Decomposing Convolutional Neural Networks into Reusable and Replaceable Modules

Rangeet Pan
rangeet@iastate.edu
Dept. of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, IA, USA

Hridesh Rajan
hridesh@iastate.edu
Dept. of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, IA, USA

ABSTRACT

Training from scratch is the most common way to build a Convolutional Neural Network (CNN) based model. What if we can build new CNN models by reusing parts from previously build CNN models? What if we can improve a CNN model by replacing (possibly faulty) parts with other parts? In both cases, instead of training, can we identify the part responsible for each output class (module) in the model(s) and reuse or replace only the desired output classes to build a model? Prior work has proposed decomposing dense-based networks into modules (one for each output class) to enable reusability and replaceability in various scenarios. However, this work is limited to the dense layers and based on the one-to-one relationship between the nodes in consecutive layers. Due to the shared architecture in the CNN model, prior work cannot be adapted directly. In this paper, we propose to decompose a CNN model used for image classification problems into modules for each output class. These modules can further be reused or replaced to build a new model. We have evaluated our approach with CIFAR-10, CIFAR-100, and ImageNet tiny datasets with three variations of ResNet models and found that enabling decomposition comes with a small cost (2.38% and 0.81% for top-1 and top-5 accuracy, respectively). Also, building a model by reusing or replacing modules can be done with a 2.3% and 0.5% average loss of accuracy. Furthermore, reusing and replacing these modules reduces CO_2e emission by ~37 times compared to training the model from scratch.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Abstraction and modularity**.

KEYWORDS

deep neural network, modularity, decomposition

ACM Reference Format:
Rangeet Pan and Hridesh Rajan. 2021. Decomposing Convolutional Neural Networks into Reusable and Replaceable Modules. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ICSE 2022, May 21-29, 2022, Pittsburgh, PA, USA

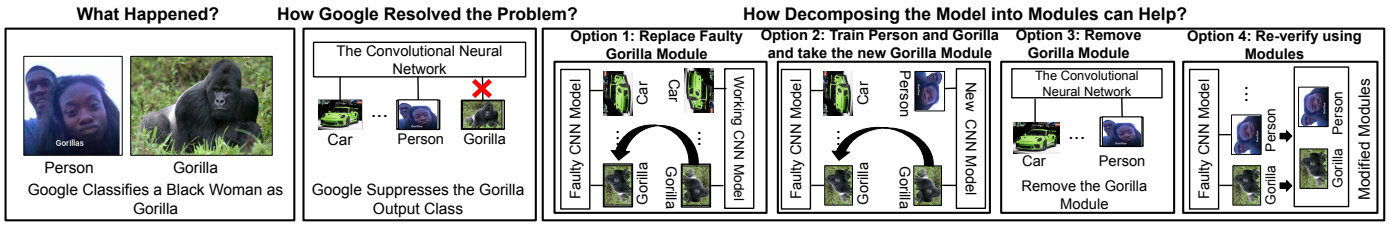
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Deep learning is increasingly being used for image segmentation, object detection, and similar computer vision tasks. With the need for bigger and complex datasets, the size and complexity of models also increase. Often, training a model from scratch needs several hours or even days. To ease the training process, layer architecture [22, 30], transfer learning [32, 35], one-shot learning [26, 36], few-shot learning [20, 29, 31, 37], etc., have been introduced. With these techniques, model structure, parameters can be reused for building a model for similar or different problems. However, in all cases, retraining is needed. Also, there may be scenarios, as shown in Figure 1, where a faulty section of the network needs to be amputated or replaced. Now, suppose we can decompose a Convolutional Neural Network (CNN) into smaller components (modules), where each component can recognize a single output class. In that case, we can reuse these components in various settings to build a new model, remove a specific output class, or even replace an output class from a model without retraining. In the past, modular networks [2, 12, 16], capsule networks [15, 27], etc., have been studied to train the network and incorporate memory into the learning process. But, these modules are not created for enabling reusability and replaceability. As others have noted [23], there are strong parallels between deep neural network development now and software development before the notion of decomposition was introduced [8, 24, 33], and developers wrote monolithic code that can be reused or replaced as easily.

Recently, decomposition has been used to enable reuse and replacement in dense-based networks [23]. This work shows that decomposed modules can be reused or replaced in various scenarios. However, this work focused on dense-based networks and did not explore a more complex set of deep learning techniques, e.g., convolutional neural networks. This work [23] relies on the dense architecture of the models, where nodes and edges (weight and bias) have a one-to-one relationship. In contrast, edges in CNN are shared among all the input and the output nodes.

In this paper, we propose an approach to decompose a CNN model used for image classification into modules, in which each module can classify a single output class. Furthermore, these modules can be reused or replaced in different scenarios. In Figure 1, we illustrate an issue faced by users while using the Google Photo app. We show how Google has resolved the problem and how replacing and reusing decomposed modules can be applied. In the past, Google Photo App tagged a black woman as a gorilla [34]. To resolve this issue, Google decided to remove the output class "Gorilla" by suppressing the output label [34]. Precisely, they have suppressed the output for "Chimpanzee", "Monkey", "Chimp", etc. Though the problem can be temporarily solved by suppressing the



Strategies	Description	Pros	Cons
Option 1	We decompose the faulty CNN model into modules for each output class. Each such module is a binary classifier that classifies whether an input belongs to the output class (for which the module is decomposed) or not. Now, suppose we have another working model trained with the same or a subset of the dataset with the gorilla and person output labels. This new model does not exhibit the behavior present in the faulty one. Then, we can replace the faulty gorilla module with the working gorilla module.	Since the module belong to the same dataset as the faulty model, the module will have sufficient information to both recognize and distinguish new inputs.	These models are massive, and training is very sufficiently costly. Thus, the availability of a similar trained model with the Gorilla class may not be feasible for all conditions.
Option 2	We decompose the faulty model into modules. Moreover, we train a new model with a person and gorilla classes and validate that the trained model does not demonstrate faulty behavior. We decompose the newly trained model into two modules (gorilla and person) and replace the faulty gorilla module with the new decomposed gorilla module.	Training with a small dataset requires less resources in comparison to training the whole dataset.	Decomposition can not alone solve the problem. However, the decomposition bundled with the traditional training could help in this situation.
Option 3	We decompose the faulty model into modules and remove the gorilla module from the collection. In this scenario, there is no cost of retraining involved.	This method is cost-effective.	The actual problem of faulty classification has not been addressed.
Option 4	We can also reuse the person and gorilla module from a working model without the faulty behavior. If any input is classified as a person or gorilla using the faulty model, we reuse the working modules to verify it further.	This approach is cost effective and used for further verification.	Since this approach involves another layer of verification using modules, there is an overhead present.

Figure 1: How Decomposing a Convolution Neural Network into Modules Can Help Solve a Real-life Problem.

output label, to fix the model, one needs to retrain the model. We propose four different solutions based on reusing and replacing decomposed modules. We discuss each approach, its pros, cons, and illustrate how retraining the model can be avoided.

The key contributions of our work are the following:

- We introduce a technique to decompose a CNN model into modules, one module per output class.
- We describe an approach for reusing these decomposed modules in different scenarios to build models for new problems.
- We replace a part of the model with decomposed modules.
- We evaluate our approach against the CO_2e consumption of models created from scratch and reusing or replacing decomposed modules.

Results-at-a-glance. Our evaluation suggests that decomposing a CNN model can be done with a little cost (-2.38% (top-1) and -0.81% (top-5)) compared to the trained model accuracy. Reusing and replacing modules can be done when the modules belong to the same datasets (reusability: +2.5%, replaceability: +1.1% (top-1) and +0.8% (top-5)) and the different datasets (reusability: -7.63%, replaceability: -2.7% (top-1) and -6.9% (top-5)). Furthermore, enabling reusability and replaceability reduce CO_2e emission by 37 times compared to training from scratch.

Outline. In §2 we describe the related work. Then in §3, we discuss our approach to decompose a CNN model into modules. In §4, we answer three research questions to evaluate the cost of decomposition, the benefit of reusing or replacing the modules, and the resource consumption. Lastly, in §5, we conclude.

2 RELATED WORKS

There is a vast body of work [1, 3–5, 7–11, 21, 24, 25, 33] on software decomposition that has greatly influenced us to decompose CNN model into reusable and replaceable modules.

The closest work is by Pan and Rajan [23], where the dense-based model has been decomposed into modules to enable reuse and replacement in various contexts. Though this work has motivated to decompose a CNN model into modules, the dense-based approach cannot be applied due to: 1) the shared weight and bias architecture in convolution layers, and 2) support for layers other than dense. Also, this work did not evaluate the impact of decomposition on CO_2e emission during training.

Ghazi *et al.* [12] have introduced modular neural networks to incorporate memory into the model. They have proposed a hierarchical modular architecture to learn an output class and classes within that output class. Though this work has been able to increase the interpretability of the network by understanding how inputs are classified, the modules are not built to enable reusability or replaceability. Other works on modular networks [2, 16] have learned how different modules communicate with others to achieve better training. Also, capsule networks [15, 27] can be utilized to incorporate memory into deep neural networks. In capsule networks, each capsule contains a set of features, and they are dynamically called to form a hierarchical structure to learn and identify objects. However, modules decomposed by our approach can be reused or replaced in various scenarios without re-training.

3 APPROACH

In this section, we provide an overview of our approach for CNN model decomposition. We discuss the challenges in general. We also discuss each step of decomposing a CNN model into modules.

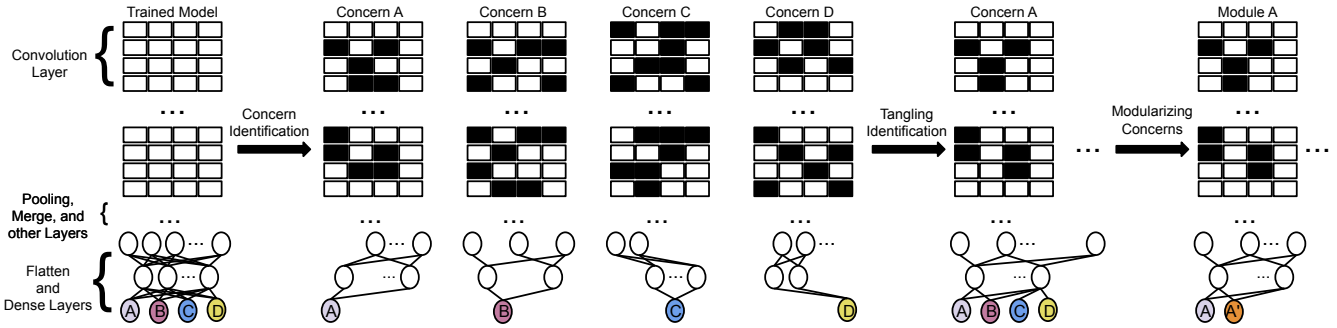


Figure 2: High-level Overview of Our Approach. Inactive nodes are denoted by black boxes.

Figure 2 shows the steps for decomposing a CNN model into modules. Our approach starts with the trained model and identifies the section in the CNN that is responsible for a single output class (Concern Identification). Since we remove nodes for all non-concerned output classes, the identified section acts as a single-class classifier. To add the notion of the negative output classes, we add a limited section of the inputs from unconcerned output classes (Tangling Identification). Finally, we channel the concerns to create a module(s) (Concern Modularization). In the example, we show the decomposition of the module for the output class A. Here, a model trained to predict four output classes has been decomposed into four modules. Each one is a binary classifier that recognizes if an input belongs to the output class. In this paper, we use *concerned* and *unconcerned* as terminologies that represent the input belonging to the output class for which module has been created and all the other output classes, respectively. For example, in Figure 2, we show a module that is responsible for identifying output class *A*. For that module, output class *A* is the concerned output class, and other output classes, e.g., *B*, *C*, and *D*, are the unconcerned classes.

3.1 Challenges

In the prior work on decomposing dense-based models, modules were created by removing edges. There, the value of the node has been computed. If it is ≤ 0, then all incoming and outgoing edges are removed. In a typical dense-based model, the first layer is a flatten layer that converts an input into a single-dimensional representation. From there, one or more dense layers are attached sequentially. In each such dense layer, nodes are connected with two different edges, 1) an edge that connects with other nodes from the previous layer (weight) and 2) a special incoming edge (bias). However, for a *Convolution* layer, this is not the case. Figure 3 illustrates a traditional *Convolution* layer. In that figure, on the left side, we have input nodes. Weight and bias are shown in the middle, and finally, on the right side, we have the output nodes. Each node in the input is not directly connected with the weight and bias, rather a set of nodes (sliding window) are chosen at a time as the input, and the output is computed based on that. The weight and bias are the same for all the input blocks. Due to these differences in the underlying architecture of a *Convolution* layer, the dense-based decomposition approach could not be applied to a *Convolution* layer directly. In the next paragraphs, we discuss each of such challenges.

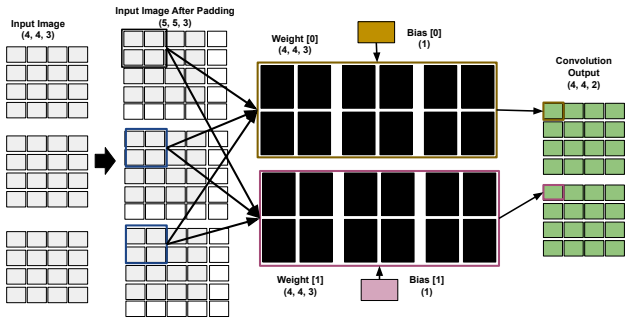


Figure 3: Architecture of a Convolutional Layer.

Challenge 1: Shared Weight and Bias. The removal of edges in dense-based layers has been done to forcing the value of the nodes that are not needed to be 0 and eventually remove them from the decision-making process in the module. This approach is possible in dense layers because there is a one-to-one relationship between two nodes that belong to subsequent layers. If we remove a part of the weight and bias for one node in the *Convolution* layer, the weight and the bias will be turned off for all other nodes as well, and there will not be any output produced.

Challenge 2: Backtracking. The prior work channels the concerns to convert the module into a binary classification problem. However, before channeling the output nodes, a backtracking approach has been applied that starts with removing nodes at the last hidden layer that are only connected to the unconcerned nodes at the output layers and backtrack to the first hidden layer. However, the same approach cannot be directly applied to the CNN models due to the differences in the architecture of the convolutional layers and other supporting layers, e.g., *Pooling*, *Merge*.

Challenge 3: Loss of Accuracy in Inter-Dataset Scenarios. Prior work evaluated their approach by reusing and replacing modules that belong to different datasets. It has been found that such scenarios involve a non-trivial cost. Since these modules involved in the reuse and replace scenario do not belong to the same datasets, they are not programmed to distinguish between them. To remediate the cost, we propose a continuous learning-based approach that enables retraining of the decomposed modules.

3.2 Concern Identification

Concern Identification (CI) involves identifying the section of the CNN model, responsible for the single output class. As a result of this process, we remove nodes and edges in the model. In traditional CNN models for image classification, both convolution, and dense layers have the notion of node and edges, and we discuss the concern identification approaches for both the layers.

Dense Layers: Here, the concerned section can be identified by updating or removing the edges connected to the nodes. In a dense-based network, there is a one-to-one relationship between the edges connecting the nodes from different layers (including the bias nodes). For each edge, the originating and the incident nodes are unique except for bias, where the incident node is unique. The edges between nodes are removed or updated based on the value associated with the nodes. In this process, we identify nodes based on the assumption that ReLU has been used as the activation function. Since our work is focused on image-based classification models, ReLU is the most commonly used activation function for the hidden layers. Also, prior work on decomposition [23] has been carried out with the same assumption.

First, we compute the value associated with the nodes by applying training inputs from the concerned output class. For an input, if the computed value at a node is ≤0, then we remove all the incident and originated edges. We do that for all the concerned inputs from the training dataset. If the value associated with a node is >0 for some input and ≤0 for other inputs, we do not remove that node. For instance, for a layer L_d, there are n_{L_d} number of nodes, and the preceding and the following layer has $n_{L_{d-1}}$ and $n_{L_{d+1}}$ nodes, respectively. For any node at the layer L_d, there will be $n_{L_{d-1}}$ incident edges and $n_{L_{d+1}}$ outgoing edges. Based on our computation, if a node n_i is inactive (value ≤0), then all the incoming and outgoing weight edges ($n_{i(L_{d-1})}$ + $n_{i(L_{d+1})}$) and one bias edge incident to n_i will be removed. We do the same for all the hidden dense layers.

Convolution Layers: In a convolutional layer, we identify the inactive sections in the output nodes by using a mapping-based technique that stores the position of the nodes that are not part of a module. In Algo.1, we describe the steps involved in building the map and storing the nodes' positions. First, we store the weight and bias for all the layers. Then, we identify the parts of the convolution layer that are not used for a single output class. We start by computing all possible combinations of sliding windows at line 12. To build the sliding windows, we use the *stride*, *padding* as input. Below, we describe each such parameter.

Sliding Window. In convolutional layers, instead of one input node at a time, a set of nodes are taken as an input for computation. For instance, in Figure 3, the blue box is a sliding window. For each sliding window, one or more output nodes are created based on the size of the shared weight in the layer.

Padding. Two variations of padding is possible in CNN, zero-padding and with-padding. In zero-padding, the input is not changed. For with-padding, the input is padded based on the size of the sliding window, and the size of the output will be the same as the input. For the example shown in Figure 3, we used the with-padding, and that adds padding with value zero and transforms the input into (5, 5, 3) size (the white boxes are the added padding).

Algorithm 1 Concern Identification (CI).

```
1:  procedure Initialization(model)
2:      convW, convB =[]
3:      for each layer ∈ model do                          ▷ Retrieve the weight and bias
4:          if layer_type=="Convolution" then
5:              convW.add(layer.Weight); convB.add(layer.Bias);
6:          else
7:              if layer_type=="Dense" then
8:                  denseW.add(layer.Weight); denseB.add(layer.Bias);
9:      return convW, convB, denseB, denseW
10: procedure CILayer (model, input, convW_Layer, convB_Layer, convMap_Layer, pad=with, Stride=1, first=False)
11:     I=input
12:     sliding_w=procSliding(I, pad, (Stride, Stride))    ▷ Sliding window
13:     Output=sliding_w * convW_Layer + convB_Layer
14:     flatOutput=flatten(Output)                         ▷ Convert into an 1-D array
15:     for j = 0 to j = |flatOutput| do
16:         if first then
17:             if flatOutput[j]<=0 then                   ▷ Identify the inactive nodes
18:                 convMap_Layer.add(j)
19:         else                ▷ Remove the inactive node if it is active for other inputs
20:             if j ∈ convMap_Layer then
21:                 if flatOutput[j]>0 then
22:                     index=findIndex(flatOutput, j)
23:                     temp=[]
24:                     for k = 0 to k = convMap_Layer do
25:                         if k! = index then
26:                             temp.add(convMap_Layer[k])
27:                     convMap_Layer=temp
28:     return convMap_Layer, Output
29: procedure CI (model, input, convMap)
30:     convW, convB, denseB, denseW=initialization(model)
31:     for each layer ∈ model do                          ▷ Perform CI for all the layers
32:         count=0
33:         if layer_type=="Convolution" then
34:             convMap[count], output = CILayer((model, input, convW[count], convB[count], convMap[count], pad=layer_pad, Stride=layer_stride))
35:         if layer_type="BatchNormalization" then
36:             output = BatchNorm((model, Gamma))
37:         if layer_type=="AveragePooling" then
38:             output = AvgPool((model, Pool_Size))
39:         if layer_type="MaxPooling" then
40:             output = MaxPool((model, Pool_Size))
41:         if layer_type="Add" then
42:             output = input+Previous_input              ▷ Add both the layers that are merged
43:         if layer_type="Dense" then
44:             output = denseMod(model, input, indicator=False, denseW[count], denseB[count])   ▷ Apply dense-based CI
45:         if layer_type="Flatten" then
46:             output = flatten(input)                    ▷ Apply flatten-based CI
47:         input=output
```

Stride. This parameter controls the movement of the sliding window while computing the output. Stride along with the padding decides the output of the layer.

Once we compute the sliding windows (line 12), we feed inputs from the training dataset to our approach and observe the output value of that particular convolution layer. At line 13, we compute the output of the convolution layer based on S*W + B, where S, W, and B denote the sliding window, shared weight, and bias, respectively. Then, we monitor the value at each node (line 14-28). If a node has a value ≤ 0, we store the position of the node in our map. We initialize the map with all the nodes (for the first input) that have value ≤ 0 (line 16-18). The *first* flag is used to denote this first input to the module. Then, we remove the nodes that are previously in the

mapping list, but the nodes have a positive value for the input under observation (line 20-27). We perform such operations to identify the section of the layer that is inactive for a particular concern. For the batch normalization layer, there is no weight or bias involved, and the layer is utilized for normalizing the input based on the values learned during the training session. Max pooling and average pooling are utilized for reducing the size of the network using the pool size. For merge or add layer, we add the value computed from the two layers connected with this layer.

3.3 Tangling Identification

In concern identification, a module is created based on identifying the nodes and edges for a single output class. Since all the nodes and edges related to the other output classes in the dataset have been removed, the module essentially characterizes any input as the concerned output class or behaves as a single-class classifier. To add the notion of unconcerned output classes and able to distinguish between the concerned and unconcerned output classes, we bring back some of the nodes and edges to the module. In Tangling Identification (TI), a limited set of inputs belonging to the unconcerned output classes have been added. Based on the prior work, we add concerned and unconcerned inputs with a 1:1 ratio. For instance, if we have a problem with 200 output classes and build a module based on observing 1000 inputs from the concerned class, then we observe 5 inputs from each unconcerned class (5x199=995≈1000).

3.4 Modularizing Concerns

So far, we identify the section of the network that is responsible for an output class and added examples from unconcerned output classes. However, the module is still an n-class classification problem. We channel the output layer for each module to convert that into a binary classification-based problem. However, before applying the channeling technique, we remove irrelevant nodes (do not participate in the classification task for a module) using a bottom-up backtracking approach.

Dense Layers: We channel the out edges as described by the prior work [23]. Instead of having n nodes at the output layer (L_d, where d is the total number of dense-based layers), two nodes (concerned and unconcerned output nodes) have been kept. For instance, the concerned output class for a module is the first output class in the dataset. We have n output classes and n nodes ($V_1, V_2, \ldots, V_n$) at the output layer. Also, the layer preceding the output layer (L_{d-1}) has $n_{L_{d-1}}$ nodes. For each node at the output layer, there will be $n_{L_{d-1}}$ incident edges. For instance, the incoming edges for V_1 node will be $E_{11}, E_{21}, \ldots, E_{n_{L_{d-1}}1}$, where $E_{n_{L_{d-1}}*1}$ (in this case, n=1) denotes that an edge is connected with $n^{th}_{L_{d-1}}$ node from L_{d-1} layer and the first node at the output layer. For the module responsible for identifying the first output label, the edges incident to the first node (as the concerned node is V_1) at the output layer have been kept intact. However, all the other edges are modified. All the edges incident to any of the unconcerned nodes ($V_2, V_3, \ldots, V_n$) at the L_d layer will be updated by a single edge. The assigned weight for the updated edge is the mean of all the edges (same for bias). Then, that updated edge has been connected to a node at the output layer, which is the unconcerned node for the module. For a module, there will be two nodes at the output layer, V_c and V_{uc}, where V_c and V_{uc} denote the concerned node and the unconcerned node. All the updated edges will be connected to the V_{uc}.

Modularizing Concern: Backtrack: Once the nodes at the output layer are channeled, we backtrack the concerned and unconcerned nodes to the previous layers. In this process, we remove the nodes that only participate in identifying the unconcerned classes. First, we discuss the backtracking approach to handle the dense layers and other non-dense layers, and then we describe how we can backtrack till the input to reduce the size of the module. In all approaches, we support the other layers, e.g., pooling, merge, flatten.

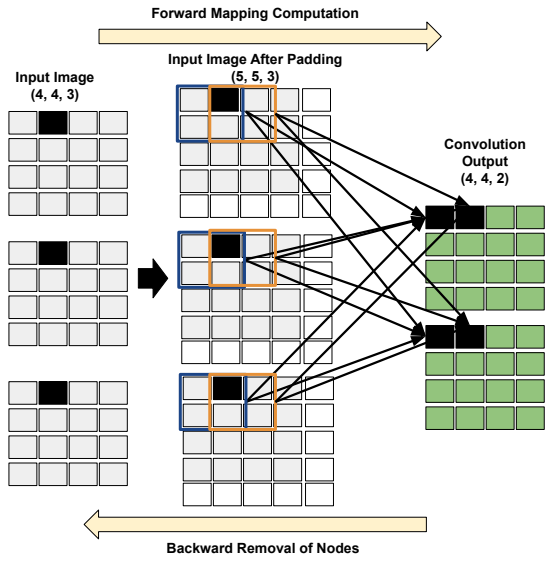


Figure 4: Backtrack Through a Convolution Layer.

Modularizing Concern: Backtrack To Last Convolutional Layer (MC-BLC). Once we channel the output layer, we prune the network based on the inactive nodes at the dense layers. In a typical CNN model, either a *Pooling* layer, *Convolution* layer, or a *Merge* layer will precede *Dense* layers and the *Flatten* layers In this approach, we leverage that information to backtrack through the *Dense* layers (including *Flatten* layer) to the last convolution layer(s) (based on the presence of *Merge* layer or not) or the pooling layer. In this process, we identify the nodes that have edges $E = \{E_{ij}; i \in L_{d-1}, j \in L_d, j \in V_{l(uc)}\}$, where all the edges are only connected to the unconcerned nodes (V_{uc}) at the d^{th} layer (line 4-7). We start the backtracking process from the output layer and move through the dense layer. For each layer, we identify the nodes connected to the unconcerned nodes in the following layer and remove them from the model. Also, we tag the removed nodes as the unconcerned nodes for that layer. To identify the nodes that strongly contribute to the unconcerned node, we introduce a constant (δ) to verify the value associated with the node. Based on the experimental evaluation, we used $\delta = 0.5$. Then, we backtrack the nodes at the layer preceding the output layer is identified at line 8-14. Finally, we backtrack to the flatten layer. In a traditional CNN model for image classification, the flatten layer is preceded

Algorithm 2 Modularing Concern: Dense-Based Backtrack (MC-DBB).

```
1: procedure CMDBB(model, num_of_labels, module_class, UpdatedW, UpdatedB)
2:     temp, tempL, tempPool = []        ▷ UpdateW: Weight for Dense Layers
3:     DenseLength = |UpdatedW|   ▷   ▷ UpdateB: Bias for Dense Layers
4:     for i = 0 to i = UpdatedW[DenseLength − 1][0] do
5:         if UpdatedW[DenseLength-1][i,v_uc]<=-δ and UpdatedW[DenseLength-1][i,v_c]>=+δ then
6:             temp.add(i)        ▷ v_c: concerned nodes, v_uc: unconcerned nodes
7:     tempL.add(temp)
8:     for i = DenseLength − 1| to i = 0 do        ▷ Backtrack till the first Dense
9:         temp = []
10:        for j = 0 to j = UpdatedW[i − 1][0] do
11:            if j ∈ tempL[i] then
12:                if UpdatedW[i-1][j:,]<=-δ then
13:                    temp.add(i) ▷ Add nodes that are only connected with v_nc
14:        tempL.add(temp)
15:    for i ∈ tempL[|tempL − 1|] do
16:        for j = 0 to j = i ∗ poolsize^2 + j do        ▷ Convert to pooling layer
17:            if Layer-1=="Convolution" then
18:                if j not ∈ convMap[depth] then
19:                    convMap[depth].add(j)        ▷ Update convolution layer map
20:            if Layer-1=="Add" then        ▷ Update two convolution layer maps
21:                if j not ∈ convMap[depth] then
22:                    convMap[depth].add(j)
23:                depthAdd2=Add.Input2
24:                if j not ∈ convMap[depthAdd2] then
25:                    convMap[depthAdd2].add(j)
```

Algorithm 3 Modularizing Concern: Convolution-Based Backtrack (MC-CBB).

```
1: procedure Sliding_Window_Mapping(input, W, pad, stride)
2:     mapping =[], count=0    ▷ Performs the forward pass and map input output nodes
3:     temp = zeros_like(input)
4:     temp = temp.flatten()
5:     for i = 0 to i = |temp| do
6:         temp[i] = i + 1
7:     temp = temp.flatten()
8:     sliding_window = sw(temp, W, pad, stride)
9:     for i = 0 to i = |sliding_window| do
10:        for j = 0 to j = W.length[3] do
11:            mapping.add(temp[i][j], count)
12: procedure Conv_Backtrack(input, W, pad, stride, B, preceeding_layer, deactive_map)
13:    convDepth = depth
14:    mapping_window = Sliding_Window_Mapping(input, W, pad, stride, B)
15:    source_mapping = mapping_window[: 0]        ▷ Input nodes
16:    sink_mapping = mapping_window[: 1]        ▷ Output nodes
17:    for each deactived_node ∈ deactive_map do
18:        source=source_mapping[sink_mapping==deactived_node]
19:        flag = True        ▷ Identify the source, where sink is deactive
20:        source_mapping = source_mapping[source_mapping] > 0]
21:        for source_node ∈ source_mapping do
22:            if flag == true then:
23:                sink_node=sink_mapping[source_mapping==source_node]
24:                if |sink_node − deactive_map| > 0 then
25:                    flag = false ▷ All sinks formed by source are not deactive
26:            if flag == true then
27:                for each source ∈ source_mapping do
28:                    if source not ∈ deactive_map then
29:                        deactive_map[−1].add(source + 1)        ▷ Update Map
30:                    if preceeding_layer=="Add" then
31:                        deactive_map[−2].add(source + 1)        ▷ Update Map
```

by a pooling layer, or a merge layer, or a convolutional layer. If the preceding layer is a convolution layer, we update the mapping (as discussed in §3.5) for that particular convolutional layer. Since the convolution layer's output is directly reshaped into the flatten layer, there is a one-to-one relationship between the two layers. If the preceding layer is a pooling layer, then there will be X^2 (X is the pool size) inactive nodes at the pooling layer for one inactive node at the flatten layer. If the preceding layer is a merge layer, then the convolution layers that are merged will be updated.

Modularizing Concern: Backtrack To Input (MC-BI). In the previous approach, we can only backtrack from the output layer to the last convolution layer. However, we cannot backtrack through the convolution layer. In a convolution layer, the input nodes cannot be directly mapped with the output nodes. For instance, in Figure 4, the input image shown on the left side is turned into the image shown in the middle, which is after adding the paddings (for this example, we choose *Valid* padding). In the output, the nodes on the top left corner for both arrays will be produced by the first sliding window (blue box) from each array shown in the middle. So, for mapping, a node in the output on the right side of the image, at least 4 (in this example, we chose the sliding window size to be 2x2) nodes can be mapped. Those four individual nodes are also mapped with other nodes in the output. The black-colored node in the middle is a part of two sliding windows (the blue box and the orange box). To remove irrelevant nodes from the convolution layer, we take a two-pass approach. First, we store the position of the nodes in each sliding window with the nodes in the output (forward pass). During the forward pass, we store the mapping $M = \{(V_i, V_j); V_j = f(V_i, W, B)\}$, where f denotes the convolution operation. During the backward pass, we remove the nodes.

In Algo. 3, we describe the step to do the mapping. From line 2-11, the forward pass has been described. In the forward pass, we store the mapping between the sliding window and output nodes. In order to denote the position of the sliding window, we mark each node with a unique number (line 5-6) before adding the padding. For padding, the nodes are marked as "0" as they are not present in the input of the *Convolution* layer. Then for each output node, the input nodes are stored in a list. In this process, we define an operation named sw that computes the sliding windows from the weight W, padding *pad*, and stride. Now, we compute the mapping with the input and the output nodes at line 13. Then, we separate the input and the output nodes at line 15 and 16. We scan through the inactivate nodes in the output and identify if they match the pattern as illustrated in Figure 4. We identify all the input nodes that are mapped with an output node and vice versa. We focus on searching nodes that are not part of the padding operation and remove the nodes marked with 0 at line 20. For each such input node, we find all the output nodes generated from the particular input node. If these output nodes are already in the deactivation node list, we add the input node to the deactivation list for the preceding convolution (output of the preceding convolution layer is the input of the next convolution layer). If the previous layer is an add layer, then the two convolution layers that are merged at the merge layer are updated with the changes. If the preceding layer is a pooling layer, the update is carried based on the pool size.

3.5 Continuous Learning

In the prior study, reusing modules that originated from different datasets involves non-trivial cost. Our intuition is that since the

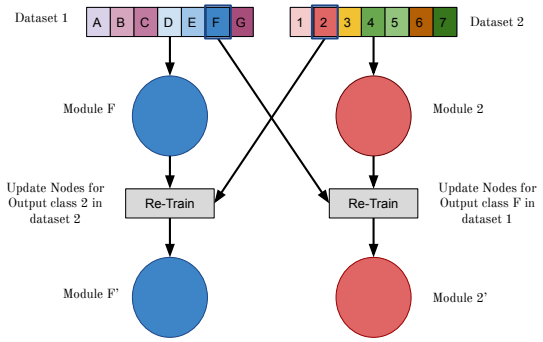


Figure 5: Continuos Learning.

modules are originated from a different dataset, they still have some traits of the parent dataset. In fact, by applying the tangling identification approach, we deliberately add some unconcerned examples to learn the modules on how to distinguish between the concerned and the unconcerned examples. However, in the inter dataset scenarios, the unconcerned output classes are not the same. To solve this problem, we propose a continuous learning-based approach. In deep learning, continuous learning Collobert and Weston [6] has been widely applied. In Figure 5, we illustrate a reuse scenario, where module F is originated from dataset 1 and module 2 is originated from dataset 2. Dataset 1 represents a set of English letters (A-G), and applying decomposition creates modules for each output class. Similarly, dataset 2 represents a set of English digits (1-7), and decomposition creates seven modules, each for one output class. When module F and module 2 are reused in a scenario, based on the prior work, each input belongs to 2, and F will be given as input to the composition of the decomposed modules. However, due to the parent dataset traits, module 2 can recognize itself but does not know how to distinguish from any input belonging to the output class F. To learn the concerned output classes in this scenario, we take the unconcerned section of the dataset. For instance, for module F, the unconcerned output class will be output class 2 from dataset 2. We take the examples from output class 2 from dataset 2 and update module F by removing the nodes responsible for detecting the output class 2. We do the same for module 2, where we remove the nodes responsible for recognizing output class F from dataset 1. Finally, the modified modules are ready to be reused.

4 EVALUATION

In this section, we discuss the experimental settings. Furthermore, we discuss the performance of decomposing the convolutional neural network into modules by answering three research questions, 1) does decomposition involve cost?, 2) how module reuse and replacement can be done compared to retraining a model from scratch?, and 3) does reusing and replacing the decomposed modules emit less CO_2?

4.1 Experimental Settings

4.1.1 Datasets. We evaluate our proposed approach based on three widely used and well-vetted datasets.

CIFAR-10 (C10) [18]: This dataset comprises of 3-D images of different objects. It has 10 output classes, and the dataset is divided into training and testing datasets. The training dataset has 50,000 images, and the testing dataset has 10,000 images.

CIFAR-100 (C100) [18]: This dataset has 100 output classes. This is widely used to evaluate CNN-based studies due to the complexity of the dataset. Here, there are 50,000 training images and 10,000 testing images. The image size is similar to the CIFAR-10.

ImageNet-200 (I200) [19]: This dataset is very popular and widely used to measure the scalability of CNN-based applications. Unlike the previous two datasets, the total number of output classes is 200. ImageNet dataset has been designed to help the computer-vision related task in machine learning. This dataset has 80,000+ nouns (name of the output class), and for each class, there are at least 500 images associated with it. This dataset has been used in training models in real-life scenarios. However, due to the complexity of the dataset, a smaller dataset with similar complexity has been made public for research purposes. This dataset (ImageNet-tiny) comprises 200 types of images. The training dataset has 100000 images, and the testing has 10000 images.

4.1.2 Models. We used the ResNet [13, 14] models to evaluate our proposed approach. In a ResNet model, there are multiple blocks present and each block consists of `Convolution`, `Add`, and `Activation` layer. There are two versions of ResNet, the original version, where there is a stack of `Convolution`, `Add`, and `Batch Normalization` layers are put together to form a residual block, and those residual blocks build the network. In the second version, the residual block is modified to form a bottleneck layer with `Convolution`, `Add`, and `Batch Normalization`. We performed our evaluation against the first version of the network for simplicity and to reduce the training time and resources. Each such version can either have `Batch Normalization` layer or not as described in the original paper. Since `Batch Normalization` is only used to train and does not participate in the prediction, we chose the model without `Batch Normization` to reduce the training time as some of our experiments include training models multiple times. We used ResNet-20, ResNet-32, and ResNet-56, where 3, 5, and 9 residual blocks are present with 21, 33, and 57 convolution layers, respectively. The reported model accuracies are different from the original paper, as we trained the model from scratch with 200 epochs.

4.1.3 Metrics Used. **Accuracy.** We compute the composed accuracy of the decomposed modules as shown in the prior work [23]. Pan *et al.* computed the top-1 accuracy for all the experiments, whereas we compute both top-1 and top-5 accuracies.

Jaccard Index. Similar to the prior work, we compute the variability between the model and the modules using Jaccard Index.

***CO_2e* Emission.** For CO_2e emission, we utilized the metrics used by Strubell *et al.* [28]. The total power consumption during the training is measured as ,

$$p_t = \frac{1.58t(p_c + p_r + gp_g)}{1000} \quad (1)$$

In this equation, p_c, p_r, p_g, and g denote the average power consumption of CPU, DRAM, GPU, and the total number of GPU cores, respectively. The t denotes the time. Here, 1.58 denotes the PUE co-efficient, which is the same value used in the prior work. We

Table 1: Decomposition Effectiveness and the Variability between the Decomposed Modules and the Trained Model.

Model	Acc		CI+TI+MC			CI+TI+MC-BLN			CI+TI+MC-BLN+MC-BI		
	Top 1	Top 5	Top 1	Top 5	JI	Top 1	Top 5	JI	Top 1	Top 5	JI
CIFAR10-R20	89.0%	99.8%	88.8%	96.7%	0.75	88.5%	99.6%	0.75	88.8%	96.5%	0.75
CIFAR10-R32	86.3%	99.5%	79.8%	96.7%	0.51	79.5%	98.2%	0.51	79.8%	96.7%	0.51
CIFAR10-R56	87.7%	99.5%	85.6%	94.1%	0.41	86.3%	99.5%	0.40	86.3%	99.5%	0.40
CIFAR100-R20	50.0%	76.4%	48.6%	75.4%	0.63	49.0%	75.6%	0.62	49.0%	75.6%	0.62
CIFAR100-R32	47.8%	73.8%	41.4%	72.4%	0.53	42.7%	72.4%	0.52	41.6%	71.4%	0.52
CIFAR100-R56	48.0%	77.6%	47.2%	77.0%	0.69	47.2%	77.0%	0.69	47.2%	77.0%	0.69
ImageNet200-R20	34.4%	58.2%	28.8%	58.2%	0.43	33.0%	58.2%	0.43	33.0%	58.2%	0.43
ImageNet200-R32	31.2%	55.2%	20.8%	21.2%	0.71	29.4%	53.6%	0.71	29.4%	53.6%	0.71
ImageNet200-R56	33.6%	57.8%	29.8%	54.8%	0.30	31.0%	56.4%	0.30	31.0%	56.4%	0.30

Acc: Accuracy, CI: Concern Identification, TI: Tangling Identification, MC: Modularizing Concern, DBB: Dense-Based Backtracking, CBB: Convolution-Based Backtracking, and JI: Jaccard Index.

performed our experiment on iMac with 4.2 GHz Quad-Core Intel Core i7 and 32 GB 2400 MHz DDR4 RAM. Since we do not have any GPU, both g and p_g are zero. The power consumption has been measured using Intel Power Gadget [17]. Finally, the CO_2e emission has been computed as,

$$CO_2e = 0.954p_t \tag{2}$$

4.2 Results

In this section, we evaluate our approach to understand the cost involved in decomposition, whether the decomposed modules can be reused or replaced, and how decomposition can be beneficial compared to training from scratch.

4.2.1 Does Decomposing CNN Model into Module Involve Cost? To understand how decomposing CNN models into modules performs, we evaluate our approach on 9 datasets and model combinations. First, we decompose the CNN model into small modules, and then we compose them to evaluate how these composed modules perform compared to the trained CNN models. In Table 1, the composed accuracy of the decomposed modules and the trained models' accuracy have been shown. We report the top-1 and top-5 accuracies of the models. Here, in the first and second columns, we show the top-1 and top-5 accuracy of the model. Whereas, in columns 3-11, the accuracy shown is from the composition of the decomposed modules. While composing the modules, we apply the voting-based approach that is similar to the prior work. Also, we compute the Jaccard index (JI) to identify the average variability between the modules and the model. Lesser value of the JI represents better decomposition as the modules are desired to be significantly different from the model. Suppose the value of the Jaccard index is very high. In that case, it denotes that the module has essentially become similar to the model. While the lower JI is a criterion for better decomposition, the cost is another criteria to be considered while decomposing a model into modules. In this study, our objective is to have the least cost of decomposition with the most dissimilarities between the modules and the model. We found that in all the cases, there is a cost involved while decomposing. For instance, our first approach identifies the concern, adds negative examples, and finally modularizes the concern involves 4.13% and 5.70% (top-1 and top-5) of loss of accuracy with an average Jaccard index 0.55. Whereas applying dense-based backtracking, the loss has reduced. The average loss with this approach is 2.38% and 0.81%, and the average Jaccard index is 0.54. For approach involving the backtrack through the convolution layer includes a loss of 2.43% and 1.43% accuracy with an average 0.54 Jaccard index. For the further experiments, we choose the second approach as the loss is the least of all. Based on these results, we can conclude that decomposition is possible in CNN models. However, it involves a small cost, and the modules produced are significantly different from the models. For further studies, we used the dense-based backtracking technique.

4.2.2 How module reuse and replacement can be done compared to retraining a model from scratch? Here, we evaluate how decomposed modules can be reused and replaced in various scenarios.

Reusability: Table 2, and 3 show two different reuse scenarios: 1) intra-dataset reuse, 2) inter-dataset reuse.

In intra-dataset reuse, we build a model by taking two output classes from a dataset and building a new model. For instance, output classes 0 and 1 have been taken from CIFAR-10, and we evaluate the accuracy by combining the decomposed modules for output classes 0 and 1 (we represent the output class with their index in the dataset). We compare the performance with retraining a model with the same structure with the same output classes and measure the accuracy. We took the best model for each dataset (based on the trained model accuracy) and the modules created from that. For CIFAR-10, CIFAR-100, and ImageNet-200, the best model in terms of training accuracy are ResNet-56, ResNet-20, and ResNet-20, respectively. Since these models take a long time to train, we performed the experiments for 4 output classes randomly chosen from the dataset to evaluate the reusability scenarios. In Table 2, the intra-dataset reuse scenarios are reported. C1, C2, C3, and C4 denote the four randomly chosen classes, and they are different for each dataset. Since we take 2 output classes in each experiment, the total number of choices for a dataset with n output classes under experiment is $\binom{n}{2}$. In our cases, we take 4 output classes from each dataset to evaluate, and the total choices would be $\binom{4}{2}$ or 6. For CIFAR-10, reusing the modules increase 6.6% accuracy, on average. For CIFAR-100, there is an average 0.67% loss, but 4/6 cases reusing the modules perform better than the trained model. For ImageNet, there is an average 1.5% increase in accuracy. Overall, the gain in the accuracy in intra-dataset reuse is 2.5%.

In inter-dataset reuse, we take two output classes from different datasets and build a binary classification-based problem. For example, output classes 1 and 2 have been taken from the CIFAR-10 and CIFAR-100 datasets, respectively. We compute the accuracy the same way as we did for intra-dataset reuse scenario. Since the

Table 2: Intra Dataset Reuse. MA: Composed Module Accuracy, TMA: Trained Model Accuracy, C10: CIFAR-10, C100: CIFAR-100, I200: ImageNet-200, and C{x}: Output Label x

(a) CIFAR-10 and CIFAR-100 Reuse. Yellow and Green represent the CIFAR-10 and CIFAR-100 reuse scenarios, respectively.

C10		C2		C3		C4	
C100	C10	TMA	MA	TMA	MA	TMA	MA
C1	C1	95.4%	**98.8%**	**84.7%**	**91.4%**	**91.9%**	**96.6%**
C2	C2	**91.0%**	**91.0%**	**93.2%**	**99.3%**	**92.2%**	**99.3%**
C3	C3	**96.0%**	**96.0%**	**95.0%**	**98.0%**	**81.2%**	**92.9%**
C4	C4	100%	95.5%	100%	96%	**98.5%**	**100%**
C100		C100-C1		C100-C2		C100-C3	

(b) ImageNet-200 Reuse. Yellow represents the ImageNet-200 reuse scenarios. Black represents the wrong combination of reuses.

I200	C2		C3		C4	
	TMA	MA	TMA	MA	TMA	MA
C1	**90.00%**	**98.00%**	**93.0%**	**94.0%**	**92.0%**	**94.0%**
C2			95.0%	88.0%	**95.0%**	**95.0%**
C3					**89.0%**	**94.0%**
C4						

Table 3: Inter Dataset Reuse.

C10 / C100	C1			C2			C3			C4		
	TMA	MA	MMA	TMA	MA	MMA	TMA	MA	MMA	TMA	MA	MMA
C1	97.8%	83.0%	91.5%	96.7%	87.9%	92.1%	96.6%	86.0%	91.5%	98.8%	87.2%	93.8%
C2	95.5%	82.4%	89.2%	98.6%	87.5%	88.1%	95.6%	85.2%	89.9%	99.7%	84.1%	89.0%
C3	98.4%	78.2%	87.5%	97.9%	79.5%	88.5%	94.6%	80.2%	89.2%	99.6%	81.6%	87.9%
C4	98.0%	88.3%	88.1%	97.2%	88.6%	88.1%	94.7%	86.7%	90.3%	99.0%	88.9%	91.9%

TMA: Trained Model Accuracy, MA: Composed Module Accuracy, and MMA: Modified Module Accuracy.

Table 4: Intra Dataset Replace.

Top-1 Accuracy

Dataset	TMA	Prior MA	RM0	RM1	RM2	RM3	RM4	RM5	RM6	RM7	RM8	RM9
CIFAR-10	86.3%	79.5%	**80.7%**	**80.4%**	**81.1%**	**82.6%**	**81.7%**	**81.8%**	**81.2%**	**80.8%**	**79.9%**	**81.0%**
CIFAR-100	47.8%	42.7%	42.3%	42.5%	**42.7%**	42.5%	42.6%	**43.0%**	42.2%	42.6%	**42.7%**	42.5%
ImageNet-200	31.2%	29.4%	**31.2%**	**30.6%**	**31.0%**	**30.6%**	**30.8%**	**31.4%**	**31.6%**	**31.4%**	**31.4%**	**31.2%**

Top-5 Accuracy

Dataset	TMA	Prior MA	RM0	RM1	RM2	RM3	RM4	RM5	RM6	RM7	RM8	RM9
CIFAR-10	99.5%	98.2%	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**
CIFAR-100	73.8%	72.4%	71.1%	71.0%	71.3%	71.4%	71.3%	71.6%	71.5%	71.5%	71.1%	71.5%
ImageNet-200	55.2%	55.6%	**57.6%**	**57.4%**	**57.2%**	**57.2%**	**56.8%**	**57.4%**	**57.4%**	**57.2%**	**57.4%**	**56.6%**

TMA: Trained Model Accuracy, MA: Composed Module Accuracy, RM{x}: Replace Module x with another Module for x.

Table 5: Inter Dataset Replace. (All results are in %.)

C10 / C100	C1				C2				C3				C4			
	M1	D1	M5	D5	M1	D1	M5	D5	M1	D1	M5	D5	M1	D1	M5	D5
C1	86.2	**87.5**	99.6	96.3	83.1	**85.8**	98.9	90.7	86.1	**89.4**	99.5	91.6	86.7	**89.1**	99.6	91.9
C2	85.3	**87.7**	99.2	95.7	82.7	**85.7**	98.9	90.3	84.6	**89.4**	99.1	90.8	87.6	**88.7**	99.4	91.5
C3	86.2	**87.1**	99.4	95.8	81.8	**85.4**	98.8	90.2	86.2	**88.9**	99.0	91.2	87.9	**88.4**	99.6	91.6
C4	84.7	**87.1**	99.2	95.8	82.6	**85.4**	98.7	89.9	85.0	**89.1**	98.8	91.2	86.6	**88.1**	99.4	91.4

M1 and M5: Top-1 and Top-5 Accuracy for Trained Model D1 and D5: Top-1 and Top-5 Composed Accuracy for Decomposed Modules.

model structure of the decomposed modules for two datasets are different, e.g., the decomposed modules for CIFAR-10 follow the model structure of ResNet-56, whereas it is ResNet-20 for CIFAR-100. The best model in terms of accuracy has been taken for retraining. We did a pilot study and found that ResNet-20 does better in terms of accuracy for the inter-dataset reuse scenario for CIFAR-10 and CIFAR-100 datasets. To overcome overfitting, we store the checkpoints for all training-based evaluations and report the last checkpoint with the best validation accuracy. In Table 3, the inter-dataset reuse scenarios are reported. We found that inter-dataset reuse has a non-trivial cost. In this case, the loss of accuracy is 12.06%. Our intuition is that since these modules originate from a different dataset, they still have some traits of the parent dataset. In fact, by applying the tangling identification approach, we deliberately add some non-concerned examples to distinguish between the concerned and the non-concerned examples. To alleviate the effect, we perform a continuous learning-based approach, where we re-apply the tangling identification with the unseen output class(es). We found that enabling continuous learning can reduce the cost of decomposition significantly. The overall loss while reusing modules is 7.63%, which is a 4.4% gain compared to the previous reusability approach. Since the input size of the CIFAR-10 and CIFAR-100 images are not the same as ImageNet-200, both training a single model and reusing modules cannot be done.

Replaceability: Similar to the reusability scenario, we evaluated both inter and intra-dataset reuses. For intra dataset reuse, we replace a module decomposed from a model with less accuracy with

a module for the same output class decomposed from a model with higher accuracy. In Table 4, we report the evaluation for replacing the module. For each dataset, we evaluate for 10 output classes to match the total number of classes for all the datasets. For CIFAR-10, we replace a module decomposed from ResNet-32 with a module for the same output class decomposed from ResNet-20. We found that the intra-dataset reuse increases the accuracy by 1.1% and 0.8% for top-1 and top-5 accuracy, respectively. In fact, 76.7% (23/30) and 66.7% (20/30) times replacing a module does the better. Although we incorporated the continuous learning approach, there is no significant increase in accuracy (top-1: 0.02% and top-5: 0.01%).

Inter-dataset replacement scenarios are reported in Table 3. To reduce the experiment time, we replace a module from CIFAR-10 with a module taken randomly from CIFAR-100 and report both composed accuracy and the accuracy of the trained model for four different output classes. We found that for top-1 accuracy, there is a 2.5% accuracy increase, and for top-5, there is a loss of 6.9%, on average.

Table 6: Scenarios Created Based on Figure 1.

Strategies	Top-1 Acc	Top-5 Acc
Initial Model	26.6%	50.4%
MA	26.0%	50.4%
Option 1	27.2%	54.6%
Option 2	25.0%	50.4%
Option 3	24.0%	48.2%
Option 4	25.3%	50.8%

Motivating Example. Here, we recreate the example shown in Figure 1. Since human face and Gorilla images are not present in the ImageNet-200 dataset, we added two classes from the ImageNet large dataset. However, there is no class specifically categorized as "human face" in the dataset. So, we take the closest class that contains the images of different persons. First, we build a model with 202 (ImageNet-200 + Person + Gorilla) output classes. For the first option, we create a hypothetical model that does not exhibit faulty behavior. We do that by training a model with more epochs (+100 epochs) and achieve higher accuracy (+1.8%). Then, we decompose the model into modules. Then, we replace the Gorilla module with the module created from the second model. In the second option, we train a model with Person and Gorilla examples and replace the faulty Gorilla module with the newly created one. In the third option, we remove the Gorilla module from the set of modules. Finally, in the fourth option, if the original model predicts Person or Gorilla, we re-verify with the modules created from the 2-class classification model in the second option. The evaluation has been shown in Table 6, and we found that enabling the replacement and reuse of modules can solve the problem. Based on the need and available resources, users can pick any of the four options.

4.2.3 Does reusing and replacing the decomposed modules emit less CO_2e? Strubell *et al.* [28] have identified that training a model often emits six times more CO_2e than a car emits over a year. To understand how decomposition can help to reduce the harmful effect, we measure the CO_2e emission for both intra and inter-dataset reuse and replace scenarios. First, we start computing the power consumption before executing the program. Then, we measure the average power drawn by other tasks running in the background for

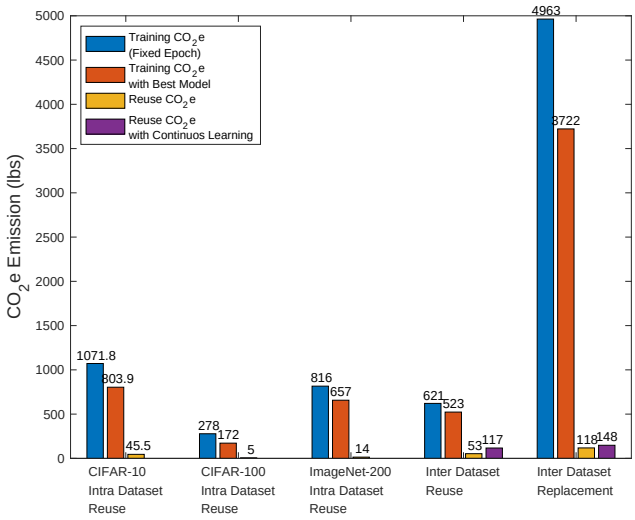


Figure 6: Comparison of CO_2e Emission.

10 sec and compute the average power drawn by CPU and DRAM. We negate these two values to separate the resource consumption of the program and other background tasks. Then, we measure the power consumption for each 100 ms (default for Intel Power Gadget). Figure 6 shows the CO_2e emission for different scenarios. We do not show intra dataset replacement scenarios as that cannot be compared with training from scratch. For training scenarios, we build the model from scratch, and after training, we predict the same set of images. This experimental setting has been done to compare a similar situation, where developers need to build and predict inputs. The value reported in the figure is the average CO_2e consumption for all the experimented scenarios described in §4.2.2. Also, since the epoch has been fixed for each retraining, the power consumption is somewhat fixed for each retraining scenario. However, the best model can be found earlier. For instance, a model is trained with 100 epochs, but the best model is found at the 20th epoch. To remove the effect of overfitting, we compute the power consumed until the return of the best model and report that in the figure. We found that for reuse scenarios, decomposition reduces the CO_2e consumption by 23.7x and 18.3x for the fixed epoch and the best model scenarios, respectively. For replacement scenarios, it is 42x and 31.5x, respectively. If we apply the continuous learning-based approach, there is a slight increase in resource consumption, but still significantly less than training from scratch. Also, we computed the additional CO_2e consumption for the one-time decomposition approach and found that on average 116, 371, and 3800 lbs of CO_2e has been generated for decomposing CIFAR-10, CIFAR-100, and ImageNet-200 models, respectively. The overhead is significantly lower for CIFAR-10 and CIFAR-100 compared to training a new model for both reuse and replace scenarios. However, for ImageNet-200, the overhead is high, but it is a one-time operation that can enable both reuses and replacements at a very low cost.

5 CONCLUSION AND FUTURE WORK

In this paper, we introduce decomposition in convolutional networks and transform a trained model into smaller components or modules. We used a data-driven approach to identify the section in the model responsible for a single output class and convert that into a binary classifier. Modules created from the same or different datasets can be reused or replaced in any scenario if the input size of the modules is the same. We found that decomposition involves a small cost of accuracy. However, both intra-dataset reuse and replaceability increase the accuracy compared to the trained model. Furthermore, enabling reusability and replaceability reduces CO_2e emission significantly. For this work, we omit the heterogeneous inputs (the input size for modules are not the same) while reusing and replacing modules, and it will be a good research direction for the future to study how an interface could be built around the modules to take different types of inputs.

REFERENCES

[1] Pierre America. 1990. Designing an object-oriented programming language with behavioural subtyping. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*. Springer, 60–90.

[2] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 39–48.

[3] John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Peter Naur, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, et al. 1960. Report on the algorithmic language ALGOL 60. *Numer. Math.* 2, 1 (1960), 106–136.

[4] John W Backus, Robert J Beeber, Sheldon Best, Richard Goldberg, Lois M Haibt, Harlan L Herrick, Robert A Nelson, David Sayre, Peter B Sheridan, H Stern, et al. 1957. The FORTRAN automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*. 188–198.

[5] Luca Cardelli. 1997. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 266–277.

[6] Ronan Collobert and Jason Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*. 160–167.

[7] Krishna Kishore Dhara. 1997. Forcing behavioral subtyping through specification inheritance. (1997).

[8] Edsger W Dijkstra. 1982. On the role of scientific thought. In *Selected writings on computing: a personal perspective*. Springer, 60–66.

[9] Edsger W Dijkstra. 2001. Go to statement considered harmful. In *Pioneers and Their Contributions to Software Engineering*. Springer, 297–300.

[10] Edsger Wybe Dijkstra et al. 1970. Notes on structured programming.

[11] Matthew Flatt and Matthias Felleisen. 1998. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 236–248.

[12] Badih Ghazi, Rina Panigrahy, and Joshua Wang. 2019. Recursive sketches for modular deep learning. In *International Conference on Machine Learning*. PMLR, 2211–2220.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity mappings in deep residual networks. In *European conference on computer vision*. Springer, 630–645.

[15] Geoffrey E Hinton, Zoubin Ghahramani, and Yee Whye Teh. 2000. Learning to parse images. *Advances in neural information processing systems* 12 (2000), 463–469.

[16] Ronghang Hu, Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Kate Saenko. 2017. Learning to reason: End-to-end module networks for visual question answering. In *Proceedings of the IEEE International Conference on Computer Vision*. 804–813.

[17] Intel Corporation. 2021. Intel Power Gadget. https://software.intel.com/content/www/us/en/develop/articles/intel-power-gadget.html.

[18] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[19] Ya Le and Xuan Yang. 2015. Tiny imagenet visual recognition challenge. *CS 231N* 7 (2015), 7.

[20] Yann Lifchitz, Yannis Avrithis, Sylvaine Picard, and Andrei Bursuc. 2019. Dense classification and implanting for few-shot learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 9258–9267.

[21] Barbara Liskov and Stephen Zilles. 1974. Programming with abstract data types. *ACM Sigplan Notices* 9, 4 (1974), 50–59.

[22] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. 2017. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*. 5058–5066.

[23] Rangeet Pan and Hridesh Rajan. 2020. On decomposing a deep neural network into modules. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 889–900.

[24] David L Parnas. 1972. On the criteria to be used in decomposing systems into modules. In *Pioneers and Their Contributions to Software Engineering*. Springer, 479–498.

[25] David Lorge Parnas. 1976. On the design and development of program families. *IEEE Transactions on software engineering* 1 (1976), 1–9.

[26] Danilo Rezende, Ivo Danihelka, Karol Gregor, Daan Wierstra, et al. 2016. One-shot generalization in deep generative models. In *International Conference on Machine Learning*. PMLR, 1521–1529.

[27] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. 2017. Dynamic routing between capsules. *arXiv preprint arXiv:1710.09829* (2017).

[28] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and policy considerations for deep learning in NLP. *arXiv preprint arXiv:1906.02243* (2019).

[29] Qianru Sun, Yaoyao Liu, Tat-Seng Chua, and Bernt Schiele. 2019. Meta-transfer learning for few-shot learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 403–412.

[30] Xiaochuan Sun, Guan Gui, Yingqi Li, Ren Ping Liu, and Yongli An. 2018. ResInNet: A novel deep neural network with feature reuse for Internet of Things. *IEEE Internet of Things Journal* 6, 1 (2018), 679–691.

[31] Flood Sung, Yongxin Yang, Li Zhang, Tao Xiang, Philip HS Torr, and Timothy M Hospedales. 2018. Learning to compare: Relation network for few-shot learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1199–1208.

[32] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. 2018. A survey on deep transfer learning. In *International conference on artificial neural networks*. Springer, 270–279.

[33] Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton. 1999. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*. IEEE, 107–119.

[34] The Verge. 2021. Google 'fixed' its racist algorithm by removing gorillas from its image-labeling tech. https://www.theverge.com/2018/1/12/16882408/google-racist-gorillas-photo-recognition-algorithm-ai.

[35] Lisa Torrey and Jude Shavlik. 2010. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, 242–264.

[36] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. 2016. Matching networks for one shot learning. *arXiv preprint arXiv:1606.04080* (2016).

[37] Davis Wertheimer and Bharath Hariharan. 2019. Few-shot learning with localization in realistic settings. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 6558–6567.