On Decomposing a Deep Neural Network into 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

Deep learning is being incorporated in many modern software systems. Deep learning approaches train a deep neural network (DNN) model using training examples, and then use the DNN model for prediction. While the structure of a DNN model as layers is observable, the model is treated in its entirety as a monolithic component. To change the logic implemented by the model, e.g. to add/remove logic that recognizes inputs belonging to a certain class, or to replace the logic with an alternative, the training examples need to be changed and the DNN needs to be retrained using the new set of examples. We argue that decomposing a DNN into DNN modules—akin to decomposing a monolithic software code into modules—can bring the benefits of modularity to deep learning. In this work, we develop a methodology for decomposing DNNs for multi-class problems into DNN modules. For four canonical problems, namely MNIST, EMNIST, FMNIST, and KMNIST, we demonstrate that such decomposition enables reuse of DNN modules to create different DNNs, enables replacement of one DNN module in a DNN with another without needing to retrain. The DNN models formed by composing DNN modules are at least as good as traditional monolithic DNNs in terms of test accuracy for our problems.

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. 2020. On Decomposing a Deep Neural Network into Modules. In *Proceedings of The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

A class of machine learning algorithms known as *deep learning* has received much attention in both academia and industry. These algorithms use multiple layers of transformation functions to convert inputs to outputs, each layer learning successively higher-level of abstractions in the data. The availability of large datasets has made it feasible to train (adjust the weights of) these multiple layers. Since these layers are organized in the form of a network, this machine learning model is also referred to as a deep neural network (DNN). While the jury is still out on the impact of deep learning on overall understanding of software's behavior, a significant uptick in its usage and applications in wide-ranging areas and safety-critical systems, e.g., autonomous driving, aviation system, medical analysis, etc, combine to warrant research on software engineering practices in the presence of deep learning.

A software component and a DNN model are similar in spirit—both encode logic and represent significant investments. The former is an investment of the developer's efforts to encode desired logic in the form of software code, whereas the latter is an investment of the modeler's efforts, an effort to label training data, and computation time to create a trained DNN model. The similarity ends there, however. While independent development of software components and a software developer's ability to (re)use software parts has led to the rapid software-driven advances we enjoy today; the ability to (re)use parts of DNN models has not been, to the best of our knowledge, attempted before. The closest approach, transfer learning [14], attempts to reuse the entire DNN model for another problem. Could we decompose and reuse parts of a DNN model?

To that end, we introduce the novel idea of *decomposing a trained DNN model into DNN modules.* Once the model has been decomposed, the modules of that model might be reused to create a completely different DNN model, for instance, a DNN model that needs logic present in two different existing models can be created by composing DNN modules from those two models, without having to retrain. DNN decomposition also enables replacement. A DNN module can be replaced by another module without having to retrain the DNN. The replacement could be needed for performance improvement or for replacing a functionality with a different one.

To introduce our notion of DNN decomposition, we have focused on decomposing DNN models for multi-label classification problems. We propose a series of techniques for decomposing a DNN model for n-label classification problem into n DNN modules, one for each label in the original model. We consider each label as a *concern*, and view this decomposition as a *separation of concerns* problem [4, 13, 18]. Each DNN module is created due to its ability to hide one concern [13]. As expected, a concern is *tangled* with other concerns [1, 6, 15] and we have proposed initial strategies to identify and eliminate concern interaction.

We have evaluated our DNN decomposition approach using 16 different models for four canonical datasets (MNIST [7], Fashion MNIST [21], EMNIST [3], and Kuzushiji MNIST [2]). We have experimented with six approaches for decomposition, each successively refining the former. Our evaluation shows that for the majority of the DNN models (9 out of 16), decomposing a DNN model into

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

modules and then composing the DNN modules together to form a DNN model that is functionally equivalent to the original model, but more modular, does not lead to any loss of performance in terms of model accuracy. We also examine intra- and inter-dataset reuse, where DNN modules are used to solve a different problem using the same training dataset or entirely different problem using an entirely different dataset. Our results show DNN models trained by reusing DNN modules are at least as good as DNN models trained from scratch for MNIST (+0.30%), FMNIST (+0.00%), EMNIST (+0.62%), and KMNIST (+0.05%), We have also evaluated replacement, where a DNN module is replaced by another and see similarly encouraging results. In the rest of this paper, we describe our initial steps toward achieving better modularity for deep neural networks starting with a motivating example in §2, related work in §3, methodology in §4 and results in §5. §6 concludes.

2 WHY DECOMPOSE A DNN INTO MODULES?

Achieving decomposition of a DNN into modules has the potential to bring many benefits of modular programming and associated practices to deep learning. To motivate our objectives, consider Figure 1 that uses DNN models for recognizing digits [7] and letters [3] to illustrate. The top-half of the figure shows two reuse scenarios. (1) If we have a DNN model to recognize 0-9, can we extract the logic to recognize 0-1 and build a DNN model for binary digits? (2) If we have two DNN models for recognizing 0-9 and A-J, can we build a DNN model for recognizing 0-9AB, essentially a duodecimal classifier? The bottom-half of the figure shows a replacement scenario. (3) If we have a DNN model that is doing a satisfactory job of classifying 0-4 and 6-9, but could improve its performance for 5, could we take the logic for classifying 5 from another DNN model and replace the faulty 5 with a new part.

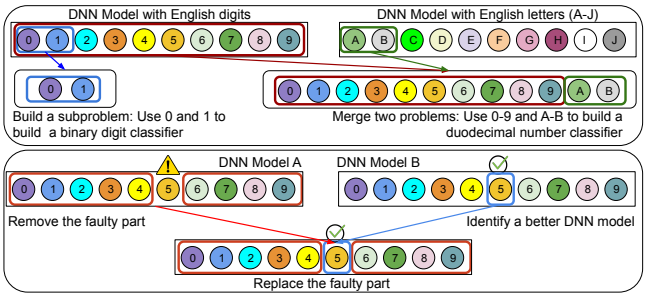

Figure 1: Examples of Fine-grained Reuse and Replacement

If the logic to recognize 0-9 and A-J were implemented as source code, modern SE practices might consider these reuse and replacement scenarios trivial. These scenarios would be far from trivial in the current state-of-the-art in deep learning. To realize the reuse scenarios, first the developer will build a model structure for binary digits for the first scenario, and duodecimal classifier for the second scenario. Then, the developer will take the training dataset for 0-9 and partition it to extract labeled training samples for 0 and 1 for the first scenario and A and B for the second scenario. Then, these new training datasets will be used to retrain the new model structures. Realizing the replacement scenario is more complicated, however. The developer might need to change the model structure of model A to match the structure of model B, which also has the potential to change model A's effectiveness for 0-4 and 6-9. Then, they can replace the training samples for 5 used to train model A with those used to train model B. Finally, the modified model A would be trained with the modified training data. Even for these simple scenarios, both reuse and replacement are complicated. Coarse-grained reuse of the entire DNN model as a black-box is becoming common. As modern software continues to integrate DNN models as components, it is important to understand whether fine-grained reuse/replacement of DNN models can be improved.

3 RELATED IDEAS

We are inspired by the vast body of seminal work on decomposing software into modules [4, 6, 13, 15, 18]. We believe that there are ample opportunities to consider a number of these ideas in the context of DNN, but focus primarily on decomposition in this work.

The decomposition of a DNN into modules has not been attempted before, but coarse-grained reuse (at the level of the entire DNN) has been. An approach for achieving reusability in deep learning is transfer learning [14]. In this technique, a DNN model is leveraged for a different setting by changing the output layer and input shape of a pre-trained network. Transfer learning can be either heterogeneous [16, 17] or homogeneous [12, 20] based on the problem domain. Zhou [22] proposed a specification based framework that uniquely identifies the goal of the model. These models can be advertised in the marketplace so that other developers can take this model as input and utilize them to build other DNN models on top of those. Li *et al.* [8] proposed that reusability can be achieved by adding different AUC metrics as tags that can help to choose the appropriate model with different parameters. Compared to the coarse-grained reuse of DNN models, the focus of this work is on fine-grained reuse and replacement.

4 DECOMPOSING A DNN INTO MODULES

Our approach, illustrated in Figure 2, decomposes a trained DNN model into modules. In this work, we focus on DNN models for multi-label classification problems. We will refer to the single, black-box, DNN model for all classes as the *monolithic model*. Our approach decomposes such models into *DNN modules*, one for each label in the original monolithic model. A DNN module accepts the same input as the monolithic model, but acts as a binary classifier. In our example in Figure 2, a multi-label classifier that classifies input into 0-2 (far-left) is decomposed into three DNN modules (far-right) that classify whether an input is 0 or 1 or 2.

DNN decomposition has three steps: *concern identification* (CI), *tangling identification* (TI), and *concern modularization* (CM). By *concern* here we mean a specific functionality, e.g. an ability to determine whether an input is 0. This meaning is consistent with the prior work on modularity [4, 6, 13, 15, 18]. *Concern identification* (CI) is the process of identifying parts of the monolithic model that are responsible for that concern. Clearly, as Figure 2 shows, concerns are tangled (mixed together) within the monolithic model and parts of the model might contribute to more than one concerns. Once concern identification is complete and parts of the monolithic model that contribute to a concern are identified, *tangling identification* (TI) is the process of identify elements among those parts that

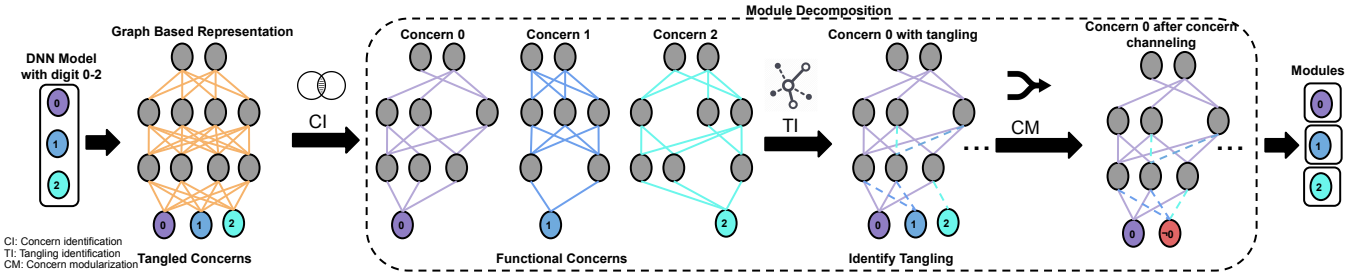

Figure 2: Overview of the approach to decompose a DNN model.

are also contributing to other concerns. Finally, *concern modularization* (CM) is the process of separating the parts of the monolithic model belonging to a concern into its own DNN module. CM also involves *concern channeling* where the effects of the non-dominant concerns within the module are channeled appropriately.

4.1 Concern Identification (CI)

Concern identification essentially identifies those parts of the monolithic model that contribute to specific functionality or concern. To paraphrase Tarr *et al.* [18], in order to achieve a better DNN quality and improve the reusability, the concerns of DNN need to be separated in such a fashion that each concern can perform a specific functionality without the intervention of the others. To untangle the concern of the output label, one could obtain a piece of the DNN that can perform a certain task, e.g., prediction for a single class, and can hide the non-dominant concern(s) to separate that concern. To illustrate, consider Figure 2. Here, the monolithic model has three tangled concerns for 0, 1, and 2. The goal of the concern identification step is to identify parts of the DNN that are responsible for classifying an image into 0, 1, and 2. Once we identify parts of the DNN related to a concern, those parts still might contribute to other concerns as well. We call those other concerns non-dominant concern(s). For example, for the concern 0, concerns 1 and 2 are non-dominant concerns in Figure 2. Before we decompose the DNN into modules, we need to identify the concerns in a monolithic DNN and separate them to build sub-networks responsible for individual concerns.

Our algorithm for concern identification is shown in Algorithm 1. We monitor the behavior of the nodes by studying the training examples for that concern (e.g., training examples for 0), and add, update, remove the edges. This algorithm forms a sub-graph that can identify the common edges for a single output label. First, the weight and bias of the DNN model are stored. In order to identify the edges that are responsible for a particular output label, we observe the behavior with the examples that belong to that label from the training dataset. With each example, we perform the weight and bias update operation. In the update operation, we provide the DNN model (*model*), input example (*input*), the updated weight (D), bias (b), and an indicator (*indicator*). The weight D and bias b are initialized with the weight and bias of the monolithic DNN model outside the Algo. 1. With every example, D and b are updated and they are returned as an output. For the next example, the modified D and b will be taken as an input. If for some inputs, the common edges for all the examples that belong to a single concern are very low, then the graph can become very sparsely connected. To avoid such circumstances, the algorithm stops removing edges from the graph once it reaches a threshold. We compute the non-negative edges outside the Algo. 1 and set the the *indicator* variable. Here, we evaluate the variable and modifies the edges based on the value. The detailed discussion is in in the Algo .3. The edge update is carried out in such a fashion that removing, updating the edges can help to identify a single concern while removing the other concerns (output labels). First, we identify the value of the nodes at each layer by applying the dense operation in the line 5-9. Here, L denotes

Algorithm 1 Concern Identification (CI).

```
1:  procedure CI (model, input, indicator, D, b)
2:      X = input
3:      W, B = extractWB(model)                          ▷ Extract weight and bias.
4:      X_0 = X
5:      for each i∈ L − 1 do
6:          X_i = X_i.W_i
7:          X_i = X_i + B_i                              ▷ Compute the value of the nodes.
8:      X_L = X_{L−1}.W_L + B_L
9:      X_L = softmax(X_L)
10:     for each i∈ L − 1 do
11:         for j = 0 to j = |X_i| do
12:             if X_i[j] ≤ 0 then
13:                 D_i[:, j] = 0 ▷ Update the value of the edges to 0 for inactive nodes.
14:                 if i!=L-1 then
15:                     D_{i+1}[j, :] = 0
16:                 b_i[j] = 0
17:             else
18:                 if indicator == True then
19:                     D_i[:, j] = W_i[:, j]            ▷ No change to edge if indicator is set.
20:                     b_i[j] = B_i[j]
21:                 else
22:                     for k = 0 to k = |W_i[:, j]| do  ▷ Update the common edges.
23:                         if W_i[j, k] < 0 then
24:                             D_i[j, k] = max(D_i[j, k], W_i[j, k])
25:                             if D_i[j, k] < 0 then
26:                                 D_i[j, k] = 0
27:                         else
28:                             D_i[j, k] = min(D_i[j, k], W_i[j, k])
29:                     b_i[k] = B_i[k]
30:     for j = 0 to j = |X_L| do
31:         if indicator == True then
32:             D_L[:, j] = W_L[:, j]
33:             b_L[j] = B_L[j]
34:         else
35:             for k = 0 to k = |W_L[:, j]| do          ▷ Update the output layer edges.
36:                 if W_L[j, k] < 0 then
37:                     D_L[j, k] = max(D_L[j, k], W_L[j, k])
38:                 else
39:                     D_L[j, k] = min(D_L[j, k], W_L[j, k])
40:             b_L[k] = B_L[k]
41:     return D, b
```

the total number of hidden and output layers. For nodes belong to the hidden layer, we monitor the value before applying the ReLU operation. However, for the output layer, the value computed after applying the *softmax* activation function, has been analyzed. At line 10, we iterate through each hidden layer and monitor the value of the nodes. From line 12-16, we remove the incoming edges to the nodes that have value *<=0* by applying the ReLU operation. Based on the definition of ReLU, any negative value will be updated to zero. We update the bias of the node to be zero as there will not be any dataflow through that node. Due to the same reason, we remove the outgoing edges from those nodes to the next layer except the edges connecting the last hidden layer and the output layer. If the value corresponding to the node is a positive one, we validate the indicator. If the indicator is *false*, the edges are updated by the minimum value of the DNN model edge ($W_i[j,k]$) and the updated edge ($D_i[j,k]$) if the value of the edge is a positive one. If the value of the edge is negative, we perform the maximum operation. Both the operation are carried out to store the semantics of the edges and their impact on the prediction. At line 25, the updated edges are validated, and if the value is negative, they are updated as zero based on the activation logic. From line 31-40, we perform a similar operation on the output layer and return the updated weight (D) and the bias (b).

4.2 Tangling Identification (CI)

Tangling identification recognizes the parts responsible for other concerns. While concern identification can separate the part of the network that contributes to a concern, it may not be able to make the separated parts functional. Using concern identification, we identify the edges that are responsible for a particular concern. However, the remaining network can only classify a single concern as all the edges correspond to the other concerns are removed or updated, and the model predicts the dominant concern irrespective of the input. Thus, the resulting network becomes a single-class classifier. This is akin to removing a conditional and a branch from a program that results in a subprogram that performs the functionality of the remaining branch but does so unconditionally. For example, in Figure 1 the concerns for 0, 1, and 2 cannot be used as they cannot distinguish between different inputs.

To solve this problem, our insight is to identify some edges and nodes back to the concern that helps us identify inputs that don't need classification by the dominant concern. In our approach, we do so by adding parts of the non-dominant concerns. By doing so, the problem of classification becomes akin to the *one against all (OAA)* classification problem. In OAA, a classifier is built from the scratch that predicts a particular output label (positive example) and can still detect any negative ones. There are a few techniques that have been proposed by prior works [9, 23] that includes introducing an imbalance between the positive and negative examples, punishing the negative example, assign higher priority to the negative examples while modifying the edges. We propose another technique that keeps the most relevant edges related to the negative examples. Before describing these techniques, we discuss the approach to add edges related to non-dominant concerns in Algorithm 2.

Algorithm 2 works as follows. First, the value of the nodes is computed similarly to the Algorithm 1. After that, the edges incident

Algorithm 2 Tangling Identification (TI).

```
1:  procedure TI(model,input,indicator,D, b)
2:      X = input
3:      W, B = extractWB(model)
4:      X_0 = X
5:      for each i∈ L − 1 do
6:          X_i = X_i .W_i                          ▷ Compute the value of the nodes.
7:          X_i = X_i + B_i
8:      X_L = X_{L−1}.W_L + B_L
9:      X_L = softmax(X_L)
10:     for each i∈ L − 1 do
11:         for j = 0 to j = |X_i| do               ▷ Update the negative edges.
12:             if X_i[j] ≤ 0 then
13:                 D_i[:, j] = D_i[:, j]
14:             else
15:                 D_i[:, j] = W_i[:, j]
16:                 b_i[j] = B_i[j]
17:     for j = 0 to j = |X_L| do                   ▷ Update the output layer edges.
18:         if X_L[j] > 0.0001 then
19:             D_L[:, j] = W_L[:, j]
20:             b_L[j] = B_L[j]
21:     return D, b
```

to the nodes with positive value (at lines 12-16), are added. For the output layer, we reintroduce the edges responsible for the negative output label classification (at lines 17-20).

Next, we discuss four different approaches for non-dominant edge addition techniques.

Tangling Identification: Imbalance (TI-I). Recall that concern identification works using training examples for a particular concern. In this approach, an imbalance is introduced while adding the number of examples from the positive and the negative output labels [11]. In the Algorithm 3, we discuss the steps followed to include the positive and negative examples and carry the edge update operations. First, weight and the bias of the DNN model are extracted at line 2. Then, training examples belonging to the positive output label are filtered. To identify the concern, the positive examples are used

Algorithm 3 Tangling Identification: Imbalance (TI-I).

```
1:  procedure TII (model, X_train, Y_train, class, negative_class)
2:      W, B = extractWB(model)                     ▷ Extract weight and bias.
3:      D = W, b = B, indicator = False
4:      X_class = X_train[Y_train == class]         ▷ Filter positive examples.
5:      for i = 0 to i = |X_class[0 : 1000]| do
6:          D, b = CI(model, X_class[i], indicator, D, b)   ▷ Update the positive
    edges.
7:          if countnonzero(D_L) ≤ .1 ∗ |D_L| then
8:              indicator = False                   ▷ Stop the update if graph is very sparse.
9:      for each i ∈ negative_class do              ▷ Update negative edges.
10:         X_negative = X_train[Y_train == i]
11:         for i = 0 to i = |X_negative[0 : ⌊10/|negative_class|⌋]| do
12:             D, b = TI(model, X_negative[i], indicator, D, b)
13:     return D, b
```

to find the common edges that correspond to them based on the approach depicted in the Algorithm 1 Our concern identification algorithm relies on an indicator to halt the process of eliminating edges. At line 7, such an indicator has been utilized that is set when the total number of nonzero edges (active) at the last layer is less than 10% of the total edges. Then, negative examples are added (we perform a floor operation to remove the floating-point value) at lines 8-11 using the Algorithm 2. Finally, the edges corresponding

to the non-dominant concerns are updated or added based on the approach discussed in the Algorithm 2.

Tangling Identification: Punish Negative Examples (TI-PN). Similar to the previous approach of introducing imbalance to tackle the OAA problem, here we punish the negative output labels by letting the positive output label update the edges first [10]. The process is similar to Algorithm 3. However, the input examples are balanced. In this approach, if 100 examples are utilized for the positive output label, then a same number of examples for the negative labels are taken e.g., for an MNIST classification problem with ten output labels, if we want to decompose a module for label 0, then we choose the ratio being 100:99 (11 from each negative label). To stop the negative edges being introduced in a module, the approach for identifying the common edges has been updated. In Algorithm 1, the edges between the output layer and the last hidden layer are updated based on the value of the weight. In addition to that, validation has been introduced before line 31, where the edges that incident to the nodes at the output layer having a value at least 0.0001 (0.01% as the last layer represents the probability value) have been added. This helps to remove the edges responsible for the negative examples leaving more edges related to the positive output label.

Tangling Identification: Higher Priority to Negative Examples (TI-HP). In this approach, the negative output labels are assigned more priority over the dominant concern by swapping the order of the edge update. For this, we update the edges correspond to the dominant concern first (lines 9-12 from Algorithm 3) and then update the edges responsible for the non-dominant concerns (lines 5-8). To keep the ratio of positive and negative examples the same, the total negative examples are equally distributed to the number of negative output labels and floor operation has been performed to avoid any floating-point number (at line 12 from Algorithm 3, we replace the number *10* with 1000 that is same as the positive examples). Other than that, everything has been kept the same.

Tangling Identification: Strong Negative Edges (TI-SNE). In this approach, the edges related to the positive label are updated, then the negative labels. Furthermore, the strong negative edges are added by introducing a validation to the Algorithm 2 at line 18 by updating the check imposed on the output node value associated with the negative examples. To do so, the value of the probability has been changed from 0.01% to 50% while keeping the other parts of the algorithm unchanged.

4.3 Concern Modularization (CM)

Concern modularization partitions the concerns into parts and builds DNN modules for each concern.

Concern Modularization: Channeling (CM-C). Concern modularization includes the abstraction of the non-dominant concerns. In this approach, we propose an approach to abstract the non-dominant concerns by combining the non-dominant nodes at the output layer. In the Algorithm 4, we discuss the methodology to channel the output nodes into one for all the non-dominant nodes. At lines 2-8, the edges between the last hidden layer and the positive node at the output layer are updated. The input *class* indicates the concerned output label. The position of the non-dominant node at the output label after the concern channeling depends on the position of the dominant node. If the dominant node is the first node, then we assign the 2nd node at the output label as the non-dominant node (lines 5-8). If not, we choose the 1st node as the non-dominant node and keep the position of the dominant node as it is (lines 12-21). All the edges incident on the pre-channeling non-dominant nodes are replaced by edges directed towards the non-dominant node after channeling with the value being the average of the edges. The remaining edges (incident on the non-dominant nodes other than the channelized one) are updated to be 0.

Algorithm 4 Concern Modularization: Channeling (CM-C).

```
1:  procedure CMC (D, b, class)
2:      for i = 0 to i = |D_L| do
3:          temp = D_L[i, :]
4:          if class = 0 then                    ▷ Assign the 2nd node as the negative.
5:              temp[1] = mean(temp[1, :])
6:              temp[2, :] = 0
7:              d_L[1] = mean(d_L[1, :])         ▷ Compute the mean of negative edges.
8:              d_L[2, :] = 0                    ▷ Update all edges to other negative nodes.
9:          else                                 ▷ Assign the 1st node as the negative.
10:             tempW, tempB = [], k = 0
11:             for j = 0 to j = |b_L| do ▷ |b_L| represents the size of the output layer.
12:                 if j! = class then           ▷ Perform for all negative nodes.
13:                     tempW[k] = temp[j]
14:                     tempB[k] = b_L[j]
15:                     k = k + 1
16:             temp[0] = mean(tempW) ▷ Compute the mean of negative edges.
17:             d_L[0] = mean(d_L[1, :])         ▷ Assign the mean value.
18:             for j = 1 to j = |b_L| do    ▷ Update all edges to other negative nodes.
19:                 if j! = class then
20:                     temp[j] = 0
21:                     d_L[2, :] = 0
22:         D_L[i, :] = temp
23:     return D, b
```

Concern Modularization: Remove Irrelevant Edges (CM-RIE). Before applying the concern channeling, we remove the irrelevant nodes at the last hidden layer that only contributes to non-dominant concerns. The edges that are connected with the negative output nodes (before channeling the output nodes) and not connected (weight is zero) with the positive output node are combined into one node. The outgoing edge(s) from the combined node to a particular negative output node, is updated with the average of the all the edges incident to that negative output node from nodes that are connected only with the negative output labels. In the Algorithm 5, we discuss the steps carried out to remove the edges picked based on the filtering criteria. At lines 4-18, the edges from the last hidden to output layer are updated if a node *n* from the last layer is only connected to the any of the negative output nodes. These edges are replaced by a single edge with the weight and bias value as the average of the edges. If the number of such nodes is more than one, we replace all the nodes by one node by removing all the edges from those nodes with one edge with the average weight and bias. We perform this operation at lines 22-28. Updating the edges and removing the connection to some nodes is similar to changing the flow of the data. In this process, the path that data flows has been updated, not the amount of the flow. Since the flow with the edges from the last hidden layer and the output layer has been updated, the same flow needs to be adjusted at the preceding layer. This

Algorithm 5 Concern Modularization: Remove Irrelevant Edges (CM-RIE).

```
1:  procedure CM-RIE (D, b, class)
2:      tempD=[], tempCount=0
3:      for i = 0 to i = |D_L| do
4:          temp1, temp2=[]
5:          if D_L[i, class]==0 then
6:              for j = 0 to |d_L| do
7:                  if j! = class then
8:                      if D_L[i, j]! = 0 then          ▷ Identify the irrelevant nodes.
9:                          temp1.add(D_L[i, j]); temp2.add(j)
10:                 ▷ Compute the mean of the negative edges for the irrelevant nodes.
11:                 if class == 0 then
12:                     D_L[i, 1] = mean(temp1)
13:                 else
14:                     D_L[i, 0] = mean(temp1)
15:                 for k ∈ temp2 do
16:                     D_L[i, k] = 0
17:             tempD.add(i)
18:     if class == 0 then
19:         negative_node = 1
20:     else
21:         negative_node = 0
22:     if len(tempD) > 1 then ▷ Merge the edges if more than 1 irrelevant nodes.
23:         for i ∈ tempD do
24:             if class == 0 then
25:                 tempCount+ = D_L[i, 1]
26:             else
27:                 tempCount+ = D_L[i, 0]
28:         D_L[tempD[0], negative_node] = mean(tempCount)
29:         for x ∈ tempD[1 : count(tempD)] do
30:             D_L[x, negative_node] = 0              ▷ Update the removed edges.
31:         for i = 0 to |D_{L-1}| do                  ▷ Compensate the flow.
32:             tempD_{L-1} = []
33:             for j ∈ tempD do
34:                 tempD_{L-1}.add(D_{L-1}[i, j])     ▷ Update the merged edges.
35:             D_{L-1}[i, tempD[0]] = mean(tempD_{L-1})
36:             for x ∈ tempD[1 : count(tempD)] do
37:                 D_{L-1}[i, x] = 0                  ▷ Update the removed edges.
38:     D, b = CMC(D, b, class)                        ▷ Apply the concern channeling approach.
39:     return D, b
```

update operations is shown at lines 31-37. The edges from the preceding layer to the last hidden layer that incident on the removed nodes are removed and updated the edges incident to a replaced node at the last hidden layer.

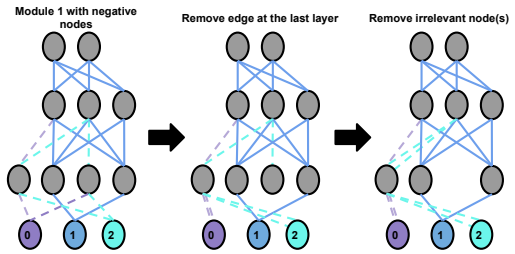

Figure 3: Concern Modularization: Remove Irrelevant Edges

In Figure 3, an example of such an operation is shown, where two nodes are connected with the negative output nodes, not with the positive output label. First, edges from the last hidden layer to the output layer are replaced with a single edge for each node. Then, all such nodes are replaced with a single node with the updated value associated with the edges. Furthermore, to compensate for the flow, all the edges that are incident on the removed nodes are removed and the value of the edges to the replaced node at the last hidden layer is updated. After this operation, we channel the behavior of the negative output label using the approach described in the Algorithm 4.

5 EVALUATION

5.1 Experimental Setup

In this section, we discuss the datasets, models, and the evaluation metrics utilized to evaluate our approach. For concern identification algorithm, we take the threshold value to be 10% of the total number of edges from the last hidden layer to the output layer, i.e. we stop removing edges at that point to prevent network from becoming too sparse. For tangling identification imbalance (TI-I) technique, we choose the imbalance to be 100:1, where for a 10-class classification problem, if 1000 examples were taken from the positive output label, 1 example from each of the other output labels (remaining nine labels) are taken for modifying the edges of the neural network.

5.1.1 Datasets. **MNIST [7].** This dataset comprises of various examples of handwritten digits (0-9). It is divided into the training and testing section. There are 60,000 training and 10,000 testing examples, and each output label has an equal number of data.
Extended MNIST (EMNIST) [3]. Similar to the MNIST, this dataset has two-dimensional images from the English alphabets (A-Z). As our approach is based on the dense hidden layer, training a DNN model with only dense layers does not achieve high testing accuracy. To remedy that problem and fixing the number of output labels for all the datasets under experiment, A-J letters (10) are taken from the dataset. Training and testing dataset contains 48000 and 8000 examples of A-J letters, respectively.
Fashion MNIST (FMNIST) [21]. This dataset is similar to the structure of the MNIST in terms of the training, testing example division, and the number of output labels. However, this dataset has 2D images from different clothes- T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, and Ankle boot.
Kuzushiji MNIST (KMNIST) [2]. The structure of the dataset is similar to the MNIST. It contains images from Japanese digits.

MNIST and FMNIST have been taken from Keras [5] and the other two datasets are extracted from Tensorflow [19].

5.1.2 Models. To evaluate our approach, we build 16 different models. These models are trained with corresponding datasets with 50 epochs, and they have 1, 2, 3, and 4 hidden layers (size of each layer is 49), respectively. The name of the DNN model has been represented as < *dataset–# of hidden layers*>, e.g., for the KMNIST dataset with 2 hidden layers, the model is referred as KMNIST-2. The testing accuracy of each model is given in Table 1.

5.1.3 Evaluation Metrics. **Accuracy.** For measuring the performance of the DNN model, we use the accuracy metrics. However, in the case of the composition of the decomposed modules, we use the same metrics based on voting. We execute N decomposed modules for a N classification problem in parallel and measure the output of each module. If only one module votes for the input, we assign the label of the input based on the dominant output label of the module, e.g., if we execute 10 modules for an MNIST problem

and for input, module 0 only votes for the positive output label, then we label that input as 0. If more than one or no module votes for the positive output label, then we choose the most confident one by picking the module with the highest probability for the positive output label. Based on this, we compute the accuracy over the test dataset. We refer to the composed accuracy of the decomposed modules as *DM* and the accuracy of the trained model *TMA*.

Jaccard Index (JI). We use the Jaccard Index to measure the similarities between the decomposed modules. First, we transform the weight and bias from all the layers into a 1-dimensional array and compare the similarities between two modules. Finally, we compute the average value of the metric and report in Table 1. If the *Jaccard Index* is 0, then there is no commonality between two compared objects, and if it is 1, then they are exactly the same.

5.2 Results

In this section, we validate our techniques to decompose DNN into modules and answer our research questions.

5.2.1 How efficient are the decomposed modules? To answer this research question, we evaluate the four different proposed techniques to identify tangling concern. We utilized the best approach based on the accuracy and the Jaccard index and used that technique to apply concern modularization to build modules. To do so, we decompose the DNN models into modules before channeling the non-dominant concerns, and run them in parallel and compute the accuracy. Finally, we compare the accuracy among different techniques and the pre-trained DNN model. Furthermore, we compute the average Jaccard index (JI) over all the decomposed modules for each technique. The results have been depicted in Table 1.

We measure the Jaccard index to identify tangling concerns among modules. We found that utilizing the TI-HP technique, where higher priority has been given to the negative examples to update the edges, the lowest Jaccard index can be achieved. This suggests that the decomposed modules have the least overlap among themselves and are significantly different in structure. However, the composed accuracy of the decomposed modules is very low (average accuracy is 22.93%). To understand the reason, we investigate the structure of the modules and find that the edges responsible for predicting the negative output labels are updated by the positive ones. As the edges related to the negative examples are updated first, then the positive ones, the edges that are responsible for negative output labels, are removed or modified by the edges from the positive output labels. Therefore, the network can only classify the dominant output label correctly not the non-dominant ones (7 out 16 scenarios have accuracy ~10% that explains the decomposed modules are not able to classify the rest of the 9 output labels). For other scenarios, especially for MNIST-3 and EMNIST-3, where all the edges related to the negative output labels are not updated or removed, that results in higher accuracy.

Utilizing TI-PN, where we punish the negative examples more than the positive examples by removing edges related to the negative output labels, achieves a lower accuracy for most of the cases (10 out 16 scenarios have accuracy <50%). In comparison to the TI-HP technique, we let the negative examples to add or update the edges responsible for the negative output labels after the positive examples identify the common edges. Here, we can see that the Jaccard index is higher than the TI-HP that indicates that there is a higher commonality among the modules, which are mostly the negative edges. However, we found that letting the negative examples add or update all the relevant edges, the composed accuracy of the decomposed modules is significantly lower than the DNN model (average loss of accuracy is 44.15%).

This problem has been partially remedied by allowing only the edges that are strongly coupled with the prediction of the negative examples. If a negative example for a module (e.g., for MNIST, any input digit other than 0 for a module responsible for 0) has been taken as input to the system, that particular model can process the input and predict as one of the non-dominant class as the strong edges still remain intact in the modules. However, this technique increases the Jaccard index, which depicts that the modules are becoming similar to each other. This results in increasing the overlap between the concerns. However, in comparison to the other three techniques, adding the strong negative edges performs the best in terms of the accuracy (average loss -0.29% and median loss +0.00%). We move forward with this technique and select this approach for utilizing our concern channeling and remedy the tangling of the concerns. To do so, we update, add, and remove edges (CM-C). The concern channelization achieves the best accuracy and better Jaccard index for 37.5% and 25% scenarios, respectively. However, to identify more tangled concerns, we utilized the CM-RIE approach, where we remove irrelevant nodes only connected with the negative output nodes and update the edges to reflect the change at the last hidden layer. We apply this technique before channelizing non-dominant concerns. From Table 1, we can validate that the CM-RIE decreases the Jaccard index from the prior techniques (CM-C), and this approach can produce modules that perform the best in terms of the accuracy for 56.25% (9 out of 16) of the cases.

With the accuracy achieved using the CM-RIE technique, we found that the accuracy after decomposing loses 0.01% on average (median is 0.00%). Also, in 9 out of 16 cases, we were able to increase or able to get the same accuracy as the trained model. To validate whether there is a significant difference between the DNN models and decomposed modules, the average number of edges with value zero (inactive) have been computed for each scenario. This metric validates how the edge removal technique to decompose the DNN model into modules performs in practice. We found that for cases where decomposing a DNN model into modules either gain accuracy or remain the same, there are, on average, 33.79% of the edges are inactive. This result shows that the modules generated are not the same as the DNN model. However, our decomposition technique can be modified in the future to increase the inactive nodes and remove the tangled concerns more efficiently. For further RQs, we evaluate our proposed approach with the decomposed modules build with the CM-RIE technique.

5.2.2 Does modularity in DNN enable reuse? In this research question, we validate whether fine-grained reuse can be achieved by utilizing the decomposed modules. To validate the reusability and answer this RQ, we evaluate decomposed modules from 16 different DNN models and reuse them in two different settings.

Intra-Dataset Reuse. In this scenario, we study two modules decomposed from the same DNN model and execute them in parallel to build a smaller problem and validate against a DNN model built

Table 1: Accuracy and Similarity of Decomposed Modules; Acc: Accuracy, JI: Mean Jaccard index

Model	Model Accuracy	TI-I		TI-PN		TI-HP		TI-SNE		CM-C		CM-RIE	
		Acc	JI	Acc	JI	Acc	JI	Acc	JI	Acc	JI	Acc	JI
MNIST-1	94.91%	**94.91%**	0.47	9.79%	0.46	23.09%	0.04	**94.91%**	0.47	**94.91%**	0.45	94.90%	**0.43**
MNIST-2	98.39%	**96.83%**	0.64	6.19%	0.64	13.09%	0.02	**96.83%**	0.65	**96.83%**	**0.63**	96.82%	**0.63**
MNIST-3	98.47%	69.19%	0.44	9.01%	0.44	96.27%	0.45	96.30%	0.45	96.30%	0.44	**96.33%**	**0.43**
MNIST-4	96.79%	96.44%	0.53	7.50%	0.54	9.80%	0.01	**96.79%**	0.55	**96.79%**	0.54	96.75%	**0.53**
FMNIST-1	85.82%	85.82%	0.78	69.21%	0.78	10.72%	0.09	85.82%	0.78	85.82%	0.76	**85.85%**	**0.75**
FMNIST-2	87.58%	**87.58%**	0.64	9.28%	0.64	18.69%	0.14	**87.58%**	0.65	**87.58%**	**0.63**	87.56%	**0.63**
FMNIST-3	87.09%	77.55%	0.52	61.95%	0.52	9.96%	0.05	87.09%	0.53	85.64%	**0.5**	**87.10%**	0.51
FMNIST-4	87.79%	87.51%	0.56	81.11%	0.57	10%	0.01	87.79%	0.57	87.79%	0.56	**87.95%**	**0.55**
KMNIST-1	76.02%	64.29%	0.51	76.32%	0.51	23.17%	0.03	76.32%	0.51	76.02%	0.50	**76.03%**	**0.48**
KMNIST-2	83.29%	**83.29%**	0.72	70.63%	0.72	10.00%	0.02	**82.09%**	0.73	**83.29%**	0.72	**83.29%**	**0.7**
KMNIST-3	83.02%	**83.02%**	0.48	42.70%	0.47	11.41%	0.01	**83.02%**	0.49	**83.02%**	**0.47**	82.97%	**0.47**
KMNIST-4	83.63%	82.89%	0.54	39.50%	0.56	10.50%	0.01	**83.63%**	0.57	83.63%	0.56	**83.63%**	**0.54**
EMNIST-1	89.00%	89.00%	0.43	81.06%	0.41	13.85%	0.04	89.00%	0.43	89.00%	0.41	**89.01%**	**0.4**
EMNIST-2	92.27%	92.10%	0.61	76.89%	0.61	10.07%	0.02	92.27%	0.62	92.27%	0.61	**92.28%**	**0.61**
EMNIST-3	92.20%	62.48%	0.53	37.90%	0.53	83.84%	0.54	**92.20%**	0.55	**92.20%**	0.54	92.16%	**0.53**
EMNIST-4	91.88%	84.33%	0.53	42.74%	0.54	12.38%	0.01	91.89%	0.55	91.89%	0.54	**95.33%**	**0.53**

Table 2: Intra-Dataset Reuse. All results in %.

MNIST and Fashion MNIST. MN: MNIST, FM: FMNIST.

MN		1		2		3		4		5		6		7		8		9	
FM \ MN		MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA
T-shirt/top	**0**	**100**	**99.9**	99.1	99.4	99.6	99.8	**99.6**	**99.6**	**99.4**	**98.8**	98.7	99.0	99.2	99.6	**99.4**	**99.2**	97.6	99.2
Trouser	**1**	98.2	98.5	**99.7**	**99.4**	**99.8**	**99.6**	**99.9**	**99.9**	**100**	**99.7**	**99.6**	**99.5**	**99.5**	**99.0**	**99.5**	**99.2**	**99.8**	**99.5**
Pullover	**2**	93.7	95.2	**99.3**	**98.9**	**99.0**	**97.6**	**99.4**	**98.5**	**99.9**	**98.5**	**99.5**	**99.0**	**98.9**	**97.4**	98.9	99.1	**99.6**	**99.2**
Dress	**3**	**92.2**	**91.5**	**98.4**	**97.5**	97.1	97.3	**99.9**	**99.4**	**98.6**	**97.7**	99.8	99.5	**99.3**	**98.6**	**98.8**	**97.5**	**99.2**	**98.6**
Coat	**4**	97.2	98.3	99.3	99.5	**88.0**	**86.3**	**96.0**	**95.1**	**99.8**	**99.0**	99.2	99.3	**99.5**	**98.6**	**99.5**	**99.2**	**98.2**	**97.7**
Sandal	**5**	97.3	99.7	99.7	100	**100**	**100**	**100**	**99.9**	**100**	**99.8**	**99.0**	**98.1**	**99.6**	**99.3**	**99.1**	**97.6**	**99.1**	**98.7**
Shirt	**6**	52.4	84.7	**99.1**	**98.4**	**88.0**	**86.1**	**95.4**	**93.6**	**90.3**	**87.3**	**100**	**99.9**	**99.7**	**99.7**	**99.2**	**99.1**	99.5	99.7
Sneaker	**7**	97.4	99.7	99.7	100	**100**	**100**	**100**	**100**	**100**	**100**	**98.0**	**96.1**	**100**	**100**	**99.6**	**98.6**	**98.6**	**98.4**
Bag	**8**	97.2	98.1	99.4	99.7	**99.1**	**98.4**	**99.1**	**99.1**	**99.2**	**99.0**	**99.6**	**99.6**	**98.2**	**97.7**	**99.6**	**99.0**	**99.2**	**97.9**
Ankle boot	**9**	97.4	100	99.6	100	**99.9**	**99.9**	**99.9**	**99.9**	**100**	**100**	**98.5**	**98.0**	**100**	**100**	**96.5**	**96.4**	**99.8**	**99.6**
FM		**T-Shirt/top**		**Trouser**		**Pullover**		**Dress**		**Coat**		**Sandal**		**Shirt**		**Sneaker**		**Bag**	

Blue represents the intra module combinations for F-MNIST and Yellow represents the the intra module combinations for MNIST.

Extended MNIST (EMNIST) and Kuzushiji MNIST (KMNIST). EM: EMNIST, KM: KMNIST.

EM		A		B		C		D		E		F		G		H		I	
KM \ EM		MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA
Japanese 0	**A**	97.8	97.9	96.6	97.8	94.8	94.9	95.1	97.4	**98.4**	**98.4**	**95.3**	**92.9**	94.2	96.5	98.5	98.8	97.8	98.6
Japanese 1	**B**	97.5	98.2	**98.9**	**98.0**	**97.6**	**96.8**	**98.6**	**98.4**	**98.9**	**98.8**	**96.1**	**95.2**	**96.8**	**97.0**	**99.4**	**98.2**	**98.9**	**97.9**
Japanese 2	**C**	**98.3**	**97.9**	**94.2**	**94.1**	**98.9**	**98.4**	**95.5**	**93.1**	**98.8**	**98.8**	**98.4**	**97.8**	**99.3**	**97.8**	**99.1**	**99.0**	**99.4**	**99.4**
Japanese 3	**D**	**97.8**	**97.0**	**98.4**	**97.2**	93.5	93.8	**99.3**	**98.0**	**99.0**	**98.3**	**97.9**	**95.2**	**97.8**	**97.3**	**99.3**	**98.1**	**97.4**	**97.2**
Japanese 4	**E**	87.7	94.3	**93.4**	**93.4**	**95.3**	**95.1**	**97.2**	**97.1**	**98.2**	**98.1**	**98.5**	**98.1**	**99.2**	**98.1**	**99.1**	**98.8**	**99.5**	**98.5**
Japanese 5	**F**	93.8	96.4	**95.2**	**94.5**	**91.5**	**91.3**	**97.3**	**97.0**	95.9	96.1	**97.7**	**97.1**	**98.9**	**97.8**	**98.4**	**96.6**	**98.8**	**97.1**
Japanese 6	**G**	97.6	99.0	**93.6**	**91.6**	95.1	96.3	**97.9**	**97.0**	**96.3**	**95.0**	**96.0**	**95.5**	98.6	97.8	**98.9**	**97.0**	97.3	97.9
Japanese 7	**H**	93.1	94.3	**95.4**	**94.2**	**95.7**	**93.9**	97.8	98.0	**92.7**	**89.7**	**97.1**	**96.3**	94.8	96.8	**99.3**	**97.9**	**99.3**	**97.3**
Japanese 8	**I**	92.8	96.9	94.6	94.8	93.1	95.7	**96.9**	**96.0**	**96.2**	**95.6**	95.0	96.3	97.6	97.7	95.2	96.2	**96.0**	**94.5**
Japanese 9	**J**	97.3	97.6	93.3	95.2	92.2	94.6	96.9	97.2	**95.0**	**94.9**	95.6	97.0	**96.4**	**94.4**	**95.1**	**92.7**	95.7	96.7
KM		**Japanese 0**		**Japanese 1**		**Japanese 2**		**Japanese 3**		**Japanese 4**		**Japanese 5**		**Japanese 6**		**Japanese 7**		**Japanese 8**	

Blue represents the intra module combinations for EMNIST and Yellow represents the the intra module combinations for KMNIST.

with the dominant examples for the picked modules. In Figure 1, we describe a similar example, where we take module responsible for the digits 0 and 1 from a pre-trained DNN model and reuse them to build a binary classifier. To validate such scenarios, we train a DNN model with the same examples (based on the example, digit 0 and 1) and the same structure as the DNN model that has been decomposed to obtain the modules. Finally, we compare the composed accuracy of the modules and the accuracy of the trained DNN model. In Table 2, we show various scenarios of intra dataset reuse. While performing this evaluation, we use the modules build from the model with the 4 hidden layers (MNIST-4, EMNIST-4, FMNIST-4, and KMNIST-4). As the total number of the output labels in each dataset is ten, choosing two modules responsible for one output label can have 45 scenarios ($\binom{10}{2}$). In Table 2 (upper half), we combine the results from MNIST and FMNIST datasets. The cells with blue and yellow have been taken from the FMNIST and MNIST datasets, respectively. We perform similar operations on KMNIST and EMNIST and depict the results in Table 2 (bottom half). Our results suggests that for 80% (36 out of 45), 68.89% (31 out of 45), 80% (36 out of 45), and 51.11% (23 out of 45) scenarios for

Table 3: Inter Dataset Reuse. All results are in %

MNIST vs Extended MNIST (EMNIST). MN: MNIST, EM: EMNIST

EM	A		B		C		D		E		F		G		H		I		J	
MN	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA
0	74.8	99.3	95.7	99.6	97.2	99.7	89.7	98.9	98.8	99.1	99.2	99.9	97.6	98.5	99.0	99.7	99.4	99.6	93.7	99.4
1	98.9	99.7	97.6	99.9	99.7	100.0	99.4	99.8	**99.7**	**99.7**	99.5	99.8	99.4	99.8	99.3	99.8	57.0	96.3	98.6	99.6
2	82.0	98.5	98.5	99.4	85.2	99.3	94.7	98.6	98.6	99.1	98.5	99.1	96.9	98.7	98.3	99.2	96.3	98.1	**97.8**	**97.2**
3	90.7	98.7	97.1	99.5	97.1	99.8	54.8	99.5	98.5	99.4	98.6	99.4	88.7	99.8	**99.3**	**99.1**	97.3	99.6	87.5	98.5
4	88.8	98.9	99.7	99.9	99.4	99.8	**99.3**	**99.3**	86.3	99.0	99.0	99.8	99.2	99.7	96.5	99.2	98.9	99.6	**99.8**	**99.7**
5	93.7	99.2	95.6	99.1	96.5	99.2	98.6	99.5	96.5	99.0	66.8	99.8	88.7	99.0	98.1	99.3	98.0	99.2	92.8	98.5
6	97.0	99.7	95.4	99.8	98.0	99.6	98.4	99.8	98.7	99.7	87.9	99.8	81.8	99.5	93.6	99.6	98.4	99.1	98.9	99.5
7	96.9	99.7	99.2	99.7	99.3	99.7	98.9	99.6	99.3	99.6	**99.3**	**99.3**	98.6	99.7	78.0	99.6	98.1	99.3	96.3	99.6
8	89.0	99.4	93.9	99.3	98.9	99.7	97.5	99.9	95.2	99.2	93.7	99.8	90.9	99.4	96.8	99.6	56.9	98.7	98.0	99.4
9	85.4	99.2	99.5	99.9	99.4	99.5	99.4	99.9	99.2	99.7	99.2	99.9	95.3	99.0	98.7	99.8	98.8	99.6	61.8	99.9

MNIST vs Kuzushiji MNIST (KMNIST). MN: MNIST, KM: KMNIST

KM	Japanese 0		Japanese 1		Japanese 2		Japanese 3		Japanese 4		Japanese 5		Japanese 6		Japanese 7		Japanese 8		Japanese 9	
MN	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA
0	86.0	99.9	93.6	99.8	98.4	99.9	97.8	99.9	95.6	99.7	96.9	99.9	97.6	99.9	91.5	99.5	93.0	99.9	98.7	99.9
1	91.8	100.0	66.0	100.0	98.5	100.0	97.1	99.8	98.8	100.0	98.4	100.0	97.3	99.9	99.6	100.0	98.5	100.0	98.7	100.0
2	73.4	99.7	93.9	100.0	77.8	99.7	93.0	99.8	86.0	99.8	93.8	99.9	87.1	100.0	88.9	99.8	95.0	100.0	93.3	100.0
3	80.3	99.9	71.7	99.9	78.1	100.0	82.1	99.9	88.0	99.8	87.5	99.9	74.0	99.9	92.9	100.0	77.1	100.0	84.4	99.1
4	95.6	99.9	99.6	99.8	99.3	99.9	99.7	99.9	73.1	99.7	99.3	99.8	99.6	100.0	99.1	99.9	99.6	99.9	99.3	100.0
5	82.7	99.9	91.0	99.8	85.8	99.6	85.8	99.8	90.1	99.8	78.5	99.9	90.9	99.7	90.3	99.5	93.9	99.8	88.2	99.9
6	95.5	100.0	98.1	99.9	97.0	99.9	98.8	99.8	94.1	99.9	97.8	99.8	79.7	99.9	95.8	99.8	98.9	99.9	97.7	99.9
7	87.8	99.9	96.6	99.9	97.6	100.0	97.4	99.9	96.7	99.8	98.3	99.9	97.2	99.9	84.1	98.4	97.1	100.0	95.1	100.0
8	92.3	99.9	90.1	99.9	81.0	100.0	98.1	99.9	93.8	99.8	94.3	99.8	90.3	99.9	91.5	99.6	71.4	100.0	87.3	99.9
9	95.8	100.0	94.8	99.9	94.6	100.0	96.6	100.0	98.0	99.7	95.8	100.0	95.5	100.0	96.4	99.8	91.6	100.0	73.7	100.0

Table 4: Intra Dataset Replacement. RM: Replaced module.

DataSet	TMA	Prior MA	RM0	RM1	RM2	RM3	RM4	RM5	RM6	RM7	RM8	RM9
MNIST	94.91%	94.90%	**94.91%**	94.59%	94.83%	92.26%	94.40%	93.68%	95.11%	94.46%	93.72%	93.29%
FMNIST	85.82%	85.93%	85.36%	**85.84%**	85.41%	**85.82%**	84.91%	**85.87%**	84.10%	**85.88%**	**85.98%**	85.78%
KMNIST	76.02%	76.03%	**76.32%**	74.18%	74.43%	75.64%	73.74%	73.77%	75.54%	**76.90%**	74.44%	**76.53%**
EMNIST	89.00%	89.01%	87.81%	87.6%	87.89%	88.40%	86.42%	**89.36%**	88.73%	88.51%	86.23%	85.57%

MNIST, FMNIST, EMNIST, and KMNIST respectively, the composed accuracy using modules is more or the same compared to the trained DNN models. Our result suggests that there is no significant change of accuracy considering all the cases (4 datasets). Average gain of accuracy is 0.03% (+0.22% median).

Inter-Dataset Reuse. While the prior experiments were done on the same dataset and model, these experiments are carried on different datasets with models build with similar architecture (same number of hidden layers). We evaluate the MNIST vs. EMNIST and MNIST vs. KMNIST with the same choice of the models. Similar to the prior experimental setup, we evaluate the composed accuracy of two decomposed modules taken from two different datasets and models and execute them on the dominant examples of the modules, e.g., we take one module from MNIST (MNIST-4 model) that can classify English 0 and one module from KMNIST (KNIST-4 model) that is responsible for Japanese 2. In this example, we validate the accuracy against the inputs and test with the example taken from the test dataset where the output label is either English 0 or Japanese 2. Also, we train a model with the same number of hidden layers (four) with the same output labels (English 0 and Japanese 2) from the training dataset and compare them. In Table 3 (upper half), we take two modules for each experiment and compare the trained DNN model build from the examples from the output labels. We report the evaluation for 100 combinations (10x10) of re(use) from MNIST and E-MNIST modules. Our result suggests that for only six scenarios, reusing the modules can perform the same as the trained model. Reusing the modules between different datasets can cause a loss of accuracy 5.36% on average (median 1.41%) in comparison to DNN models trained with those examples. Similarly, we evaluate with MNIST and KMNIST and report the results in Table 3 (bottom half). Our result suggests that loss of accuracy is 8.28% on average (-5.67% median).

Also, we build a duodecimal classifier based on our motivating example depicted in the Figure 1, where all the modules decomposed from MNIST-4 and decomposed modules responsible for English letter *A* and *B* run in parallel, and the composed accuracy of the decomposed module is 83.40%. Furthermore, a DNN model has been trained based on the 0-9 and A-B, and the testing accuracy of that model is 91.45%. This shows that decomposing the DNN into modules and reusing them to build a problem to recognize the duodecimal (0-9AB) digits is possible. However, the accuracy of the composition of the modules is lower (8.05%) than the model trained from scratch. This could be a potential direction for future work.

5.2.3 Does modularity in DNN enable replacement? In this research question, we answer whether the decomposed modules decomposed can be replaced by other modules. Replacing a module from a set of modules built by decomposing a DNN model can help in either of these two directions. First, referring to the bottom half of the Figure 1, where the faulty part of the DNN has been replaced with a better-fitted part from a different DNN model trained on the

Table 5: Inter-Dataset Replacement. All results are in %

MNIST vs Extended MNIST (EMNIST). MN: MNIST, EM: EMNIST

EM	A		B		C		D		E		F		G		H		I		J	
MN	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA
0	94.0	94.8	93.7	94.2	94.7	95.5	94.4	95.0	94.4	95.3	94.3	94.9	93.9	94.8	94.1	95.4	94.4	95.1	94.6	95.7
1	94.3	94.4	94.0	94.2	94.5	94.5	94.7	95.7	94.4	95.8	94.9	**94.7**	**94.7**	95.2	94.4	95.6	95.2	95.8	94.6	96.0
2	**95.5**	**95.2**	94.0	96.1	**94.5**	**94.5**	94.8	95.5	94.4	95.3	94.9	95.1	94.4	95.2	94.4	95.6	94.7	95.8	94.7	96.0
3	**94.8**	**94.5**	93.4	94.7	94.5	95.8	94.4	95.3	93.8	95.5	94.0	95.4	93.7	94.7	93.8	95.1	94.3	94.9	94.1	95.9
4	**95.1**	**94.7**	**94.9**	**94.3**	**95.4**	**94.9**	**95.5**	**95.3**	95.3	95.7	**95.7**	**95.2**	**95.1**	**93.9**	95.3	96.0	96.1	96.3	**95.5**	**95.4**
5	92.6	94.9	92.5	95.3	93.0	95.2	93.0	95.1	92.9	95.8	94.0	95.5	**94.4**	**91.5**	92.9	96.0	93.7	96.1	93.2	96.6
6	91.8	95.7	91.4	94.7	92.1	95.7	92.8	95.1	91.9	95.4	92.9	95.6	91.8	95.4	91.9	94.1	92.8	96.0	92.4	96.0
7	94.6	94.9	**94.4**	**93.8**	**95.0**	**94.9**	94.9	95.3	95.3	95.5	94.9	96.0	**95.4**	**95.2**	94.9	95.5	**95.3**	**95.3**	95.1	95.4
8	86.8	93.8	**94.9**	**94.3**	87.5	94.6	87.5	93.9	87.1	93.9	87.4	94.7	87.0	94.7	87.2	94.2	87.5	94.2	87.3	94.5
9	93.2	94.8	93.2	94.6	93.7	95.0	95.1	95.4	93.2	95.7	94.0	95.6	93.1	94.9	93.6	95.2	93.5	95.6	93.4	95.0

MNIST vs Kuzushiji MNIST (KMNIST). MN: MNIST, KM: KMNIST

KM	Japanese 0		Japanese 1		Japanese 2		Japanese 3		Japanese 4		Japanese 5		Japanese 6		Japanese 7		Japanese 8		Japanese 9	
MN	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA	MA	TMA
0	90.3	95.9	89.9	94.2	93.0	94.6	91.1	95.4	90.4	96.0	91.1	96.3	90.3	94.9	90.4	95.5	90.7	95.1	90.6	96.5
1	88.7	96.2	88.2	95.4	89.0	95.7	92.0	95.9	88.6	94.8	89.2	96.4	88.5	94.9	88.9	95.9	89.5	96.4	89.0	96.2
2	88.9	94.7	88.6	94.9	89.3	95.9	90.1	95.7	89.0	96.1	89.5	96.1	89.0	96.1	89.2	95.3	90.8	96.1	89.4	95.8
3	90.8	94.8	90.1	95.5	91.3	95.1	92.1	95.7	90.5	95.9	91.6	96.4	90.4	95.5	90.6	96.6	90.8	95.9	90.8	96.2
4	90.3	95.5	89.4	96.1	91.4	95.4	90.4	94.9	89.8	95.2	90.7	95.8	90.1	95.5	89.9	96.0	90.2	96.2	90.1	96.5
5	91.7	95.6	90.8	95.3	91.9	94.6	91.9	94.9	91.2	95.8	92.2	95.7	91.1	95.8	91.3	95.7	91.5	95.9	91.5	96.6
6	88.7	95.2	88.3	94.9	89.6	95.7	90.8	95.5	88.7	96.1	89.3	96.0	88.8	95.6	88.8	96.2	89.4	96.0	89.1	96.2
7	91.1	94.7	89.9	95.0	91.6	95.6	90.6	95.6	90.4	95.2	91.2	94.7	90.4	95.5	90.8	95.3	90.9	95.8	90.7	96.4
8	90.3	95.6	88.8	94.7	89.6	96.0	91.3	95.2	89.1	95.9	89.5	96.0	89.0	95.2	89.2	95.4	89.6	96.0	89.6	96.6
9	89.9	95.4	89.7	95.2	90.7	95.7	91.1	95.7	90.1	95.1	90.9	96.2	90.1	95.6	90.3	96.0	91.4	96.5	90.6	95.9

same problem, and this scenario we refer to as the intra dataset replacement. Second, a part of the DNN model can be replaced by a part from a DNN model that represents a different concern. We represent these scenarios as inter problem replacement, where the dataset of the problems is different. These broader categories of the replaceability study have discussed in the next few paragraphs.

Intra Dataset Replacement. In this scenario, we replace a module from a set of modules decomposed from a DNN model with a module built on the same dataset but with different configurations (decomposed from a different DNN model). To evaluate these scenarios, we replace each module from the least complex model (based on the number of hidden layers (1)) from each dataset and replace that with a module of same output label from a more complex model (model with four hidden layers). Finally, we compute the composed accuracy of the modules and compare them with the accuracy of the DNN model from which the modules were decomposed and the prior accuracy of the modules. Table 4 depicts the scenarios for four datasets and we report the composed accuracy for MNIST-1, FMNIST-1, KMNIST-1, and EMNIST-1 with modules replaced from MNIST-4, FMNIST-4, KMNIST-4, and EMNIST-4, respectively. We found that replacing modules with decomposed modules from more complex models can increase the composed accuracy of the decomposed modules for 10 out of 40 scenarios. On average, there is an average 0.76% (median is 0.50%) drop in the accuracy when compared to the composed accuracy of the modules before replacement. Furthermore, we evaluate the inter dataset replacement on the example depicted in Figure 1. To make a part of the model faulty, we impose bias in the training dataset. We build a DNN model with 6000 training examples for all the output labels except the output label 5, where we use 500 examples. The trained DNN model achieves testing accuracy of 96.82%. Then, we decompose the DNN model intro modules and replace the module 5 with a module responsible for the same output label, decomposed from MNIST-4. Our result shows that the accuracy after the replacement is 98.66% (+1.84%). Thus, we can conclude that our approach can be utilized to replace a faulty piece of DNN model with a part taken from a better DNN model.

Inter Dataset Replacement. While a module can be replaced with a module with the same concern, there can be a situation that needs a module to be replaced with a different concern. Here, we replace one module from each dataset and replace that with a module taken from a different dataset, e.g., the module responsible for classifying English 1 replaced with a module for classifying Japanese 1. We validate our approach to replacing the modules from different datasets by conducting the experiments on the modules decomposed from the DNN models with four hidden layers (MNIST-4, EMNIST-4, and KMNIST-4). Our scenarios involve replacing modules from MNIST with KMNIST and modules from MNIST with EMNIST. Our results suggest that the accuracy of the replaced modules perform worse than the DNN models trained with the same structure (four hidden layers) with the training examples from the same output classes. In Table 5, we depict such scenarios. Our evaluation shows that by replacing modules from MNIST with Extended MNIST, the accuracy is decreased 1.62% on average (median 1.16%) in comparison to the DNN models trained with the same configurations. In the case of substituting modules from MNIST with Kuzushiji MNIST, the accuracy drop is 5.44% (median 5.40%) compare to the models trained from scratch.

Based on the overall evaluation, we found that replacing a module with similar concerns and different concerns can be achieved. However, there is a loss of accuracy in comparison to the models trained from scratch.

6 CONCLUSION

In this work, we explored whether a DNN can be decomposed so that parts of the DNN, which we call DNN modules, can be reused to build different DNNs or replaced with better DNN modules. We described our technique that relies on concern identification to identify the sub-network, identifying tangling of other concerns, and finally decomposing the sub-network into DNN modules. We described four different techniques for reducing tangling. We have evaluated our approach using four canonical datasets and sixteen different models. Often decomposition into modules has some costs, but we find that the costs for DNN decomposition is already very minimal. In 56.25% of cases, decomposed modules are slightly more accurate (0.00%-3.45%), and in remaining cases lose very little accuracy (0.01%-2.14%). The benefits of decomposition are observed in enabling reuse and replacement. We observe that for our datasets and models both reuse and replacement is possible. Based on these results, we believe that this work takes the first step toward enabling more modular designs for deep learning.

ACKNOWLEDGMENTS

This work was supported in part by US NSF under grants CNS-15-13263, and CCF-19-34884. All opinions are of the authors and do not reflect the view of sponsors. We thank ESEC/FSE'20 reviewers for constructive comments that were very helpful.

REFERENCES

[1] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature Interaction: A Critical Review and Considered Forecast. *Comput. Netw.* 41, 1 (Jan. 2003), 115–141. https://doi.org/10.1016/S1389-1286(02)00352-3

[2] Tarin Clanuwat, Mikel Bober-Irizar, Asanobu Kitamoto, Alex Lamb, Kazuaki Yamamoto, and David Ha. 2018. Deep Learning for Classical Japanese Literature. arXiv:cs.CV/cs.CV/1812.01718

[3] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. 2017. EMNIST: Extending MNIST to handwritten letters. *2017 International Joint Conference on Neural Networks (IJCNN)* (2017). https://doi.org/10.1109/ijcnn.2017.7966217

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

[5] Keras. 2020. Keras Dataset. https://keras.io/datasets/.

[6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 220–242.

[7] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[8] Nan Li, Ivor W Tsang, and Zhi-Hua Zhou. 2012. Efficient optimization of performance measures by classifier adaptation. *IEEE transactions on pattern analysis and machine intelligence* 35, 6 (2012), 1370–1382.

[9] Yi Liu and Yuan F Zheng. 2005. One-against-all multi-class SVM classification using reliability measures. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, Vol. 2. IEEE, 849–854.

[10] Eneldo Loza Mencía and Johannes Furnkranz. 2008. Pairwise learning of multilabel classifications with perceptrons. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*. IEEE, 2899–2906.

[11] Guobin Ou and Yi Lu Murphey. 2007. Multi-class pattern classification using neural networks. *Pattern Recognition* 40, 1 (2007), 4–18.

[12] Sinno Jialin Pan, Ivor W Tsang, James T Kwok, and Qiang Yang. 2010. Domain adaptation via transfer component analysis. *IEEE Transactions on Neural Networks* 22, 2 (2010), 199–210.

[13] D. L. Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. https://doi.org/10.1145/361598.361623

[14] Lorien Y. Pratt, Jack Mostow, and Candace A. Kamm. 1991. Direct Transfer of Learned Information among Neural Networks. In *Proceedings of the Ninth National Conference on Artificial Intelligence - Volume 2 (AAAI'91)*. AAAI Press, 584–589.

[15] Christian Prehofer. 1997. Feature-oriented programming: A fresh look at objects. In *ECOOP'97 — Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 419–443.

[16] Peter Prettenhofer and Benno Stein. 2010. Cross-language text classification using structural correspondence learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*. 1118–1127.

[17] Xiaoxiao Shi, Qi Liu, Wei Fan, S Yu Philip, and Ruixin Zhu. 2010. Transfer learning on heterogenous feature spaces via spectral transformation. In *2010 IEEE international conference on data mining*. IEEE, 1049–1054.

[18] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. 1999. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. IEEE Computer Society, Los Alamitos, CA, USA, 107–119. https://doi.ieeecomputersociety.org/

[19] Tensorflow. 2020. Tensorflow Dataset. https://www.tensorflow.org/datasets.

[20] Rui Xia, Chengqing Zong, Xuelei Hu, and Erik Cambria. 2013. Feature ensemble plus sample selection: domain adaptation for sentiment classification. *IEEE Intelligent Systems* 28, 3 (2013), 10–18.

[21] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747* (2017).

[22] Zhi-Hua Zhou. 2016. Learnware: on the future of machine learning. *Frontiers Comput. Sci.* 10, 4 (2016), 589–590.

[23] Weiwei Zong and Guang-Bin Huang. 2011. Face recognition based on extreme learning machine. *Neurocomputing* 74, 16 (2011), 2541–2551.