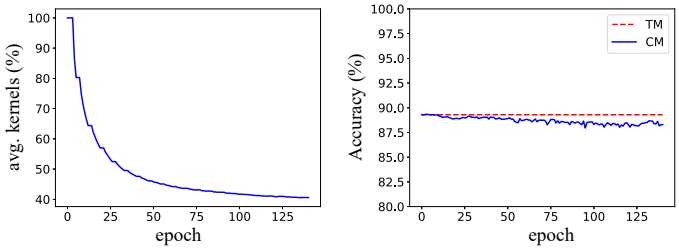

Fig. 9. The convergence process of GradSplitter, including the average percentage of retained kernels in a module (the left sub-figure) and the validation accuracy of the *TM*/*CM* (the right sub-figure) during modularization.

Table 1. The modularization results on four strong models.

Model	TM		Accuracy of a CM			Average percentage of kernels in a module		
	Acc.	# Kernels	CNNSplitter	GradSplitter	Increment	CNNSplitter	GradSplitter	Reduction
SimCNN-CIFAR10	89.77%	4224	86.07% (-3.70%)	88.90% (-0.87%)	2.83%	61.96%	41.22%	20.74%
SimCNN-SVHN	95.41%	4224	93.85% (-1.56%)	95.67% (+0.26%)	1.82%	52.79%	35.18%	17.61%
ResCNN-CIFAR10	90.41%	4288	85.64% (-4.77%)	89.08% (-1.33%)	3.44%	58.26%	40.63%	17.63%
ResCNN-SVHN	95.06%	4288	93.52% (-1.54%)	94.70% (-0.36%)	1.18%	54.03%	30.48%	23.55%
Average	**92.66%**	**4256**	**89.77% (-2.89%)**	**92.09% (-0.58%)**	**2.32%**	**56.76%**	**36.88%**	**19.88%**

1) Setup. **Training settings.** To answer RQ1, SimCNN and ResCNN are trained on CIFAR10 and SVHN, resulting in four strong CNN models: SimCNN-CIFAR, SimCNN-SVHN, ResCNN-CIFAR, and ResCNN-SVHN. The training datasets of CIFAR10 and SVHN are divided into two parts in the ratio of 8:2, respectively. The 80% samples are used as the training set while the 20% samples are used as the validation set. On both CIFAR10 and SVHN datasets, SimCNN and ResCNN are trained with mini-batch size 128 for 200 epochs. The initial learning rate is set to 0.01 and 0.1 for SimCNN and ResCNN, respectively, and the initial learning rate is divided by 10 at the 60th and 120th epoch for SimCNN and ResCNN respectively. All the models are trained using data augmentation [63] and SGD with a weight decay [28] of 10^{-4} and a Nesterov momentum [69] of 0.9. After completing the training, the trained models are evaluated on testing datasets.

Modularization settings. CNNSplitter applies a genetic algorithm to search CNN modules, following the common practice [56, 57, 68], the number of individuals N_I and the number of parents N_P in each generation are set to 100 and 50, respectively. The mutation probability p_M is generally small [56, 68] and is set to 0.1. The weighting factor α is set to 0.9. For the sake of time, an early stopping strategy [18, 84] is applied, and the maximum number of generations is set as $T = 200$. A trained CNN model $\mathcal{M}$ is modularized with reference to the validation set, which was not used in model training. After completing the modularization, the resulting modules are evaluated on the testing dataset.

GradSplitter initializes *masks* by filling positive values and initialize *heads* randomly. When initializing *actions* (Line 1 in Algorithm 3), we set $actions[1{:}5]=[0]\times5$ and $actions[6{:}E]=[1,1,1,1,1,0,0]\times\frac{E-5}{7}$. $actions[e]=0$ indicates that GradSplitter trains only the *heads* in the epoch e. $actions[e]=1$ indicates that GradSplitter jointly trains both *masks* and *heads* in the epoch e. As a result, GradSplitter trains only the *heads* in the first 5 epochs. Then, GradSplitter trains only the *heads* for 2 epochs after every 5 epochs of joint training. The training process iterates 145 epochs (*i.e.* E=145) with a learning rate of 0.001. By default, β in the weighted sum of $loss_1$ and $loss_2$ is set to 0.1. We also investigate the impact of β on modularization in Section 6.2.

Under Accepted

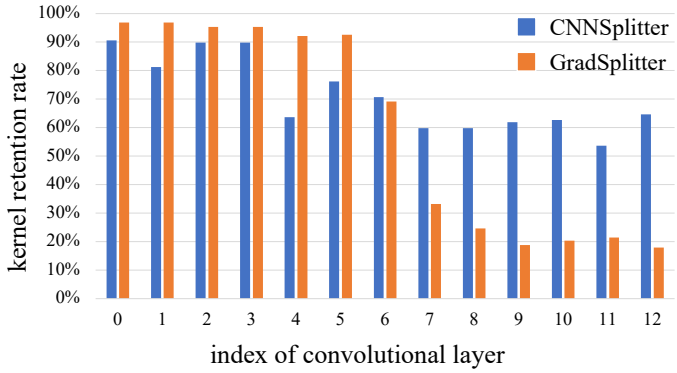

Fig. 10. The convolution kernel retention rate for each convolutional layer of SimCNN-CIFAR10.

2) Results. Figure 9 shows the trend of average percentage of retained convolution kernels in a module and the validation accuracy of the *CM* along with training epochs. In this example, the corresponding *TM* is obtained by training SimCNN on CIFAR-10, and the validation accuracy of the *TM* on the validation dataset is 89.29%. As shown in the left sub-figure in Figure 9, modules retain all of the kernels at the beginning of training because masks are initialized with random positive values. GradSplitter also tried to initialize masks using random real and random negative values. Compared to the initialization with positive values, initialization with random real values could lead to the removal of relevant kernels, causing GradSplitter to converge more slowly. Initializing masks with negative values makes it difficult for GradSplitter to train the masks, as all convolutional layer outputs are zeros at the beginning. Consequently, masks are initialized with positive values by default. During the training process, the average percentage of retained kernels in a module decreases quickly in the first 40 epochs and then gradually converges. As shown in the right sub-figure in Figure 9, despite the decreasing number of convolution kernels of modules, the validation accuracy of the *CM* is maintained close to the validation accuracy of the *TM*. As each *head* in a module has only 11 (10+1) neurons (detailed in Section 4.2), the optimization of randomly initialized *heads* in *CM* is fast, and the validation accuracy of *CM* is close to that of the *TM* at the first epoch.

Table 1 presents the modularization results of GradSplitter and CNNSplitter on four strong models, with a comparison between the two approaches. For instance, as shown in the 3rd row, the trained model SimCNN-CIFAR10 achieves a test accuracy of 89.77% (2nd column) with 4224 kernels (3rd column). GradSplitter decomposes SimCNN-CIFAR10 into 10 modules, each retaining an average of 41.22% of the model's kernels (penultimate column). The *CM*, which is composed of the 10 modules and classifies the same classes as the *TM*, obtains a test accuracy of 88.90%, with a loss of only 0.87% compared to SimCNN-CIFAR10 (5th column). In contrast, the modules generated by CNNSplitter retain more kernels, averaging 61.96% (7th column), and the accuracy of the *CM* is lower at 86.07%, with a loss of 3.70% (4th column). Compared to CNNSplitter, GradSplitter achieves improvements in both accuracy and module size, with an increase of 2.83% (6th column) and a reduction of 20.74% (last column), respectively. As shown in the last row, on average, the accuracy of composed models produced by CNNSplitter and GradSplitter are 89.77% and 92.09%, respectively, with the latter achieving an improvement of 2.32%. Compared with the accuracy of the trained model of 92.66%, the accuracy losses caused by CNNSplitter and GradSplitter are 2.89% and 0.58%, respectively, indicating that GradSplitter causes much fewer loss of accuracy. Regarding the module size, the modules generated by both CNNSplitter and GradSplitter are smaller than the models, with only 56.76% and 36.88% of kernels retained, respectively, indicating that the module incurs fewer memory and computation costs than models.

Just Accepted

Table 2. The FLOPs of the models and decomposed modules.

Model	Model FLOPs (M)	CNNSplitter		GradSplitter	
		Module FLOPs (M)	Reduction	Module FLOPs (M)	Recudtion
SimCNN-CIFAR10	313.7	164.3	47.60%	168.9	46.16%
SimCNN-SVHN		107.8	65.60%	121.2	61.37%
ResCNN-CIFAR10	431.2	225.4	47.70%	250.2	41.97%
ResCNN-SVHN		142.1	67.00%	158.5	63.24%
Average	**372.45**	**159.9**	**56.98%**	**174.7**	**53.18%**

We observed that GradSplitter retains fewer convolution kernels than CNNSplitter but incurs less accuracy loss. To explain this outcome, we first analyze the retention of kernels in every convolutional layer and find the difference in kernel retention between modules generated by the two approaches. As shown in Figure 10, GradSplitter retains more kernels in the lower layers (*i.e.* the first 6 convolutional layers) and fewer in the higher layers (*i.e.* the last 7 layers) compared to CNNSplitter. Studies [10, 81] have shown that lower layers (closer to the input layer) learn general features, while higher layers (closer to the output layer) learn specific features of certain classes. Therefore, intuitively, an appropriate distribution of kernel retention for a module should retain more kernels in the lower layers and fewer in the higher layers. GradSplitter performs better than CNNSplitter regarding the distribution of kernel retention, thus retaining fewer kernels while incurring less accuracy loss.

Moreover, the modules generated by GradSplitter have a larger capacity, which is another potential explanation for the outcome. The capacity of a model can be measured by the number of floating-point operations (FLOPs) required by the model. Larger FLOPs mean that the model has a larger capacity [15, 62]. Table 2 presents the FLOPs required by the modules and models to classify an image. Take the SimCNN-CIFAR10 model as an example (3rd row), on average, a module generated by CNNSplitter requires 164.3 million FLOPs, while a module generated by GradSplitter requires 168.9 million FLOPs. For all four models, the modules generated by GradSplitter require more FLOPs than those generated by CNNSplitter. The reason why a module generated by GradSplitter retains fewer kernels but requires more FLOPs is that its lower layers retain more kernels. Due to max pooling operations, the inputs of lower layers are larger than those of higher layers, and thus a kernel in the lower layer could incur more FLOPs than a kernel in the higher layer.

As shown in Table 2, our proposed techniques can significantly reduce the number of FLOPs required by modules, with average reductions of 56.98% and 53.18% for CNNSplitter and GradSplitter, respectively. In contrast, modules produced by uncompressed modularization approaches [43, 44] retain all weights or kernels, resulting in more memory and computation costs. Since the tools [54, 55] published by [43, 44] and our proposed techniques are implemented on Keras and PyTorch, respectively, they cannot directly decompose each other's trained models. We attempted to convert PyTorch and Keras trained models to each other; however, the conversion incurs much loss of accuracy (5% to 10%) due to differences in the underlying computation of PyTorch and Keras. Thus, we analyze the open source tools [54, 55], including source code files and the experimental data (*e.g.* the trained CNN models and the generated modules). The open-source tool keras-flops [73] is used to calculate the FLOPs for the approach described in [43]. The FLOPs required by a module in [43] are the same as those required by the model. For the project of [44], the modules are not encapsulated as Keras model, and there are no ready-to-use, off-the-shelf tools to calculate the FLOPs required by the modules. Therefore, we manually analyze the number of weights of the module and confirm that a module has the same number of weights as the model. In summary, the experimental results indicate that our compressed modularization approaches outperform the uncompressed modularization approaches [43, 44] in terms of module's size and its computational cost.

We also evaluate the effectiveness of importance-based grouping, sensitivity-based initialization, and pruning-based evaluation in CNNSplitter. Table 3 shows the results of CNNSplitter under different grouping settings

JSEP Accepted

Table 3. The results of CNNSplitter in different settings.

Model	Settings		Composed Model	
	Grouping	Initialization	Generation	Accuracy
SimCNN-CIFAR10	no	sensitivity-based	194	0.2754
	random	sensitivity-based	190	0.3650
	importance-based	random	192	0.3702
	importance-based	**sensitivity-based**	**123**	**0.8607**
SimCNN-SVHN	no	sensitivity-based	200	0.2430
	random	sensitivity-based	200	0.2512
	importance-based	random	188	0.9204
	importance-based	**sensitivity-based**	**79**	**0.9385**
ResCNN-CIFAR10	no	sensitivity-based	83	0.7271
	random	sensitivity-based	193	0.8420
	importance-based	random	197	0.8432
	importance-based	**sensitivity-based**	**185**	**0.8564**
ResCNN-SVHN	no	sensitivity-based	140	0.9027
	random	sensitivity-based	162	0.9249
	importance-based	random	179	0.9332
	importance-based	**sensitivity-based**	**107**	**0.9352**

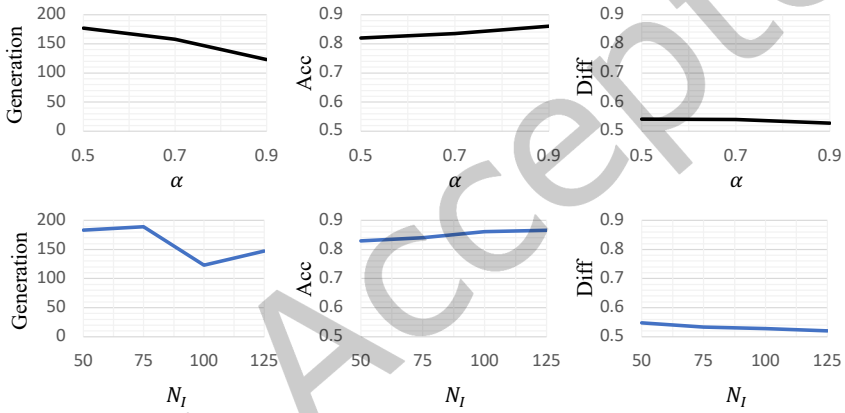

Fig. 11. The impact of major parameters in CNNSplitter.

(*no*, *random*, *importance-based*) and initialization settings (*random*, *sensitivity-based*). (1) comparing the results under ⟨importance-based, sensitivity-based⟩ to the results under ⟨no, sensitivity-based⟩ and ⟨random, sensitivity-based⟩, we found that modularization without importance-based grouping would cause more accuracy loss and require more generations (e.g., for ResCNN-SVHN) and may even fail due to significant accuracy loss (e.g., for SimCNN-CIFAR and SimCNN-SVHN). The time cost for grouping mainly involves collecting the importance of convolution kernels. This process takes about 30 seconds for both SimCNN-CIFAR10 and ResCNN-CIFAR10, and 60 seconds for both SimCNN-SVHN and ResCNN-SVHN. (2) comparing the results under ⟨importance-based, sensitivity-based⟩ to the results under ⟨importance-based, random⟩, we observed that sensitivity-based initialization could improve search efficiency and reduce accuracy loss. Analyzing the sensitivity of convolutional layers takes 169 seconds, 156 seconds, 221 seconds, and 204 seconds for SimCNN-CIFAR10, ResCNN-CIFAR10, SimCNN-SVHN, and ResCNN-SVHN, respectively. (3) regarding pruning-based evaluation, in the absence of the pruning strategy, modularization fails due to a timeout (i.e., requires more than several years). In contrast, with the pruning strategy, the time cost per generation for SimCNN-CIFAR, SimCNN-SVHN, ResCNN-CIFAR, and ResCNN-SVHN is 83 seconds, 95 seconds, 80 seconds, and 93 seconds, respectively.

Just Accepted

Table 4. The impact of β in GradSplitter.

β	Accuracy	Loss of Accuracy	Avg. #k (%)
0.01	89.19	0.58	2297 (54.37)
0.05	88.92	0.85	1853 (43.87)
0.10	88.90	0.87	1741 (41.22)
0.50	89.12	0.65	1947 (46.09)
1.00	89.66	0.11	4224 (100.0)

In addition, we investigate the impact of major parameters on CNNSplitter, including α (the weighting factor between the *Acc* and the *Diff*, described in Sec. 3.3) and N_I (the number of modules in each generation, described in Sec. 3.2). Figure 11 shows the "Generation", "Acc", and "Diff" of the composed model on SimCNN-CIFAR with different α and N_I. We find that, CNNSplitter performs stably under different parameter settings in terms of *Acc* and *Diff*. The changes in the number of generations show that a proper setting can improve the efficiency of CNNSplitter. The results also show that our default settings (*i.e.* $\alpha = 0.9$ and $N_I = 100$) are appropriate.

As for GradSplitter, we investigate the impact of β on modularization (see Algorithm 3). Table 4 shows the test accuracy and the number of kernels of *CMs* on SimCNN-CIFAR with different β. As GradSplitter considers the accuracy of *CMs* first when selecting the modules (see Line 9 in Algorithm 1), the accuracy of *CMs* with different values of β is similar; however, the average number of convolution kernels in a module is different. As the value of β increases from 0.01 to 0.1, the number of kernels decreases, since GradSplitter prefers to reduce the number of kernels to minimize the weighted loss. However, the value of β should be small, as the excessive value of β (*e.g.* β=1.0) could lead to removing kernels dramatically in an epoch, resulting in a sharp decrease of accuracy. The results show that the default β=0.1 is appropriate.

> Both CNNSplitter and GradSplitter strike a balance between the module's ability and its size. In particular, GradSplitter outperforms CNNSplitter, which causes a negligible accuracy loss of 0.58% and produces smaller modules with only 36.88% kernels retained.

RQ2: Can the recognition ability of a weak model for a target class be improved by patching?

1) Setup. **Design of weak models.** To answer RQ2, we conduct experiments on three common types of weak CNN models, *i.e. overly simple models*, *underfitting models*, and *overfitting models*. The modules generated from strong CNN models in RQ1 will be reused to patch these weak CNN models.

An overly simple model has fewer parameters than a strong model. To obtain the overly simple models, simple SimCNN and ResCNN models are used. Specifically, a simple SimCNN contains 2 convolutional layers and 1 FC layer, while a simple ResCNN contains 4 convolutional layers, 1 FC layer, and 1 residual connection.

An underfitting model has the same number of parameters as a strong model but is trained with a small number of epochs. To obtain the underfitting models, the model is trained at the $\frac{n_{best}}{2}$th epoch, which can neither well fit the training dataset nor generalize to the testing dataset. The accuracy of the underfitting model is low on both the training dataset and the testing dataset, indicating the occurrence of underfitting.

An overfitting model is obtained by disabling some well-known Deep Learning "tricks", including dropout [66], weight decay [28], and data augmentation [63]. These tricks are widely used to prevent overfitting and improve the performance of a DL model. The overfitting model can fit the training dataset and the accuracy on the training dataset is close to 100%; however, its accuracy on the testing dataset is much lower than that on the training

Just Accepted

Table 5. The comparison between GradSplitter and CNNSplitter in patching weak CNN models. The columns "CS" and "GS" present the performance of patched models based on CNNSplitter and GradSplitter, respectively. The column "GS-W" indicates the improvement of patched models based on GradSplitter against weak models. The column "GS-CS" indicates the improvement of GradSplitter against CNNSplitter in patching weak models. All results in %.

Metric	Model	Simple					Underfitting					Overfitting				
		Weak	CS	GS	GS-W	GS-CS	Weak	CS	GS	GS-W	GS-CS	Weak	CS	GS	GS-W	GS-CS
Precision	SimCNN-CIFAR	73.30	82.75	90.05	16.75	7.30	49.44	70.93	76.76	27.32	5.83	57.34	71.64	88.36	31.03	16.72
	SimCNN-SVHN	92.32	96.27	98.34	6.02	2.07	78.09	91.17	98.12	20.02	6.95	93.76	95.33	98.14	4.38	2.81
	ResCNN-CIFAR	78.10	89.99	90.75	12.64	0.76	45.55	81.66	83.22	37.67	1.56	57.50	76.08	75.85	18.35	-0.23
	ResCNN-SVHN	89.33	95.86	98.29	8.97	2.43	81.39	91.53	98.06	16.68	6.53	92.27	95.60	98.05	5.78	2.45
Recall	SimCNN-CIFAR	74.50	70.20	73.60	-0.90	3.40	36.90	65.00	78.10	41.20	13.10	59.40	57.70	61.50	2.10	3.80
	SimCNN-SVHN	93.04	91.31	93.67	0.63	2.36	77.55	85.96	92.16	14.61	6.20	92.43	91.71	92.10	-0.33	0.39
	ResCNN-CIFAR	74.30	68.70	69.90	-4.40	1.20	39.00	53.60	57.30	18.30	3.70	57.90	55.80	55.10	-2.80	-0.70
	ResCNN-SVHN	94.68	90.91	91.59	-3.09	0.68	84.18	79.14	80.00	-4.18	0.86	93.28	91.79	91.61	-1.67	-0.18
F1-score	SimCNN-CIFAR	73.76	75.68	80.57	6.81	4.89	35.46	64.59	77.07	41.61	12.48	58.14	63.60	72.18	14.04	8.58
	SimCNN-SVHN	92.67	93.71	95.94	3.27	2.23	77.71	88.39	95.00	17.29	6.61	93.08	93.46	95.00	1.91	1.54
	ResCNN-CIFAR	75.66	77.13	78.47	2.80	1.33	34.43	63.05	66.95	32.52	3.90	57.51	64.09	63.42	5.92	-0.67
	ResCNN-SVHN	91.88	93.30	94.79	2.91	1.49	79.63	82.31	86.27	6.63	3.96	92.70	93.61	94.65	1.95	1.04

Just Accepted

dataset, indicating the occurrence of overfitting. Except for the special design above, the same settings are applied as that of strong models.

Training dataset for weak models. Considering that the classification of the weak and existing strong models may not be identical in real-world scenarios (which is a reason for patching the weak model rather than directly replacing it with the strong model), weak models are not trained on the same training datasets used by strong models. For instance, a weak model for the classification of "cat" and "dog" performs poorly in identifying the class "cat". Supposing there is a trained model capable of classifying both "cat" and "fish" and performs better than the weak model in identifying the class "cat", the trained model cannot substitute the weak model but can be used to patch the weak model through modularization. To construct the training datasets for weak models, a subset of CIFAR-100 and a subset of SVHN are used. The former consists of 9 classes: "apple", "baby", "bed", "bicycle", "bottle", "bridge", "camel", "clock", and "rose", which do not overlap with the CIFAR-10 dataset. Each class in CIFAR-10 is considered as a target class in turn and merged with the subset, resulting in ten 10-class classification datasets, named CIFAR-W. The latter consists of 4 fixed classes: "6", "7", "8", and "9". Each class of "0", "1", "2", "3", and "4" is considered as a target class in turn and merged with the subset, resulting in five 5-class classification datasets, named SVHN-W. Consequently, fifteen datasets are used to train weak models. The proportion for training, validation, and testing data is 8:1:1.

As a result, 10 CIFAR-W datasets and 5 SVHN-W datasets are constructed to train 3 types of weak models. A total of 90 weak models are obtained, among which, 60 weak models for CIFAR-W (10 datasets with each dataset having 3 weak models for SimCNN and ResCNN, respectively) and 30 weak models for SVHN-W (5 datasets with each dataset having 3 weak models for SimCNN and ResCNN, respectively).

Metrics. Given a set of overly simple, underfitting, and overfitting models, the effectiveness of using modules as patches can be validated by quantitatively and qualitatively measuring the improvements of patched models against weak models. Specifically, the ability of weak models and patched models to recognize a TC can be evaluated in terms of precision and recall. Precision is the fraction of the data belonging to TC among the data predicted to be TC. Recall indicates how much of all data, belonging to TC that should have been found, were found. F1-score is used as a weighted harmonic mean to combine precision and recall.

Table 6. The comparison between GradSplitter and CNNSplitter regarding the accuracy of non-TCs for patched models. All result in %.

Model	Simple					Underfitting					Overfitting				
	Weak	CS	GS	GS-W	GS-CS	Weak	CS	GS	GS-W	GS-CS	Weak	CS	GS	GS-W	GS-CS
SimCNN-CIFAR	77.44	78.23	78.57	1.12	0.34	42.72	43.31	43.52	0.80	0.21	58.59	59.47	60.07	1.48	0.60
SimCNN-SVHN	88.88	89.94	90.53	1.65	0.59	73.97	75.13	76.15	2.17	1.02	91.14	91.52	92.19	1.05	0.67
ResCNN-CIFAR	81.16	81.96	82.04	0.88	0.08	42.80	44.53	44.61	1.81	0.08	60.25	61.28	61.31	1.06	0.03
ResCNN-SVHN	88.06	90.14	90.89	2.83	0.75	74.99	78.82	80.52	5.53	1.70	90.36	91.53	92.18	1.82	0.65

2) Results. Table 5 summarizes the precision, recall, and F1-score of weak models and patched models produced by CNNSplitter and GradSplitter. For instance, in the 3rd row, the 3rd and 5th columns show the average precision of 10 overly simple SimCNN-CIFAR models before and after patching with GradSplitter, respectively. The patched models significantly outperform the weak models (90.05% vs 73.30%), representing an improvement of 16.75% in precision (as shown in the 6th column). Overall, GradSplitter could improve all types of weak models in terms of precision, with an average improvement of 17.13%, and generally improves the weak models in terms of recall, with an average improvement of 4.95%. We observed that the recall values of some patched models decrease (*e.g.* the overly simple SimCNN-CIFAR model), as there is often an inverse relationship between precision and recall [7, 8]. Nevertheless, the improvement in F1-score indicates that all types of weak models could be improved through patching, with an average improvement of 11.47%.

We further compare GradSplitter with CNNSplitter in terms of improvement in recognition of TCs. The columns "GS-CS" in Table 5 present the improvements of GradSplitter against CNNSplitter. Positive improvements are highlighted with a grey background. Except for the overfitting ResCNN-CIFAR and overfitting ResCNN-SVHN models, the patched models produced by GradSplitter are superior to those produced by CNNSplitter. Overall, GradSplitter outperforms CNNSplitter on average by 4.60%, 2.90%, and 3.95% in terms of precision, recall, and F1-score, respectively. The detailed results in terms of precision, recall, and F1-score are available at the project webpage [2].

Besides the improvement in recognizing TC, another concern is whether the patch affects the ability to recognize other classes (*i.e.* non-TCs). To evaluate the patch's effects on non-TCs, the samples belonging to TC are removed, and weak models and patched models are evaluated on the samples belonging to non-TCs. Finally, the effect of the patch on non-TCs is validated by comparing the accuracy of weak models to patched models. The experimental results [2] are summarized in Table 6. Overall, when using GradSplitter to patch weak models, 92% (83/90) of patched models outperform the weak models, and the average accuracy improvement of 90 patched models is 1.85%. The reason for performance improvement is that some samples that belong to non-TCs but were misclassified as TC are correctly classified as non-TCs after patching. The results indicate that the patching does not impair but rather improves the ability to recognize non-TCs. Comparing GradSplitter with CNNSplitter, as shown in the columns "GS-CS", GradSplitter performs better than CNNSplitter across all types of weak models, with an average improvement of 0.56% in non-TCs recognition accuracy.

Both CNNSplitter and GradSplitter effectively enhance the recognition performance of weak CNN models on TCs and non-TCs. Notably, GradSplitter outperforms CNNSplitter, which improves weak CNN models in recognizing TCs with an average increase of 17.13%, 4.95%, and 11.47% in precision, recall, and F1-score, respectively.

RQ3: Can a composed model, built entirely by combining modules, outperform the best trained model?

Just Accepted

Table 7. The modularization results of GradSplitter. The columns "TM" and "CM" present the test accuracy of the *TM* and *CM*. "# K" denotes the average number of kernels in a module.

Idx	SimCNN-CIFAR10			SimCNN-SVHN			ResCNN-CIFAR10			ResCNN-SVHN			InceCNN-CIFAR10			InceCNN-SVHN		
	TM	CM	# K	TM	CM	# K	TM	CM	# K	TM	CM	# K	TM	CM	# K	TM	CM	# K
0	79.28	**78.96**	1842	86.91	**86.86**	1995	80.16	**80.25**	1706	85.07	**86.42**	1679	80.71	**79.81**	1275	80.95	**81.49**	1223
1	78.45	**78.29**	1933	87.41	**87.68**	2002	78.99	**78.11**	1903	82.78	**83.50**	1641	78.26	**77.90**	1334	79.06	**78.79**	1289
2	77.80	**77.33**	1858	84.45	**83.55**	1985	77.53	**77.15**	1907	80.96	**79.99**	2169	79.43	**78.55**	1386	80.48	**80.05**	1452
3	80.10	**80.29**	2025	82.28	**81.45**	2086	81.88	**81.34**	2036	80.96	**80.91**	1828	82.35	**81.81**	1543	83.19	**82.65**	1225
4	77.19	**76.61**	1913	84.33	81.31	1890	78.09	**77.96**	2052	84.54	**84.91**	1718	81.65	**80.79**	1591	81.57	**81.14**	1388
5	79.66	78.49	1938	87.51	85.98	1833	77.93	**77.50**	1964	80.45	**79.97**	2384	79.11	77.90	1513	76.86	**77.88**	1039
6	77.30	**77.09**	2043	78.94	**78.68**	1930	81.06	**80.95**	1977	78.11	**78.51**	1452	82.70	**82.48**	1417	73.03	**73.84**	1213
7	81.01	**80.29**	1821	77.75	76.34	2077	80.88	**80.33**	1821	74.84	**76.01**	1842	80.18	**79.58**	1383	76.61	**76.26**	1168
8	77.23	**76.76**	2035	81.31	**80.55**	1773	76.85	**76.62**	1946	80.54	**79.85**	1817	79.66	**78.81**	1730	81.19	**80.27**	1145
9	78.20	**77.66**	1953	74.82	**73.97**	1803	77.00	**76.76**	1842	82.75	**82.55**	1698	83.06	**82.33**	1398	69.28	**68.37**	1203
Avg.	78.62	78.18	1936	82.57	81.64	1937	79.04	78.70	1915	81.10	81.26	1823	80.71	80.00	1457	78.22	78.07	1234

RQ1 and RQ2 have verified the effectiveness of CNNSplitter and GradSplitter in modularizing CNN models and patching weak models, respectively. Also, the results demonstrate that GradSplitter performs better than CNNSplitter in both modularization and composition. Therefore, in RQ3 to RQ5, we will focus on GradSplitter.

1) Setup. **Dataset construction.** To investigate whether GradSplitter can construct better composed models than trained models by reusing optimal modules from different models, we conducted experiments on 6 pairs of datasets and CNNs. For each pair, we train 10 CNN models (*TMs*) for 10-class classification and decompose each *TM* into 10 modules. In practice, the trained models shared by third-party developers could be trained on datasets with different distributions. Therefore, in the experiment, instead of training 10 *TMs* on the initial training set D, we draw 10 subsets $\{S_j\}_{j=1}^{10}$ from D and train a *TM* on each subset. Each subset $S_j = \{S_j^n\}_{n=1}^{10}$ consists of 10 classes of samples, similar to $D = \{D^n\}_{n=1}^{10}$, where S_j^n and D^n indicate the samples in S_j and D belonging to class n, respectively. To ensure that the sampling is reasonable, the sampling is performed according to Dirichlet distribution [50], which is an appropriate choice to make subsets similar to the real-world data distribution [34] and is the hypothesis on which many works are based [37, 42].

Specifically, we first assign the class n a proportion value p_j^n ($0 < p_j^n \leq 1$) that is sampled from the Dirichlet distribution $Dir(\beta)$. Then, we draw $p_j^n \times |D^n|$ samples from D^n randomly to construct S_j^n. Finally, S_j is constructed once all classes have been sampled. Here, $Dir(\beta)$ denotes the Dirichlet distribution and β is a concentration parameter ($\beta > 0$). If β is set to a smaller value, then the greater the difference between $\{p_j^n\}_{n=1}^{10}$, resulting in a more unbalanced proportion of sample size between the 10 classes. For CIFAR-10, we set $\beta = 1$, and for SVHN with more data, we set $\beta = 0.5$. In addition, a threshold t is set to ensure that a CNN model has sufficient samples to learn to recognize all classes. p_j^n and S_j^n are resampled when $p_j^n \times |D^n| < t$. We set t=100 for both CIFAR-10 and SVHN.

Each subset S_j is used to train a *TM* and modularize the *TM*. Specifically, S_j is randomly divided into two parts in the ratio of 8:2. The 80% samples are used as *training dataset* to train the *TM* and modularize the *TM*. The 20% samples are used as *validation dataset* to evaluate the *TM* and the *CM* during training and modularization.

The initial test dataset is randomly divided into two parts in the ratio of 8:2. The 80% samples are used as the *test dataset* to evaluate *TMs* and *CMs* after training or modularization. The 20% samples are used as the *module evaluation dataset* to evaluate and recommend modules (see Section 5.2.1).

With 6 pairs of datasets and CNNs, we train 10 models for each pair, resulting in 60 CNN models, and then use GradSplitter to decompose these models. The training and modularization settings of the 60 models are the same as that of strong models mentioned in RQ1.

Just Accepted

Table 8. The summarized results of modularization. "Acc." denotes the average test accuracy, and "# Kernels (%)" denotes the average number (and percentage) of kernels in a module.

Model	Trained Model		Composed Model		
	# Kernels	Acc.	Acc.	Loss	# Kernels (%)
SimCNN-CIFAR10	4224	78.62	78.18	0.44	1936 (46%)
SimCNN-SVHN	4224	82.57	81.64	0.93	1937 (46%)
ResCNN-CIFAR10	4288	79.04	78.70	0.34	1915 (45%)
ResCNN-SVHN	4288	81.10	81.26	-0.16	1823 (43%)
InceCNN-CIFAR10	3200	80.71	80.00	0.71	1457 (46%)
InceCNN-SVHN	3200	78.22	78.07	0.15	1234 (39%)
Average				**0.40**	**1717 (44%)**

2) Results. **Modularization.** Table 7 shows the modularization results of GradSplitter for six pairs of datasets and CNNs, with 10 *TMs* per pair, for a total of 60 *TMs*. Each *TM* is decomposed into 10 modules, and a *CM* is composed of the 10 modules and classifies the same classes as the *TM*. We evaluate *TMs* and *CMs* on the test dataset and compare the test accuracy of *TMs* and *CMs*. For each *TM*, Table 7 shows the test accuracy of the *TM* and the corresponding *CM*, as well as the average number of kernels in a module (the column "# K"). Among the 60 *CMs*, the 55 *CMs* with less than 1% accuracy loss are highlighted in bold when compared to *TMs*.

Table 8 summarizes the results of six pairs of datasets and CNNs, with 10 *TMs* per pair. For SimCNN-CIFAR, SimCNN-SVHN, ResCNN-CIFAR, ResCNN-SVHN, InceCNN-CIFAR, and InceCNN-SVHN, the average test accuracy of 10 *TMs* and the average test accuracy of 10 *CMs* are (78.62%, 78.18%), (82.57%, 81.64%), (79.04%, 78.70%), (81.10%, 81.26%), (80.71%, 80.00%), and (78.22%, 78.07%), respectively. As shown in the column "Loss", for each pair, the average loss of accuracy for *CMs* compared to *TMs* is less than 1%. Overall, the average loss of accuracy of 60 *CMs* is only 0.4%, which demonstrates that the *CMs* have comparable accuracy to the *TMs* on the 10-class classification tasks. The comparable accuracy of the *CMs* suggests that the modules have sufficient ability to recognize the features of the target classes.

We also count the number of convolution kernels in *TMs* and the number of retained convolution kernels in modules. The column "# Kernels" of Table 8 presents the number of kernels of each *TM*. And as shown in the last column "# Kernels (%)" of Table 8, for SimCNN-CIFAR, SimCNN-SVHN, ResCNN-CIFAR, ResCNN-SVHN, InceCNN-CIFAR, and InceCNN-SVHN, a module retains about 40% convolution kernels of a *TM*. The average number and percentage of retained convolution kernels in a module for the 60 *CMs* are 1717 and 44% respectively, indicating that the size of modules is much smaller than that of *TMs*. The small size of modules leads to a lower prediction overhead than that of *TMs* (as discussed in RQ5). Consequently, the modularization results of 60 *TMs* demonstrate that GradSplitter can strike a balance between the module's recognition ability and the module size. The resultant modules will be reused to build better *CMs* than *TMs* (discussed in RQ3) and construct *CMs* for new tasks (as discussed in RQ4).

Composition. To develop a *CM* that outperforms all *TMs*, modules are first tested on the module evaluation dataset (see Dataset construction in setup). Each class in a task is considered in turn as the target class (TC). For each TC, the corresponding modules that can recognize the TC form a set of candidate modules, and GradSplitter evaluates the candidate modules and recommends the module with the best recognition ability to the developer (detailed in Section 5.2.1). Candidate modules can come from *TMs* with the same network structure or *TMs* with different network structures. We refer to the former as *intra-network reuse* and the latter as *inter-network reuse*.

Intra-network reuse. Table 9 shows the testing results of candidate modules from 10 SimCNN-CIFAR *TMs*, which is used for *intra-network* reuse. For each class (TC), there are 10 modules from each of the 10 *TMs*. Thus, in rows 2 to 11 of Table 9, each row shows the performance of the modules on the corresponding TC in terms of F1-score, with the best F1-score highlighted in bold. The last row shows the accuracy of the 10 *TMs* on the

Just Accepted

Table 9. The evaluation results of modules of 10 SimCNN-CIFAR *TMs* in terms of F1-score on module evaluation dataset. The test accuracy of the *TMs* is presented in the last row for comparison. All results in %.

TC \ TM	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	81.62	76.75	80.60	**86.27**	77.20	77.73	72.04	84.58	77.88	81.13
1	**92.09**	88.67	90.13	90.09	78.82	77.68	82.32	91.04	87.84	85.87
2	74.03	68.90	74.16	70.74	75.76	**78.62**	70.53	78.01	50.36	53.42
3	60.76	59.70	64.15	**67.18**	64.07	62.84	59.06	66.17	65.09	63.66
4	70.46	**81.98**	81.29	69.19	81.73	74.48	62.82	81.15	75.63	80.18
5	74.33	**76.78**	60.40	74.93	35.09	67.11	72.88	60.63	71.22	69.18
6	84.70	74.14	72.78	75.52	84.26	**86.57**	83.02	83.73	85.32	74.11
7	76.61	79.17	72.57	**88.38**	81.92	84.40	80.79	84.40	82.13	84.11
8	**89.73**	79.79	88.29	88.74	80.08	88.02	84.28	89.69	80.10	83.06
9	**91.61**	83.33	82.94	88.43	76.96	83.26	83.53	81.34	84.26	85.57
Acc.	79.28	78.45	77.80	80.10	77.19	79.66	77.30	**81.01**	77.23	78.20

Table 10. Intra-network reuse. All results in %.

Dataset	**CNN**	**Accuracy**		**Improvement**
		Best TM	**CM**	
CIFAR-10	SimCNN	81.01	86.26	5.25
	ResCNN	81.88	85.95	4.07
	InceCNN	83.06	86.94	3.88
SVHN	SimCNN	87.51	93.12	5.61
	ResCNN	85.07	90.55	5.48
	InceCNN	83.19	90.22	7.03
Average				**5.22**

10-class classification task, with the best accuracy highlighted in bold. The testing results show that (1) the *TM* with an index of 7 has the highest accuracy (*i.e.* 81.01%) on 10-class classification; however, the modules of this *TM* are not optimal in terms of F1-score, and (2) the modules with the best performance for different TCs could come from different *TMs*.

Based on the evaluation results of modules, the modules with the best performance on the module evaluation dataset are reused to build a *CM* for the 10-class classification task. We evaluate the *CMs* on the test dataset and compare the test accuracy of *TMs* and *CMs*. As shown in the 3rd row of Table 10, the test accuracy of the *CM*, which is composed of the modules highlighted in bold in Table 9, is 86.26%. The *CM* outperforms the best *TM*, and the improvement in accuracy is 5.25%. Table 10 also shows the results for the other five cases of intra-network reuse. In all six cases, the *CMs* outperform the corresponding best *TMs*. On average, the improvement in terms of accuracy obtained by modularization and composition is 5.22%.

Inter-network reuse. Table 11 shows the testing results of modules that come from 10 SimCNN-CIFAR *TMs* and 10 ResCNN-CIFAR *TMs*. For each TC, there are 20 candidate modules. Rows 3 to 12 of Table 11 show the performance of the modules in terms of F1-score. Similar to intra-network reuse, the evaluation results show that the ResCNN-CIFAR *TM* with index 3 has the highest accuracy (*i.e.* 81.88%) on 10-class classification; however, not all modules of this *TM* are optimal in terms of F1-score. For the 10 TCs, Table 11 highlights the corresponding modules with the highest F1-score. Among the 10 modules, 5 modules are from the SimCNN-CIFAR *TMs*, and 5 modules are from the ResCNN-CIFAR *TMs*. As shown in the 3rd row of Table 12, the test accuracy of the *CM* composed of these 10 modules is 87.24%. Compared to the best *TM*, *i.e.* the ResCNN-CIFAR *TM* with index 3, the improvement in accuracy is 5.36%. Table 12 also shows the results of inter-network reuse for the other seven cases. In all eight cases, the *CMs* outperform the corresponding best *TMs* with an average improvement of 5.14% in accuracy.

Just Accepted

Table 11. The evaluation results of modules of 10 SimCNN-CIFAR *TMs* and 10 ResCNN-CIFAR *TMs* in terms of F1-score on module evaluation dataset. The test accuracy of the *TMs* is presented in the last row for comparison. All results in %.

TM / TC	SimCNN-CIFAR										ResCNN-CIFAR									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	81.62	76.75	80.60	**86.27**	77.20	77.73	72.04	84.58	77.88	81.13	78.37	75.73	81.48	79.59	83.25	75.26	81.98	84.92	72.94	84.01
1	92.09	88.67	90.13	90.09	78.82	77.68	82.32	91.04	87.84	85.87	**92.57**	89.31	88.83	89.98	81.84	87.82	89.18	86.36	87.07	90.13
2	74.03	68.90	74.16	70.74	75.76	**78.62**	70.53	78.01	50.36	53.42	72.23	72.11	75.85	69.89	72.80	75.13	74.40	74.01	50.71	53.10
3	60.76	59.70	64.15	67.18	64.07	62.84	59.06	66.17	65.09	63.66	64.77	65.02	62.38	**72.64**	64.10	59.83	68.98	66.93	68.83	63.55
4	70.46	81.98	81.29	69.19	81.73	74.48	62.82	81.15	75.63	80.18	66.52	81.40	75.90	82.24	68.10	63.38	61.18	**85.58**	76.06	80.28
5	74.33	**76.78**	60.40	74.93	35.09	67.11	72.88	60.63	71.22	69.18	68.24	73.12	61.89	75.28	64.65	72.36	76.14	58.78	70.34	67.65
6	84.70	74.14	72.78	75.52	84.26	86.57	83.02	83.73	85.32	74.11	82.40	78.55	82.17	85.14	86.10	82.11	**86.82**	85.43	83.26	68.41
7	76.61	79.17	72.57	**88.38**	81.92	84.40	80.79	84.40	82.13	84.11	79.56	80.00	71.56	86.49	83.87	79.31	82.09	84.68	82.85	83.45
8	89.73	79.79	88.29	88.74	80.08	88.02	84.28	89.69	80.10	83.06	**89.80**	73.97	89.04	84.29	84.87	86.94	86.03	84.38	79.51	87.91
9	**91.61**	83.33	82.94	88.43	76.96	83.26	83.53	81.34	84.26	85.57	85.15	80.60	85.20	86.82	83.97	82.87	88.89	83.78	87.23	84.80
Acc.	79.28	78.45	77.80	80.10	77.19	79.66	77.30	81.01	77.23	78.20	80.16	78.99	77.53	**81.88**	78.09	77.93	81.06	80.88	76.85	77.00

Table 12. Inter-network reuse. All results in %.

Dataset	CNN	Accuracy		Improvement
		Best TM	**CM**	
CIFAR-10	SimCNN-ResCNN	81.88	87.24	5.36
	SimCNN-InceCNN	83.06	87.18	4.12
	ResCNN-InceCNN	83.06	86.95	3.89
	SimCNN-ResCNN-InceCNN	83.06	87.34	4.28
SVHN	SimCNN-ResCNN	87.51	93.35	5.84
	SimCNN-InceCNN	87.51	93.17	5.66
	ResCNN-InceCNN	85.07	91.23	6.16
	SimCNN-ResCNN-InceCNN	87.51	93.35	5.84
Average				**5.14**

Both intra-network reuse and inter-network reuse demonstrate that, using GradSplitter, a composed CNN model with higher accuracy than all the trained CNN models can be developed through modularization and composition. Overall, the average improvement on all 14 cases (6 cases for intra-network reuse and 8 cases for inter-network reuse) is 5.18%. Furthermore, comparing intra-network reuse (shown in Table 10) with inter-network reuse (shown in Table 12), we found that the more candidate modules there are, the higher the accuracy of the *CM*. For instance, as shown in Table 12 (Row 3) and Table 10 (Rows 3 and 4), the *CM* composed of the modules from SimCNN-CIFAR *TMs* and ResCNN-CIFAR *TMs* outperforms the *CM* composed of the modules only from SimCNN-CIFAR *TMs* or only from ResCNN-CIFAR *TMs*. This suggests that GradSplitter is promising and can benefit from a large number of shared modules.

Both intra-network and inter-network reuse can develop accurate composed CNN models, and the average accuracy improvement is 5.18%. Experimental results demonstrate the feasibility of reusing modules generated by GradSplitter to build more accurate models.

RQ4: Can a CNN model for a new task be built through modularization and composition while maintaining an acceptable level of accuracy?

1) Setup. For a new task, if existing models cannot satisfy the functional requirement, a developer can only reuse the model structure and retrain the model from scratch (the retrained model is referred to as *RTM* for short). However, through modularization and composition, the modules of existing models can be reused to create a *CM*

Just Accepted

Table 13. The results of reusing SimCNN modules for developing *CMs* for new tasks. The row "C" and column "S" indicate the class index of CIFAR10 dataset and SVHN dataset, respectively. All results are test accuracy and are in %.

C / S	0		1		2		3		4		5		6		7		8		9	
	RTM	**CM**	**RTM**	**CM**	**RTM**	**CM**	**RTM**	**CM**	**RTM**	**CM**	**RTM**	**CM**	**RTM**	**CM**	**RTM**	**CM**	**RTM**	**CM**	**RTM**	**CM**
0	99.95	99.09	100.0	98.91	99.85	96.54	99.37	92.06	99.90	99.50	99.66	96.33	99.90	99.45	99.90	99.04	99.80	99.31	99.85	99.54
1	99.76	98.63	99.95	99.06	99.81	95.19	99.66	93.12	99.74	99.18	99.71	97.35	99.92	98.91	99.81	98.75	99.84	99.10	99.97	99.41
2	99.74	98.46	99.84	98.77	99.74	94.65	99.59	91.94	99.87	99.01	99.62	96.83	99.81	99.49	99.81	98.70	99.68	98.30	99.94	99.18
3	99.42	96.87	99.78	97.19	99.33	94.06	98.29	86.10	99.60	98.13	99.78	90.77	99.85	98.21	99.75	97.77	99.78	97.92	99.85	99.19
4	99.76	96.41	99.92	98.19	99.75	96.34	99.15	91.14	99.24	98.61	99.61	94.72	99.76	98.82	99.72	98.05	99.88	97.78	99.88	98.89
5	99.80	98.31	99.92	98.82	99.67	95.22	99.56	89.00	99.71	98.97	99.64	95.46	99.88	99.00	99.71	99.05	99.79	98.74	99.84	98.89
6	99.86	98.48	99.95	98.61	99.72	96.63	99.45	92.23	99.68	99.16	99.64	95.51	100.0	99.32	99.86	98.86	99.91	98.17	100.0	99.32
7	99.81	98.19	99.95	98.56	99.72	93.26	99.54	91.36	99.76	98.81	99.77	96.77	100.0	99.17	99.81	98.56	99.86	99.13	100.0	99.13
8	99.85	97.56	99.95	98.40	99.75	96.63	99.07	84.18	99.71	98.31	99.76	93.14	100.0	98.67	99.80	98.46	100.0	98.77	100.0	98.92
9	99.64	98.36	99.90	98.84	99.46	94.63	99.38	87.35	99.58	98.84	99.69	93.84	99.84	99.17	99.84	98.74	99.69	98.48	99.84	98.64

Table 14. The summarized results of reusing modules for developing *CMs* for new tasks. All results are test accuracy and are in %.

CNN	**RTM**	**CM**	**Loss**
SimCNN	99.75	97.10	2.65
ResCNN	99.75	97.22	2.53
InceCNN	99.83	97.64	2.19
Average			**2.46**

that satisfies the task without costly retraining. Specifically, to develop a *CM* for a new task, each class of the task is treated in turn as TC, and the candidate modules are evaluated on the module evaluation dataset. Then, for each class, the module that achieves the best classification result is recommended to be reused. For instance, a new binary classification task is to classify two classes that come from CIFAR-10 and SVHN, respectively. The modules from SimCNN-CIFAR *TMs* and SimCNN-SVHN *TMs* (produced in RQ3) can be composed to satisfy the binary classification task. Moreover, the test accuracy of a *CM* should be close to that of the *RTM*. The *RTM* is obtained by retraining SimCNN on the binary classification dataset from scratch with the settings described in RQ1, except that the number of epochs is changed to 30.

2) Results. Table 13 shows the test accuracy of *RTMs* and *CMs*. A binary classification task is formed by one of CIFAR-10's 10 classes and one of SVHN's 10 classes, for a total of 100 binary tasks. *CM* refers to the composed model constructed by the best module from the 10 SimCNN-CIFAR models and the best module from the 10 SimCNN-SVHN models. *RTM* refers to the binary classification SimCNN trained from scratch. The time cost caused by training a binary model from scratch is 150 seconds for SimCNN and ResCNN (calculated by 5 seconds per epoch times 30 epochs), and 180 seconds for InceCNN (calculated by 6 seconds per epoch times 30 epochs). Table 14 summarizes the results of reusing SimCNN, ResCNN, and InceCNN modules, showing the average test accuracy of 300 *RTMs* and 300 *CMs*, respectively. The results show that the accuracy of *CMs* is slightly lower than that of *RTMs*. Compared to retraining a model from scratch, reusing modules causes an average accuracy loss of 2.46%.

The loss of test accuracy could be due to a slight decrease in module recognition ability caused by modularization (see RQ1). On the other hand, since each module in a *CM* has not processed the samples belonging to non-TC during training the *TM*, the modules could misclassify non-TC during prediction. For example, the SimCNN-CIFAR modules have not processed the samples from SVHN during training SimCNN-CIFAR models and thus could misclassify. Overall, compared to the model trained from scratch, the composed CNN models can achieve similar

Just Accepted

Table 15. Overhead for modularization.

Dataset	CNN	Time		GPU Memory
		time per epoch	total time	
CIFAR-10	SimCNN	28 sec	67 min	3.9 GB
	ResCNN	32 sec	77 min	5.7 GB
	InceCNN	56 sec	136 min	8.9 GB
SVHN	SimCNN	33 sec	79 min	3.9 GB
	ResCNN	36 sec	88 min	5.7 GB
	InceCNN	57 sec	138 min	8.9 GB
Average		**40 sec**	**98 min**	**6.1 GB**

accuracy. The experimental results demonstrate that reusing modules to build a new CNN model for a new task is feasible.

Compared to the models retrained from scratch, the composed models can achieve similar accuracy (the average loss of accuracy is only 2.46%). Experimental results demonstrate that it is feasible to reuse modules to build CNN models for new tasks.

RQ5: How efficient is GradSplitter in modularizing CNN models and how efficient is the composed CNN model in prediction?

Besides the accuracy of *CMs*, the modularization efficiency of GradSplitter and the prediction efficiency of *CMs* are also of concern. Specifically, decomposing a trained CNN model should not bring too much time and computational overhead to developers. Also, the composed CNN models should not incur too much additional overhead than the trained CNN models in the prediction phase.

Overhead for Modularization. Table 15 shows the overhead for decomposing a trained CNN model, including time overhead and computational overhead (*i.e.* the GPU memory usage). The time and computational overhead is brought by training masks and heads. For each pair of dataset and CNN, there are 10 trained CNN models (introduced in RQ3); thus, the values of time overhead and the GPU memory usage presented in Table 15 are the average values of 10 models. On average, the time overhead per epoch is 40 seconds, and the total time for decomposing a trained CNN model is 98 minutes. The time overhead on SVHN is larger than that on CIFAR-10, as the former has more data than the latter. For different CNNs, the time overhead for modularization is positively correlated with the number and the size of convolution kernels in the CNN. ResCNN has more convolution kernels than SimCNN; thus, the time overhead for ResCNN is larger. Since the convolution kernel size of InceCNN is 5×5, which is larger than that of SimCNN and ResCNN (*i.e.* 3×3), the time overhead for InceCNN is the largest. The GPU memory usage is also positively correlated with the number and the size of convolution kernels; thus, the GPU memory usage for ResCNN is larger than SimCNN, and the GPU memory usage for InceCNN is the largest. On average, the GPU memory usage is 6.1GB. The experimental results demonstrate that both time overhead and computational overhead are not expensive.

Moreover, modularization can be a one-time and offline process performed by the model sharer. With the shared modules, third-party developers could build models without costly training. For instance, modularization on SimCNN-CIFAR10 and SimCNN-SVHN takes approximately 146 minutes (calculated by $67min + 79min$). The resulting modules can be used to build 100 binary classification CMs without additional training. In contrast, training 100 binary classification SimCNN models from scratch would take about 250 minutes (calculated by $150s \times 100 \div 60$). Although training a single binary classification model is cost-effective, the total time costs could

Just Accepted

Table 16. Prediction overhead in parallel prediction mode. "Inc." denotes the increased overhead of *CMs* over *TMs*.

Dataset	CNN	Time (s)			GPU Memory (MB)		
		TM	CM	Inc. (%)	TM	CM	Inc. (%)
CIFAR-10	SimCNN	4.0	2.6	-35.0	405.4	3303.4	714.9
	ResCNN	5.4	3.5	-35.2	441.1	3579.0	711.4
	InceCNN	9.0	7.4	-17.8	727.0	5949.5	718.3
SVHN	SimCNN	8.8	6.2	-29.5	405.4	3265.8	705.7
	ResCNN	12.3	7.3	-40.7	441.1	3194.1	624.1
	InceCNN	21.8	18.9	-13.3	727.0	5338.6	634.3
Average		**10.2**	**7.7**	**-28.6**	**524.5**	**4105.1**	**684.8**

be significant when dealing with a large number of models. For model sharing platforms with a massive user base, the value generated by avoiding retraining through modularization may be even more substantial.

Overhead for Reusing. The overhead for reusing refers to the prediction overhead of the composed models (*CMs*), which consists of time overhead and GPU memory usage. Since the time overhead for predicting a single input is very small, it is measured using all test data. The GPU memory usage comes from two sources: the weights of a *TM*/*CM* and the intermediate results, which can be measured using the open-source tool *torchsummary* [65]. Since each module can be reused as a binary classifier, there are two prediction modes for the *CM*, including parallel prediction and serial prediction. In parallel prediction, the modules within a *CM* predict simultaneously, while in serial prediction, the modules predict sequentially (one after the other). The time overhead and GPU memory usage for the *CM* prediction vary in different modes.

Table 16 shows the time overhead and GPU memory usage for *TMs* and *CMs* in parallel prediction. As a module retains only about 44% convolution kernels (shown in Table 8), the time overhead of a module is less than that of the corresponding *TM*. In parallel prediction, a *CM* predicts through executing the modules in parallel; thus, the time overhead of a *CM* is less than that of the corresponding *TM*. For all pairs of datasets and CNNs, the *CM* incurs less time overhead than the *TM*. On average, reusing modules reduces the average prediction time overhead by 28.6%. For the GPU memory usage, a *CM* requires more GPU memory than the corresponding *TM*, as the former has more convolution kernels and intermediate results. On average, the GPU memory usage for a *TM* and a *CM* is 524.5MB and 4105.1MB, respectively. And reusing modules increases the average GPU memory usage by 684.8%.

Table 17 shows the time overhead and GPU memory usage for *TMs* and *CMs* in serial prediction. Though a module incurs less time overhead than the corresponding *TM*, 10 modules predicting in serial incurs more time overhead than the corresponding *TM*. On average, reusing modules increases the average prediction time overhead by 770.6%. For the GPU memory usage, reusing modules increases the average GPU memory usage by 55.88%. A *CM* requires more GPU memory than the corresponding *TM*; however, the additional GPU memory usage is much less than that in parallel prediction. Since modules are executed in serial, only one module produces intermediate results at each moment. Consequently, the GPU memory usage for a *CM* consists of the weights of all modules and the intermediate results of only one module; hence the serial prediction takes much less GPU memory usage than the parallel prediction.

In serial prediction, *CMs* based on ResCNN and InceCNN increase the GPU memory usage by 12.8% and 3.7% on CIFAR-10 and 3.2% and 0.6% on SVHN, respectively. Compared to *CMs* based on SimCNN, *CMs* based on ResCNN and InceCNN incur much less GPU memory usage, as the ResCNN modules and InceCNN modules retain fewer weights of the corresponding *TM*. SimCNN has 3 FC layers, with only one FC layer connecting to the final convolutional layer having its weights reduced. ResCNN and InceCNN both have a single FC layer that connects to the final convolutional layer and has its weights reduced. Compared to SimCNN modules, ResCNN modules

Just Accepted

Table 17. Prediction overhead in serial prediction mode. "Inc." denotes the increased overhead of *CMs* over *TMs*.

Dataset	CNN	Time (s)			GPU Memory (MB)		
		TM	CM	Inc. (%)	TM	CM	Inc. (%)
CIFAR-10	SimCNN	4.0	37.1	827.3	405.4	1040.7	156.7
	ResCNN	5.4	43.4	703.3	441.1	497.6	12.8
	InceCNN	9.0	79.6	784.4	727.0	753.8	3.7
SVHN	SimCNN	8.8	90.4	926.7	405.4	1046.9	158.3
	ResCNN	12.3	89.3	625.9	441.1	455.3	3.2
	InceCNN	21.8	186.6	756.1	727.0	731.2	0.6
Average		**10.2**	**87.7**	**770.6**	**524.5**	**754.3**	**55.88**

and InceCNN modules retain fewer weights of the corresponding *TM*, resulting in less GPU memory usage. The experimental results indicate that GradSplitter is more suitable for CNNs with fewer FC layers. Fortunately, the trend in CNN model design is to reduce the number of FC layers [32]. For instance, the FC layers are replaced with average pooling layers in recent work [21, 25, 71].

When comparing parallel and serial prediction, there is a trade-off between time overhead and GPU memory usage. With sufficient GPU memory, reusing modules in parallel prediction not only achieves better accuracy than the *TM* but also speeds up the prediction. With limited GPU memory, reusing modules in serial prediction can improve accuracy while incurring only a small increase in GPU memory usage.

> Modularization only incurs low time and computational overhead. In prediction, *CMs* incur additional overhead; however, running modules in parallel can balance the time overhead and GPU memory usage. In the case of all modules running in parallel, the time overhead of *CMs* can be significantly lower than that of *TMs*.

7 DISCUSSION

This section discusses the difference between compressed and uncompressed modularization techniques, as well as the threats to the validity of the proposed approaches and experimental results.

7.1 Compressed Modularization *vs.* Uncompressed Modularization

CNNSplitter and GradSplitter are *compressed modularization* approaches that remove convolution kernels, instead of individual weights, from a trained CNN model (*TM*). The module generated by compressed modularization has fewer weights than the *TM*. On the contrary, *uncompressed modularization* [43, 44] removes individual weights or neurons from a *TM* and generates modules with sparse weight matrices (as illustrated in Figure 12). As a result, a module has the same number of weights as the *TM*, and the prediction overhead of a *CM* could be significantly higher than that of the *TM* due to much more weights.

To understand the limitations of uncompressed modularization, we analyze the open source project [55] published by Pan *et al.* [44], including source code files and the experimental data (*e.g.* the trained CNN models and the generated modules). Based on the default settings and the published experimental data, we run the project to evaluate how much additional prediction overhead the *CM* requires compared to the *TM*. In that project, the test dataset is CIFAR-10, and the *CM* is composed of 10 modules. Since the *CM* in that project runs each module serially to predict, we only discuss the overhead for the serial prediction mode.

For the *TM*, the prediction time overhead is 2.2s, while the time overhead of the *CM* is 4542.6s. The results show that reusing the modules generated by the uncompressed modularization approach [44] increases the prediction time overhead by 2,064.8%. The significant additional time overhead is caused by (1) serial prediction

Just Accepted

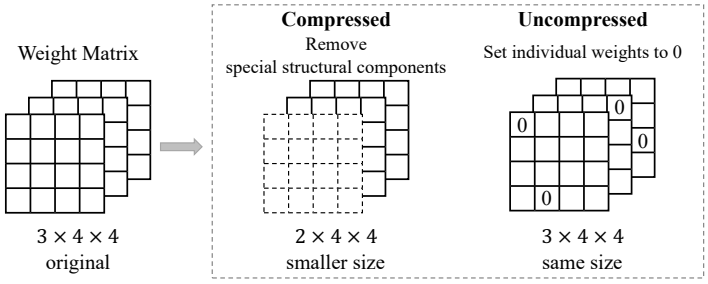

Fig. 12. An illustration of the two types of modularization

and (2) the additional operations (*i.e.* setting the values of neurons to zero according to the map). Since each module in the *CM* has the same number of weights as the *TM*, 10 modules predict serially causing about 10 times (1,000%) overhead than the *TM*. Moreover, given an input image, the output (*i.e.* a vector of neuron values) of each convolutional layer cannot be fed directly to the next layer, as part of neuron values in the output need to be set to zeros.

The computational overhead (*i.e.* GPU memory usage) comes from two sources: the weight matrices in the *TM*/*CM* and the intermediate results. As a module has the same number of weights as the *TM*, the additional overhead incurred by more weights is about 1,000%. On the other hand, there are more additional intermediate results in the *CM* due to the additional operations, which incurs more overhead. Since the additional operations are custom operations (*e.g.* iterating the map) that are not integrated into modules as a layer, the overhead caused by intermediate results cannot be computed using existing open-source tools. Complex manual computations are required; thus, the overhead of intermediate results is not discussed here. Overall, in [44], reusing modules increases the prediction time overhead by 2,064.8% and the computational overhead by about 1,000%.

Consequently, compressed modularization, such as GradSplitter proposed in this paper, can decompose a trained CNN model into small modules with fewer weights. The additional prediction overhead in serial prediction mode is significantly less than that of the uncompressed modularization approach.

7.2 Threats to Validity

External validity: Threats to external validity relate to the generalizability of our results. First, some CNN models have different structures and sizes compared to SimCNN, ResCNN, and InceCNN. The results might not be generalizable to these CNN models. For instance, CNNSplitter's performance is subject to the size of search space exponentially, leading to a trade-off between the efficiency and quality of modularization. It remains to be validated whether the default settings, such as the number of groups, can achieve a good balance for different models. Second, the results are not validated on other datasets for different tasks. However, SimCNN, ResCNN, and InceCNN are representative CNN models, and their structures are widely used in various tasks. Many different CNN models can be seen as variants of SimCNN, ResCNN, and InceCNN. In addition, CIFAR-10, CIFAR-100, and SVHN are representative datasets and are widely used for evaluation in related research [13, 44]. In our future work, more experiments will be conducted on a variety of CNN models and datasets to alleviate this threat. Finally, a threat relates to the conclusion that a composed model will outperform trained models. In our experiments, the performance of models trained on different sampled distributions could vary (See RQ3); thus, a composed model could outperform trained models by combining their advantage in classifying different classes. However, if models are trained on a similar distribution, the conclusion may not necessarily hold.

Internal validity: An internal threat comes from the implementation of trained CNN models. The results of modularization and composition could vary in different training settings, such as hyperparameters and training

Just Accepted

strategies. In our experiments, the training settings follow the common settings [21, 64, 70], which are widely used.

Construct validity: In this study, a threat relates to the suitability of our evaluation metrics. The accuracy and the number of retained convolution kernels are used as the metrics to evaluate modularization. Moreover, the accuracy, time overhead, and GPU memory usage are used to evaluate the composed model. These metrics have also been used in other related work [43, 44].

8 RELATED WORK

8.1 Modularization of Deep Neural Networks

Neural network modularization [3, 4, 6, 24, 31, 43, 44, 52, 53, 61, 79] has attracted increasing interest in both the AI and SE communities. We classify existing works into three categories: modularizing *before*, *while*, and *after* training. *Modularizing before training* [3, 4, 6, 31, 61, 79] refers to adopting a modular design during the phase of building model architecture. For instance, Mixture-of-Experts (MoE) [11, 31, 61] is a typical modular design. The MoE layer consists of multiple experts (similar to modules), each being a fully connected neural network. It also has a gating network that selects a combination of experts to process each input. Through such modular design, these works aim to significantly increase model capacity without a proportional increase in computational overhead.

Different from the aforementioned work, our work belongs to *modularizing after training* [24, 43, 44, 52], which focuses on decomposing a trained DNN model into modules, with each module responsible for one subtask of the trained model. Existing approaches [24, 43, 44] design heuristic metrics based on neuron activation to indirectly measure the relevance between weights (or neurons) and sub-tasks. They then remove irrelevant neurons or weights from the model by setting them as zeros to generate modules. These modularization approaches could have limitations in practice, as the size of the weight matrix of a module is the same as that of the original trained model, resulting in a composed model incurring several times higher prediction overhead than the original model. In contrast, our work is search-based, which allows for a more direct measurement of the relevance between weights and subtasks according to the impact of weight removal on modules' performance for subtasks. Moreover, our work realizes modularization with reduced network size, helping to decrease the overhead of module reuse.

In addition, Qi et al. [53] recently propose a new paradigm of modularization, namely *modularizing while training* (MwT). MwT introduces the concepts of cohesion and coupling from modular development into the process of training common DNN models from scratch. It designs cohesion and coupling loss functions and guides the training process to follow the modular criteria of high cohesion and low coupling. In contrast, our work belongs to modularizing after training, focusing on decomposing trained models.

8.2 Reusing Deep Neural Networks

Our work is related to the work on model reuse, such as direct reuse [39, 75] and transfer learning [9, 10, 45, 81]. Direct model reuse aims to recommend a trained model for developers and enables developers to reuse the model for their new tasks directly. For instance, Wu *et al.* [75] use the reduced kernel mean embedding (RKME) as a specification for a trained model and then recommend a trained model according to the relatedness of the new task and trained models, which is measured based on the value of the RKME specification. Transfer learning techniques develop a new model by reusing the entire or a part of a model trained on the other dataset and then fine-tuning the reused model on the new dataset. For instance, BERT [9] can be used to develop new models for various downstream tasks by changing the heads (*i.e.* the output layers) and fine-tuning on the new dataset. The techniques mentioned above reuse an entire (or vast majority of) trained model, while our work reuses the optimal modules through modularization.

Just Accepted

8.3 DNN Debugging

Existing DNN debugging techniques improve DNNs mainly by providing more training data [38]. One of the mainstream DNN debugging techniques is the generation technique [5, 72, 77, 83], which generates new training samples that are similar to the provided input data samples. For instance, DeepHunter [77] and DeepTest [72] generate new images by mutating an original image with metamorphic mutations such as pixel value transformation and affine transformation. DeepRoad [83] and infoGAN [5], both based on generative adversarial networks, train a generator and a discriminator and then use the generator to generate new images. Another popular technique is the prioritization technique [13, 74], which can find the possibly-misclassified data from massive unlabeled data. The possibly-misclassified data, rather than total data, are manually labeled first and added into the training dataset to improve a DNN. Unlike the existing DNN debugging techniques focusing on retraining models with more training data, this work focuses on patching models without retraining.

8.4 Neural Architecture Search

Neural architecture search (NAS) techniques [35, 56] construct the optimal neural network structure by searching combinations of network layers, layer connections, activation methods, and so on. CNNSplitter searches modules from a trained CNN model. Apart from their differences in objectives, there are some other differences between CNNSplitter and NAS. For instance, genetic CNN [35] encodes CNN model architectures into bit vectors and applies a genetic algorithm to search. Each bit of a bit vector represents whether or not a connection between two convolutional layers is required. While in CNNSplitter, each bit of bit vectors represents whether a kernel group is retained; thus, the approach of genetic CNN cannot be directly applied to modularization. Moreover, CNNSplitter includes three heuristic methods (see Section 3) to improve the efficiency of search, which are also different from genetic CNN.

8.5 DNN Pruning

DNN pruning techniques [14, 19, 30, 59, 85] are employed to remove weights that are not important for the whole task, resulting in a smaller model and reducing the resources and time required for inference on the initial task. For instance, magnitude-based pruning [14, 32, 40, 59, 85], which is one of the mainstream techniques, iteratively prunes the individual weights [14, 59, 85] or convolution kernels [32, 40] with the smallest absolute values and trains the retained weights or kernels to recover from pruning-induced accuracy loss. In contrast, our work removes convolution kernels that are irrelevant to the sub-task (*i.e.* classifying one of all classes) to decompose the model into modules, thus facilitating model development and improvement through module reuse.

9 CONCLUSION

In this work, we explore how a trained CNN model can be decomposed into a set of smaller and reusable modules. The resulting modules can be reused to construct completely different CNN models or more accurate CNN models without costly training from scratch. We propose two compressed modularization approaches including CNNSplitter and GradSplitter to address the modularity issue of CNN models with a genetic algorithm and gradient-based optimization, respectively. We also propose a module evaluation method to guide module reuse, thus providing a new solution for developing accurate CNN models. We have evaluated CNNSplitter and GradSplitter with three representative CNN models on three widely-used datasets. The experimental results confirm the effectiveness of our approaches.

In the future, we will explore search-based modularization on more neural networks, such as Transformers at the granularity of attention heads, and further study the modularization and module reuse on large language models such as GPT. Additionally, hierarchical modularization, such as decomposing models based on the semantic

Just Accepted

hierarchy of ImageNet, where a module could be further decomposed into more granular modules, holds potential. This approach could enhance the flexibility of module reuse and the comprehension of model properties.

Our source code and experimental data are available at **https://github.com/qibinhang/CNNSplitter** and **https://github.com/qibinhang/GradSplitter**.

ACKNOWLEDGEMENT

This work was supported partly by National Natural Science Foundation of China under Grant Nos.(61972013, 61932007) and partly by Guangxi Collaborative Innovation Center of Multi-source Information Integration and Intelligent Processing.

REFERENCES

[1] 2022. CNNSplitter. https://github.com/qibinhang/CNNSplitter
[2] 2023. GradSplitter. https://github.com/qibinhang/GradSplitter
[3] Amjad Almahairi, Nicolas Ballas, Tim Cooijmans, Yin Zheng, Hugo Larochelle, and Aaron Courville. 2016. Dynamic capacity networks. In *International Conference on Machine Learning*. PMLR, 2549–2558.
[4] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. 2013. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation. *CoRR* abs/1308.3432 (2013).
[5] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. 2016. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *International Conference on Neural Information Processing Systems*. 2180–2188.
[6] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).
[7] Cyril W Cleverdon. 1972. On the inverse relationship of recall and precision. *Journal of documentation* (1972).
[8] Leon Derczynski. 2016. Complementarity, F-score, and NLP Evaluation. In *International Conference on Language Resources and Evaluation*. 261–266.
[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*. Association for Computational Linguistics, 4171–4186.
[10] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. 2014. DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition. In *31st International Conference on Machine Learning*, Vol. 32. 647–655.
[11] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. 2022. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*. PMLR, 5547–5569.
[12] Thomas Elsken, Jan Hendrik Metzen, Frank Hutter, et al. 2019. Neural architecture search: A survey. *J. Mach. Learn. Res.* 20, 55 (2019), 1–21.
[13] Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. 2020. Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks. In *International Symposium on Software Testing and Analysis*. 177–188.
[14] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *International Conference on Learning Representations*. OpenReview.net.
[15] Mengya Gao, Yujun Wang, and Liang Wan. 2021. Residual error based knowledge distillation. *Neurocomputing* 433 (2021), 154–161.
[16] Xiang Gao, Ripon K. Saha, Mukul R. Prasad, and Abhik Roychoudhury. 2020. Fuzz Testing Based Data Augmentation to Improve Robustness of Deep Neural Networks. In *The ACM/IEEE 42nd International Conference on Software Engineering*. 1147–1158.
[17] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*. 580–587.
[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.
[19] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems* 28 (2015).
[20] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and Software Technology* 43, 14 (2001), 833–839.
[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.
[22] Christopher R Houck, Jeff Joines, and Michael G Kay. 1995. A genetic algorithm for function optimization: a Matlab implementation. *Ncsu-ie tr* 95, 09 (1995), 1–10.

Just Accepted

[23] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. *Advances in neural information processing systems* 29 (2016).
[24] Sayem Mohammad Imtiaz, Fraol Batole, Astha Singh, Rangeet Pan, Breno Dantas Cruz, and Hridesh Rajan. 2023. Decomposing a Recurrent Neural Network into Modules for Enabling Reusability and Replacement. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1020–1032.
[25] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*. 448–456.
[26] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).
[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems* 25 (2012), 1097–1105.
[28] Anders Krogh and John A Hertz. 1992. A simple weight decay can improve generalization. In *Advances in neural information processing systems*. 950–957.
[29] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of IEEE* 86, 11 (1998), 2278–2324.
[30] Yann LeCun, John S Denker, and Sara A Solla. 1990. Optimal brain damage. In *Advances in neural information processing systems*. 598–605.
[31] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. {GS}hard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *International Conference on Learning Representations*.
[32] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning Filters for Efficient ConvNets. In *International Conference on Learning Representations*.
[33] Lingbo Li, Mark Harman, Fan Wu, and Yuanyuan Zhang. 2016. The value of exact analysis in requirements selection. *IEEE Transactions on Software Engineering* 43, 6 (2016), 580–596.
[34] Tao Lin, Lingjing Kong, Sebastian U Stich, and Martin Jaggi. 2020. Ensemble distillation for robust model fusion in federated learning. *Advances in Neural Information Processing Systems* 33 (2020), 2351–2363.
[35] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436* (2017).
[36] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully convolutional networks for semantic segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*. 3431–3440.
[37] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. 2010. Bug localization using latent dirichlet allocation. *Information and Software Technology* 52, 9 (2010), 972–990.
[38] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: automated neural network model debugging via state differential analysis and input selection. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 175–186.
[39] Linghan Meng, Yanhui Li, Lin Chen, Zhi Wang, Di Wu, Yuming Zhou, and Baowen Xu. 2021. Measuring Discrimination to Boost Comparative Testing for Multiple Deep Learning Models. In *43rd International Conference on Software Engineering*. IEEE, 385–396.
[40] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2017. Pruning Convolutional Neural Networks for Resource Efficient Inference. In *International Conference on Learning Representations*. OpenReview.net.
[41] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. 2011. Reading digits in natural images with unsupervised feature learning. (2011).
[42] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In *26th International Conference on Automated Software Engineering*. IEEE, 263–272.
[43] Rangeet Pan and Hridesh Rajan. 2020. On decomposing a deep neural network into modules. In *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 889–900.
[44] Rangeet Pan and Hridesh Rajan. 2022. Decomposing Convolutional Neural Networks into Reusable and Replaceable Modules. In *In 44th International Conference on Software Engineering*. 524–535.
[45] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2009), 1345–1359.
[46] David Lorge Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (1972), 1053–1058.
[47] David Lorge Parnas. 1976. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering* 1 (1976), 1–9. https://doi.org/10.1109/TSE.1976.233797
[48] David Lorge Parnas, Paul C Clements, and David M Weiss. 1983. Enhancing reusability with information hiding. In *Proceedings of the ITT Workshop on Reusability in Programming*. 7–9.

Just Accepted

[49] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *26th Symposium on Operating Systems Principles*. 1–18.
[50] Jonathan K Pritchard, Matthew Stephens, and Peter Donnelly. 2000. Inference of population structure using multilocus genotype data. *Genetics* 155, 2 (2000), 945–959.
[51] Binhang Qi, Hailong Sun, Xiang Gao, and Hongyu Zhang. 2022. Patching Weak Convolutional Neural Network Models through Modularization and Composition. *37th International Conference on Automated Software Engineering* (2022).
[52] Binhang Qi, Hailong Sun, Xiang Gao, Hongyu Zhang, Zhaotian Li, and Xudong Liu. 2023. Reusing Deep Neural Network Models through Model Re-engineering. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 983–994.
[53] Binhang Qi, Hailong Sun, Hongyu Zhang, Ruobing Zhao, and Xiang Gao. 2023. Modularizing while Training: A New Paradigm for Modularizing DNN Models. *46th International Conference on Software Engineering* (2023).
[54] rangeetpan. 2020. *Decompose a DNN model into Modules.* https://github.com/rangeetpan/decomposeDNNintoModules
[55] rangeetpan. 2022. *Decompose a CNN model into Modules.* https://github.com/rangeetpan/Decomposition
[56] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *AAAI Conference on Artificial Intelligence*. 4780–4789.
[57] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. 2017. Large-scale evolution of image classifiers. In *International Conference on Machine Learning*. PMLR, 2902–2911.
[58] Colin R Reeves. 1995. A genetic algorithm for flowshop sequencing. *Computers & Operations Research* 22, 1 (1995), 5–13.
[59] Jonathan S Rosenfeld, Jonathan Frankle, Michael Carbin, and Nir Shavit. 2021. On the predictability of pruning across scales. In *International Conference on Machine Learning*. 9075–9083.
[60] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition*. 4510–4520.
[61] Noam Shazeer, *Azalia Mirhoseini, *Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *International Conference on Learning Representations*.
[62] Jieke Shi, Zhou Yang, Bowen Xu, Hong Jin Kang, and David Lo. 2022. Compressing Pre-trained Models of Code into 3 MB. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[63] Connor Shorten and Taghi M Khoshgoftaar. 2019. A survey on image data augmentation for deep learning. *Journal of Big Data* 6, 1 (2019), 1–48.
[64] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*.
[65] sksq96. 2021. *pytorch-summary.* https://github.com/sksq96/pytorch-summary
[66] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.
[67] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. 2021. Improving Test Case Generation for REST APIs Through Hierarchical Clustering. In *The IEEE/ACM 36th International Conference on Automated Software Engineering*. 117–128.
[68] Masanori Suganuma, Mete Ozay, and Takayuki Okatani. 2018. Exploiting the potential of standard convolutional autoencoders for image restoration by evolutionary search. In *International Conference on Machine Learning*. 4771–4780.
[69] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. 2013. On the importance of initialization and momentum in deep learning. In *International Conference on Machine Learning*. 1139–1147.
[70] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.
[71] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *IEEE conference on computer vision and pattern recognition*. 2818–2826.
[72] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *International conference on software engineering*. 303–314.
[73] tokusumi. 2020. *keras-flops.* https://github.com/tokusumi/keras-flops
[74] Zan Wang, Hanmo You, Junjie Chen, Yingyi Zhang, Xuyuan Dong, and Wenbin Zhang. 2021. Prioritizing Test Inputs for Deep Neural Networks via Mutation Analysis. In *43rd International Conference on Software Engineering*. IEEE, 397–409.
[75] Xi-Zhu Wu, Wenkai Xu, Song Liu, and Zhi-Hua Zhou. 2021. Model reuse with reduced kernel mean embedding specification. *IEEE Transactions on Knowledge and Data Engineering* (2021).
[76] Lingxi Xie and Alan Yuille. 2017. Genetic cnn. In *IEEE International Conference on Computer Vision*. 1379–1388.
[77] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *International Symposium on Software Testing and Analysis*. 146–157.

Just Accepted

[78] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. 2018. Convolutional neural networks: an overview and application in radiology. *Insights into imaging* 9, 4 (2018), 611–629.
[79] Qinghao Ye, Haiyang Xu, Guohai Xu, Jiabo Ye, Ming Yan, Yiyang Zhou, Junyang Wang, Anwen Hu, Pengcheng Shi, Yaya Shi, et al. 2023. mplug-owl: Modularization empowers large language models with multimodality. *arXiv preprint arXiv:2304.14178* (2023).
[80] Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley J. Osher, Yingyong Qi, and Jack Xin. 2019. Understanding Straight-Through Estimator in Training Activation Quantized Neural Nets. In *International Conference on Learning Representations*.
[81] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How Transferable Are Features in Deep Neural Networks?. In *27th International Conference on Neural Information Processing Systems - Volume 2*. 3320–3328.
[82] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide Residual Networks. In *BMVC*. BMVA Press.
[83] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *International Conference on Automated Software Engineering*. IEEE, 132–142.
[84] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. 2018. Practical Block-Wise Neural Network Architecture Generation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2423–2432. https://doi.org/10.1109/CVPR.2018.00257
[85] Michael Zhu and Suyog Gupta. 2018. To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression. In *International Conference on Learning Representations*. OpenReview.net.

Just Accepted