Decomposing a Recurrent Neural Network into Modules for Enabling Reusability and Replacement

Sayem Mohammad Imtiaz*, Fraol Batole*, Astha Singh*, Rangeet Pan†§, Breno Dantas Cruz*, and Hridesh Rajan*
* *Department of Computer Science, Iowa State University*, Ames, IA, USA
*{sayem, fraol, asthas, bdantasc, hridesh}@iastate.edu
† *IBM Research*, Yorktown Heights, NY, USA, rangeet.pan@ibm.com

***Abstract*—Can we take a recurrent neural network (RNN) trained to translate between languages and augment it to support a new natural language without retraining the model from scratch? Can we fix the faulty behavior of the RNN by replacing portions associated with the faulty behavior? Recent works on decomposing a fully connected neural network (FCNN) and convolutional neural network (CNN) into modules have shown the value of engineering deep models in this manner, which is standard in traditional SE but foreign for deep learning models. However, prior works focus on the image-based multi-class classification problems and cannot be applied to RNN due to (a) different layer structures, (b) loop structures, (c) different types of input-output architectures, and (d) usage of both non-linear and logistic activation functions. In this work, we propose the first approach to decompose an RNN into modules. We study different types of RNNs, i.e., Vanilla, LSTM, and GRU. Further, we show how such RNN modules can be reused and replaced in various scenarios. We evaluate our approach against 5 canonical datasets (i.e., Math QA, Brown Corpus, Wiki-toxicity, Clinc OOS, and Tatoeba) and 4 model variants for each dataset. We found that decomposing a trained model has a small cost (Accuracy: -0.6%, BLEU score: +0.10%). Also, the decomposed modules can be reused and replaced without needing to retrain.**

***Index Terms*—recurrent neural networks, decomposing, modules, modularity**

I. Introduction

Recurrent neural networks (RNNs), like fully-connected neural networks (FCNN) and convolutional neural networks (CNN), are a class of deep learning (DL) algorithms that are critical for important problems such as text classification. Depending on their architecture, they are further classified into vanilla RNN, LSTM (Long Short Term Memory), or GRU (Gated Recurrent Unit). To build such models, the most common way is by training from scratch. Otherwise, developers can also use transfer learning [1, 2] to reuse a model by retraining its last few layers. Such types of model reuse are coarse-grained, relying on the original model's entire structure. In contrast, we propose to decompose a trained RNN model to enable fine-grained reuse without needing to retrain.

The term '*modules*' has also appeared in the AI/ML community; however, it serves a different purpose [5, 6, 7, 8, 9]. They aim to add external memory capability to the DL models [9]. To illustrate, Ghazi *et al.* presents an example of a room and objects within [9]. A DL model is excellent in answering immediate questions such as "Is there a cat in the room"? However, suppose a person often visits a room over many years and later ponders an indirect question, "How often was there a cat?". In that case, a DL model is incapable of answering it. As a remedy, this line of work proposes to build a deep modular network consisting of many independent neural networks [8, 9, 5]. Essentially, they view a module as a function created in advance. Once a problem is given, they dynamically instantiate a composition of the modules to answer such questions. In all these cases, the end result is still akin to a monolithic model tasked with solving a particular problem. On the contrary, we aim to decompose a trained RNN model to enable the benefit of software decomposition.

TABLE I: Comparison with the existing works

Functionality		FCNN-D	CNN-D	Our Work
Input	Image	✓	✓	✓
	Sequential Data (e.g., Text)	X	X	✓
Model Properties	Models with more than one output	X	X	✓
	Models classify into one of the labels	✓	✓	✓
	Models with loops	X	X	✓
	Models with more than one input	X	X	✓
	Shared weight and bias	X	✓	✓
	Gated layer architecture	X	X	✓
	Non-linear activation function	✓	✓	✓
	Logistic activation function	X	X	✓

* FCNN-D: Fully Connected Neural Network decomposition approach [3], CNN-D: Convolutional Neural Network decomposition approach [4]

Along this line, recent work has proposed an approach to decompose FCNN and CNN models into modules and enable their reuse [3, 4]. However, those approaches cannot be applied to RNNs due to the challenges listed in Table I. For instance, RNNs, particularly LSTM and GRU, incorporate activation functions such as Tanh and Sigmoid in their internal architecture, which are unsupported by prior works. RNNs have five types of architecture depending on the network's input-output. Moreover, RNN includes a loop structure to process sequence data effectively. These differences render prior approaches inapplicable to RNNs.

Therefore, in this work, we ask: can we identify the RNN model parts responsible for each task and decompose them into modules? Doing so would allow us to (a) build a new problem or (b) replace existing model functionality. In both

§At the time this work was completed, Rangeet Pan was a graduate student at Iowa State University

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00093

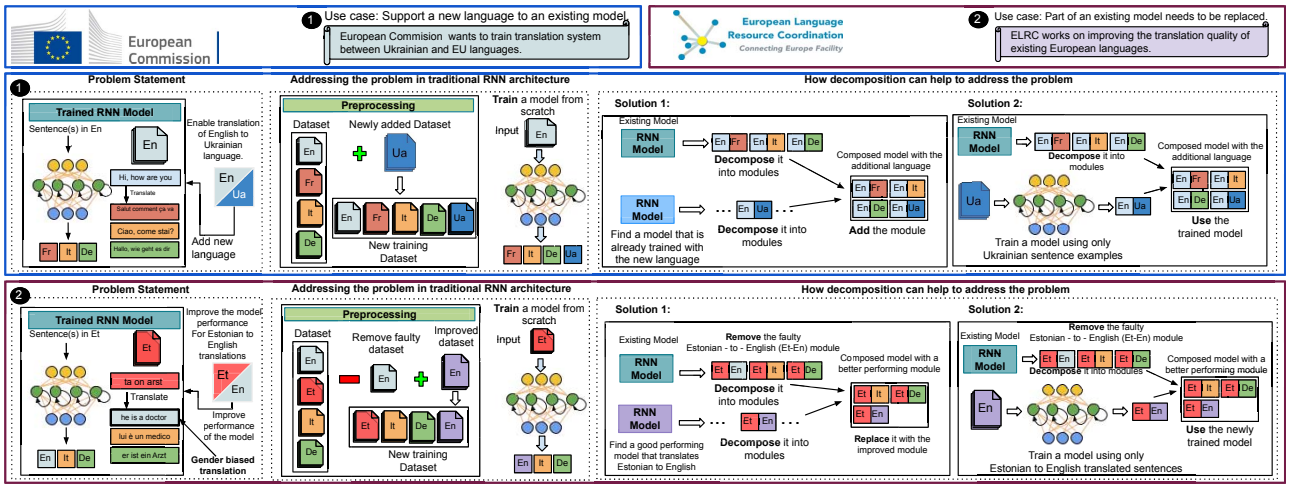

Fig. 1: Motivating Examples. De: German, En: English, Et: Estonian, Fr: French, It: Italian, Ua: Ukrainian Language.

cases, this type of reuse removes the need for additional data and training. To that end, we propose an approach for decomposing RNNs into modules. A key innovation in our work is handling the loop in RNN in both time insensitive and sensitive manner. Inspired by prior works on understanding loops in SE [10, 11, 12], we propose to identify nodes and edges responsible for each output class - (a) over all the iterations of the loop (*rolled*) and (b) individually, for each iteration in the loop (*unrolled*). *Unrolled*-variant is aware of the time dimension of the model, while *rolled* is not. Second, to handle different RNN architectures, we identify the concern (i.e., parts of the network responsible for classifying an output label) and untangle each output timestep at a time. In prior works, each concern is identified and untangled separately; this does not apply to models that produce many outputs. Third, we support models built using logistic activation functions, i.e., Tanh, Sigmoid, etc., which are commonly used in RNN [13, 14, 15]. In addition, we propose a decomposition approach assuming *ReLU* activation as well.

To evaluate our approach, we apply it to five different input-output (I/O) architecture types. Moreover, we evaluate different RNN-variants (LSTM, GRU, Vanilla) for each architecture. To that end, we utilize Math QA [16], Brown Corpus [17], Wiki-toxicity [18], Clinc OOS [19], and Tatoeba [20] datasets for training models in different setups, and decompose them. In total, our benchmark consists of 60 models, i.e., 4 (# models) * 3 (# RNN-variants) * 5 (# I/O architectures). In this work, we use the terms "RNN" or "recurrent model" interchangeably to refer to three RNN-variants collectively.

Key Results: To evaluate our approach, first, we measure the cost of decomposition by comparing the accuracy of the model composed using decomposed modules and the monolithic model from which the modules are decomposed. We found that the loss of accuracy is trivial (Avg.:-0.6%, median: -0.24%). For language translation models, there is a slight gain in performance (Avg.: +0.10%, median: +0.01%), measured in BLEU score [21]. We also find that the decomposition of models producing more than one output must be time-sensitive or aware of what output appears at what time. Second, we evaluated our approach to reusing and replacing the decomposed modules to build various new problems. We compared the accuracy of the models composed using decomposed modules with monolithic ones, trained from scratch. When reusing and replacing, we found that the performance change is (accuracy: -2.38%, BLEU: +4.40%) and (accuracy: -7.16%, BLEU: +0.98%), on average, respectively. **All the results and code for replication is available here [22].**

The key contributions of this work are as follows:

- We propose an approach to decompose an RNN model.
- Our proposed approach is applicable for all five types of I/O architectures.
- We propose two variants to support loops in RNN.
- Our approach supports both logistic and *ReLU* activation and all 3 RNNs, i.e., Vanilla, LSTM, and GRU.
- We show that our approach can reuse and replace the modules to build new problems without retraining.

II. Motivating Examples

In this section, we show two examples of using RNN models and how decomposing them into modules could help. The RNN models are used for multilingual language translation. Note that it is preferably performed in a multilingual setup to improve the overall performance of the translation task [23, 24]. Fig. 1 shows two examples in which decomposition can assist developers when building RNN models for translation.

Adding a new language to the European Union system. To introduce a new input (new language) to an existing RNN model, it must be trained from scratch. For instance, recently, the European Commission (EC) presented a new requirement to support the translation of languages that are not part of the European Union (EU) [25, 26]. It needs to be retrained to support the additional language. This process is expensive and time-consuming as developers must preprocess the data and train a model. Fig. 1 on the top right section shows how decomposition could help in this regard. Here, we have a monolithic model that translates an input in English into

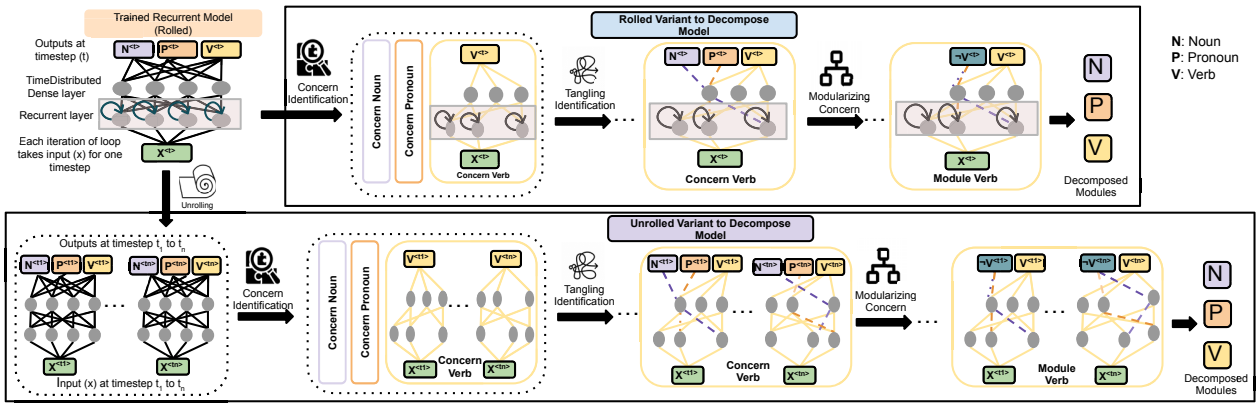

Fig. 2: High-level overview of our approach.

French, Italian, and German. Traditionally, a new model needs to be trained by adding new examples to the existing dataset to add the translation of English to the Ukrainian language.

In contrast, using our approach, one can take the existing trained model and decompose it into modules, in which each module is responsible for translating English to another non-English language. Then, find a different multi-lingual model trained to translate multiple languages, including English to Ukrainian, and decompose it. Next, take the module responsible for translating English to Ukrainian and compose it with the existing model's modules. Moreover, one can also train a small model that only supports English to Ukrainian translation. Then, compose it with the decomposed modules from the existing model. In both cases, training the large model can be avoided. The choice of approach depends on the available resources and module(s).

Altering a part of a trained model. Like any software system, neural network models can exhibit faulty behavior. For instance, in Estonian, "ta on arst" means "She is a doctor" in English. However, since Estonian is a gender-inclusive language, when we used the Google translator [27], the output was "He is a doctor", which is incorrect and gender-biased. In such scenarios, the European Language Resource Coordination (ELRC) is set to actively enhances the multilingual translation services of EU languages [25]. The most common approach to updating existing models is two-pronged: 1) introducing new examples to improve the faulty data, and 2) retraining the whole model. However, this approach can be resource-intensive because of the retraining.

In contrast, using our approach, one can handle the same problem in two different ways (Fig. 1 bottom right part). In the first approach, we decompose the existing trained model into modules. Then, we remove the Estonian to English translation module. Lastly, we use decomposition to select a replacement from a non-biased Estonian to English model. We retrain a small model using an improved dataset in the second approach. Then we compose it with the previously decomposed modules. In both cases, retraining the large model can be avoided. Thus, saving computational resources and hardware costs.

III. Related Works

In the SE community, a vast body of work in software decomposition [28, 29, 30, 31, 32] exists. The notion of modularity exists in the ML community too [5, 6, 7, 8, 9], however, to address different issues than what this work aims to deliver. They eventually produce a monolithic model in the sense that it does not enable fine-grained reuse.

Many studies reuse a DNN model to solve a software engineering task [33, 34, 35]. Transfer learning is a common technique to reuse the knowledge and structure of a trained model [1, 2]. However, retraining and modification are required when applying transfer learning to a different task. Moreover, replacing the model's logic cannot be achieved by transfer learning.

Along this line, the closest work was introduced by Pan and Rajan [3, 4]. They propose an approach to decompose an FCNN and CNN multi-class model into modules, which can be (re)used with other module(s) or be replaced by other modules to solve various problems. SaiRam et al. [36] identified sections of a trained CNN model to reuse when the target problem requires a set of output classes, i.e., the subset of the original model. In contrast, our decomposed modules can be reused and replaced with modules originating from the same and different datasets.

While these works introduce the notion of decomposition in various DL models, they cannot be directly applied to RNNs for the following reasons: (a) loops in the architectures, (b) logistic activations, and (c) different I/O modes. In contrast, our work addresses these novel technical challenges and proposes an approach to decompose RNN models into modules.

IV. Approach

In this paper, we propose a decomposition technique for recurrent models, where a binary module is produced for each output label. Figure 2 shows the overall approach. It starts with a trained model and produces decomposed modules. We propose two variants (*rolled* and *unrolled*) to that end. Each variant uses a different strategy to handle the loops in the RNN architecture. Like prior work [3, 4], broadly, the process of RNN decomposition has three steps – Concern

Identification (CI), Tangling Identification (TI), and Concern Modularization (CM). First, CI identifies the model parts contributing to a concern (an output class). After CI, a model can mostly recognize the target output class. Therefore, TI aims to add/update some parts responsible for negative output classes. Lastly, CM modularizes the concerns and creates a module that can recognize a single output class. Next, we describe technical challenges first, then each step in detail.

A. Challenge

Previous studies have proposed a decomposition approach for FCNN and CNN [3, 4]. However, RNN models significantly differ from other model types rendering prior approaches inapplicable to RNNs. This section discusses such differences that our approach addresses.

Challenge 1: Weight Sharing Across Time. Traditional neural networks (i.e., CNNs and FCNNs) do not have loops in their architecture – data directly flows from the input to the output layer. However, for RNNs, loops are leveraged to process sequential data. By doing so, the weights and biases are shared across time. Therefore, the decomposition algorithm must be aware of the time dimension in the network.

Challenge 2: Additional Learned Parameters. Unlike traditional neural networks, every RNN cell (node) receives two sets of learnable weights, i.e., weights associated with input at a timestep, W, and internal recurrent state or memory weights, U. The decomposition algorithm must decompose these additional learned parameters associated with memory.

Challenge 3: Gated layer architecture. Improved RNN variants, such as LSTM and GRU, perform gate operations, which enable both long and short-term memory, unlike vanilla RNN, which only involves short-term memory. For example, LSTM involves three gates in its internal architecture: input, forget, and output gate. Internal operations of an LSTM cell is shown in $lstm\_op$ method in algorithm 3. Each of these gates incorporates learnable (weights and biases) parameters of its own. Hence, the decomposition technique must consider these gates, i.e., identify relevant nodes across all gates.

Challenge 4: New activation function. Previous works on DNN decomposition only support *ReLU* activation function [3, 4], which are common in FCNN and CNN. However, in RNNs, logistic activation functions, such as *Tanh, Sigmoid*, are the most common [13, 14, 15]. Moreover, popular DL libraries such as *Keras, TensorFlow, and PyTorch* use *Tanh* as the default activation function for RNNs [37, 38, 39]. Therefore, to apply decomposition, we must support these functions.

Challenge 5: Multiple I/O Architecture. FCNN and CNN models process a single output from one given input (*1:1*). As a result, prior works [3, 4] were proposed for networks with a *1:1* architecture. However, there are 5 different input-output architectures in RNNs, as shown in Figure 3, which also need to be supported. **1:1** models receive a single input and produce one output. **1:N** models take a single input and produce sequential outputs. **M:1** models receive sequential inputs and produce one output. **M:N** models receive a sequence of inputs and produce many outputs. Figure 3 shows the two sub-types for *M:N* models. On the right, the architecture takes inputs and produces outputs at different timesteps (e.g., encoder-decoder models). The left one produces the output at the same timestep as it receives its inputs.

B. Concern Identification

Concern identification (CI) aims to identify parts of the network (nodes and edges) responsible for classifying an output label, OL, referred to as a concern. The definition of concern is aligned with the traditional SE and prior works on decomposing DNN into modules [3, 4]. The output label for which the concerned nodes and edges are identified is the dominant output class in the concern, while other classes are non-dominant. For instance, in Figure 2, for the *concern verb*, verb (*V*) is the dominant class, and pronoun (*P*) and noun (*N*), are the non-dominant classes. In that example, we show how a model, taking many inputs (words) and tagging each word with a part-of-speech (POS), can be decomposed into modules for each POS.

On a high level, CI aims to identify nodes and edges relevant to a particular concern or output class. In a ReLU-based network, relevant nodes can be identified by observing activation values for a sample of that class, and nodes that always remain active can be treated as relevant. However, because of logistic activation used in RNNs, where the notion of "active" or "inactive" is not as distinct as in ReLU, we identify relevant nodes by comparing the central activation level of a node in both positive and negative samples. Besides that, CI for RNN must also be aware of the time loop, multi-output, and gated architecture of RNNs. Our approach unrolls the time loop in RNN and performs CI (i.e., identifying relevant nodes) in both a time-sensitive (unrolled) and insensitive manner (rolled). Moreover, improved RNN variants (i.e., LSTM) involve gates in its architecture, which we handle by identifying relevant nodes across all gates. Finally, a different CI approach is needed for multi-output models from a single one, as one input example can be labeled with multiple classes (concerns). While CI for single output can monitor a single input sample for one particular class, we must enable simultaneous detection of multiple concerns for multi-output models. Next, we discuss our approaches to handling these challenges while decomposing an RNN model in detail.

1) Identifying Concerns Across Time: First, in the RNN models, the weights are shared across timesteps (in RNN notations, each point in time is called a *timestep*). As a result, different nodes may activate at distinct timesteps. Therefore, in our approach, we propose two CI variants. In particular, the *rolled*, which does not take the timesteps of a node into account, and the *unrolled*, which is timestep-sensitive.

Rolled: In this variant, we identify the concern and remove the unrelated nodes and edges. Instead of keeping multiple copies of the updated edges and nodes (one for each timestep), we keep a single copy of the modified edges and nodes. All timesteps share the same weights obtained after removing edges and nodes.

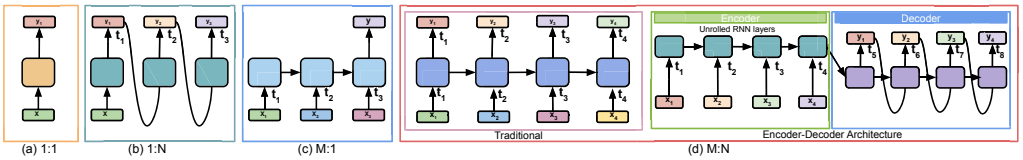

Fig. 3: Different RNN input/output (I/O) architectures.

Algorithm 1 Decomposition:{One, Many}-to-One Models.

```
procedure ONE(model, activation, rolled, input_timestep, X, Y)
    modules = [ ]
    for every output label, ol do
        p_in = sample(X, Y, only = ol, size = M)
        n_in = sample(X, Y, not = ol, size = M/|class| − 1)
        concern_o = initConcern(model, timestep, rolled)
        h_val_pos = monitor(p_in, model, rolled)
        h_val_neg = monitor(n_in, model, rolled)
        if rolled then
            flat_p = flatten_obs(model, h_val_pos, timestep)
            flat_n = flatten_obs(model, h_val_neg, timestep)
            update_concern(concern_o, flat_p, flat_n, activation)
        else
            for ts ∈ input_timestep do
                p_ts = obs_at(model, h_val_pos, ts)
                n_ts = obs_at(model, h_val_neg, ts)
                con_ts = initConcern(model, timestep, True)
                update_concern(con_ts, p_ts, n_ts, activation)
                merge(concern_o, con_ts, ts)
        cur_module = channel(concern_o)
        append(modules, cur_module)
    return modules
```

Unrolled: In traditional SE, to understand the impact of each loop iteration, prior works have proposed approaches [10, 11, 12] to unroll the loops. Inspired by such works, the unrolled variant follows a two-step concern identification process. In particular, it first unrolls the RNN model loop while transforming it into an equivalent sequence of operations. Then, it identifies the nodes and edges responsible for each concern at each timestep.

Algorithm 2 Decomposition: {One, Many}-to-Many Models.

```
procedure MANY(model, activation, rolled, output_timestep, X, Y)
    modules = [ ]
    for every output label, ol do
        concern_o = initConcern(model, timestep, rolled)
        flat_p = []
        flat_n = []
        for every output_timestep, ts do          ▷ monitor one step at a time
            p_in = sample(X, Y, ts, only = ol, size = M)
            n_in = sample(X, Y, ts, not = ol, size = M/|class| − 1)
            h_val_pos = monitor(p_in, model, rolled)
            h_val_neg = monitor(n_in, model, rolled)
            p_ts = obs_at(h_val_pos, ts)
            n_ts = obs_at(h_val_neg, ts)
            if rolled then
                concat(flat_p, h_val_pos_ts, axis = 0)
                concat(flat_n, h_val_neg_ts, axis = 0)
            else
                con_ts = initConcern(model, timestep, True)
                update_concern(con_ts, p_ts, n_ts, activation)
                concern_o[ts].W, U, B = con_ts.W, U, B        ▷ Merge
        if rolled then
            update_concern(concern_o, flat_p, flat_n, activation)
        cur_module = channel(concern_o)
        append(modules, cur_module)
    return modules
```

2) Support logistic activation: Unlike *ReLU*, logistic activations squash the given input within a certain range [40]. For example, Tanh squashes input in range $[-1, +1]$, and Sigmoid in range $[0, 1]$ [40]. In that regard, Lipton [15] *et al.* reported that the hidden layers of a model using logistic activation are rarely sparse, unlike *ReLU*-based layers. This is because a node value is rarely exactly zero for logistic activation-based layers, unlike *ReLU*-based ones [15]. As a result, prior *ReLU*-based decomposition techniques, which leverages the notion of "on" or "off" ("off", if node value is 0, otherwise, "on") to identify concerned nodes, do not apply to the logistic activation functions.

Algorithm 3 Helper algorithms for algorithm 1 and 2

```
procedure INITCONCERN (model, timestep, rolled)
    con : object                                  ▷ Initialize new model object
    for each layer, l ∈ model do
        if layer.generic_type == "Recurrent" and not rolled then
            for each ts ∈ timestep do
                con[l].W[ts], U[ts], B[ts] = l.get_weights()
        else
            con[l].W, U, B = l.get_weights()      ▷ U=N/A if dense layer
    return concern
procedure MONITOR(samples, model)
    hidden_val = [ ]
    for x ∈ samples do
        for each layer ∈ model do
            if layer.generic_type == "Recurrent" then
                for ts ∈ timestep do
                    if layer.type == "LSTM" then
                        h_t, c_t = lstm_op(layer, x, h_{t−1}, c_{t−1})
                    ...
                    append(hidden_val, ts, layer, h_t)
            if layer.type == "Dense" then
                h_t = x.dot(layer.W) + layer.B
                append(hidden_val, layer, h_t)
    return hidden_val
procedure FLATTEN_OBS(model, hidden_val, timestep)
    flattened = [ ]
    for s ∈ timestep do
        for each layer ∈ model do
            append(flattened, layer, hidden_val[ts][layer])
    return flattened
procedure OBS_AT(model, hidden_val, ts)
    values = [ ]
    for each layer ∈ model do
        append(values, layer, hidden_val[ts][layer])
    return values
procedure UPDATE_CONCERN(concern, h_pos, h_neg, activation)
    if activation == "logistic" then
        ct_pos = central_tendency(h_pos, model)
        ct_neg = central_tendency(h_neg, model)
        CI_logistic(ct_pos, ct_neg, concern)      ▷ identify concern
    if activation == "relu" then
        rate_pos = active_rate(h_pos, model)
        rate_neg = active_rate(h_neg, model)
        CI_relu(rate_pos, concern)                ▷ identify dominant nodes
        TI_relu(rate_neg, concern)                ▷ identify tangling nodes
    return concern                                ▷ Return updated concern
procedure LSTM_OP (layer, x_t, h_{t−1}, c_{t−1})  ▷ Apply LSTM Op.
    s_t = x_t.dot(layer.W) + h_{t−1}.dot(layer.U) + layer.B
    n = layer.num_hidden_neuron
    i, f, o = σ(s_t[:, : n]), σ(s_t[:, n : 2n]), σ(s_t[:, 3n :])      ▷ Gates.
    c_t = i ∗ tanh(s_t[:, 2n : 3n]) + f ∗ c_{t−1}                     ▷ Cell state.
    h_t = o ∗ tanh(c_t)                                               ▷ Hidden state.
    return h_t, c_t
```

To address this issue, our intuition is that a logistic activation value can be regarded as the excitation level of a node. A higher activation value indicates to what extent it lets information through that node. For instance, an activation value 1 for *Sigmoid* allows maximum information through that node, while 0.0 is the lowest. Leveraging this insight, we compare the central activation tendency of a neuron in positive and negative examples and identify concerns (Algorithm - 4). To that end, first, we sample a set of positive (inputs with target output label) and negative examples (inputs with other output labels). Then, we monitor the activation of nodes for both sets while comparing their central activation tendency. The method $central\_tendency$ measures the central activation tendency of a neuron. First, it retrieves the distribution of activation values for the observed examples (Line 22 of Algorithm 4) for a neuron, $node$. Note that the absolute value of observed activations is taken (Line 23 of Algorithm 4) as *Tanh*-activated values could lie in the range [-1,+1] and -1 allows as much information as +1, but in the opposite direction. Then, it computes the mean as a measure of central activation tendency for that neuron after discarding outlier observations (Line 26 of Algorithm 4). In method $CI\_logistic$, a node is considered more relevant to the dominant output label if its central tendency is higher in positive examples than in negative ones (Line 8 of Algorithm 4). Because it tends to allow more information to pass through for the dominant class than for other classes, we keep this node and edges going in/out of it for the dominant class. The method stops removal if the graph becomes too sparse. To that end, taking the 10% removal threshold used by Pan *et al.* [3] as a starting point, we evaluate the decomposition quality at a 5% interval (i.e., 10%, 15%, 20%, etc.). For RNN models, we found that beyond 20%, the graph starts to become too sparse, affecting decomposition adversely, and we selected this as the threshold in our experiments.

3) Support gated layer: To support the decomposition of gated layer architecture, we propose two approaches: i) decompose each gate based on its own activation, and ii) decompose all gates based on LSTM/GRU final hidden state. Note that hidden states or values refer to the output of the model's intermediate or hidden layers (between input and output layers). In the first approach, we identify dominant nodes in each gate based on their own activation values, while, in the latter, we remove a node from all gates if its corresponding hidden state value is found to be less significant in positive samples. The hidden state at a particular timestep dictates the output at that step. As a result, a node's hidden value indicates its relevance to the output produced at that step. Therefore, in the second approach, we compare a node's central hidden state tendency for positive and negative examples. We remove it from each gate if found to be more relevant to non-dominant classes or negative examples (Line 8-15 in Algorithm 4).

4) Support Multiple I/O Architectures: To support the CI in models with multiple input and output classes (i.e., *1:N*, and *M:N*), we propose two approaches for two different output modes, one and many. Algorithm 1 and 2 show the details as described next.

Algorithm 4 Concern identification for logistic activation.

```
1: procedure CI_LOGISTIC (ct_pos, ct_neg, concern, thres)
2:     d = {}
3:     for each layer, l ∈ concern do
4:         for each node, n ∈ l do
5:             d[l][n] = ct_pos[l][n] − ct_neg[l][n]
6:     d = sort(d, order = "ascending")
7:     for each layer, node ∈ d.keys() do
8:         if d[layer][node] < 0.0 then
9:             removeNode(concern, layer, node)
10:            if layer.type = "LSTM" or layer.type = "GRU" then
11:                hunit = layer.num_hidden_neurons
12:                removeNode(concern, layer, hunit + node)
13:                removeNode(concern, layer, 2 * hunit + node)
14:                if layer.type = "LSTM" then
15:                    removeNode(concern, layer, 3 * hunit + node)
16:        if removedPercent >= thres then
17:            stop
       return concern                                  ▷ Return updated concern
18: procedure CENTRAL_TENDENCY(hidden_val, concern, activation)
19:    tendency = {}
20:    for each layer, node ∈ concern do
21:        all_obs = []
22:        for each observation, o ∈ hidden_val do
23:            all_obs.append(abs(hidden_val[layer][node][o]))
24:
25:        d = remove_outliers(all_obs)
26:        tendency[layer][node] = mean(d)
       return tendency
```

{One, Many}-to-One. Like any traditional network, RNN can take a single input and produce a single output. However, RNN can also take many inputs (i.e., sequential data). Algorithm 1 shows our approach for concern identification in the presence of an input loop and models that produce a single output. Algorithm 1 receives a trained model, activation type, modularization mode (i.e., *rolled* or *unrolled*), *timesteps* (1 for *one*, > 1 for *many* at input end), and input examples. Next, it iterates through every output label (OL) to create a module for each output label. To identify the concern for OL, first, our approach selects the examples labeled as the dominant output class in the training dataset (Line 4 of Algorithm 1), which we call a positive sample. Also, it selects negative samples proportionally from each negative class (Line 5 of Algorithm 1).

Then, in line 6 of algorithm 1, *initConcern* method initializes the modular weights. In particular, it initializes the modular weights *W*, *U*, and *B* with the trained model ones. These nodes and edges from the weights are removed and updated as the algorithm proceeds to identify a concern. In the *unrolled* mode, for *recurrent* layers, *initConcern* creates *timestep* copies of *W*, *U* and *B* to allow individual timestep-specific weight pruning (Line 5-6) in Algorithm 3).

In the next step, each neuron activation is monitored for both positive and negative samples (Line 7-8 of Algorithm 3) by feeding them to the trained model. $monitor$ method of the Algorithm 3 shows how the nodes are observed. In particular, each example is propagated through the trained model while recording the node hidden values for each example across all timesteps. The method handles different types of trainable layers found in a recurrent model. For recurrent layers, the method handles the presence of a loop by implementing a

feedback loop (Line 14 of Algorithm 3). For example, for LSTM layers, LSTM cell, $lstm\_op$, is repeatedly fed with an input, x_t, at a particular *timestep*, *t*, previous hidden and cell state (Line 16 of Algorithm 3). The cell performs a stateful input transformation, x_t, by using the contextual information from the previous cell and recording hidden values at each timestep (Line 18 of Algorithm 3).

Algorithm 5 Concern identification for *ReLU*.

```
1: procedure ACTIVE_RATE(hidden_val, concern)
2:     rate = {}
3:     for each node, n ∈ concern do
4:         activeCounter = 0
5:         for each observation, o ∈ hidden_val do
6:             if isNodeActive(hidden_val[layer][n][o]) = True then
7:                 activeCounter+ = 1
8:         rate[n] = (activeCounter/len(hidden_val)) * 100.0
       return rate
9: procedure CI_RELU(active_rate, concern, thres)
10:    for each node n ∈ concern do
11:        if active_rate[n]==0.0 then
12:            removeNode(concern, layer, n)
13:        if removedPercent >= thres then
14:            stop
       return concern                          ▷ Return updated concern
15: procedure TI_RELU(active_rate, concern)
16:    for each node, n ∈ concern do
17:        if active_rate[n] > 0.0 then
18:            restoreNode(concern, layer, n)
       return concern                          ▷ Return updated concern
```

Rolled-variant of the algorithm identifies concern for the dominant class in a timestep-insensitive manner. Instead of identifying concerns in each timestep separately, all timesteps share the same identified nodes and edges for an output class. To that end, method $flatten\_obs$ is called to flatten observations/hidden values from all timesteps (Line 10 of Algorithm 1). This method essentially treats observed activation values of nodes at each timestep as a distinct observation of its own. For example, consider a model with timestep, 10, and 100 input samples to observe. In this case, each neuron will have $100 * 10$ observations after invoking $monitor$. In particular, it will observe a neuron, X, 100 times in each timestep. However, *rolled*-variant will treat hidden values from different timesteps for a neuron as a separate observation as if there were $100 * 10$ input samples. Hence, in this mode, each neuron will have 1000 observations.

Then, the concern is updated based on the 1000 observations in the $update\_concern$ method (Line 12 of Algorithm 1). For logistic activations, in $update\_concern$, it first computes the central activation tendency for these 1000 observations (Line 33-34 of Algorithm 3). Then, it identifies relevant nodes and edges for the current concern, OL. Similarly, for *ReLU*, it computes the active percent of a node given these observations and identifies relevant nodes accordingly (Line 37-40 of Algorithm 3). In particular, we keep nodes that are observed to be always *active* (for Relu-based models) or comparatively more intensely activated (for logistic-based models); otherwise, removed as shown in Algorithm 4 and 5.

However, in *unrolled* mode, the algorithm is timestep-sensitive as it goes to identify dominant nodes at each timestep separately (Line 14-19 of Algorithm 1). It iterates through each timestep and retrieves observations at that timestep (Line 15,16 of Algorithm 1). Then, it creates a temporary object, con_ts, to represent concern at this particular timestep. Finally, it merges the identified relevant nodes at this timestep to the concern under analysis, $conern\_o$, by adding them at that timestep (line 19 of Algorithm 1).

{One, Many}-to-Many. Unlike {*One, Many*}*-to-One*, an input example can be associated with multiple output classes in {*One, Many*}*-to-Many* models. For instance, the POS-tagging example shown in Figure 2 has many outputs. In this example, every individual word in a given sentence is associated with a POS tag in the output. This kind of many-output problem poses unique challenges to the decomposition technique proposed for DNNs in the past [3, 4]. The one-output-based technique can uniquely monitor a single input sample for one particular output label, as shown in Algorithm 1. However, it is not possible for many-output models as multiple concerns may be present simultaneously in a single input. Therefore, to decompose such models, our insight is to monitor each output timestep at a time, as shown in Algorithm 2.

The Algorithm 2 starts by receiving the same parameters as Algorithm 1. Then, to build one module for each output class, it identifies dominant nodes in each timestep separately (Line 7 of Algorithm 2). For each output timestep, it similarly samples positive and negative examples to Algorithm 1. However, the sampling procedure is timestep-sensitive (Line 8 of Algorithm 2). For example, consider an output label 'V' and timestep 2. Then, input with the label 'V' at the second timestep will be treated as positive if 'V' is present at timestep 2 and negative otherwise, regardless of other labels in other timesteps. After sampling, the algorithm monitors the neurons (Line 10 of Algorithm 2). Next, it retrieves only observations at the current timestep, as other observations at other timesteps are irrelevant as they can be associated with other output labels.

Next, in *rolled* mode, all observations at the currently monitored timestep are concatenated with previous observations at other timesteps as a distinct observation (Line 15 of Algorithm 2). For example, for a model with 10 timesteps and 100 examples at each timestep, there will be 1000 observations per neuron in *rolled* mode (Line 22 of Algorithm 2). However, unlike Algorithm 1, it takes 100 observations from timestep 0 when CI is done for timestep 0, ignoring other (9*100) observations. Then, the next 100 observations are taken from timestep 1 when CI is done for timestep 1, ignoring the other 900 observations and so on. Finally, it uses all 1000 observations to identify the concerns (Line 22 of Algorithm 2). As such, in many-output models, *rolled*-variant is unaware of the association between an output label and timestep, i.e., contextual information on what labels appear at what step usually. However, in *unrolled* mode, dominant nodes are identified only based on current observations (Line 18 of Algorithm 2), and therefore, this variant is capable of identifying concerns in a timestep-wise output-sensitive manner.

C. Tangling Identification

The CI stage mostly identifies nodes relevant to positive samples. However, a module is still required to recognize negative classes. Therefore, tangling identification (TI) aims to bring back some nodes after observing negative samples. TI is particularly important for *ReLU*-based decomposition as the CI stage only keeps the most active nodes after observing positive examples. As a result, it may only recognize the dominant output class, thus becoming a single class classifier. However, for logistic activation-based models, we use a different approach during CI that keeps nodes showing higher central activation tendencies in positive examples. This technique also keeps some tangled nodes as it does not remove nodes that are almost similarly activated in both samples.

For *ReLU*-based models, we bring back a few nodes and edges related to the non-dominant concerns by observing the negative examples. For one-output models, in *rolled* mode, the observations are flattened, similar to the CI stage (Line 11 of Algorithm 1). Then, it restores a node if it is active in some negative examples (TI_relu method in Algorithm 5). In *unrolled* mode, TI is performed in a timestep-sensitive manner as was done for CI. It restores a node in a timestep if it is found to be active in some negative examples in a timestep (Line 16 of Algorithm 1).

D. Concern Modularization

For Concern Modularization, we channel the output layer to convert N output nodes into two types, dominant (D) and non-dominant (ND) nodes. For each node in the layer before the output layer, we average all the edges connecting to non-dominant output class nodes. Then, we connect these nodes to the newly introduced non-dominant ones. Next, we remove all other edges from nodes in the preceding layer. Thus, it converts the output layer into a binary classification-based problem. For example, given an input, the module recognizes whether it belongs to a dominant output class. For many-output models, it performs this operation for each timestep, except for the encoder-decoder architecture. For instance, in language translation models, each module produces a single operation translating an input sentence to a different language.

V. Evaluation

This section describes the experimental setup and evaluates our approach using three research questions.

A. Experimental Setups

1) Datasets: We perform our experiment on five widely used datasets for text-based sequential problems. Each dataset is used to train different types of RNN models.

MathQA [16]: This dataset contains a series of mathematical questions. Each question has a particular tag (e.g., geometry, physics, probability, etc.). Also, the dataset has a total of 6 output classes.

Wiki-toxicity [18]: This dataset contains Wikipedia pages' comments. Each comment is annotated with seven toxicity labels (e.g., toxic, severe toxicity, obscene, threat, insult, etc.).

Clinc OOS [19]: In NLP, intent classification is a well-known problem in which an input text is categorized based on a user's needs. This dataset has ten output classes.

Brown Corpus [17]: It contains English linguistic data. Each word is annotated with a part-of-speech tag from 12 different tags.

Tatoeba [20]: This dataset contains sentences in more than 400 languages. Each sentence in English is translated into other languages. Thus, the dataset is especially used for multilingual evaluation [**?** 41, 23]. We selected English, Italian, German, and French languages as they have the richest (# of training data) corpus.

2) Models: For every RNN variant, we built four models for each of the five I/O architectures. Specifically, we used 1, 2, 3, and 4 RNN layers to build models and named them RNN-<no. of RNN layers> (RNN refers to either of LSTM, GRU, or Vanilla). The structure of the models has been inspired by prior works [3]. Moreover, the data pre-processing and architecture of the language models are based on a real-world example [24]. In model architecture, we use a combination of 8 different Keras layers; (a) `Embedding` represents words as a fixed-length high-dimensional vector, (b) `RepeatVector` repeats the input n times, (c) `Flatten` converts a multidimensional input into a single dimension, (d) `SimpleRNN` is the vanilla RNN layer, (e) `LSTM`, (f) `GRU`, (g) `Masking` ignores the padded inputs, and (h) `TimeDistributed` applies the same layer across timesteps.

3) Evaluation metrics: To evaluate, we use three metrics.

Accuracy. For comparing the trained model with the modules, we use testing accuracy as one of the metrics. We interchangeably use the trained model accuracy (TMA), monolithic model accuracy (MMA), etc. For the modules, we use a voting-based approach (similar to [3, 4]) to compute the composed accuracy. All modules in a problem receive the same input, with a joint decision computed at the end. We use the following terminologies interchangeably – composed model accuracy (CMA) and module accuracy (MA).

BLEU Score [21]. For language translations, we use the BLEU score as it is widely applied [42, 43].

Jaccard Index. [3] We compute the Jaccard index (JI) to measure the similarity of the model and the modules.

B. Results

In this section, we present the results and discuss them briefly. We evaluated the decomposition on 60 models (20 for each RNN variant). Moreover, we repeat the experiments in both *rolled*, and *unrolled* modes. Due to space limitations, we only present a summary of the results (detailed results can be found here [22]). Moreover, for gated RNN variants, we repeat all experiments in two proposed approaches for decomposing gates in § IV-B2. We found that the decomposition cost of both approaches is comparable, and only results from the second approach are presented here.

1) RQ1: Does decomposing RNN model into modules incur cost? : In this research question, we evaluate the cost of decomposition in 60 scenarios. To that end, we determine

TABLE II: Cost of Decomposing RNN into Modules

I/O Type	Dataset	Mode		LSTM-1	LSTM-2	LSTM-3	LSTM-4	GRU-1	GRU-2	GRU-3	GRU-4	Vanilla-1	Vanilla-2	Vanilla-3	Vanilla-4	Avg. JI
1:1	Math QA	Rolled		**+0.13**	**+0.07**	-0.03	**+0.20**	**+0.10**	**+0.10**	**+0.10**	-0.07	-0.30	-0.67	-0.64	-1.01	0.75
		Unrolled														
M:1	Clinc OOS	Rolled		-0.13	-0.44	-1.40	-0.44	-0.20	**+0.22**	-0.22	-0.47	-1.58	-5.51	-2.27	-2.40	0.85
		Unrolled		**+0.96**	**+0.51**	**+0.07**	-0.67	**+0.09**	**+0.00**	-0.09	-0.69	-0.07	-1.38	-0.87	-0.11	0.86
1:N	Toxic Comment	Rolled		-14.73	-14.98	-94.59	-80.78	-0.49	-1.82	-6.05	-1.02	-15.21	-14.96	-15.15	-15.28	0.86
		Unrolled		-0.01	**+0.05**	**+0.17**	**+0.44**	-0.27	-0.76	**+0.04**	-0.14	-0.97	-1.48	-0.36	-2.78	0.86
M:N	Brown Corpus	Rolled		-77.07	-76.56	-80.55	-84.63	-75.28	-75.48	-78.04	-78.11	-80.13	-80.18	-79.93	-79.93	0.83
		Unrolled		-0.52	-0.22	-2.04	-2.84	-0.02	-0.77	-0.52	-2.59	-1.02	-1.64	-2.90	-3.37	0.83
M:N (Encoder - Decoder)	Tatoeba	Rolled	EN-FR	**+0.02**	**+0.18**	-0.45	-0.49	**+0.25**	-0.06	**+0.02**	-0.05	**+0.13**	**+0.16**	-0.68	-1.04	0.80
			EN-DE	**+0.05**	**+0.06**	-0.07	-0.15	-0.16	**+0.03**	**+0.01**	**+2.01**	-0.02	-0.48	-0.56	-0.28	0.80
			EN-IT	**+0.00**	**+0.22**	**+0.08**	-0.18	**+0.44**	**+0.45**	**+0.99**	**+0.33**	-0.57	-1.60	**+2.38**	**+2.51**	0.80
		Unrolled	EN-FR	**+0.04**	**+0.31**	-0.18	**+0.14**	**+0.44**	-0.45	-0.50	-0.11	-0.25	**+0.25**	-0.38	-2.46	0.79
			EN-DE	**+0.01**	**+0.00**	-0.35	-0.35	**+0.15**	-0.17	-0.65	-0.02	-0.85	-0.26	-3.00	-0.34	0.79
			EN-IT	-0.10	**+0.12**	-0.03	-0.25	-0.51	-0.26	-0.26	-0.50	**+0.76**	-0.52	**+3.59**	-0.48	0.79

All values are in % except Avg. JI. Here, 1:1=one-to-one,M:1=many-to-one,1:N=one-to-many,M:N=many-to-many

the quality of the decomposition and composition approaches. First, we decompose a trained RNN model into modules. Each module receives the same input and recognizes an output class. Next, we use the modules to compose a new model using a voting-based approach. The modules' decisions are combined into one that matches the output type. For example, the final decision is a single output class for {one, many}-to-one. Whereas, for {one, many}-to-many, the final decision will be a list of output classes. Then we compare the composed accuracy with the monolithic one.

We apply the *rolled* and *unrolled*-variants to decompose the 60 models, and the decomposition cost is depicted in Table II in terms of accuracy difference, $\delta = CMA - MMA$. In the *rolled*-variant, we identify the concern for all the timesteps at once. For the *unrolled*-variant, we check the concerns for each timestep separately after unrolling loops in the RNN model. We evaluate the *unrolled*-variant with the *M:1*, *1:N*, and *M:N* architectures. The unrolled-variant does not apply to the *1:1* architecture as it does not contain loops. We found an average loss of 25.8% (median: -2.04%) accuracy for the *rolled*-variant. In contrast, the average accuracy loss is 0.74% (median: -0.44%) for the *unrolled* one. Moreover, in 31.25% scenarios, CMA remains the same or improves (considering *rolled* mode for *1:1* and *unrolled* for others).

For the *rolled*-variant, the bulk of the accuracy losses come from the many-output models, while the differences in the one-output model are trivial (Table II). In *rolled* mode, the average accuracy loss for one-output models is -0.7%, while it is -50.87% for many-output models. However, in *unrolled* mode, decomposition quality for both one and many-output models are comparable (avg. loss for one-output: -0.2% and for many-output: -1.02%). This is because *rolled*-variant is insensitive to timestep, therefore, more likely to lose output-related contextual information at a timestep. In other words, timestep-specific output sensitivity is lost in *rolled*-variant. Many-output models, where each timestep has an output, require that concern is identified for that output based on what is observed in that timestep, which *unrolled*-variant does. On the other hand, one-output models only rely on the hidden state of the final RNN cell, requiring no output sensitivity for different timesteps. Therefore, we recommend *unrolled*-variant for many-output models, particularly where each timestep-output is subject to decomposition.

For language models, we compute the BLEU score for each pair of languages. This score measures their translation quality [21]. We found an average gain of 0.10% (median: +0.01%) for the *rolled*-variant in the BLEU score. While, in *unrolled* mode, the average BLEU score loss is -0.2% (median: -0.25%). Based on the argument by [44], such a change in the BLEU score does not affect the quality of the translation. Furthermore, for 52.8% cases in *rolled* mode, the composed model's BLEU score remains the same or improves compared to the original one.

Similarity: Apart from the cost, we also measure the structural similarity (in terms of learned parameters) between the monolithic model and module to assess the effectiveness of decomposition. A high similarity indicates an ineffective decomposition approach, creating modules replicating the original model. To evaluate the variability among modules and models, we compute Jaccard Index (JI). We found that, on average, the *JI* value for the rolled-based approach is 0.82, and for the unrolled one is 0.83. This result shows that the modules are significantly different from the parent models. Overall, we found that the RNN model can be decomposed into modules at a very small cost. Also, the decomposed modules are significantly different from the original model.

2) RQ2: Can Decomposed Modules be Reused to Create a New Problem?: In this RQ, we reuse the decomposed modules. Our evaluation focuses on two reusability cases: a) (re)use the modules within the same input-output (I/O) type, and b) from a different type, to create a new problem. We perform these experiments separately for three RNN variants (LSTM, GRU, and Vanilla). Next, we discuss the results from each case.

TABLE III: Summary of intra and inter-reuse experiments

Reuse Type	I/O Type	LSTM		GRU		Vanilla	
		Mean	Median	Mean	Median	Mean	Median
Intra	1:1	-0.07	**+0.00**	-0.03	**+0.00**	-0.05	**+0.00**
	M:1	-0.82	-0.67	-0.73	-0.44	-0.53	-0.44
	1:N	-2.20	-0.31	-5.49	-1.25	**+0.38**	**+0.02**
	M:N	-0.50	-0.71	**+0.42**	**+0.24**	**+0.01**	-0.02
	M:N Encoder-Decoder	**+5.20**	**+3.49**	**+3.83**	-1.31	**+4.17**	**+3.00**
Inter	1:1-1:N	-2.49	-1.25	-7.93	-3.10	-8.23	-7.68
	M:1-M:N	-2.92	-1.68	-3.28	-2.50	-3.55	-2.63

All values are in %.

Intra RNN Type Reuse. To evaluate this reuse type, we

take modules from the same I/O type of RNN. To do so, we use the dataset available in our benchmark. We take two modules, compose them, and evaluate the accuracy of the composed models. Additionally, we train a model with the same model architecture of the modules with examples of the dominant classes of the modules. For example, consider the case of intra-reuse for a *M:1 GRU* model trained on the Clinc OOS dataset. In this case, consider the two output toxicity levels: *severe* and *threat*. First, a model is trained from scratch using inputs of these two labels alone to reuse them. Then, we take corresponding modules from a previously decomposed model with all labels and compare their composed accuracy with that of the trained one newly.

We take 2 modules from each trained model and build a sub-problem. The total possible combinations for taking 2 modules from a model trained with a dataset having N output classes are $\binom{N}{2}$. Our benchmark has 6, 7, 10, and 12 output classes in the datasets used to train the models for *1:1*, *1:N*, *M:1*, and *M:N* (traditional) architectures, respectively. So, the total possible combinations can be 152 ($\binom{6}{2}$ + $\binom{7}{2}$ + $\binom{10}{2}$ + $\binom{12}{2}$). For each case, we train a model from scratch using two labels. Then, to get composed accuracy from reused modules, we consider the modules decomposed from RNN-4 (one with the highest number of layers), similar to prior work [3]. Then, we compare their accuracies to understand the effectiveness of intra-reuse. We found that for *1:1*, *1:N*, *M:1*, and *M:N* (traditional), the change of accuracy is -0.05% (median 0%), -2.44% (median -0.17%), -0.69% (median -0.56%), and -0.02% (median 0%), respectively.

In Table III, we report the results. Overall, there is a slight loss (mean: -0.58%, median: -0.11%) of accuracy when modules are reused. We also perform a similar evaluation for language translation modules. Since there are 3 modules produced from the trained model, taking 2 modules at a time can create 3 possible scenarios. We also train a multi-lingual model that takes English sentences as input and converts it into the 2 chosen non-English languages. We report the results in Table III. We found that there is an average gain of 4.40% BLEU score (median: +2.66%) for each language pair.

Inter RNN Type Reuse. We use modules from different I/O architecture types to evaluate this reuse type to build a new problem. However, to compose the modules, all of them should be able to process similar input types. For instance, assume taking a module from an RNN model that receives one input (1-to-{1, N}) and reusing it with a module from a model that receives several inputs at a time (M-to-{1, N}). This composition would not work because the two modules do not satisfy the same input constraints. For that, we evaluate in two different settings. First, **1-to-{1, N}**, in which we take one module decomposed from a *1:1* model and another from a *1:N* model. Then we compose them together to form a model. Second, **M-to-{1, N}**, in which we take one module decomposed from the *M:1* model and another from *M:N* (traditional) model. The encoder-decoder architecture prevents model reuse with other I/O types for language-translation models. However, such modules can be reused if decomposed from different datasets. We discuss such an experiment in §V-B4, when we recreate the motivation scenario and show the possibilities of solving the problem. We found that for the former scenario (1-to-{1, N}), there is an -6.22% (median -2.95%) loss of accuracy, on average. Whereas, for the latter scenario (M-to-{1, N}), the loss is, on average, -3.25% (median -2.20%).

TABLE IV: Summary of intra and inter-replace experiments

Replace Type	I/O Type	LSTM		GRU		Vanilla	
		Mean	Median	Mean	Median	Mean	Median
Intra	1:1	**+0.52**	-0.49	**+0.26**	**+0.25**	-0.55	-0.54
	M:1	-0.30	-0.13	-0.20	-0.14	-2.57	-1.52
	1:N	-0.74	**+0.04**	-0.07	-0.24	-2.39	-2.48
	M:N	-1.41	-1.56	-1.81	-2.90	-4.76	-4.04
	M:N Encoder-Decoder	**+0.30**	**+0.16**	**+1.72**	**+1.88**	**+0.92**	**+0.96**
Inter	1:1-1:N	-4.75	-0.02	-5.10	-3.30	-4.08	-6.60
	M:1-M:N	-10.48	-12.23	-6.71	-7.11	-12.23	-13.68

All values are in %.

3) RQ3: Can Decomposed Modules be Replaced?: This RQ investigates how to replace a decomposed module with another one. Similar to RQ2, we evaluate two different scenarios–(a) replacing a module with another performing the same operation within the same I/O type and (b) between different I/O types (in our case, different datasets too). We perform these experiments for different RNN-variants separately. We discuss each scenario in the following paragraphs.

Intra RNN Type Replacement. Here, a module is replaced with another one performing the same operation. The rationale of this experiment is to investigate how decomposition can help fix faulty models (e.g., low accuracy). To that end, we take the model with the lowest performance score. Then, we replace a module with one decomposed from the model with the highest accuracy in that category. For instance, consider the case of intra-replace for *1:1 LSTM* models trained on the Math QA dataset. In this case, LSTM-1 performs best and worst for LSTM-4. Therefore, we replace a module from LSTM-4 with one decomposed from LSTM-1. Then, we compute the accuracy of the composed model. Table IV shows the result of the experiments for all types of RNN models. We found that for a model trained with (*1:1*, Math QA), (*M:1*, Clinc OOS), (*1:N*, Toxic comment), and (*M:N*, Brown corpus) architecture-dataset pair, the average change of accuracy is +0.07% (median +0.25%), -1.03% (median -0.42%), -1.07% (median -0.27%), and -2.66% (median -2.90%), respectively. For the *M:N* encoder-decoder architecture, we replace the modules from the lowest average BLEU score model with the highest. As a result, we observed a 0.98% increase in the BLEU score compared to the monolithic model's BLEU score.

Inter RNN Type Replacement. Here, we replace a module with another one between different I/O types. We use the resulting composed model to perform different tasks. For instance, we replace a module from a model with a *1:1* architecture with one from a *1:N* model. Similar to the RQ2, the replaced module must accept a similar input type. For this reason, we perform two different experiments. First, for **1-to-{1, N}** architectures, we replace a module decomposed from *1:1* with one from an RNN model using the *1:N* architecture. Second, for the **M-to-{1, N}**, we replace a module decom-

posed from a *M:1* with a module from an RNN model with *M:N* architecture (traditional). Like RQ2, for *M:N* encoder-decoder architecture, we cannot perform the inter-RNN type replaceability due to the difference in the I/O architecture.

We found that for the 1-to-{1, N} replaceability, there is an average -4.64% (median -3.85%) accuracy loss. Whereas for M-to-{1, N} replaceability, the loss is -9.81% (median -11.49%) (Table IV). Overall, the loss is -8.47% (median -10.79%) for the inter-model type replaceability.

4) Recreating Motivating Examples: Here, we evaluate the scenarios discussed in the §II. For the first use case, a new language needs to be added to an existing model. We created a model with the languages from the motivating example (i.e., English, French, German, and Italian). Then, we decompose it to create modules. We train another model with the Ukrainian language as one of the target languages and decompose it too. Next, we compose the modules from the original model with the module that translates English to Ukrainian. In the second approach, we train a new model that translates only English to Ukrainian and uses it as a module. Lastly, we compose it with the previously decomposed modules. We found that both approaches can address the problem. However, the average BLEU score of modules is slightly less than that of the monolithic model for the first approach (see Table V). Here, we only report the results for LSTM models. Results for other models are similar and included in the replication package [22].

For the second scenario, we replace the module from a model that performs badly with a module decomposed from a model that performs better. We build a model that translates the Estonian language into English, Italian, and German. We decompose the model into modules and replace the Estonian with the English module with two approaches described in the examples. We found that both approaches perform better than the trained model from scratch.

TABLE V: Motivating Scenarios (Results for LSTM models)

Scenario	TMA	MA1	MA2
Add Ukranian Language	32.12%	31.94%	**32.53%**
Update Estonian-English Translation	20.80%	**20.89%**	**21.30%**

* MA{X}, TMA: Avg. BLEU score for scenario X and trained model.

a) Summary: We found that decomposing trained RNN models into modules has a trivial cost (accuracy: -0.6% and BLEU score: +0.10%). Also, these decomposed modules can be reused (accuracy: -2.38%, BLEU: +4.40%) and replaced (accuracy: -7.16%, BLEU: +0.98%) in various scenarios.

VI. Threats to validity

Internal threat: An internal threat can be the trained models. To mitigate, we follow prior works [3, 4, 42] to build the model (details in §-V.A). Another threat can arise from the stochastic nature of DL. To mitigate, in RQ1, each task is evaluated on four different model architectures, and in RQ2 and RQ3, every combination is exhaustively evaluated.

External threat: An external threat can be the experimental datasets. To mitigate this, we chose canonical datasets already used in the literature [16, 17, 18, 19, 24]. These datasets have a rich corpus and are adequately diverse to allow evaluation of our technique in different practical usage of NLP, i.e., single-output, multi-output, and generation (language translation).

VII. Conclusion and future directions

Modularization and decomposition have been shown to enable many benefits in traditional software, such as reuse, replacement, hiding changes, and increased comprehensibility of the modules [28, 45]. Recent works have demonstrated that DL systems can also benefit from such a decomposition and demonstrate these advantages for FCNN and CNN networks [3, 4]. This paper further advances our knowledge of modularity in the context of DL systems by extending it to RNNs, an important class of DNNs. It shows that different RNN models can be effectively decomposed and reused in different scenarios. Practitioners can use modules to compose new models. Also, they can leverage modules to replace faulty parts of existing models. The approach has been evaluated extensively on a benchmark of 60 models in different setups, i.e., different input/output types, RNN variants, and assuming both non-linear and logistic activation functions, etc. We found that decomposition has a small cost in terms of performance (accuracy and BLEU score). While this work limits its focus on the reuse and replace dimension of the modularity, we envision this decomposition can also enable/facilitate other benefits such as:

Hiding Changes: One of the key benefits of modularization is its ability to isolate and hide changes to a smaller number of components. This notion could be extended to deep learning software, making maintenance of large models, particularly in NLP, more manageable. Given the significance of change hiding in these scenarios, it is worth exploring the potential of the proposed modularization to streamline the maintenance process.

Increase Comprehensibility: Modularization has been shown to enhance our understanding of program logic, as noted by Dijkstra [45]. In the context of deep learning models, modularization could help reveal the internal logic more efficiently by breaking down a monolithic black-box model into distinct, functional units.

VIII. Acknowledgement

This work was supported in part by US NSF grants CNS-21-20448 and CCF-19-34884. We want to thank the reviewers for their valuable and insightful comments. The views expressed in this work are solely those of the authors and do not reflect the opinions of the sponsors.

References

[1] L. Y. Pratt, J. Mostow, C. A. Kamm, A. A. Kamm *et al.*, "Direct transfer of learned information among neural networks." in *Aaai*, vol. 91, 1991, pp. 584–589.

[2] X.-Y. Zhang, C. Li, H. Shi, X. Zhu, P. Li, and J. Dong, "Adapnet: Adaptability decomposing encoder-decoder network for weakly supervised action recognition and

localization," *IEEE transactions on neural networks and learning systems*, 2020.

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

[4] ——, "Decomposing convolutional neural networks into reusable and replaceable modules," in *ICSE'22: The 44th International Conference on Software Engineering*, May 21-May 29, 2022 2022.

[5] J. Andreas, M. Rohrbach, T. Darrell, and D. Klein, "Neural module networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 39–48.

[6] R. Hu, J. Andreas, M. Rohrbach, T. Darrell, and K. Saenko, "Learning to reason: End-to-end module networks for visual question answering," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 804–813.

[7] G. E. Hinton, Z. Ghahramani, and Y. W. Teh, "Learning to parse images," *Advances in neural information processing systems*, vol. 12, pp. 463–469, 2000.

[8] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," *arXiv preprint arXiv:1710.09829*, 2017.

[9] B. Ghazi, R. Panigrahy, and J. Wang, "Recursive sketches for modular deep learning," in *International Conference on Machine Learning*. PMLR, 2019, pp. 2211–2220.

[10] J. J. Dongarra and A. Hinds, "Unrolling loops in fortran," *Software: Practice and Experience*, vol. 9, no. 3, pp. 219–226, 1979.

[11] S. Weiss and J. E. Smith, "A study of scalar compilation techniques for pipelined supercomputers," *ACM SIGARCH Computer Architecture News*, vol. 15, no. 5, pp. 105–109, 1987.

[12] J. W. Davidson and S. Jinturkar, "Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation," in *Proceedings of the 28th annual international symposium on Microarchitecture*. IEEE, 1995, pp. 125–132.

[13] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[14] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[15] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," *arXiv preprint arXiv:1506.00019*, 2015.

[16] A. Amini, S. Gabriel, P. Lin, R. Koncel-Kedziorski, Y. Choi, and H. Hajishirzi, "Mathqa: Towards interpretable math word problem solving with operation-based formalisms," *arXiv preprint arXiv:1905.13319*, 2019.

[17] W. N. Francis and H. Kucera, "Computational analysis of present-day american english," *Providence, RI: Brown University Press. Kuperman, V., Estes, Z., Brysbaert, M., & Warriner, AB (2014). Emotion and language: Valence and arousal affect word recognition. Journal of Experimental Psychology: General*, vol. 143, pp. 1065–1081, 1967.

[18] E. Wulczyn, N. Thain, and L. Dixon, "Ex machina: Personal attacks seen at scale," in *Proceedings of the 26th international conference on world wide web*, 2017, pp. 1391–1399.

[19] S. Larson, A. Mahendran, J. J. Peper, C. Clarke, A. Lee, P. Hill, J. K. Kummerfeld, K. Leach, M. A. Laurenzano, L. Tang, and J. Mars, "An evaluation dataset for intent classification and out-of-scope prediction," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 1311–1316. [Online]. Available: https://aclanthology.org/D19-1131

[20] J. Tiedemann, "Parallel data, tools and interfaces in opus," in *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, N. C. C. Chair), K. Choukri, T. Declerck, M. U. Dogan, B. Maegaard, J. Mariani, J. Odijk, and S. Piperidis, Eds. Istanbul, Turkey: European Language Resources Association (ELRA), may 2012.

[21] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[22] (2022) Rnn decomposition - replication package. [Online]. Available: https://huggingface.co/datasets/Anonymous1234/RNNDecomposition/tree/main

[23] A. Fan, S. Bhosale, H. Schwenk, Z. Ma, A. El-Kishky, S. Goyal, M. Baines, O. Celebi, G. Wenzek, V. Chaudhary *et al.*, "Beyond english-centric multilingual machine translation," *Journal of Machine Learning Research*, vol. 22, no. 107, pp. 1–48, 2021.

[24] F. Chollet, *Deep learning with Python*. Simon and Schuster, 2021.

[25] European Language Resource Coordination, "Supporting Multilingual Europe," 2022, https://www.lr-coordination.eu/.

[26] ——, "Call For Support Ukraine," 2022, https://www.lr-coordination.eu/sites/default/files/common/Call%20for%20support_Ukraine.pdf.

[27] Google, "Google Translator," 2022, https://translate.google.com/?sl=auto&tl=en&text=ta%20on%20arst&op=translate.

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

[29] B. Liskov and S. Zilles, "Programming with abstract data types," *ACM Sigplan Notices*, vol. 9, no. 4, pp. 50–59, 1974. 13

[30] D. L. Parnas, "On the design and development of program families," *IEEE Transactions on software engineering*, no. 1, pp. 1–9, 1976.

[31] E. W. Dijkstra, "On the role of scientific thought," in *Selected writings on computing: a personal perspective*. Springer, 1982, pp. 60–66.

[32] L. Cardelli, "Program fragments, linking, and modularization," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997, pp. 266–277.

[33] Z. Gao, X. Xia, D. Lo, J. Grundy, and T. Zimmermann, "Automating the removal of obsolete todo comments," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 218–229.

[34] Y. Chai, H. Zhang, B. Shen, and X. Gu, "Cross-domain deep code search with few-shot meta learning," *arXiv preprint arXiv:2201.00150*, 2022.

[35] J. Lin, Y. Liu, Q. Zeng, M. Jiang, and J. Cleland-Huang, "Traceability transformed: Generating more accurate links with pre-trained bert models," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 324–335.

[36] K. SaiRam, J. Mukherjee, A. Patra, and P. P. Das, "Hsdcnn: Hierarchically self decomposing cnn architecture using class specific filter sensitivity analysis," in *Proceedings of the 11th Indian Conference on Computer Vision, Graphics and Image Processing*, 2018, pp. 1–9.

[37] (2021) Keras rnn layers. [Online]. Available: https://keras.io/api/layers/recurrent_layers/

[38] (2021) Tensorflow rnn layers. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers

[39] (2021) Pytorch rnn layers. [Online]. Available: https://pytorch.org/docs/stable/nn.html#recurrent-layers

[40] H. Salehinejad, S. Sankar, J. Barfett, E. Colak, and S. Valaee, "Recent advances in recurrent neural networks," *arXiv preprint arXiv:1801.01078*, 2017.

[41] M. Artetxe and H. Schwenk, "Massively multilingual sentence embeddings for zero-shot cross-lingual transfer and beyond," *Transactions of the Association for Computational Linguistics*, vol. 7, pp. 597–610, 2019.

[42] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[43] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[44] Google, "Understanding the BLEU Score," 2022, https://cloud.google.com/translate/automl/docs/evaluate#:~\protect\protect\leavevmode@ifvmode\kern+.2222em\relaxtext=BLEU%20(BiLingual%20Evaluation%20Understudy)%20is,of%20high%20quality%20reference%20translations.

[45] E. W. Dijkstra *et al.*, "Notes on structured programming," 1970.