DeepArc: Modularizing Neural Networks for the Model Maintenance

Xiaoning Ren1, Yun Lin2∗, Yinxing Xue1∗, Ruofan Liu3, Jun Sun4, Zhiyong Feng5, Jin Song Dong3
1 University of Science and Technology of China, China, hnrnxn@mail.ustc.edu.cn, yxxue@ustc.edu.cn
2 Shanghai Jiao Tong University, China, lin_yun@sjtu.edu.cn
3 National University of Singapore, Singapore, e0134091@u.nus.edu, dcsdjs@nus.edu.sg
4 Singapore Management University, Singapore, junsun@smu.edu.sg
5 Tianjin University, China, zyfeng@tju.edu.cn

Abstract—Neural networks are an emerging data-driven programming paradigm widely used in many areas. Unlike traditional software systems consisting of decomposable modules, a neural network is usually delivered as a monolithic package, raising challenges for some maintenance tasks such as model restructure and re-adaption. In this work, we propose DeepArc, a novel modularization method for neural networks, to reduce the cost of model maintenance tasks. Specifically, DeepArc decomposes a neural network into several consecutive modules, each of which encapsulates consecutive layers with similar semantics. The network modularization facilitates practical tasks such as refactoring the model to preserve existing features (e.g., model compression) and enhancing the model with new features (e.g., fitting new samples). The modularization and encapsulation allow us to restructure or retrain the model by only pruning and tuning a few localized neurons and layers. Our experiments show that (1) DeepArc can boost the runtime efficiency of the state-of-the-art model compression techniques by 14.8%; (2) compared to the traditional model retraining, DeepArc only needs to train less than 20% of the neurons on average to fit adversarial samples and repair under-performing models, leading to 32.85% faster training performance while achieving similar model prediction performance.

Index Terms—architecture, modularization, neural networks

I. INTRODUCTION

Deep neural network (DNN) models have attracted increasing attention in both industry and academia, widely incorporated in many modern software systems [1]–[6]. As a new programming paradigm, neural network models are programmed with predefined structures of mathematical functions [7], [8], whose parameters are learned automatically to fit a given training dataset. While a network model can be learned to function well, its hundreds of thousands of stacked “tiny functions” make it hard to interpret how the decisions are made, which in turn makes model maintenance challenging.

In practice, a trained DNN model needs to evolve with new requirements. Analogous to code logic modification and redundant code refactoring in traditional software, programmers and data scientists need to adapt the model regarding the new samples, or prune the model while preserving its behaviors (e.g., adapting a functional model into mobiles or IoT devices). However, existing practice usually requires restructuring the model with the new and existing training dataset [9], which induces large maintenance costs. Note that classical neural networks usually consist of millions of weights (Resnet-50 [10] has about 25.6 million weights, VGGNet19 [11] has 143 million weights, and the emerging GPT-3 [12] even has 175 billion weights), which makes the current practice to maintain the models computationally expensive.

For the maintenance of DNN models, there are emerging works in SE community to decompose a model for improving its reusability. Pan et al. proposed to decompose a multi-classification model into a set of binary classification models for fully connected networks [13] and convolutional networks [14]. Technically, given a network model $G$ and prediction class set $C$, their solution extracts a subgraph $G' \in G$ as a binary classifier to preserve the behavior of $G$ on $C'$ ($C' \subset C$). However, this technique can provide limited support if we need to restructure the model $G$ while preserving its behaviors or improve $G$’s performance with minimum modifications.

In this work, we aim for a model modularization solution focusing on the encapsulation nature. Specifically, we investigate whether we can change only a few modularized neurons and layers in the model maintenance tasks. We propose to extract the semantic architecture of a neural network by encapsulating consecutive model layers as a semantic module. In a semantic module, (1) all layers extract features with similar expressiveness/separability for training/testing samples, and (2) some layers can be removed and duplicated without changing much model behavior (e.g., model prediction and robustness).

Those properties can facilitate applications such as model compression and model enhancement. The network modularization provides the following maintenance benefits:

- **Decomposition Overview**: By encapsulating similar layers, we can have an abstract view on how many actual transformation functions a network model can be decomposed. Each actual transformation (i.e., semantic module) is essentially a modularized feature-extraction subfunction.

- **Modularized Restructure**: Similar layers indicate potential redundant layers, which provides us the flexibility to remove and add some layers in a module without changing much model behavior.

- **Modularized Re-adaption**: Each semantic module can represent a unique feature-extraction function, which allows us to localize and retrain a certain critical module (instead of

∗ Yun Lin and Yinxing Xue are the corresponding authors.
II. FRAMEWORK

Figure 1 shows an overview of our DeepArc framework. Given a DNN model, DeepArc first extracts several modules to form a semantic architecture, each of which includes semantically similar model layers. Based on the semantic architecture, we can support (1) model restructure to modify the network while preserving its behaviors, and (2) model re-adaption to fix mis-prediction or fit a model on new samples. Moreover, the restructured or improved DNN can be further modularized to support new model restructuring or re-adaption tasks.

A. Semantic Architecture

Typically, a neural network can be decomposed into a sequence of network layers, i.e., \( f_n() = l_n(l_{n-1}(\ldots l_1())(n > 1) \), where each subfunction \( f_i() \) takes the output of its former subfunction \( f_{i-1}() \) as input. We denote each layer as \( l_i() : \mathbb{R}^{n_i-1} \rightarrow \mathbb{R}^{n_i} \) where \( n_i \) and \( n_{i-1} \) are the number of neurons in \( l_i \) and \( l_{i-1} \) respectively. Given an input \( x \in \mathbb{R}^{n_i} \) and a network \( f_n() \) with \( n \) layers, \( f_n() \) can generate \( n - 1 \) representations each of which is the output of \( f_k() = l_k(l_{k-1}(\ldots l_1()))(2 \leq i \leq n) \).

- **Layer Similarity:** Given a dataset \( D = \{x_1, \ldots, x_m\} \), we denote \( x_j \in D \) as an input and \( f_i(x_j) \) as the output representation of \( x_j \) on layer \( l_i \). With a similarity measurement on \( l_i \), denoted as \( sim_{l_i}(\cdot, \cdot) \), we call the matrix \( M_{l_i} \) as the similarity matrix of \( l_i \) where \( M_{l_i}[p, q] = sim_{l_i}(x_p, x_q) \). Here \( x_p \) and \( x_q \) are the indexes of a pair of samples \( x_p \) and \( x_q \). Therefore, the matrix at each layer has the same shape as \([m, m] \) and contains the similarity of all pairs of samples. Intuitively, \( M_{l_i} \) indicates the embedding landscape of dataset on the \( i \)th layer. Figure 2 demonstrates the embedding landscape of 16 data samples. Figure 2a and Figure 2b share similar landscape as the samples are in similar spatial distribution. Specifically, the data points share similar neighbours in the embedding space. In contrast, Figure 2a and Figure 2c share different landscape. A similar embedding landscape indicates that the model extracts similar semantics from the input samples. Typically, we implement \( sim_{l_i}(\cdot, \cdot) \) as:

\[
sim_{l_i}(x_p, x_q) = f_i(x_p) \cdot f_i(x_q)
\]

Further, given a dataset \( D \) and two layers \( l_i \) and \( l_j \), we can define a layer similarity measurement \( sim_{layer}(\cdot, \cdot) \) as:

\[
sim_{layer}(M_{l_i}, M_{l_j}) = \frac{M_{l_i} \cdot M_{l_j}}{||M_{l_i}|| \cdot ||M_{l_j}||}
\]

- **Module Similarity:** Given a module \( p_k \) consisting of \( m \) consecutive layers starting at \( l_{k_1} \), i.e., \( p = \{l_{k_1}, l_{k_2}, \ldots, l_{k_m}\} \) where \( k_i (1 \leq i \leq m) \) denotes the index.
Fig. 3: An example of a modularized DNN for classification.

Fig. 4: Layers within the same modules share similar expressiveness: The x-axis represents layers, and y-axis represents the classification accuracy of a linear classifier trained on output representation of layers.

of the i-th layer in module \( p_k \), we define the intra-module similarity measurement of \( p \) as:

\[
\text{sim}_{\text{module}}(p_k) = \frac{\sum_{i=1}^{p_k-1} \text{sim}_{\text{layer}}(M_{l_i}, M_{l_{i+1}})}{|p_k| - 1}
\]  

(3)

If \(|p_k| = 1\), we define \( \text{sim}_{\text{module}}(p_k) = 0 \). Intuitively, if the output representations of \( l_i \) and \( l_{i+1} \) are similar, the transformation of \( f_{i+1}(\cdot) \) preserves the similar output (embedding) semantics as \( f_i(\cdot) \). Moreover, we call a module \( p_k \) as single-layer module if \(|p_k| = 1\); and call \( p_k \) as multiple-layer module if \(|p_k| > 1\).

Given a network model \( f_{\cdot}(\cdot) \) and a similarity threshold \( \text{th}_s \), the semantic architecture of \( f_{\cdot}(\cdot) \) is a partition \( \mathcal{P} = \{ p_1, p_2, \ldots, p_M \} \) (\( M \) is the number of modules) on a sequence of consecutive model layers so that

\[
\max_{\mathcal{P}} \sum_{k=1}^{M} \text{sim}_{\text{module}}(p_k) \quad \text{s.t.} \quad \forall p_k(|p_k| > 1), \text{sim}_{\text{module}}(p_k) > \text{th}_s
\]  

(4)

In Equation 4 the objective function requires that the overall layer similarity of each module in the partition is maximized while the layer similarity between different modules should be minimized. In addition, the condition requires that the overall layer similarity of each module in the partition should be larger than the predefined threshold \( \text{th}_s \).

Example. Next, we use a RestNet-110 [10] model trained on CIFAR-10 [24] dataset to illustrate the modularization. The trained network has 54 blocks, which can be decomposed into 26 modules including multiple-layer and single-layer modules. Three largest multiple-layer modules are highlighted in Figure 3. Module A corresponds to 5-9th blocks, Module B corresponds to 22-35th blocks, Module C corresponds to 38-49nd blocks in ResNet-110. Specifically, Figure 3 shows an optimal partition \( \mathcal{P}^* = \{ \cdot, \cdot, [5-9], \cdot, [22-35], \cdot, \cdot, [38-49], \cdot, \cdot, [53-54] \} \).

Given a dataset \( D \) and a layer \( l_i \), we say that \( D \) is \( k\%-\text{linear separable} \) on \( l_i \) if we can achieve a prediction accuracy of \( k\% \) through training a logistic regression classifier to fit the output representation of \( D \) on \( l_i \). The linear separability of layer \( l_i \) indicates how expressive the \( i\)-th layer can be used to fit the dataset. Figure 4 demonstrates how the layers within a module are similar in expressing samples. Overall, the layers within a same module share similar expressiveness.

Layer Modularization. Equation 4 indicates the modularization (i.e., the search for an optimal partition) is an NP-complete problem, the global optimization is computationally expensive. We design our efficient modularization algorithm based on the observation of a phenomenon of \textit{neighboring closeness} in neural network layers. Specifically, given layer \( i \), \( i + k_1 \), and \( i + k_2 \) where \( k_2 > k_1 > 0 \), it is more likely that \( \text{sim}_{\text{layer}}(l_i, l_{i+k_2}) > \text{sim}_{\text{layer}}(l_i, l_{i+k_1}) \). Intuitively, closer layers go through less transformation, which are more likely to be similar. Figure 5 shows a visualized layer matrix where each row or column represents a layer and a matrix entry represents the similarity between two layers. The darker the...
Given that there is a high correlation between layer proximity and layer similarity, we design a progressive semantic architecture algorithm as a sub-optimal solution to Algorithm 1 which takes as input a layer-similarity matrix $sim$ (as in Figure 5) and a modular similarity threshold ($th_s$), and generates a layer partition as the semantic architecture $S_{layer}$. We represent one network layer with its layer index, one network module as a set of layer indexes, and the semantic architecture as a set of modules. In Algorithm 1 we progressively merge the layers in sequential order (line 2). Specifically, we add a layer into a module if (1) the module is empty (line 3) or (2) the inner-module similarity of the to-be-merged module is above the threshold $th_s$. If a layer cannot be merged into the current module (line 7), we add the module into the semantic architecture, and create a new module for further analysis (lines 8-9). Overall, the complexity of Algorithm 1 is $O(n)$ where $n$ is the model layer size.

B. Model Restructure

Given that the layers within a module share similar feature extraction, their “redundancy” can be used to reduce the model size. Some layers inside a module can be removed without modifying much of the model structure, while largely preserving the model behaviors such as prediction accuracy. Figure 6 shows our architecture-based compression acceleration technique. Our architecture-based compression acceleration technique firstly obtains the reduced depth network B by removing the redundant network layers from network A, and then performs the standard compression technique to obtain network C. Larger granularity of pruning will lead to effective speedup. Overall, the extracted architecture serves as an informative guidance to prune redundant layers to have a pre-processed model. Then, the follow-up model compression techniques such as [16–20] can be applied to further fine-grained compression of the pre-processed model to drop the neurons/channels.

Layer Reduction. Given a module $pk = \{l_{k_1}, l_{k_2}, ..., l_{k_n}\}$, for each layer $l_{k_i}$, we denote its input shape as $S_{in}(l_{k_i})$ and its output shape as $S_{out}(l_{k_i})$. Also, we denote the layer after the module is $l_{next}$. We reduce the module by keeping the first layer which acts as the interface layer, and removing the rest layers which represent redundant trans-

informative guidance to drop the neurons/channels.

Layer Reduction. Given a module $pk = \{l_{k_1}, l_{k_2}, ..., l_{k_n}\}$, for each layer $l_{k_i}$, we denote its input shape as $S_{in}(l_{k_i})$ and its output shape as $S_{out}(l_{k_i})$. Also, we denote the layer after the module is $l_{next}$. We reduce the module by keeping the first layer which acts as the interface layer, and removing the rest layers which represent redundant trans-
After achieving the semantic architecture, we apply a module-wise root-cause analysis technique to locate the most critical modules. Finally, we isolate the critical modules and apply traditional retraining process on them.

**Rationale.** The transformation between two module serves as the module interface. Suppose a module outputs a deviated representation from the “normal” representation, the deviation can cause all the follow-up modules to output unexpected representations, leading the model to make an unexpected prediction. Therefore, our approach aims to locate the most significant module where the representations of some samples are most deviated and fix it back to a normal representation. Figure 3 shows an example where a normal input and its derived adversarial sample are fed into a modularized model. Taking the normal representation as a reference, we can retrain a single module (instead of the whole network) to correct the prediction. Therefore, the training can be applied in an encapsulated module instead of the whole network. Next, we introduce how to localize critical modules.

1) Critical Module Location: We locate the critical module by measuring the deviation of the new samples from the original dataset. Given a model $m$ and its semantic architecture $P^* = \{p_1, p_2, ..., p_n\}$, we know that $m$ can be decomposed into $n$ transformations. For each transformation (or module), we denote it as $M_i: R^n \rightarrow R^n$, where $R^n$ is the input space of the module and $R^n$ is the output space. Given an input $x$, any module $M_i$ can have its input space representation, denoted as $x_{M_i}$. In other words, $M_i(x_{M_i}) = x_{M_i}$. We assume that we have a dataset $D$ where $m$ can perform well and $D'$ where $m$ cannot perform well. For any module, taking its output space representation of $D$ as a reference, we can evaluate the deviation of the representations of $D'$ from that of $D$. Our goal is to find a module where such deviation is significant, as the critical module for a re-adaptation task. From the other perspective, we also identify the most non-robust features (or transformation) for its significant deviation.

**Deviation Measurement.** Let the dataset to be enhanced/repaired as $D'$. We use $Dev(D')$ as the deviation measurement for $D'$. Given an input $x \in D'$, we denote its deviation measurement as $Dev(x)$. We design different deviation measurements regarding explicit reference and implicit reference as follows.

• **Explicit Reference:** Given a dataset $D'$, for each $x \in D'$, if we can infer its exact ground truth output representation (denoted as $x_{M_{i}^{D'}}$) in any modules, we call $D'_{M_{i}^{D'}}$ as the explicit reference of $D'$. For example, assuming that we know some adversarial inputs $D'$ are derived from a normal dataset $D$, we can take the explicit reference $D_{M_{i}^{D'}}$ as $D'_{M_{i}^{D'}}$ (see Figure 3). Specifically, we use $sim_{layer}(\cdot, \cdot)$ distance between $D_{M_{i}^{D'}}$ and $D'_{M_{i}^{D'}}$ as $Dev(D')$.

• **Implicit Reference:** In contrast, if we need to estimate the ground truth output representation in any modules, we call the estimated ground truth as the explicit reference of $x$. Intuitively, we estimate how likely $x$ is to fall into the distribution of a subset of well-predicted dataset $D_s \in D$. For example, in a classification task, we estimate the probability that $x$ can fall into the distribution of the sample set $D_C \subset D$ where (1) the label of each $s \in D_C$ is $C$, and (2) the model $m$ can perform well on $D_C$. Specifically, we estimate a multi-variable Gaussian probability distribution function $P$ for $D_C$, then calculate the probability of $x$ being in the distribution of $D_C$ by $P(x)$. As a result, we use $P(x)$ as $Dev(x)$ and $\sum_{x \in D_{\text{normal}}} P(x)$ as $Dev(D')$.

2) Modularized Training: Given a critical module $M$, we retrain the model by only fine-tuning the weights in $M$. Given the original dataset $D$, the to-be-adapted dataset $D'$, and the to-be-adapted model $m$, we train $m$ with the following loss function:

$$L(D, D', model) = \alpha \cdot L(D \cup D', model) + \beta \cdot Dev(D') \quad (5)$$

In Equation 5, the first item $L(D \cup D', model)$ is the original loss function to train a model. Note that the deviation measurements for explicit and implicit reference are differentiable. Intuitively, the deviation measurement can be regarded as an additional constraint (or soft invariant) forced by the “module interface”. In addition, $\alpha$ and $\beta$ are two user-defined hyperparameters.

### III. Evaluation

We evaluate DeepArc with the following research questions:

• **RQ1 (Modularization Prevalence):** To what extent can a neural network be modularized? Under what circumstances can and cannot the modularization disappear?

• **RQ2 (Modularization Semantics):** What semantics can the layers in the same module share?

• **RQ3 (Model Restructure):** Can our architecture based model restructure technique effectively boost the state-of-the-art model compression tasks?

• **RQ4 (Model Re-adaption):** How effective is our architecture based model re-adaption technique that can improve the model retraining tasks?

RQ1 and RQ2 evaluate the effectiveness of our modularization method and the properties of the module itself. RQ3 and RQ4 evaluate the usefulness on two application tasks.

#### A. Dataset Setup

1) **Model Architectures and Datasets:** We choose the classical ResNet-110 [10], Wide-ResNet-38 [21], and VGGNet-19 [11] as the subject model architectures. In addition, we train the models on MNIST [22], FMNIST [23], and CIFAR-10 [24] datasets until the model performance converges. Moreover, we train each configuration (i.e., architecture-dataset combination) 10 times. We choose $\theta_0 = 0.99$ (in Algorithm 1) to generate their semantic architectures. More detailed configurations (e.g., hyperparameters such as learning rate and decay ratio to train the models) can be referred to in [25].
B. Modularization Prevalence Experiment (RQ1)

1) Experiment Setup: To answer the RQ1, we design an experiment to observe the semantic architecture of a deep classifier by controlling three variables, i.e.,
- **Model Size (MS):** The model architecture with different numbers of parameters. We take ResNet architecture [10] as the subject model because we can flexibly prune its size.
- **Dataset Size (DS):** The datasets with different sizes. We take CIFAR-10 as the subject dataset and take 20%, 40%, 60%, 80%, 100% of the total samples in the experiment.
- **Task Scale (TS):** Given an N-classification task, we choose five subsets of the task to control the classification task into 0.2N, 0.4N, 0.6N, 0.8N and N-classification.

The number of layers and parameters of the ResNet model includes five configurations of <14,0.176M>, <44,0.663M>, <86,1.34M>, <110,1.73M>, <152,2.42M>, each of which is trained 10 times on 5 dataset sizes and 5 task scales of CIFAR-10, with a total of 1250 (5*5*10) models.

**Modularity Metrics.** We define the modularity metrics $Y_{\text{modular}} = \frac{N_{\text{layer}}}{N_{\text{module}}}$, indicating how modularized a model architecture is, and conduct a correlation analysis with $Y_{\text{modular}}$ as the dependent variable. In the metrics, $N_{\text{layer}}$ is the number of layers in multi-layer modules and $N_{\text{module}}$ is the total number of layers. The fact that $Y_{\text{modular}} = 0$ means that the model has only single-layer modules, while the fact that $Y_{\text{modular}} = 1$ means all modules are multiple-layer.

We define the high modularity threshold to be 0.25, which means that if over 1/4 of layers can be encapsulated, we think it can be effectively modularized. We further record the location of multi-modules to study the potential pattern of module distribution. Besides, to show the impact of the choice of threshold, we run our experiment with thresholds $th = 0.9, 0.91, 0.92, 0.93, 0.94, 0.95, 0.96, 0.97, 0.98, 0.99$ (see algorithm 1).

2) Results: Figure 9 shows the correlation analysis results of $MS$, $DS$, $TS$ and $Y_{\text{modular}}$. The regression slope with the model size as the variable is positive and largest, while ones with the dataset and task are negative and relatively small. This shows that when the model size changes, it has the greatest impact on the modularity degree. Each independent variable has a certain correlation with the degree of modularity, in which $DS$ and $TS$ are negatively correlated with $Y_{\text{modular}}$, but $MS$ positively. Considering all the variables together, a multivariate retrospective analysis yields a regression equation: $Y_{\text{modular}} = 0.78 * MS - 0.10 * DS - 0.16 * TS + 0.31 \ (6)$

This equation intuitively shows that $Y_{\text{modular}}$ has the highest sensitivity to model size, and relatively low sensitivity to dataset and task size, despite they all have a strong correlation. Besides, for popular configurations such as ResNet86 and ResNet110 on CIFAR-10, the high modularity threshold is exceeded, i.e. above the red line in the figure. In summary, the network modularity is usually present and particularly evident with a large model, simple task, and tiny data. In other words, the more redundant information the model has, the greater the degree of modularity. Typically, popular configurations can achieve effective modularity.

As shown in Figure 10 the multiple-layer modules are mainly found in the later layers of the model, because the features extracted later are more refined, and prone to redundant information due to slow feature transformation. Also, there are almost all single-layer modules on the boundary, where the feature dimension changes, usually acting as an interface.
layer. Besides, Figure 11 shows how the degree of modularity varies with the threshold \( t_{h} \) of Algorithm 1. The threshold is a custom tradeoff, which needed to be set larger when we require a high level of information redundancy within the module, and vice versa. The threshold refers to the similarity of adjacent layers, and this similarity will become smaller as the layers are stacked. Therefore, we choose a threshold of 0.99 to guarantee a high degree of semantic similarity within modules.

**Answer to RQ1:** Usually, there is a modular phenomenon in the layers of the model. Model, dataset and task size will affect modularity and have a certain linear correlation. The modularization will almost disappear with a small model, complex task and huge dataset, that is, each layer is a module representing different semantics.

**C. Modularization Semantics Experiment (RQ2)**

To answer RQ2, we evaluate the semantic properties of the module in the following two parts:

- **RQ2.1 (Dynamic Semantic):** How much can the model behaviors change if we remove and replicate the layers in multi-layer modules?
- **RQ2.2 (Static Semantic):** How much the prediction can vary if we use the intra/inter layers for the prediction?

1) **Dynamic Semantics Design:** We apply intra-module and inter-module changes on the models. We evaluate how the model prediction and robustness change when we remove and duplicate layers in intra-module and inter-module layers. Furthermore, in order to avoid introducing additional neurons when modifying the network model, we only add/remove the layers where the output dimension of their previous layer can match the input dimension of their following layer.

**Intra-module Change.** Given a trained model \( m \) and its derived semantic architecture \( \mathcal{P}^* = \{p_1, \cdots, p_k\} \), where each \( p_i \) represents a module consisting of consecutive layers in \( m \), we change \( m \) into its variant \( m' \) by removing or adding layers in multi-layer module \( p_i \) as follows.

We denote the original model as \( m_0 \), the model variant with \( k \) intra-module layers removed as \( m_{-k} \), and that with \( k \) intra-module layers added as \( m_{+k} \). Algorithm 2 shows the order we use to add and remove layers in the modules. Specifically, we sequentially parse the multi-layer modules in the semantic architecture (lines 2-3). For each module, we process its layer addition or removal from the last layer to its first layer (lines 4-7). Algorithm 2 provides a progressive way of adding and removing the layers.

**Inter-module Change.** Given a derived semantic architecture \( \mathcal{P}^* = \{p_1, \cdots, p_k\} \), we can consider the layers in the single-layer modules as module interfaces. We remove them to have inter-module change. Their addition/removal order is the order they appear in the model.

**Semantic Preserving Evaluation.** Given the original model \( m_0 \) and its variant model \( m_{+/-k} \), we evaluate how the semantics has been changed regarding the changed model prediction and robustness. We evaluate the model robustness change by how many adversarial samples for \( m_0 \) can be reused to compromise the model \( m_{+/-k} \). Specifically, given an adversarial attack approach \( A \), we generate \( N (N=1000 \text{ in the study}) \) adversarial samples for \( m_0 \). If \( M \) out of \( N \) adversarial samples can still compromise \( m_{+/-k} \), then we say that the model robustness change is \( 1 - \frac{M}{N} \). In this study, we use FGSM [20], DeepFool [21], C&W [22], and PGD [23] as adversarial attack approaches. Note that fine-tuning with few data is required for VGG19, while the ResNet family does not.

2) **Results of Dynamic Semantics:** Figure 12 and Figure 13 show how the semantics are preserved for the model variants under different model architectures (i.e., ResNet-38, ResNet-110, and VGGNet-19). Given the space limit, we only show diversified model architectures on the CIFAR-10 dataset. More details and figures are available at [25]. Overall, the other two datasets (and with more training trials) share similar effects.

**Preserving Model Prediction.** In Figure 12 the x-axis shows the model variants caused by adding or removing \( k \) intra-module layers according to Algorithm 2. For example, “0” represents the original model, “-3” represents the model variant by removing 3 layers, and “8” represents the model variant by adding 8 layers. In the following, we denote a model by adding \( k \) layers as \( k \)-model. We can see that, the model prediction performance on both the training and testing dataset is largely preserved even if a large number of intra-module layers are removed. Besides, the same magnitude of variation indicates that changes within the module will preserve the model prediction ability, not causing overfitting or underfitting.

**Preserving Robustness.** Figure 13 shows the performance of model variants on the adversarial samples compromising the original model. Note that, the accuracy of 0-model is the performance of the original model on those adversarial samples. Except for FGSM, almost all the attacks make the accuracy of the original model drop to almost 0. We can see that the adversarial samples of different attack methods have different semantic preserving effects. Overall, the adversarial samples from PDG and FGSM can still compromise the model variants regardless of the number of added/removed layers. In contrast, there is a “jump” of prediction accuracy from 0-model to 1-model and -1-model for the adversarial samples from DeepFool and C&W (see the red and purple lines). Our
Fig. 12: Training and testing accuracy by removing and adding intra-module layers.

Fig. 13: The demonstration of how the adversarial samples for the original model can further compromise its model variants by removing/adding intra-module layers.

Fig. 14: Different adversarial samples, the distribution of different adversarial samples is different, which makes some samples fail to compromise the model variants.

investigation shows that, compared to the samples from PDG and FGSM, that from DeepFool and C&W are more likely to lie closer to the classification decision boundaries. The average model confidence score (the difference between the top-2 prediction before the softmax layer) on the adversarial samples of DeepFool is 2.38; in contrast, FGSM is 9.4 and PGD is 25, far larger than that of DeepFool. Figure 14 shows an illustration. Each deep classifier has its high-dimensional classification boundary (the solid curve in Figure 14), and its mutant can be considered as a perturbation of the boundary (the dashed curve in Figure 14). If some adversarial samples lie close to the boundary, the boundary perturbation can lead to the prediction change. Although the impact of deleting/adding one layer is very small, the boundary samples are sensitive to the layer change. Hence, DeepFool and C&W (with adversarial samples generated on the boundary) have larger performance changes while others have not.

Interface Importance. Figure 15 shows the change of testing accuracy if we gradually remove inter-module layers, compared to the removal of intra-module layers. For example, for the ResNet-110 architecture Figure 15a, the model accuracy drops from 0.96 to 0.1, with the inter-module layer removal. In contrast, the model accuracy is still preserved around 0.9. This shows that the interface is responsible for transforming useful information. If the interface is broken, the overall information will be greatly affected.

Finding 1: The intra-module change can essentially preserve the semantics, regarding both model prediction and model robustness. However, the inter-module change can cause the model semantics to change radically.

3) Static Semantics Design: We quantify the static semantics of a layer $l$ by how sufficient the output representation of $l$ can be used to classify a training dataset linearly. Given a layer $L$, and a dataset $D$, we use $D_L$ to represent the set of output representation of $D$. Specifically, we use a linear classifier (i.e., we use one linear-layer neural network in this work) to classify $D_L$ and achieve its classification accuracy, denoted as $Acc_{linear}(L)$. Intuitively, we measure the linear expressiveness of each layer to distinguish $D$.

Linear Expressiveness Evaluation. Given an architecture as $P^* = \{p_1, p_2, ..., p_n\}$, we sample $N$ intra-module layer pair set $P_{intra} = \{(L_i, L_j) \mid L_i \in p_k, L_j \in p_k, p_k \in P^*, i \neq j\}$ and $N$ inter-module layer pair set $P_{inter} = \{(L_i, L_j) \mid L_i \in p_{k_1}, L_j \in p_{k_2}, k_1 \neq k_2, i \neq j\}$. For each $p = (L_i, L_j)$, we calculate the distance of linear expressiveness by $dis_{exp}(p) = |Acc_{linear}(L_i) - Acc_{linear}(L_j)|$. Then, given a model and its semantic architecture, we compare
TABLE I: The difference of linear expressiveness of inter-module layers and intra-module layers

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Wide-ResNet38</th>
<th>ResNet110</th>
<th>VGGNet19</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$P_{\text{intra}}$</td>
<td>$P_{\text{inter}}$</td>
<td>$P_{\text{intra}}$</td>
</tr>
<tr>
<td>MNIST</td>
<td>0.044±0.001</td>
<td>0.127±0.006</td>
<td>0.265±0.003</td>
</tr>
<tr>
<td>FMNIST</td>
<td>0.071±0.002</td>
<td>0.142±0.008</td>
<td>0.275±0.006</td>
</tr>
<tr>
<td>CHAR16K</td>
<td>0.071±0.001</td>
<td>0.246±0.012</td>
<td>0.392±0.009</td>
</tr>
</tbody>
</table>

4) Results of Static Semantics: Table 1 shows that the linear expressiveness of intra-module layers is much more similar than that of inter-module layers. For the Wide-ResNet38 trained on MNIST, the average difference of the linear expressiveness of each intra-module layer is 0.044, while the average difference of each inter-module layer is 0.217. This shows that the difference score of inter-module linear expressiveness is nearly five times that of the difference score of intra-module linear expressiveness. On average, the difference score of intra-module linear expressiveness is 0.056, in contrast to that of inter-module linear expressiveness is 0.18. For the layers within the modules, the semantic transformations between them are more redundant and bring little help to the classification task.

Finding 2: The intra-module representations of a semantic architecture have similar linear separability. In contrast, the inter-module representations have more obvious differences.

D. Compression Efficiency Experiment (RQ3)

1) Experiment Setup: To answer the RQ3, we select 4 state-of-the-art model compression techniques proposed in the last two years (i.e., LAMP [30], Global [31], Uniform+ [32] and ERK [33]) to see whether our DeepArc framework can speed up the model compression efficiency. Following the above four works, we use ResNet-101 trained on CIFAR10 as the target model. We conduct controlled experiments according to whether our framework is used for pre-processing before compression. Specifically, we collect the compression rate, model accuracy and number of iterations under the premise of guaranteeing similar prediction accuracy. The three indicators are as follows, 1) compression rate: the proportion of compressed parameters 2) number of iterations: the number of iterations required to complete the compression 3) model accuracy: the accuracy of the model on testset.

Answer to RQ3: Based on the removal of redundant information in semantic modules, DeepArc accelerates model restructure efficiency significantly, while resulting in models with comparable compression ratios and model accuracy.

E. Model Re-adaption Experiment (RQ4)

1) Experiment Setup: To answer the RQ4, we design two tasks to see whether DeepArc can improve the re-training efficiency with little loss, compared to an entire training.
TABLE III: The model re-adaption performance with/without the acceleration of DeepArc

<table>
<thead>
<tr>
<th>Re-adaption Type</th>
<th>Training Approach</th>
<th>Dropped Accuracy</th>
<th>WideResNet38</th>
<th>ResNet110</th>
<th>VGGNet19</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Trained Model Size</td>
<td>Original Accuracy</td>
<td>Retraining Accuracy</td>
<td>Training Time (s)</td>
</tr>
<tr>
<td>Model Enhancement</td>
<td>Fully Training</td>
<td>56.07M</td>
<td>0.0751</td>
<td>0.9980</td>
<td>294.63</td>
</tr>
<tr>
<td></td>
<td>Modularized Training</td>
<td>7.38M</td>
<td>0.9747</td>
<td>0.9974</td>
<td>158.21</td>
</tr>
<tr>
<td>Model Repair</td>
<td>Fully Training</td>
<td>56.07M</td>
<td>0.0840</td>
<td>1.0000</td>
<td>762.01</td>
</tr>
<tr>
<td></td>
<td>Modularized Training</td>
<td>7.38M</td>
<td>0.9747</td>
<td>0.9974</td>
<td>158.21</td>
</tr>
</tbody>
</table>

Model Enhancement Task. In this experiment, we use four adversarial attacks, i.e., DeepFool [27], C&W [28], FGSM [26], and PGD [29] to generate 1000 adversarial samples. We compare the retraining accuracy and efficiency between our modularized training approach and traditional approach. Both aim to force the original model to perform well on those adversarial samples.

Model Repair Task. In this experiment, given a trained model m, we mutate m by progressively replacing its model weights with random values until its training accuracy drops by about 10%, 30%, and 50%. For each mutation configuration (i.e., a model architecture, a dataset, and a dropping accuracy rate), we generate 3 model mutants. For each model mutant, we compare DeepArc with traditional model retraining regarding the retraining accuracy and efficiency.

2) Results: Table III shows how the modularized training technique in DeepArc framework helps to accomplish the model enhancement task (a.k.a., adversarial training task) and the model repair tasks. Note that the models are trained on CIFAR-10, more details are available at [25].

Model Enhancement Task. As shown in Table III, modularized training of ResNet110 reduces trained model size from 1.73M to 0.13M, and decreases training time by about 32.4% with 2.1% accuracy loss. Across all the models, modularized training can reduce training time by 41.8% on average with comparative accuracy. Overall, compared to fully training the whole model, DeepArc allows us to achieve comparable model enhancement performance by only training a small subset of neural layers.

Model Repair Task. In the model repair task, DeepArc achieves a similar performance boost as in model enhancement task. We can also see that, even trained with a localized modular, DeepArc allows us to successfully achieve on average 0.976 training accuracy and reduce training time by 23.9%.

Answer to RQ4: In summary, (1) model re-adaption tasks are not necessary to retrain the whole model, and (2) localized/modularized training can boost the training efficiency with little accuracy loss.

F. Threats to Validity

External Validity. Threats to external validity arise from that our experiments are conducted on three model architectures and three datasets. To mitigate the risk, we included three most popular model architectures in the computer vision domain. These models including fully connected layers, convolutional layers, residual blocks, etc., are therefore already widely representative for DNNs. In the future, we will conduct more thorough studies to evaluate whether the effectiveness of DeepArc can be generalized to more diversified model architecture such as Transformer and BERT in NLP domains.

Internal Validity. Threats to internal validity may arise from the randomness of model training. To mitigate this threat, we prepared 10 seeds to retrain the models for the experiment.

IV. DISCUSSION

We have shown that the semantic architecture of DeepArc is designed to inform (1) the number of actual transformation operations in the network, (2) a hint to statically remove or add model layers without radically changing model semantics, and (3) modularized layers that can be compressed by model restructure and attributed by model re-adaption tasks, just like a function in the functional program that can be attributed by bug-fixing and feature-enhancement tasks.

Nevertheless, the modularization in this work is still in the grain of transformation operations. While it is useful for tasks like refactoring and performance enhancement, it is not sufficient for module reuse, in comparison with the software modules in the traditional system. In practice, there are still many problems to be solved in reuse tasks. For example, the existing methods for reuse [13], [14] aim to extract sub-networks, which change the model structure in a significant way. In this work, the module interfaces of the semantic architecture are mainly used for encapsulation and localized retraining. In future work, we will explore how to extract human-readable interfaces, so that we can facilitate module-level model reuse.

V. RELATED WORK

A. Software Architecture Recovery

Researchers have investigated the software architecture recovery for decades. The solutions are generally based on clustering, where the similarity measurement is defined based on concerns, code hierarchy or information entropy, which include Comprehension-Driven Clustering (ACDC) [34], Architecture Recovery using Concerns (ARC) [35], Bunch [36], scalable InforMation BOttleneck (LIMBO) [37], Weighted Combined Algorithm (WCA) [38], and Zone-Based Recovery (ZBR) [39]. Based on the recovered code architecture, programmers and architects can have a broader overview and...
Deeper insight into the software system. For example, Wong et al. use a design structure matrix (i.e., DSM) \[40\] to extract the architecture and suggest potential design violations.

Unlike traditional software systems, the deep learning network is simpler in architecture (i.e., a sequence of transformation operations) while much harder to interpret. Our DeepArc framework takes the first step towards the modularity of the network layers. Moreover, the usefulness of our recovered modularity is not limited to comprehension, it can also facilitate model restructuring and model re-adaptation tasks.

B. Modularity of Neural Network

The research on the modularity of the neural network model remains relatively under-explored. Although there are many DNN-related works that introduces the concept of modules \[41\]–\[46\] in the AI community, these mainly focus on formulating some specific functional DNN components for better interpretability in some specific domains. There are emerging works in the SE community to decompose a neural network for improving the model reusability. Pan et al. propose to decompose a multi-classification model into a set of binary classification models for fully connected networks \[13\] and convolutional networks \[4\]. These works extract the same number of multiple sub-networks as categories, by removing and modifying nodes and edges in the neural network. However, these binary classifiers are not conducive to acceleration of training and inference due to the discrete nature of the sub-networks, which is especially costly for tasks with a particularly large number of categories. Besides, these techniques can provide limited support if we need to restructure the model while preserving its behaviors or improving performance with the minimum modifications. Our modular approach operates directly on model layers, scaling the depth of the network without affecting the acceleration. Yet, the approach is designed to tune the network itself rather than extract new networks, and hence it can help with the maintenance of the model itself.

VI. Conclusion and Future Work

In this work, we started a first attempt to encapsulate a monolithic DNN into several consecutive modules and investigate whether it can be beneficial. We propose the framework DeepArc which allows us to extract the semantic architecture of a DNN, facilitating applications such as model restructuring and enhancement tasks. We show that the architecture has some properties, largely preserving the fitting, robustness and linear separability. Besides, the tasks like model compression, enhancement, and repair can largely benefit from the modularity of network layers. In the future, we will further explore whether DeepArc framework can help modular reuse.

Acknowledgment

This research is supported in part by the National Nature Science Foundation of China (grant No.61832014, No.61972373), the Basic Research Program of Jiangsu Province, China (BK20201192), the Minister of Education, Singapore (T1-251RES1901, T2EP20120-0019, MOET32020-0004), NUS-NCS Joint Laboratory for Cyber Security, Singapore, the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity Research and Development Programme (Award No. NRF-NCR_TAU_2021-0002) and A*STAR, CISCO Systems (USA) Pte. Ltd and National University of Singapore under its Cisco-NUS Accelerated Digital Economy Corporate Laboratory (Award I21001E0002). Dr. Xue’s research is also supported by CAS Pioneer Hundred Talents Program.

References


