Reusing Deep Neural Network Models through Model Re-engineering

Binhang Qi*
SKLSDE Lab, Beihang University
Beijing, China
binhangqi@buaa.edu.cn

Hailong Sun*†
SKLSDE Lab, Beihang University
Beijing, China
sunhl@buaa.edu.cn

Xiang Gao*†
SKLSDE Lab, Beihang University
Beijing, China
xiang_gao@buaa.edu.cn

Hongyu Zhang
Chongqing University
Chongqing, China
hyzhang@cqu.edu.cn

Zhaotian Li*
SKLSDE Lab, Beihang University
Beijing, China
lizhaotian@buaa.edu.cn

Xudong Liu*
SKLSDE Lab, Beihang University
Beijing, China
liuxd@act.buaa.edu.cn

Abstract—Training deep neural network (DNN) models, which has become an important task in today's software development, is often costly in terms of computational resources and time. With the inspiration of software reuse, building DNN models through reusing existing ones has gained increasing attention recently. Prior approaches to DNN model reuse have two main limitations: 1) reusing the entire model, while only a small part of the model's functionalities (labels) are required, would cause much overhead (e.g., computational and time costs for inference), and 2) model reuse would inherit the defects and weaknesses of the reused model, and hence put the new system under threats of security attack. To solve the above problem, we propose SEAM, a tool that re-engineers a trained DNN model to improve its reusability. Specifically, given a target problem and a trained model, SEAM utilizes a gradient-based search method to search for the model's weights that are relevant to the target problem. The re-engineered model that only retains the relevant weights is then reused to solve the target problem. Evaluation results on widely-used models show that the re-engineered models produced by SEAM only contain 10.11% weights of the original models, resulting 42.41% reduction in terms of inference time. For the target problem, the re-engineered models even outperform the original models in classification accuracy by 5.85%. Moreover, reusing the re-engineered models inherits an average of 57% fewer defects than reusing the entire model. We believe our approach to reducing reuse overhead and defect inheritance is one important step forward for practical model reuse.

Index Terms—model reuse, deep neural network, re-engineering, DNN modularization

I. INTRODUCTION

Software reuse is the process of using existing software artifacts that would be otherwise created from scratch [1]–[3], which is widely deemed essential to improve software quality and development productivity. Instances of software reuse include the reuse of software libraries, components, APIs, etc. As today's software systems are increasingly incorporating AI techniques (e.g., deep learning), training DNN models has become an important task in the software development lifecycle. However, training DNN models is often known to be very costly, especially for models with billions of parameters and large datasets. To solve this problem, with the inspiration of software reuse, the software engineering community is paying more attention to DNN model reuse [4]–[10].

A trained model can be directly reused if it fits the target problem domain. However, reusing entire trained models may cause large overhead (e.g., inference time). Just like traditional software libraries which implement a large number of functions, a trained model may also have multiple functionalities (e.g., classification for multiple categories). When reusing a trained model, often only part of functionalities are required to solve the target problem. For instance, Google Vision API provides the service of multi-class classification with around 20,000 classes, but not all classes are necessary in practical scenarios. Suppose that a developer needs to build a fire alarm application [11] for determining whether a given image indicates "fire". Although only two classes ("fire" and "non-fire") are needed, if the developer directly invokes Google Vision API, all the 20,000 classes will be involved, which can incur much inference overhead caused by the unnecessary weights/neurons in the underlying DNN model.

A model trained to solve a similar problem can also be indirectly reused via transfer learning [12], [13]. Transfer learning consists of taking relevant features learned on a similar problem and optionally fine-tuning the trained model using the dataset of the target problem. Although effective in classification accuracy and training efficiency, reusing trained models may inherit their defects [14]–[16]. It has been shown that AI models are notoriously brittle to small perturbations on input data [15], [17], which allows attackers to craft adversarial examples for malicious attacks. When reusing a model, the weakness of a trained model can be inherited, and hence putting the system under the threats of adversarial attacks.

To address the weaknesses of existing model reuse methods, one idea is to only reuse some parts of a trained model (e.g., by eliminating some weights or neurons) that are relevant to the target problem, as the weaknesses correlate with the

---

* Also with Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, Beijing 100191, China.
† Corresponding authors: Hailong Sun and Xiang Gao.

weights of a trained model [7], [18]. Identifying the relevant weights/neurons can be achieved with the fundamental concept of *re-engineering* in software engineering [19], [20], which aims to improve software maintainability and reusability by enhancing or altering existing software. Borrowing the idea of software re-engineering, we propose *model re-engineering* for DNN models, which searches for the target problem-related weights with the guidance of target problem-related metrics (e.g., classification accuracy) and removes irrelevant weights from an *original model* (i.e., trained model), resulting in a re-engineered model. When solving a certain problem through direct or indirect reuse, the re-engineered model, which retains only relevant weights to certain functionalities (e.g., a part of classes in classification), is reused, hence reducing the reuse overhead and mitigating the defect inheritance.

Existing work, including model modularization [8]–[10] and model slicing [7], has preliminarily explored the idea of reusing part of trained models based on neuron activation and neuron coverage [14], [15]. For instance, relying on neuron coverage, model slicing [7] first computes the relevance between weights and the target problem, then deletes the irrelevant weights. Unfortunately, due to the lack of interpretability of DNN models, the effectiveness of using neuron coverage is still questionable [21], [22]. The neuron coverage-based work [7]–[9] is not accurate enough in identifying relevant weights and hence prefers to be conservative in removing weights, i.e., only a small number of weights are removed to avoid removing relevant weights. Therefore, the models obtained with the existing approaches [7]–[9] will retain lots of irrelevant weights or neurons, having the limitations of reuse overhead and defect inheritance. CNNSplitter [10] introduces the first search-based approach for modularizing CNNs. As CNNSplitter achieves modularization by searching for relevant convolution kernels with a genetic algorithm, this approach cannot be applied directly to other neural networks, such as the fully connected neural networks.

In this paper, we propose SEAM, a **Sea**rch-based **M**odel re-engineering approach that can accurately identify relevant weights and hence removes as many irrelevant weights as possible. Different from the neuron coverage-based approaches [7]–[9], SEAM is directly guided by the target problem-related metrics, e.g., classification accuracy, to search for the relevant weights. Moreover, SEAM applies a gradient-based search method to identify relevant weights, which is more general and efficient than CNNSplitter [10]. Specifically, SEAM consists of three components: search space, performance estimation strategy, and search strategy. The search space consists of the masks of all candidate re-engineered models. The mask of a candidate records which weights of the original model should be retained or removed. The performance estimation strategy defines the objective function of the search as the weighted sum of the candidate's weight retention rate and its cross-entropy loss on the target problem's dataset (denoted as target dataset). The objective function is used to evaluate a candidate's performance, and the objective function value is sent to the search strategy to guide the
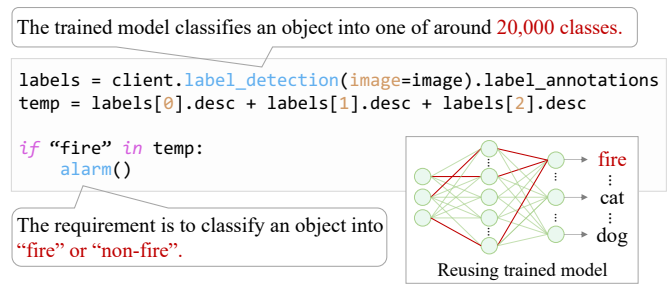

Fig. 1. An example of direct model reuse.

next round of search. The search strategy applies a gradient-based search method to explore the search space efficiently. In each search round, the search strategy finds a candidate with better performance by minimizing the objective function value. SEAM performs the search and estimation processes iteratively, and stops when the objective function value converges. The candidate with the minimal objective function value will be regarded as the resultant re-engineered model. The re-engineered model can be reused directly, or indirectly via fine-tuning, which helps reduce reuse overhead and lower the risk of defect inheritance while achieving comparable performance (e.g., classification accuracy) to the original model.

We evaluate SEAM using four representative CNN models on eight widely-used datasets. The experimental results first demonstrate that SEAM can accurately identify relevant weights and thus remove a large number of irrelevant weights. On average, a re-engineered model contains 89.89% fewer weights than the original model, and outperforms the original model by 5.85% in classification accuracy. Moreover, reusing a re-engineered model incurs less reuse overhead than reusing an original model, e.g., the average reduction in time cost for inference is 42.41%. Regarding defect inheritance, reusing the re-engineered model inherits an average of 57% fewer defects than reusing the original model.

The main contributions of this work are as follows:

- We propose the notion of *model re-engineering*, which re-engineers a trained deep learning model to improve its reusability.
- We propose a search-based model re-engineering approach named SEAM, which can accurately identify the weights relevant to a target problem and hence allows the re-engineered model to retain as few irrelevant weights as possible. SEAM can reduce the reuse overhead and lower the risk of defect inheritance in model reuse.
- We conduct extensive experiments using four representative CNN models on eight widely-used datasets. The results show that SEAM can remove a large number of irrelevant weights from the original models. Also, the experiments demonstrate the effectiveness of SEAM in overcoming the limitations of existing approaches.

II. Motivating Examples

Reusing a re-engineered model containing fewer irrelevant weights rather than an original model has several benefits. In this section, we introduce the applications and benefits of model re-engineering with two examples.

A. Reducing reuse overhead in direct reuse

When a trained model satisfies the requirement of a target problem, a common way of reuse is to reuse the entire trained model on the target problem directly. However, there may be redundancy in the functionalities provided by the trained model [8], [23]. Redundancy in a trained model's functionality implies redundant weights, which may incur significant *reuse overhead*, including computational and time costs for inference, that is unnecessary for the target problem.

As shown in Figure 1, a simple fire alarm application [11] is used to illustrate the problem. In this example, the developer reuses a trained model (by calling the Google `label_detection` API) to classify an input image. An alarm will be triggered if the top-3 classification labels returned by the trained model include the keyword "fire". The requirement of the target problem is to classify an image into "fire" or "non-fire", while the reused trained model classifies an image into one of around 20,000 classes. As different weights could recognize features of different classes [24], [25], only a few relevant weights recognize the features of "fire". However, when reusing the trained model for inference, a lot of irrelevant weights are loaded into memory and involved in computation to produce intermediate results, incurring memory, computational, and time costs.

The example demonstrates that the requirement of a target problem may be only a small part of a trained model's functionality. Model re-engineering can remove part of the original model's weights irrelevant to the target problem and allows developers to reuse only the relevant weights. In this example, the weights irrelevant to the target problem are removed, resulting in a re-engineered model that only classifies "fire" and "non-fire". Compared to directly reusing the trained model, reusing the re-engineered model containing fewer weights could reduce the reuse overhead.

B. Mitigating defect inheritance in transfer learning

When a trained model cannot satisfy the requirement of a target problem, a common form of reuse is to transfer learning [26]–[28]. That is, a developer reuses a trained model and fine-tunes the trained model on the target dataset to build a fine-tuned model that satisfies the requirement. This form of reuse is widely-used and effective; however, it faces the problem of defect inheritance [7], [16], [29], [30]. An example shown in Figure 2 is used to illustrate the defect inheritance and potential attacks to face. In this example, a public model trained on ImageNet [31] can perform classification with 1000 classes (including 59 bird classes [32]). To build a model for classifying birds with 200 classes, a developer reuses the trained model and fine-tunes the trained model on the target dataset Caltech-UCSD Birds [33]. During fine-tuning, most of
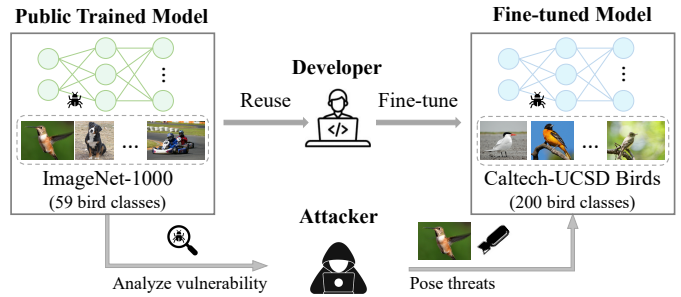

Fig. 2. Fine-tuning a publicly available trained model. Inherited defects could be exploited by attackers.

the weights in the pre-trained model are retained in the fine-tuned model. The adversarial examples that can fool the public trained model are still likely to be able to fool the fine-tuned model, which is called defect inheritance [7], [16], [29], [30].

The major reason for defect inheritance is indiscriminate reuse [7], [18]. Specifically, in conventional transfer learning, all the trained model's weights are reused, including both the relevant and the irrelevant ones to the target problem. As the target dataset is usually not very large, fine-tuning will not have much effect on changing the weights irrelevant to the target problem. As a result, the defects are mostly inherited in the fine-tuned model [7], [34].

Model re-engineering alters the original model by removing irrelevant weights, thus avoiding the inheritance of defects associated with these weights when the re-engineered model is reused. In this example, a re-engineered model retains only the weights relevant to the features of "bird". As a result, compared to reusing the original model, reusing the re-engineered model can reduce the defect inheritance while achieving comparable accuracy.

III. Our Approach

In this section, we introduce SEAM, a search-based approach to model re-engineering, which uses a gradient-based search method to find the target problem-related weights.

A. Overview

As illustrated in Figure 3, the workflow of SEAM consists of three components: *search space*, *performance estimation strategy*, and *search strategy*. Given an original model (a 3-class classification in Figure 3), which consists of three neural network layers with fifteen weights, and a target dataset (binary classification in Figure 3), the model re-engineering process is summarized as follows:

(1) *Construction of Search Space*: A re-engineered model selectively removes part of the original model's weights according to a *mask*. A *mask* is a bit vector $[0, 1]^L$, where L is the number of weights in the original model, and each bit represents whether the corresponding weight is removed. In total, there are 2^L candidate masks, each of which corresponds to a candidate re-engineered model. Consequently, the search space consists of 2^L candidates. The mask is initialized
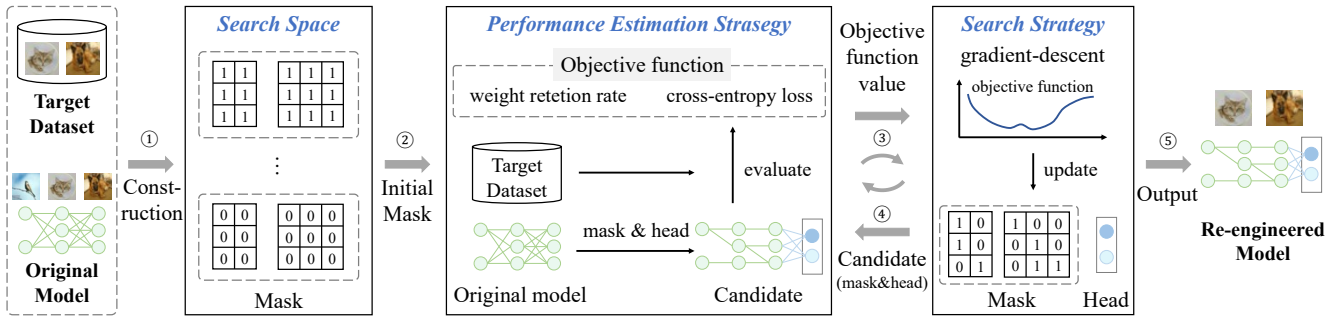
Fig. 3. The workflow of model re-engineering with SEAM.

with all element values as 1, representing that all weights are retained initially. The first and second steps along with the component Search Space in Figure 3 display the above process, where $L = 15$ and the search space size is 2^{15}.

(2) *Performance Estimation*: Given a candidate mask, the performance estimation strategy first constructs a candidate re-engineered model by removing weights according to the mask and appending a *head* as the output layer. The *head*, which is a fully connected layer, is used to enable the candidate to adapt to the target problem, i.e., adapt the original *N*-classification model to the target *K*-classification problem. Then, the objective function is defined as the weighted sum of the *weight retention rate* of the candidate and the *cross-entropy loss* between the candidate's predictions and corresponding actual labels on the *target dataset*. The objective function is used to evaluate the performance of a candidate. The resulting objective function value will be fed back to the searching process to guide the next search round. The third step along with the component Performance Estimation Strategy in Figure 3 display the estimation process.

(3) *Searching Candidates*: The search strategy applies a gradient-based search method to explore the search space with the guidance of the objective function. The gradient-based search method not only efficiently explores the huge search space, but also optimizes the head at the same time. In each search round, the search strategy sends the updated mask and head as a new candidate to the performance estimation strategy. The fourth step along with the component Search Strategy in Figure 3 display the search process, where the head has two neurons as the target problem is binary classification.

SEAM iterates the search and estimation processes. When the objective function value converges, SEAM outputs the re-engineered model. In the example shown in Figure 3, the re-engineered model retains 7 out of 15 weights of the original model and performs binary classification. We present the technical details of each step in the following.

B. Construction of Search Space

The goal of model re-engineering is to obtain a new model which retains only the target problem-related weights of the original model. Model re-engineering is formulated as a problem of searching for a new model from all candidate models,
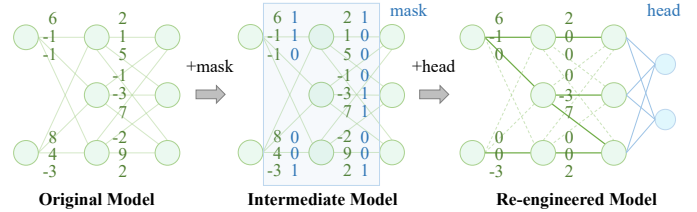

Fig. 4. The construction of a re-engineered model using the mask and head.

which selectively removes part of the original model's weights. If the searched model retains only the target problem-related weights, it is regarded as the re-engineered model. In this problem, the search space consists of all possible candidate re-engineered models. To facilitate a technical solution to this problem in practice, a *mask* that records which weights are removed and retained in a candidate is used to represent a candidate, thereby omitting unnecessary details of a candidate, such as Max-pooling and Dropout layers. Consequently, in SEAM, the search space consists of all candidate masks.

Specifically, a mask is a bit vector $[0, 1]^L$, where L is the number of weights in the original model, and 0 (or 1) represents the corresponding weight removed (or retained). Figure 4 illustrates the use of a mask to remove weights from the original model. By multiplying the weights of the trained model with the mask, SEAM sets the values of irrelevant weights to zero and keeps the values of relevant weights. The weights with values set to zero are involved in the computation but have no effect on the prediction, thus achieving the effect of removing irrelevant weights. Note that, after model re-engineering, the computation of a re-engineered model involving the weights with zero values could be eliminated by special libraries (e.g., DeepSparse [35]), which will be discussed in Section IV-B.

After the construction of search space, a mask initialized to all element values of 1 is fed to the performance estimation strategy. That is, the starting point of the search is a candidate that retains all the original model's weights.

C. Performance Estimation

The search aims to find the optimal mask, which corresponds to a candidate re-engineered model that retains only

the target problem-related weights and can classify well on the target problem. To achieve the goal, the performance estimation strategy defines the objective function of the search as the weighted sum of *weight retention rate* and *cross-entropy loss*. The weight retention rate can measure the number of weights retained by the candidate. The cross-entropy loss on the target dataset can measure the classification performance of the candidate on the target problem.

Specifically, when evaluating a candidate's performance, SEAM first constructs the candidate, as the computation of cross-entropy loss requires running the candidate on the target dataset. Figure 4 illustrates the construction of a candidate re-engineered model. SEAM first multiplies the weights of the original model with the mask to remove part of the original model's weights, resulting in an intermediate model. As the output layer has three neurons, the intermediate model is still a model for 3-class classification. To adapt the candidate to the number of classes of the target problem, the head, a fully connected layer, is appended after the intermediate model as the output layer of the candidate. The head is randomly initialized in the first search round and will be updated along with the mask in the subsequent rounds. In this example, the head has two neurons, which transforms the 3-class prediction of the intermediate model to the binary prediction, allowing the candidate to adapt to the target problem.

After constructing the candidate, the cross-entropy loss $\mathcal{L}_{ce}$ between the candidate's predictions on the target dataset and the actual labels is computed as follows:

$$\mathcal{L}_{ce} = - \sum_{i=1}^{K} t_i \log(P_i(\mathcal{M}, \mathcal{H})), \qquad (1)$$

where K is the number of classes in the target problem, $\mathcal{M}$ and $\mathcal{H}$ are the mask and head, $P_i(\mathcal{M}, \mathcal{H})$ is the prediction for the i-th class by a candidate constructed with $\mathcal{M}$ and $\mathcal{H}$, and t_i is the probability of the i-th class in the one-hot representation of the actual label, with a value of 0 or 1. A lower cross-entropy loss indicates that the candidate retains more target problem-related weights and hence achieves higher classification accuracy on the target dataset.

The weight retention rate $\mathcal{L}_{wr}$ is computed directly from the mask:

$$\mathcal{L}_{wr} = \frac{1}{L} \sum_{i=1}^{L} \mathcal{M}[i], \qquad (2)$$

where L is the number of weights in the original model. A lower weight retention rate indicates that the candidate retains fewer weights. Based on $\mathcal{L}_{ce}$ and $\mathcal{L}_{wr}$, the objective function $\mathcal{O}$ is defined as follows:

$$\mathcal{O} = \mathcal{L}_{ce} + \alpha \times \mathcal{L}_{wr}, \qquad (3)$$

where α is a weighting factor and is empirically set to 1.0. To minimize $\mathcal{O}$, SEAM tends to search for a candidate that retains only the target problem-related weights, as this candidate can achieve the highest classification accuracy while retaining as few weights as possible.

D. Searching Candidates

Large models can have billions of parameters, resulting in super huge search space. To explore the huge search space efficiently, our search strategy applies a gradient-based search method. In each search round, the search strategy finds a new candidate with a smaller objective function value by gradient descent based on the objective function value of the candidate in the previous round. That is, the mask is updated by descending the gradient as follows:

$$\mathcal{M}' = \mathcal{M} - \xi \times \nabla_{\mathcal{M},\mathcal{H}} \mathcal{O}, \qquad (4)$$

$$\nabla_{\mathcal{M},\mathcal{H}} \mathcal{O} = \nabla_{\mathcal{M},\mathcal{H}} \mathcal{L}_{ce} + \alpha \times \nabla_{\mathcal{M}} \mathcal{L}_{wr}, \qquad (5)$$

where ξ is the learning rate, and $\mathcal{M}'$ is the updated mask corresponding to a new candidate with a smaller objective function value.

When applying gradient descent to update a mask, it is important to note that gradient descent requires the search space to be continuous and differentiable [36], while the search space composed of masks is discrete and non-differentiable. Inspired by DARTS [36], the search strategy attaches a continuous number to each element of the mask, which can be considered as the relevance of the weight to the target problem. Then an indicator function $\mathbb{1}_{(0,+\infty)}: X \rightarrow \{0,1\}$ is used to set the element values corresponding to the weights with relevance greater than zero to 1 and the other element values to 0. As the relevance is continuous, the search strategy uses gradient descent to update the relevance and thus can update the mask.

After satisfying the condition that the search space is continuous, another problem is that the indicator function is non-differentiable at $x=0$, and the derivative of the indicator function equals 0 everywhere except at $x=0$. This problem prevents the common backward propagation based on gradient descent from directly applying to update relevance [37], [38]. To address the problem, the technique named straight-through estimator [38], [39] is used to estimate the gradient of the indicator function, which directly uses the gradient of the previous neural network layer as the gradient of the current neural network layer.

The head is updated along with the mask by descending the gradient $\nabla_{\mathcal{M},\mathcal{H}} \mathcal{L}_{ce}$. After updating the mask and head, the search strategy sends them as a new candidate to the performance estimation strategy and starts the next round of search after getting the objective function value.

IV. EXPERIMENTS

To evaluate the effectiveness of SEAM, in this section, we introduce the benchmarks and experimental setup as well as the experimental results. Specifically, we evaluate SEAM by answering the following research questions:

- RQ1: How effective is our model re-engineering approach in reusing trained models?
- RQ2: Does reusing a re-engineered model incur less overhead than reusing the original model?
- RQ3: Does reusing the re-engineered model mitigate the defect inheritance?

A. Experimental Setup

RQ1: How effective is our model re-engineering approach in reusing trained models? Three representative CNN models are used in this research question, including VGG16 [40], ResNet20, and ResNet50 [41]. The three CNN models are trained on three public classification datasets, including CIFAR-10 [42], CIFAR-100 [42], and ImageNet [31]. In total, there are five trained CNN models in this experiment, including VGG16-CIFAR10, VGG16-CIFAR100, ResNet20-CIFAR10, ResNet20-CIFAR100, and ResNet50-ImageNet. Among these trained models, the first four models are publicly available from the third-party GitHub repositories [43], and the last model is provided by PyTorch [44].

Given a trained model for N-class classification, we perform model re-engineering to alter the trained model on two types of target problems, including binary and multi-class classification problems. For the binary classification problem, each class of the trained model corresponds to a target problem. In total, there are N target problems. A re-engineered model needs to classify whether an input belongs to the corresponding class or not. In this scenario, VGG16-CIFAR10, VGG16-CIFAR100, ResNet20-CIFAR10, and ResNet20-CIFAR100 are altered, and there are 220 re-engineered models in total. Due to the significant overhead of generating 1000 re-engineered models, ResNet50-ImageNet is not used here. We count the number of removed weights and compare the accuracy of re-engineered models and trained models on target problems to validate the effectiveness of SEAM. Also, we compare SEAM with the state-of-the-art modularization approach [8] to demonstrate the improvement achieved by our approach.

For the multi-class classification problem, a re-engineered model classifies an input into one of the concerning classes. In this scenario, we use CIFAR-100 and ImageNet as our datasets since there are publicly available schemes for dividing them into superclasses [42], [45]. A small-size model ResNet20-CIFAR100 and a large model ResNet50-ImageNet are chosen for a more comprehensive evaluation. Specifically, CIFAR-100 has divided the 100 classes into 20 superclasses, each containing 5 classes with semantically similar labels [42]. For ResNet20-CIFAR100, we follow this division; thus, there are 20 target problems, each corresponding to a superclass. For ResNet50-ImageNet, following the public division [45], the 1000 classes are divided into 67 superclasses, of which 3 superclasses are discarded because they contain only 1 class. The remaining 64 superclasses with a number of classes ranging from 2 to 119 form 64 target problems. In total, there are 84 re-engineered models. We count the number of removed weights and compare the accuracy of re-engineered models and trained models on target problems to validate the effectiveness of SEAM. Since the modularization approaches [8], [9] are designed for binary classification (i.e., each module performs binary classification) and cannot be applied to multi-class classification directly, we compare SEAM with the method of retraining from scratch.

When re-engineering an original model on a target problem,

we follow the settings of our baselines [7], [43] to divide the target dataset into training and testing sets. The training set is used to search for a candidate, and the testing set is used to evaluate the candidate. The major parameters in SEAM include weighting factor α (see Equation 3) and learning rate ξ (see Equation 4). The appropriate values of α and ξ could vary from different trained models and are generally set to 1.0 and 0.05, respectively. The detailed settings and their impact on model re-engineering are described in the project webpage [46].

RQ2: Does reusing a re-engineered model incur less overhead than reusing the original model? In this experiment, the trained models and re-engineered models from RQ1 are reused, and we compare the reuse overhead of re-engineered models with the original models. Two metrics are used to measure the reuse overhead, including the number of floating point operations (FLOPs) [47], [48] and time cost for inference. An open-source tool fvcore [49] is used to calculate the FLOP. Regarding inference time cost, an open-source tool DeepSparse [35] is used to run both original and re-engineered models and compute the inference time cost.

RQ3: Does reusing the re-engineered model mitigate the defect inheritance? In transfer learning, a pre-trained model generally has a large number of weights and classifications, and a target problem has insufficient data. Therefore, VGG16 and ResNet20 are not suitable to be transferred, and CIFAR-10 and CIFAR-100 are unsuitable for target problems. Following the state-of-the-art approach ReMos [7], two widely-used transfer learning CNN models, ResNet18 and ResNet50, are used as trained models (i.e., the pre-trained models in transfer learning), which are trained on ImageNet and are provided by PyTorch [44]. Five popular transfer learning datasets are used as target datasets, including MIT Indoor Scenes [50], Caltech-UCSD Birds [33], 102 Category Flowers [51], Standford 40 Actions [52], and Standford Dogs [53].

We first apply SEAM to alter the trained model on the target dataset, resulting in a re-engineered model. Then we use the standard fine-tuning approach [26], [27] to fine-tune the re-engineered model on the target dataset, resulting in a fine-tuned model. We compare SEAM with two baselines, standard fine-tuning [26], [27] and the state-of-the-art approach ReMos [7]. Standard fine-tuning fine-tunes all of the trained model's weights on the target dataset. ReMos first sets a trained model's weights irrelevant to the target problem to zeros, and then uses standard fine-tuning to fine-tune the sliced model on the target dataset, resulting in a fine-tuned model. Following the setup of ReMos, we use accuracy (ACC) and defect inheritance rate (DIR) to measure and compare the effectiveness of SEAM and the baselines. The accuracy is computed as the correct classification rate on the target dataset D^T:

$$ACC = \frac{1}{|D^T|} \sum_{(x,y) \in D^T} \mathbb{1}[f(x) = y]. \qquad (6)$$

The defect inheritance rate is computed as the misclassification

rate on a set of malicious inputs S^M:

$$DIR = \frac{1}{|S^M|} \sum_{(\hat{x},y) \in S^M} \mathbb{1}[f(\hat{x}) \neq y]. \quad (7)$$

Same as ReMos [7], open source tool *advertorch* [54] is used to generate S^M based on the trained model and D^T. We use the same parameters as ReMos when using advertorch.

In this experiment, we set the learning rate ξ=0.05 and weighting factor α=0.5. Regarding the standard fine-tuning approach and ReMos, we also use the open source project [55] published by ReMos.

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.

B. Experimental Results

RQ1: How effective is our model re-engineering approach in reusing trained models?
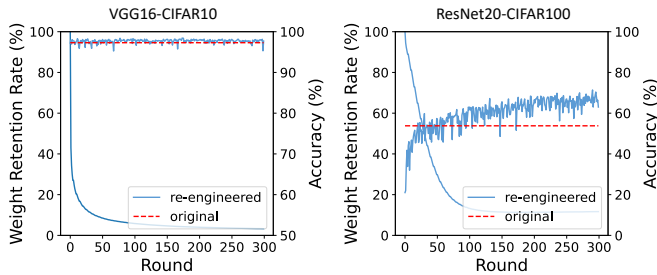

Fig. 5. The convergence process of SEAM on binary (left sub-figure) and multi-class (right sub-figure) classification problems.

In this research question, we present the model re-engineering results of SEAM for two types of target problems (i.e., binary and multi-class classification). Figure 5 shows the convergence process of SEAM on two types of target problems. For instance, the left sub-figure shows the trend of weight retention rate and classification accuracy along with search rounds during re-engineering VGG16-CIFAR10 on a binary classification problem. The weight retention rate descends quickly in the first 50 rounds and then gradually converges. Although many weights are removed, the re-engineered model maintains a comparable accuracy to the original model. The right sub-figure depicts the convergence process of ResNet20-CIFAR100 on a 5-class classification problem. Similar to re-engineering VGG16-CIFAR10, the weight retention rate descends quickly in the first 100 rounds and then gradually converges. The difference is that the accuracy of the re-engineered model may be lower than that of the original model at the beginning of search. The reason is that the target dataset of the 5-class classification problem contains fewer samples than that of the binary classification problem (500 *vs.* 10,000). Thus, the former requires more rounds to optimize the mask and head. As optimization rounds increase, the mask retains more related weights, and the head learns to classify better, so the re-engineered model can recover accuracy and eventually exceed that of the original model. The

Target Problem	Model Name	# Weights (million)		
		Original	Re-engineered	Reduction(%)
Binary Classification	VGG16-CIFAR10	15.25	0.62	95.93
	VGG16-CIFAR100	15.29	1.47	90.39
	ResNet20-CIFAR10	0.27	0.03	88.89
	ResNet20-CIFAR100	0.28	0.03	89.29
Multi-class Classification	ResNet20-CIFAR100	0.28	0.04	85.71
	ResNet50-ImageNet	25.50	2.77	89.16
Average		**-**	**-**	**89.89**

time cost of the search varies by the models, target problems, and target datasets. The sizes of target datasets vary from 500 to 140,000 samples. For the binary classification problem, each round takes several seconds. For the multi-class classification problem, re-engineering ResNet20-CIFAR100 takes 2s per round. For ResNet50-ImageNet, as each superclass contains a different number of classes, the time cost varies from several seconds to a few minutes per round. In this example, re-engineering VGG16-CIFAR10 and ResNet20-CIFAR100 takes 4s and 2s per round, respectively.

Table I shows the results regarding the number of weights for the original and re-engineered models. For each trained model, we count the number of the original model's weights and the number of weights retained (i.e., non-zero weights) in the re-engineered model[1]. For instance, VGG16-CIFAR10 is altered on 10 target problems, resulting in 10 re-engineered models. The average number of weights retained (i.e., non-zero weights) in a re-engineered model is 0.62 million. Compared to the original model having 15.25 million weights, a re-engineered model retains only 4.07% of the original model's weights, which means that SEAM achieves a 95.93% reduction in the number of weights. It is worth mentioning that, for multi-class modularization, although a re-engineered model requires the classification of more classes, it still has much fewer weights than the trained model. For instance, a re-engineered model obtained by altering ResNet20-CIFAR100 can classify five classes; however, the re-engineered model has only an average of 0.04 million weights, and the reduction in the number of weights is 85.71%. The reason is that different classes may contain the same features, which means that the weights needed to identify one more class may already be included in the existing weights. Consequently, for all six trained models, the number of weights retained in re-engineered models is significantly smaller than the number of weights in original models. On average, for the six trained models, SEAM achieves an 89.89% reduction in the number of weights.

Table II shows the averaged accuracy of original and re-engineered models. The original and re-engineered models are

---

[1]As a head contains a negligible number of weights (e.g., 0.43% at most) compared to the original model, the head weight count is omitted in the experiment.

Target Problem	Model Name	Avg. ACC (%)		
		Original	Re-engineered	Increase(%)
Binary Classification	VGG16-CIFAR10	96.50	97.12	0.62
	VGG16-CIFAR100	86.82	92.93	6.12
	ResNet20-CIFAR10	95.64	95.81	0.17
	ResNet20-CIFAR100	83.97	90.92	6.95
Multi-class Classification	ResNet20-CIFAR100	68.29	82.76	14.47
	ResNet50-ImageNet	78.83	85.63	6.80
Average		-	-	**5.85**

TABLE III
THE RESULTS OF MODULARIZATION [8] ON BINARY CLASSIFICATION
PROBLEMS.

Model Name	# Neurons (million)			Avg. ACC (%)		
	Original	Module	Reduction(%)	Original	Module	Increase(%)
ResNet20-CIFAR10	0.20	0.17	14.33	92.85	91.81	-1.04
ResNet20-CIFAR100		0.15	22.84	72.92	59.45	-13.47
Average	-	-	**18.59**	-	-	**-7.25**

TABLE IV
THE MODEL RE-ENGINEERING AND MODEL RETRAINING RESULTS ON
MULTI-CLASS CLASSIFICATION.

Model Name	Avg. ACC (%)		
	Retrained	Re-engineered	Increase (%)
ResNet20-CIFAR100	77.15	82.76	5.61
ResNet50-ImageNet	75.56	85.63	10.08
Average	-	-	**7.84**

evaluated on the corresponding target problems. Again using VGG16-CIFAR10 as an example, the average accuracy of the 10 re-engineered models on the 10 target problems is 97.12%. The original model is also evaluated on the 10 target problems, and the average accuracy is 96.50%. Compared to the original model, the re-engineered models achieve comparable accuracy on target problems, and the averaged accuracy increases by 0.62%. The reason for the improvement may be that model re-engineering enables the re-engineered model to fit the target problem during altering the original model. Note that, the fitting is mainly achieved by removing irrelevant weights instead of training the weights of the original trained model. On both binary and multi-class classification problems, for all the six trained models, re-engineered models can achieve comparable accuracy to original models, and the averaged accuracy increases by 5.85%. Due to space limitation, the detailed results regarding the number of weights and accuracy are available at the project webpage [46].

When comparing SEAM with the existing modularization approach [8], we directly use the open source project [56] published by [8], which decomposes a trained model into modules, each for a binary classification problem. Since tool [8] and SEAM are implemented on Keras and PyTorch, respectively, they cannot directly alter 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 the differences in the underlying computation of PyTorch and Keras. To make the comparison as fair as possible, we run the modules and trained models published by [8] and compare SEAM to [8] based on the results of ResNet20-CIFAR10 and ResNet20-CIFAR100, as the two models are also used in [8]. Specifically, we analyzed the accuracy and the number of neurons of the original models and modules (re-engineered model of SEAM). As modularization [8] decomposes a CNN model mainly by removing neurons (i.e., setting neurons to zero but retaining all weights) from convolutional layers, we analyzed the number of neurons rather than the number of weights.

As shown in Table III, for both ResNet20-CIFAR10 and ResNet20-CIFAR100, a module retains fewer neurons than the original model; however, the number of neurons in a module is reduced by only 18.59% on average. In addition,

a module retains all the weights of the convolutional layers. Regarding accuracy, modules achieve a lower accuracy than the trained models on target problems, and the accuracy of a module reduces by 7.25% on average. Compared to modularization [8], model re-engineering can remove a large number of weights without impairing the accuracy. A major reason for the improvement of SEAM over [8] is that SEAM identifies the target problem-related weights more accurately. SEAM is a search-based approach that identifies the target problem-related weights directly based on the classification accuracy, while [8] identifies the target problem-related weights and neurons based on the neuron activation that indirectly correlates with the accuracy.

We also compare SEAM to model retraining on multi-class classification problems. Model retraining reuses the architecture and hyperparameters of the trained model to retrain a new model from scratch on the target dataset. As both model re-engineering and retraining alter/train the same model (architecture) on the same target problem, while the latter may fit more slowly and even run several times, the time cost of the retraining would be higher than that of re-engineering. Regarding accuracy, as shown in Table IV, re-engineered models outperform retrained models for both ResNet20-CIFAR100 and ResNet50-ImageNet, and the average improvement is 7.84%. The reason for the improvement of model re-engineering may be the difference in the amount of data. The original model is trained on a large-scale dataset, while the retrained model is trained on a small-scale target dataset. The model re-engineering alters the original model to fit the target problem; thus, the re-engineered model achieves higher accuracy than the retrained model.

> On average, a re-engineered model contains 89.89% fewer weights than the original model but outperforms the original model in accuracy by 5.85%.

RQ2: Does reusing a re-engineered model incur less overhead than reusing the original model?

TABLE V
THE NUMBER OF FLOPS REQUIRED BY ORIGINAL AND RE-ENGINEERED
MODELS.

Target Problem	Model Name	FLOPs (million)		
		Original	Re-engineered	Reduction (%)
Binary Classification	VGG16-CIFAR10	314.28	75.53	75.97
	VGG16-CIFAR100	314.33	111.35	64.58
	ResNet20-CIFAR10	41.22	9.34	77.35
	ResNet20-CIFAR100	41.22	9.36	77.30
Multi-class Classification	ResNet20-CIFAR100	41.22	9.76	76.32
	ResNet50-ImageNet	4111.53	955.85	76.75
Average		-	-	74.71

TABLE VI
THE INFERENCE TIME COST REQUIRED BY ORIGINAL AND
RE-ENGINEERED MODELS.

Target Problem	Model Name	Time Cost (ms/batch)		
		Original	Re-engineered	Reduction (%)
Binary Classification	VGG16-CIFAR10	6.82	3.79	44.43
	VGG16-CIFAR100	6.37	5.55	12.87
	ResNet20-CIFAR10	1.40	0.74	47.14
	ResNet20-CIFAR100	1.43	0.64	55.24
Multi-class Classification	ResNet20-CIFAR100	1.45	0.61	57.93
	ResNet50-ImageNet	64.19	40.55	36.83
Average		-	-	42.41

One of the benefits of model re-engineering is to reduce the reuse overhead. As mentioned in Section IV-A, the number of FLOPs and inference time cost are used to measure the reuse overhead. We evaluated the original and re-engineered models from RQ1 on the two metrics to answer this research question.

Table V shows the number of FLOPs required by the original and re-engineered models to classify an image with resolution 32×32, respectively. Note that, following the related work [57], [58], when computing the number of FLOPs required by a re-engineered model with a sparse weight matrix, only the computations involved in non-zero weights are considered. For instance, despite having the same number of weights as the original model, a re-engineered model obtained by altering VGG16-CIFAR10 has 95.93% (see Table I) of its weights set to zero. As the calculations associated with these zero weights can be eliminated by special libraries [59], the calculations associated with these weights are not considered when calculating FLOPs. VGG16-CIFAR10 requires 314.28 million FLOPs, while the average number of FLOPs required by a re-engineered model is 75.53 million. SEAM achieves 75.97% reduction in terms of FLOPs. On average, for the six trained models, SEAM reduces the FLOPs by 74.71%.

To verify that the reduction in the number of FLOPs can reduce the inference time cost, the open-source library DeepSparse [35] is used to deploy and run the original and re-engineered models. Given an input with batch size 16, each re-engineered model or original model classifies the input 200 times, and the average time cost of classification is used to measure the inference time cost of a re-engineered model or an original model. Table VI shows the average inference time cost of each trained model and its corresponding re-
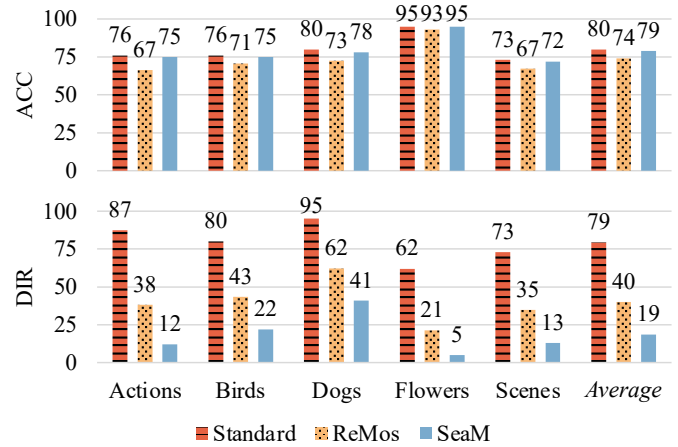

Fig. 6. The accuracy (ACC) and defect inheritance rate (DIR) on ResNet18.

engineered models. For instance, the inference time cost of VGG16-CIFAR10 is 6.82ms/batch, which means that VGG16-CIFAR10 requires 6.82ms to classify an input with batch size 16. The re-engineered model obtained by altering VGG16-CIFAR10 incurs an average of 3.79ms/batch inference time cost. The reduction in inference time cost is 44.43% (calculated by $(1-3.79/6.82)*100$). For all the six trained models, SEAM achieves an average of 42.41% reduction in inference time cost, which demonstrates that the reduction in the number of weights and FLOPs can reduce the inference time cost.

FLOP focus on the computation of neural network layers containing weights. While apart from the layers containing weights, the time cost for inference also involves other operations, such as activation functions, dropout, tensor reshape, and so on. Therefore, the reductions in the number of FLOPs and the time cost differ.

> Reusing a re-engineered model incurs less reuse overhead than reusing an original model, while achieving even higher accuracy in inference than the original model.

RQ3: Does reusing the re-engineered model mitigate the defect inheritance?

In this experiment, following ReMos, we evaluate SEAM and the baselines using two metrics, i.e., accuracy (*ACC*) and defect inheritance rate (*DIR*). Figure 6 and Figure 7 show the results of ResNet18 and ResNet50 on all of five datasets. In each figure, the first row displays the accuracy (*ACC*) of the fine-tuned model, and the second row displays the defect inheritance rate (*DIR*) of the fine-tuned model. On average, for ResNet18, the ACC and DIR achieved by the standard fine-tuning, ReMos, and SEAM are (80%, 79%), (74%, 40%), and (79%, 19%), respectively. For ResNet50, the ACC and DIR are (85%, 66%), (82%, 19%), (84%, 14%), respectively.

The standard fine-tuning approach achieves higher accuracy on the target datasets than SEAM and ReMos; however, the cost is much higher DIRs. Compared to the standard fine-tuning approach, both SEAM and ReMos can achieve lower
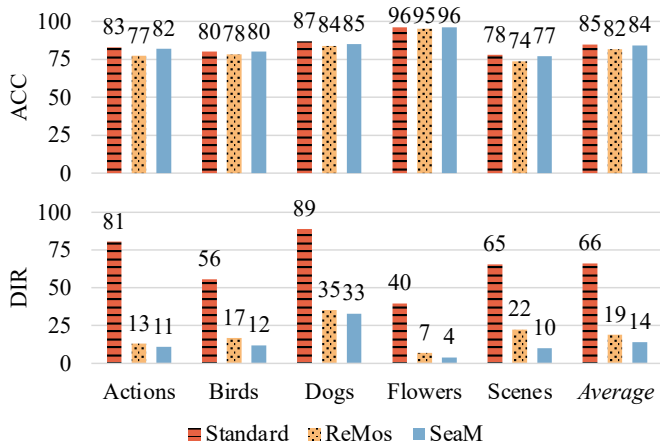
Fig. 7. The accuracy (ACC) and defect inheritance rate (DIR) on ResNet50.

Fig. 8. The accuracy (ACC) and defect inheritance rate (DIR) on ResNet18 with Dropout layers.

DIRs at the cost of a small accuracy loss, indicating that removing the weights that are not relevant to the target dataset can reduce DIRs and improve the robustness of the fine-tuned model. Overall, for the two models on five datasets, the averaged DIRs for fine-tuning the re-engineered model and fine-tuning the original model (i.e., standard fine-tuning approach) are 16% and 73%, respectively. The reduction in DIR is 57%, demonstrating the effectiveness of SEAM in reducing defect inheritance.

Compared to ReMos, SEAM can achieve lower DIRs and higher ACC. For instance, for ResNet18, the average DIRs achieved by SEAM and ReMos are 19% and 40%, respectively. The DIR achieved by SEAM is roughly half of that achieved by ReMos. Regarding ACC, the average ACC achieved by SEAM and ReMos is 79% and 74%, respectively. Overall, for the two models on five datasets, the average DIRs and ACC for SEAM and ReMos are (16%, 82%) and (29%, 78%), respectively. SEAM is 13% lower and 4% higher than ReMos in terms of DIR and ACC, respectively. The reason for the improvement in DIR achieved by SEAM is the considerable reduction in the number of weights. ReMos removes only 10% and 3% weights for ResNet18 and ResNet50, respectively. Compared to ReMos, SEAM can remove more irrelevant weights. The reduction in the number of weights is about 50% for both ResNet18 and ResNet50.

It is worth mentioning that there are some differences between the results shown in Figure 6 and Figure 7 and the results shown in ReMos [7], especially in terms of DIR. For instance, for ResNet18, the average DIRs achieved by ReMos shown in [7] and our work are 15% and 40%, respectively. The reason for the differences is that ReMos uses additional Dropout layers for fine-tuning while ours does not. To make a more comprehensive comparison of ReMos and SEAM, we follow the experimental setup of ReMos [7] and plot the results on ResNet18 in Figure 8. As shown in Figure 8, after adding Dropout layers, both SEAM and the baselines achieve better results, as Dropout layers help increase the robustness of models. The average DIRs achieved by the standard fine-
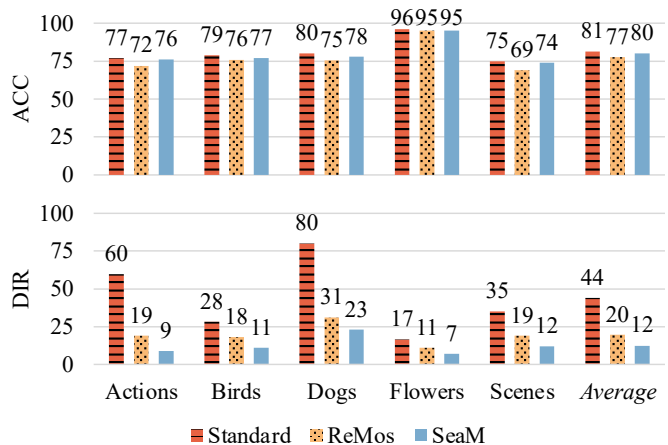
tuning approach, ReMos, and SEAM are 44%, 20%, and 12%, respectively. Consistent with the above conclusion, our approach can outperform ReMos. Moreover, we observe that the DIRs of ResNet50 are lower than that of ResNet18. The reason for this could be that the increased number of weights helps increase the robustness. This observation aligns with the prior works [7], [60].

> Overall, SEAM inherits much fewer defects compared to standard fine-tuning and the state-of-the-art approach.

V. THREATS TO VALIDITY

External validity: Threats to external validity relate to the generalizability of our results. While the notion of re-engineering a trained model to improve its reusability is general, we have only evaluated our approach on CNN models in this paper. The effectiveness on other types of DNNs, such as LSTM and transformer, remains to be evaluated. However, during the search, the objects removed are weights, not CNN-specific structures such as convolutional kernels. Also, the search is guided by the classification accuracy and the number of retained weights. Therefore, the principles of our proposed approach are not specific to CNN and are applicable to other types of DNNs as well. We will further investigate it in our future work.

Internal validity: An internal threat comes from the choice of trained models and datasets. To mitigate this threat, we use four representative trained CNN models and evaluate SEAM on eight well-organized and widely-used datasets.

Construct validity: A threat relates to the suitability of our evaluation metrics. Evaluating the quality of DNN models remains an open problem. Measuring only the misclassification rate of the adversarial samples may not be comprehensive enough. However, the misclassification rate of adversarial samples is a representative metric and has also been widely used in related work [7], [30].

VI. Related Work

Reusing trained DNN models: Our work is related to reusing DNN models, including direct reuse [4], [5] and transfer learning [12], [61]. The work related to direct reuse recommends a trained model for developers and allows developers to reuse the model on the target problem directly. For instance, SDS [4] evaluates trained models using a few efficient test data that could discriminate multiple trained models and then recommends the best one to reuse. Transfer learning techniques reuse a model trained to solve a similar problem and fine-tune the reused model on the target problem. For instance, ResNet [41] trained on ImageNet for 1000-class classification is widely reused to develop new models for various target problems by fine-tuning its weights on the target datasets [61], [62]. The techniques mentioned above support model reuse; however, they reuse the entire trained model or the vast majority of model's weights. In contrast, this work allows developers to reuse only the target problem-related weights, thus reducing reuse overhead and defect inheritance.

DNN modularization and slicing: Similar to our work, DNN modularization [8], [9] and slicing [7] attempt to reuse part of trained models. For instance, DNN modularization [8], [9] decomposes a trained model into modules based on neuron activation [21], [22]. A module retains part of trained model's neurons and can be reused to solve a binary classification problem. Relying on neuron coverage [21], [22], DNN slicing [7] removes irrelevant weights and reuses the slice with relevant weights for fine-tuning. Compared to DNN modularization and slicing, our work is search-based model re-engineering, which can remove much more irrelevant weights and hence reduce more reuse overhead and defect inheritance. Our previous work CNNSplitter [10] concerns the modularization of CNN models through searching with genetic algorithms and fixing the weakness of a model by replacing the corresponding part with a better module. In contrast, this work can realize the modularization of general neural network models and the searching algorithm is more efficient.

DNN pruning: Iterative magnitude pruning [57], [63], [64] is one of the mainstream network pruning techniques, which prunes part of weights that are not important for the original problem to reduce the computational overhead required by inference on the original problem. Our work removes part of weights that are irrelevant to a target problem to reduce reuse overhead and defect inheritance on the target problem. Apart from their differences in objectives, iterative magnitude pruning compresses a model by repeatedly removing unimportant weights and retraining the retained weights over several rounds, while SEAM removes irrelevant weights without changing retained weights.

VII. Conclusion

In this work, we propose the notion of *model re-engineering*, which re-engineers a trained DNN model to improve its reusability. Based on the notion, we propose a search-based model re-engineering approach named SEAM, which can re-engineer a trained model by removing many irrelevant weights.

Extensive experiments with four representative CNN models on eight widely-used datasets demonstrate the effectiveness of SEAM in reusing trained models as well as reducing reuse overhead and defect inheritance.

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

References

[1] I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success.* USA: ACM Press/Addison-Wesley Publishing Co., 1997.

[2] J. S. Poulin, *Measuring Software Reuse: Principles, Practices, and Economic Models.* USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

[3] C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992.

[4] L. Meng, Y. Li, L. Chen, Z. Wang, D. Wu, Y. Zhou, and B. Xu, "Measuring discrimination to boost comparative testing for multiple deep learning models," in *43rd International Conference on Software Engineering.* IEEE, 2021, pp. 385–396.

[5] Y. Ji, X. Zhang, S. Ji, X. Luo, and T. Wang, "Model-reuse attacks on deep learning systems," in *ACM SIGSAC conference on computer and communications security*, 2018, pp. 349–363.

[6] Y. Li, Z. Zhang, B. Liu, Z. Yang, and Y. Liu, "Modeldiff: Testing-based dnn similarity comparison for model reuse detection," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, p. 139–151.

[7] Z. Zhang, Y. Li, J. Wang, B. Liu, D. Li, Y. Guo, X. Chen, and Y. Liu, "Remos: Reducing defect inheritance in transfer learning via relevant model slicing," in *2022 IEEE/ACM 44th International Conference on Software Engineering*, 2022, pp. 1856–1868.

[8] R. Pan and H.Rajan, "Decomposing convolutional neural networks into reusable and replaceable modules," in *44th International Conference on Software Engineering*, 2022.

[9] R. Pan and H. Rajan, "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*, 2020, pp. 889–900.

[10] B. Qi, H. Sun, X. Gao, and H. Zhang, "Patching weak convolutional neural network models through modularization and composition," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[11] Phoenix. A fire-detection application. [Online]. Available: https://github.com/yunusemreemik/Phoenix

[12] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *27th International Conference on Neural Information Processing Systems*, 2014, p. 3320–3328.

[13] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT (1).* Association for Computational Linguistics, 2019, pp. 4171–4186.

[14] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems," in *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 739–743.

[15] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[16] B. Wang, Y. Yao, B. Viswanath, H. Zheng, and B. Y. Zhao, "With great training comes great vulnerability: Practical attacks against transfer learning," in *27th USENIX Security Symposium*, 2018, pp. 1281–1297.

[17] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar, "Adversarial machine learning," in *4th ACM workshop on Security and artificial intelligence*, 2011, pp. 43–58.

[18] S. Rezaei and X. Liu, "A target-agnostic attack on deep models: Exploiting security vulnerabilities of transfer learning," in *8th International Conference on Learning Representations*, 2020.

[19] L. H. Rosenberg and L. E. Hyatt, "Software re-engineering," *Software Assurance Technology Center*, pp. 2–3, 1996.

[20] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE software*, vol. 7, no. 1, pp. 13–17, 1990.

[21] F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, and M. Kim, "Is neuron coverage a meaningful measure for testing deep neural networks?" in *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 851–862.

[22] Z. Li, X. Ma, C. Xu, and C. Cao, "Structural coverage criteria for neural networks could be misleading," in *41st International Conference on Software Engineering: New Ideas and Emerging Results*, 2019, pp. 89–92.

[23] C. Wan, S. Liu, S. Xie, Y. Liu, H. Hoffmann, M. Maire, and S. Lu, "Automated testing of software that uses machine learning apis," in *44th International Conference on Software Engineering*, 2022, pp. 212–224.

[24] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, "Convolutional neural networks: an overview and application in radiology," *Insights into imaging*, vol. 9, no. 4, pp. 611–629, 2018.

[25] D. Bau, J.-Y. Zhu, H. Strobelt, A. Lapedriza, B. Zhou, and A. Torralba, "Understanding the role of individual units in a deep neural network," *Proceedings of the National Academy of Sciences*, vol. 117, no. 48, pp. 30 071–30 078, 2020.

[26] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" *Advances in neural information processing systems*, vol. 27, 2014.

[27] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2014, pp. 580–587.

[28] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[29] Y. Chen, C. Shen, C. Wang, and Y. Zhang, "Teacher model fingerprinting attacks against transfer learning," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3593–3610.

[30] T.-W. Chin, C. Zhang, and D. Marculescu, "Renofeation: A simple transfer learning method for improved adversarial robustness," in *Conference on Computer Vision and Pattern Recognition*, 2021, pp. 3243–3252.

[31] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[32] G. Van Horn, S. Branson, R. Farrell, S. Haber, J. Barry, P. Ipeirotis, P. Perona, and S. Belongie, "Building a bird recognition app and large scale dataset with citizen scientists: The fine print in fine-grained dataset collection," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2015, pp. 595–604.

[33] P. Welinder, S. Branson, T. Mita, C. Wah, F. Schroff, S. Belongie, and P. Perona, "Caltech-ucsd birds 200," 2010.

[34] B. Liu, Y. Guo, and X. Chen, "Wealthadapt: A general network adaptation framework for small data tasks," in *27th ACM International Conference on Multimedia*, 2019, pp. 2179–2187.

[35] DeepSparse. Sparsity-aware neural network inference engine for gpu-class performance on cpus. [Online]. Available: https://github.com/neuralmagic/deepsparse

[36] H. Liu, K. Simonyan, and Y. Yang, "DARTS: differentiable architecture search," in *7th International Conference on Learning Representations*, 2019.

[37] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, "Binary neural networks: A survey," *Pattern Recognit.*, vol. 105, p. 107281, 2020.

[38] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 4107–4115.

[39] Y. Bengio, N. Léonard, and A. C. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *CoRR*, vol. abs/1308.3432, 2013.

[40] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *ICLR*, 2015.

[41] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[42] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[43] chenyaofo. Pytorch cifar models. [Online]. Available: https://github.com/chenyaofo/pytorch-cifar-models

[44] PyTorch. Models and pre-trained weights. [Online]. Available: https://pytorch.org/vision/stable/models.html

[45] novelty detection. Imagenet division. [Online]. Available: https://github.com/noameshed/novelty-detection

[46] B. Qi. SeaM. [Online]. Available: https://github.com/qibinhang/SeaM

[47] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," in *International Conference on Learning Representations*, 2017.

[48] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *IEEE International Conference on Computer Vision*, 2017, pp. 5058–5066.

[49] FAIR. fvcore. [Online]. Available: https://github.com/facebookresearch/fvcore

[50] A. Quattoni and A. Torralba, "Recognizing indoor scenes," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 413–420.

[51] M.-E. Nilsback and A. Zisserman, "Automated flower classification over a large number of classes," in *2008 Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, 2008, pp. 722–729.

[52] B. Yao, X. Jiang, A. Khosla, A. L. Lin, L. Guibas, and L. Fei-Fei, "Human action recognition by learning bases of action attributes and parts," in *2011 International Conference on Computer Vision*, 2011, pp. 1331–1338.

[53] A. Khosla, N. Jayadevaprakash, B. Yao, and F.-F. Li, "Novel dataset for fine-grained image categorization: Stanford dogs," in *CVPR workshop on fine-grained visual categorization*, vol. 2, no. 1. Citeseer, 2011.

[54] BorealisAI. Advertorch. [Online]. Available: https://github.com/BorealisAI/advertorch

[55] ziqi zhang. Remos: Reducing defect inheritance in transfer learning via relevant model slicing. [Online]. Available: https://github.com/ziqi-zhang/ReMoS_artifact

[56] rangeetpan. Decomposing a convolutional neural network into modules. [Online]. Available: https://github.com/rangeetpan/Decomposition

[57] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *Advances in neural information processing systems*, vol. 28, 2015.

[58] C. Louizos, M. Welling, and D. P. Kingma, "Learning sparse neural networks through l_0 regularization," in *International Conference on Learning Representations*, 2018.

[59] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[60] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *6th International Conference on Learning Representations*, 2018.

[61] Y. Guo, H. Shi, A. Kumar, K. Grauman, T. Rosing, and R. Feris, "Spottune: transfer learning through adaptive fine-tuning," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4805–4814.

[62] S. Kornblith, J. Shlens, and Q. V. Le, "Do better imagenet models transfer better?" in *IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 2661–2671.

[63] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *International Conference on Learning Representations*, 2019.

[64] J. S. Rosenfeld, J. Frankle, M. Carbin, and N. Shavit, "On the predictability of pruning across scales," in *International Conference on Machine Learning*, 2021, pp. 9075–9083.