Apricot: A Weight-Adaptation Approach to Fixing Deep Learning Models †

Hao Zhang
Department of Computer Science
City University of Hong Kong
Hong Kong
hzhang339-c@my.cityu.edu.hk

W.K. Chan ‡
Department of Computer Science
City University of Hong Kong
Hong Kong
wkchan@cityu.edu.hk

***Abstract*—A deep learning (DL) model is inherently imprecise. To address this problem, existing techniques retrain a DL model over a larger training dataset or with the help of fault injected models or using the insight of failing test cases in a DL model. In this paper, we present *Apricot,* a novel weight-adaptation approach to fixing DL models iteratively. Our key observation is that if the deep learning architecture of a DL model is trained over many different subsets of the original training dataset, the weights in the resultant reduced DL model (rDLM) can provide insights on the adjustment direction and magnitude of the weights in the original DL model to handle the test cases that the original DL model misclassifies. Apricot generates a set of such reduced DL models from the original DL model. In each iteration, for each failing test case experienced by the input DL model (iDLM), Apricot adjusts each weight of this iDLM toward the average weight of these rDLMs correctly classifying the test case and/or away from that of these rDLMs misclassifying the same test case, followed by training the weight-adjusted iDLM over the original training dataset to generate a new iDLM for the next iteration. The experiment on five state-of-the-art DL models shows that Apricot can increase the test accuracy of these models by 0.87%-1.55% with an average of 1.08%. The experiment also reveals the complementary nature of these rDLMs in Apricot.**

Keywords—Deep Neural Networks, Optimization, Model Evolution, Debugging, Model Fixing, Model Repair

I. INTRODUCTION

Deep Learning (DL) [28] systems have found successful applications in many areas, such as natural language processing, computer vision, data processing, and computer games, to make non-trivial decisions. Their ***deep learning models*** (referred to as ***DLMs***) often deliver higher accuracy than the prior state-of-the-arts that were invented and significantly improved after years of research investigation. However, these models are also inherently imprecise. Using them in, for example, controlling driverless cars or making decisions in surgeries may incur significant threats. Making them more accurate is essential.

A DLM is a deep learning network (DNN) [28] initialized with a set of weights for the connection between its neurons, where such weights are obtained from a training process. Thus, by pairing the same DNN with different sets of weights (called weight matrices [45]), the corresponding DLMs may exhibit different classification behaviors.

To train a DLM, one needs a ***training dataset***, which is a set of ***inputs***, referred to as ***test cases***. Each test case is labeled with an expected classification result (or ***expected result*** for short), such as the output class of the target DLM that should receive the highest likelihood value. Running a DLM to classify a test case produces a classification result, which is referred to as the ***actual result***. If a DLM classifies an input inconsistent to its expected result, a misclassification is identified. The input is said to be a ***failing test case***, and we also say that the DLM ***fails*** on this test case, ***incorrectly classifies*** the test case, or ***experiences a failure***. If there is no misclassification detected, the test case is a ***passing test case***, and the DML ***correctly classifies*** it. The ***accuracy*** of a DLM over a dataset is the percentage of test cases that the DLM correctly classifies them. For a training dataset, it is called ***training accuracy***, and for a testing dataset, it is called ***test accuracy***.

If a DLM is not accurate enough, it should be "fixed". One category of strategies is to expand training datasets with additional test cases. Collecting these test cases, such as via automated generations of adversarial examples [14] or from other sources [59], need additional costs and time. Besides, these test cases should be labeled with expected results. Take the huge ImageNet dataset [8] for example. Many state-of-the-art DLMs, such as VGG16 [43] and ResNet [16], can achieve 90% or higher in accuracy. Further expanding the dataset with the aim of observable increase in accuracy of such DLMs is challenging.

Another category of strategies is to further optimize a DLM with the original training dataset by, for example, changing parameters progressively [9][25][58], exploring different weight assignments [1][7][17][19][20][34][52][56]**,** producing variants of the DNN underlying the DLM [2][60], or a mix of them [40] to generate evolved DLMs before training them. A common characteristic of these strategies is to generate a set of DLM variants (often through evolutionary algorithms) in each iteration, and these DLM variants are trained with the original training dataset. These strategies also do ***not*** use the expected results of the training inputs in generating these DLMs.

The third category of strategies is to optimize a DLM with the original training dataset but, unlike the category stated in the paragraph above, without using multiple DLM variants at each iteration in their optimization processes. For instance, MODE [33] identifies a hidden layer where features contained in its neurons are most representative for exposing erroneous behaviors of DLM. Layers from the input layer to this identified layer are reconstructed to form a feature model, and such a feature model can guide on selecting inputs for retraining the DLM. Regularization techniques [30][42][54][57] make a

† This research is supported in part by the GRF of HKSAR Research Grants Council (project nos. 11214116 and 11200015), the HKSAR ITF (project no. ITS/378/18), the CityU MF_EXT (project no. 9678180), the CityU SRG (project nos. 7004882 and 7005216) and the CityU SGS Conference Grant.
‡ Correspondence author.

** This version supersedes the camera-ready version.*

DLM's weight matrices [15] sparser to reduce the impacts of unimportant weights on the classification results. These techniques typically make a coarse adjustment by adding a regularization term (e.g., L_p norm [30]) to the loss function [45] in the training process of the DLM. Intuitively, many ill-trained weights are small in value, but not all weights with small values are ill-trained ones. Fixing a DLM at a coarse level is challenging to fix the problem in selected few (and unknown) ill-trained weights in the DLM.

In this work, we present ***Apricot***, a novel weight-adjustment approach to fixing deep learning models. Apricot is built on top of two intuitions: (1) As the number of inputs in a training dataset T_0 increases, in general, training a DL model tends to become more difficult for the resultant DL model to retain a large proportion of its weights well-trained to capture all essential features in T_0. Let us consider a scenario. Suppose that there is a pair of DLMs (denoted as D_0 and D_{sub}, respectively) where their DNNs and the training processes are identical except that D_0 and D_{sub} are trained with T_0 and a subset S_0 of T_0, respectively. Further, suppose that a test case x in S_0 is classified correctly and incorrectly by D_{sub} and D_0, respectively. An intuition for D_{sub} to classify x correctly is that $D_{sub}$is trained on a smaller dataset S_0, which makes D_{sub} to probabilistically retain some features essential in classifying x correctly, which D_0 has unfortunately lost them to a large extent in its training process (and thus, x represents a failing test case). The weight assignments for D_{sub} likely capture insights on the direction and magnitude to adjust the weight matrices of D_0 to make D_0 more likely to classify inputs similar to x correctly. For ease of our reference, we refer to such a D_{sub} as a ***reduced deep learning model*** (rDLM). (2) One rDLM may not capture the essential features to classify a particular test case correctly. If there is a set of rDLMs such that on average, each rDLM is more likely to classify a test case correctly than the otherwise, then, intuitively, the central tendency of this set of rDLMs is also more likely to classify the test case correctly.

Given a pair of DLM D_0 and a training dataset T_0, Apricot first generates a set of rDLMs, each training with a random subset of T_0. It then goes into an iterative process with D_0 as the input model (denoted as iDLM D_1) of the first iteration. At the k-th iteration, for each failing test case x of the iDLM D_k, Apricot divides the set of rDLMs into two partitions, denoted as $IncorrectSubModels_k(x)$ and $CorrectSubModels_k(x)$, where each rDLM in these two partitions classifies x incorrectly and correctly, respectively. Then, for each weight assignment w of D_k, Apricot takes the mean of the corresponding weight assignments of the DLMs in $IncorrectSubModels_k(x)$ and $CorrectSubModels_k(x)$, and combines these two mean values into a single value. It then adjusts w of D_k accordingly. (Apricot can be configured to use either both average values or only one of them.) Next, it trains the adjusted D_k with T_0 to produce the iDLM D_{k+1} for the (k+1)-th iteration.

The output of the above procedure is a DLM with repaired weights. One clear merit of Apricot is that it only changes the weights of a DLM and needs fewer epochs to train DLM compared to the original training process applied to DLM.

We have evaluated Apricot with five DL models over the CIFAR-10 dataset. The empirical results show that for the strategy using the mean value of correctly classified rDLMs (called Strategy 2), Apricot increases the test accuracy on three CNN models by 0.89%–1.55% with an average of 1.19% and on two ResNet models by 0.87%–0.92% with an average of 0.895%. By using both correctly and incorrectly classified rDLMs (called Strategy 1), Apricot achieves the highest increase in test accuracy on four out of five models by 1.00-2.18% with an average of 1.502%. By combining the results of all three strategies initialized with Apricot, an average improvement of 1.08% across all models is obtained, and the average maximum gain is 1.536%. Apricot is fully automated. It can be initialized with different weight adjustment strategies and applied to other kinds of DL models. In view of the present result, we believe that Apricot delivers a promising result and represents a feasible direction to produce DLMs with repaired weights.

The main contribution of this paper is threefold.

- This paper presents Apricot. Apricot does not need any additional inputs. It connects the original DLM and a set of rDLMs via failing test cases, which provides a guide to repair ill-trained weights of the original DLM, which sets it apart from regularization and other debugging work. It does not generate any fault-based DLMs, which makes it different from fault-based (testing, debugging, or repair) techniques.
- This paper presents an evaluation of Apricot, which shows that Apricot is viable to improve model accuracy.
- This work demonstrates that using multiple rDLMs with failing test cases of the original DLM can provide interesting insights for fixing the original DLM by exploring the connections among all these DLMs.

The organization of the remaining sections is as follows. Section II revisits the background. Section III presents Apricot in detail, followed by its evaluation in Section IV. Section V discusses related works. We conclude this work in Section VI.

II. Background

A. Deep Neural Networks (DNNs)

A typical deep neural network (DNN) [10][12][18][27] contains a set of layers, e.g., an input layer, an output layer, and multiple hidden layers, and each layer contains a set of neurons. Neurons of adjacent layers are connected. A neuron is a function that receives one or more inputs from its incoming connections. Each such connection carries a value called weight. The neuron sums these weighted inputs, possibly with some bias constant or regularization term, to compute a weighted sum and passes this sum to an activation function (usually has a non-linear shape) to produce an output, representing whether there is an input, to its output connections (which serve as incoming connection to other neurons for example).

Formally, a deep neural network [28] is a triple tuple $D = \langle L, T, \Phi \rangle$, where

- $L = \{ l_i \mid i \in \{1, 2, \ldots, k\}\}$ is a set of k layers where l_1 is called the input layer, and l_k is called the output layer. Other layers are called hidden layers. Each layer l_i contains n_i neurons.

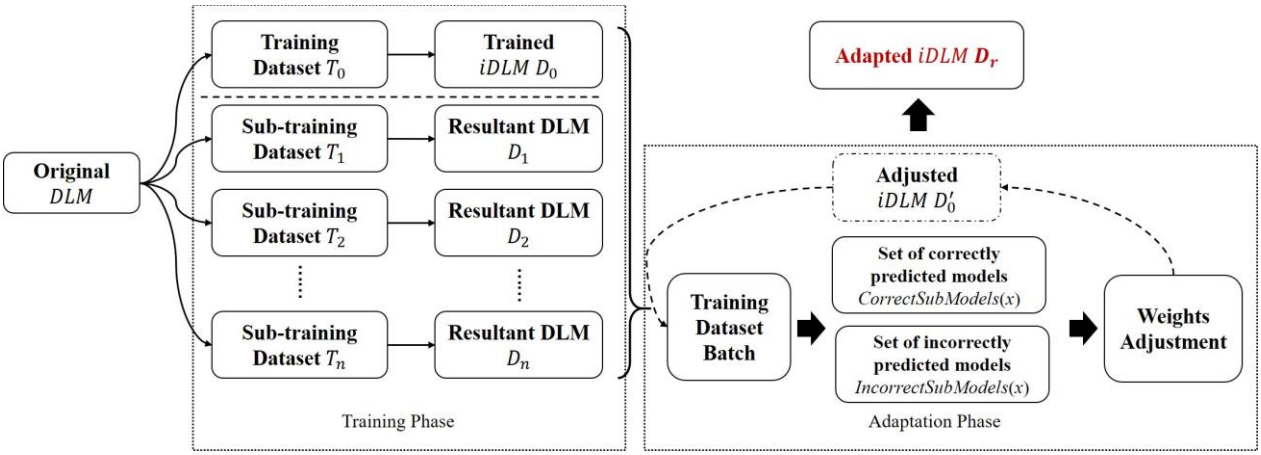

Fig. 1 Overall process of repairing deep neural networks by Apricot

- $T \subseteq L \times L$ is the set of connections between two layers, representing the edges between neurons of these two layers. Each connection is assigned with a specific value called ***weight***, and the weights of all the connections between a pair of layers in T forms a *weight matrix* [45].
- $\Phi = \{ \phi_i \mid i \in \{1, 2, \ldots, m\} \}$ is a set of activation functions [45]. In many configurations, neurons belonging to the same layer have the same activation function.

If a deep neural network is initialized with a set of weight matrices, it is called a ***deep learning model*** (***DLM***). Assigning a weight with a particular value is called a ***weight assignment***.

To obtain the set of weight matrices for a DLM, we may train the DLM over a training dataset. The involved training process may directly train the DLM over the training dataset or firstly divide the training dataset into multiple subsets (where each such subset is called a ***batch***) followed by training the DLM over a sequence of these batches. On training a DLM over a dataset or a batch, it often requires iterating many times, and each such iteration is called an ***epoch***.

However, the presence of ill-trained weights in a DLM may lead the DLM to behave erroneously. Apart from the limitation of the data quality of a given training dataset and whether the DLM is over- or under-fitted through its training process, a widespread belief [11] of incurring ill-trained weights in a DLM is that some inputs in the training dataset may conflict with each other, i.e., the updating directions of parameters (i.e., weights) between two inputs or two batches of them may be opposite.

The output of the hidden layers of a DLM can be regarded as an abstract representation of what the DLM has learned.

B. Ill-trained Weights in DLM and Repair

In the literature, DLM bugs can be classified into two kinds [33]: structural bugs and training bugs. Structural bug refers to the structure (including the type of layers, the connections between layers, the types, and allocations of activation function) of the DLM being not well designed. Fixing this type of bugs automatically is slow because apart from making a candidate DLM sufficiently trained and tested, the number of candidates is enormous, making the search space much larger than fixing training bugs.

Training bug [33] refers to some weights in the weight matrices of a DLM being not well-trained due to insufficient training process or bias in the training dataset. We refer to these weights as ***ill-trained weights*** in this paper.

It is challenging to isolate the impact of one weight (or a set of them) on the performance (e.g., accuracy) of a DLM. Precisely identifying ill-trained weights is still unreachable yet.

A related approach to addressing the problem of ill-trained weights is to modify the weights in some weight matrices through retraining directly. Retraining a DLM with a different dataset (such as adding more inputs to the original training dataset) or using a different set of weight matrices for the initialization of the DLM before retraining will produce a new set of weight matrices.

Adding more inputs to a training dataset is widely considered as a practical and most common approach to repairing ill-trained weights. Nonetheless, using more inputs for retraining is laborious in data collection and labeling, and finding sufficiently more inputs may be impractical in many cases.

It is interesting to develop effective techniques to utilize the existing datasets to adjust the weight matrices to alleviate the problem of ill-trained weights. As we have summarized in Section I, there are two categories of work in this area. Apricot is also in this direction but different from them.

III. APRICOT

In this section, we present Apricot.

A. Overview

Apricot is a novel weight-adaptation approach to fixing deep learning models. We first recall from Section I that given a DLM D_0 with its training dataset T_0, a ***reduced deep learning model*** (***rDLM*** for short) is obtained from training the deep neural network of D_0 but with a subset of T_0 as its training dataset using the same training procedure that has trained D_0. Apricot explores the relationship between D_0 and a set of its rDLMs on the *failing* test cases experienced by D_0 over T_0. It does not need any mutated or faulty versions of D_0.

In general, once a (state-of-the-art or good) DLM is well-trained with a large dataset, such as the CIFAR-10 dataset or the

ImageNet dataset, it is difficult to improve its accuracy further significantly. Our work shows a possibility to make such an improvement without needing extra information, which we will present evidence in Section IV (evaluation).

The overall process of Apricot is depicted in Fig. 1. There are two phases: the training phase and the adaptation phase.

In the training phase, the training dataset T_0 is divided into multiple subsets $T_{sub} = \{T_1, T_2, \dots, T_n\}$, where $\bigcup_{i=1}^{n} T_i = T_0$. (To ensure rDLMs to be trained sufficiently, the size of each subset shall be large enough.) Note that there may or may not be some overlapping between subsets. Next, Apricot trains D_0 on T_0 followed by training all rDLMs D_i on T_i for i = 1 to n, respectively.

In the adaptation phase, Apricot divides T_0 into a sequence of batches. It processes these batches in turn. On processing each batch, it invokes a sequence of two activities: weight adjustment followed by retraining. In the weight adjustment activity, each batch *b* accepts a DLM as an input (called input DLM, or iDLM for short). Apricot identifies all the test cases in *b* that the iDLM fails on each of them. For each such failing test case *x*, it divides the set of rDLMs into two partitions: one partition (denoted as *CorrectSubModels*(*x*)) contains these rDLMs each classifying *x* correctly, and the other partition (denoted as *IncorrectSubModels*(*x*)) contains all the remaining rDLMs (i.e., these each classifies *x* incorrectly). Then, it adjusts the weights of the iDLM *toward* the average weight of these rDLMs in *CorrectSubModels*(*x*) and/or *away from* that of these rDLMs in *IncorrectSubModels*(*x*). In the retraining activity, Apricot trains the iDLM (which is produced by the weight adjustment activity for the same batch) using the entire training dataset T_0 to produce a DLM serving as the input for the iteration on handling the next batch.

Our experiment (see Section IV) shows that using rDLMs that *correctly* classify these failing test cases experienced by D_0 can outperform using only *incorrectly* classified rDLMs or both of these two kinds of rDLMs on average but using both types of rDLMs can achieve the highest increase in test accuracy. In fact, all three strategies show improvements in accuracy, and an average of 1.08% improvement in accuracy is obtained. This result indicates the complementary nature of these two kinds of rDLMs used in Apricot.

Algorithm 1 Generating rDLMs

```
Input:  train_x, train_y ← inputs and labels of training dataset
        n ← the number of generated rDLMs
        dnn ← the DNN of original DLM
Output: set of generated rDLMs
1:  subTrainingDataset ← SplitDataset(train_x, train_y, n)
2:  subModelList ← []
3:  for each subset in subTrainingDataset do
4:  |  subModel ← CopyModelStructure(dnn)
5:  |  subModel ← TrainingModel(subModel, subset)
6:  |  subModelList.append(subModel)
7:  end for
8:  return subModelList
```

B. Adjusting weights of DLM by rDLMs

The algorithms for the training and adaptation phases of Apricot are shown in Algorithms 1 and 2, respectively.

In Algorithm 1, the training dataset and the DNN of the original DLM (e.g. type of layers and parameters for each layer) are given as inputs to the algorithm. In line 1, the algorithm creates *n* subsets of the original training dataset where the original training dataset is specified as inputs (*train_x*) and the corresponding expected results (*train_y*). Then, it feeds each such subset to a corresponding rDLM that has the same structure as the given DLM by firstly copying the DNN from *dnn* followed by training the rDLM using the subset (lines 3-7). It finally collects the trained rDLMs as the outputs (lines 6 and 8).

We note that in the training process of the DLM and these rDLMs, only models with highest test accuracy are kept, which is a typical setting chosen in training a DLM in practice [53].

Algorithm 2 Adjusting weights of DLM by rDLMs

```
Input:  train_x, train_y ← inputs and labels of training dataset
        test_x, test_y ← inputs and labels of test dataset
        subModelList ← the list of trained rDLMs
        dnnModel ← the trained iDLM D0
        strategy ← strategy of adjusting weights
Output: DNN model with repaired weights
1:  bestWeights ← GetWeights(dnnModel)
2:  bestAcc ← EvaluateModel(dnnModel, test_x, test_y)
3:  for each xBatch, yBatch in GetBatch(train_x, train_y) do
4:  |  baseWeights ← GetWeights(dnnModel)
5:  |  for each x, y in xBatch, yBatch do
6:  |  |  ySubList ← ModelPrediction(x, subModelList)
7:  |  |  yOrigin ← ModelPrediction(x, dnnModel)
8:  |  |  if yOrigin equals y then
9:  |  |  |  continue
10: |  |  else
11: |  |  |  correctSubModels ← CorrectlyPredict(ySubList, y)
12: |  |  |  if correctSubModels is empty then //no correct predictions
13: |  |  |  |  continue
14: |  |  |  end if
15: |  |  |  incorrectSubModels ← IncorrectlyPredict(ySubList, y)
16: |  |  |  correctWeights ← AvgWeights(correctSubModels)
17: |  |  |  incorrWeights← AvgWeights(incorrectSubModels)
18: |  |  |  corrDiff ← baseWeights – correctWeights
19: |  |  |  incorrDiff ← baseWeights – incorrectWeights
20: |  |  |  baseWeights ← AdjustWeights(baseWeights, corrDiff,
        incorrDiff, strategy)
21: |  |  |  dnnModel.setWeights(baseWeights)
22: |  |  end if
23: |  end for
    |  // evaluate adjustments
24: |  currAcc ←EvaluateModel(dnnModel, test_x, test_y)
25: |  if bestAcc < currAcc then
26: |  |  bestWeights ← baseWeights
27: |  |  dnnModel.setWeights(baseWeights)
28: |  |  dnnModel ← TrainingModel(dnnModel, train_x, train_y)
29: |  |  bestAcc ←currAcc
30: |  else
31: |  |  baseWeights ← bestWeights
32: |  |  dnnModel ← TrainingModel(dnnModel, train_x, train_y)
33: |  end if
34: |  dnnModel.setWeights(bestWeights)
35: end for
36: return dnnModel
```

Algorithm 2 presents the procedure of adjusting the weights of a DLM using a set of trained rDLMs.

In lines 1-2, the weights of the given iDLM and test accuracy achieved by this iDLM are collected via functions GetWeights() and EvaluateModel(), respectively. Then, the whole training dataset is divided into a set of batches via the function GetBatch(), where each batch contains a specific number of inputs *xBatch* and corresponding expected results *yBatch* of these inputs (line 3). For each test case x and its corresponding expected result y in each batch, the classification results of all rDLMs, as well as iDLM on x, are collected (lines 6-7) through the function ModelPrediction(). If iDLM classifies x correctly, the weights of iDLM are not adjusted (lines 8-9). On the other hand, if iDLM classifies x incorrectly, then the set of rDLMs is divided into two partitions: *correctSubModels* and *incorrectSubModels* at line 11 and line 15, respectively. If no models classify x correctly, no adjustment will be applied (lines 12-13). It is because our adjustment strategies (will be presented in the next paragraph) require correctly classified rDLMs to provide directions of weights adjustment. Note that the adjustment procedure continues if no rDLMs classify x incorrectly. If *correctSubModels* is non-empty, then the average weights of both *correctSubModels* and *incorrectSubModels* are calculated via the function AvgWeights() at lines 16-17. Apricot calculates average weights as follows:

$$W_{avg} = \frac{1}{n}\sum_{i=1}^{n} W_i \quad (1)$$

where n is the total number of rDLMs, W_i is the weight matrices for each rDLM. All matrices of rDLMs are added by adding corresponding elements together, and then means of weights are calculated. In lines 18-19, the difference in weights between iDLM and *correctSubModels* as well as that between the weights of iDLM and in*correctSubModels* are calculated, which are kept by the variables *corrDiff* and *incorrDiff*, respectively. In line 20, it adjusts the weights of iDLM by updating the value of each weight of iDLM via the function setWeights().

The function AdjustWeights() requires *strategy* as its input. Apricot includes three strategies of weight adjustments (denoted as Strategies 1, 2, and 3, respectively) as follows:

$$w^{(k)} = w^{(k-1)} - r_l \cdot p_{corr} \cdot corrDiff + r_l \cdot p_{incorr} \cdot incorrDiff \quad (2)$$

$$w^{(k)} = w^{(k-1)} - r_l \cdot p_{corr} \cdot corrDiff \quad (3)$$

$$w^{(k)} = w^{(k-1)} + r_l \cdot p_{incorr} \cdot incorrDiff \quad (4)$$

where $w^{(k)}$ is the adjusted weight for the corresponding edge in the iDLM after k-th iteration of the for-loop spanning over lines 5-23, r_l is a constant representing the learning rate, and p_{corr} and p_{incorr} are proportions of correctly and incorrectly classified rDLMs, respectively. That is, let a and b be the numbers of rDLMs in the sets *correctSubModels* and *incorrectSubModels*, respectively. We have $p_{corr} = a / (a + b)$, and $p_{incorr} = b / (a + b)$.

In (2), both weight components from correct and incorrect rDLMs are utilized. In (3), it ignores the component for *incorrectSubModels*, and in (4) it ignores the components for *correctSubModels*. The intuition behind these equations is that Apricot aims to minimize the distance between current weights and weights of *correctSubModels* and/or maximize the distance to weights of *incorrectSubModels*.

TABLE I STRUCTURE OF CNNS

Model	CNN_1	CNN_2	CNN_3
Block 1	Conv(64) Conv(64) MaxPooling	Conv(32) BatchNormalization Conv(32) BatchNormalization MaxPooling	Conv(96) Conv(96) Conv(96) MaxPooling
Block 2	Conv(128) Conv(128) MaxPooling	Conv(64) BatchNormalization MaxPooling	Conv(192) Conv(192) Conv(192) MaxPooling
Block 3	-	Conv(128) BatchNormalization Conv(128) BatchNormalization MaxPooling	Conv(10) GlobalAvgPooling
Output	Dense(256) Dense(256) Dense(10) Softmax	Flatten() Dense(10) Softmax	Dense(10) Softmax

After the above weight adjustment activity for the current batch, the algorithm evaluates the adjusted iDLM on the test dataset (line 24). It trains the adjusted iDLM on the whole training dataset for several epochs (the detailed settings will be described in the next section). It also applies a greedy algorithm to retain the DLM with the highest test accuracy. More specifically, in lines 25-29, if the test accuracy of an adjusted iDLM is higher than that of the best case obtained so far, then the weights of this particularly adjusted iDLM are saved. If not, the weights of the iDLM are rolled back to the best case obtained so far (lines 31-32). In line 34, the weights of iDLM are set to the best case founded so far.

IV. EXPERIMENT AND DATA ANALYSIS

In this section, we evaluate Apricot on the CIFAR-10 dataset with five state-of-the-art deep learning models.

A. Experimental Setup

We have implemented Apricot on the top of Keras 2.2.2 [23] and TensorFlow 1.13.1 [49], which are popular machine learning libraries in Python. All experiments were performed on Windows 10 running on an Intel i7-8700K CPU, 32GB RAM, a

TABLE II DESCRIPTIVE SUMMARY OF DNN MODELS

Model	Num. of parameters	Avg. training accuracy	Avg. test accuracy
CNN_1	2.43M	99.09%	73.37%
CNN_2	272K	96.07%	86.23%
CNN_3	1.00M	99.64%	76.98%
ResNet-20	274K	97.75%	91.00%
ResNet-32	470K	98.74%	91.95%

TABLE III SUMMARY OF TRAINING TIME OF DLM AND RDLMS

Model	Training time t_o of DLM	Total training time t_{total} of rDLMs	Num. of rDLMs	Avg. training time t_r	$\frac{t_0 - t_r}{t_0} \times 100\%$
CNN_1	1min46s	15min24s	20	46s	56.6%
CNN_2	31min39s	26min44s	20	1min20s	95.8%
CNN_3	5min47s	43min48s	20	2min11s	62.0%
ResNet-20	28min30s	2h56min3s	20	8min48s	69.1%
ResNet-32	31min21s	2h58min35s	20	8min55s	71.6%

TABLE IV SUMMARY OF RDLMS AFTER TRAINING PHASE

rDLM	Avg. training accuracy	Training accuracy std.	Avg. test accuracy	Test accuracy std.	Percentage of correct classification while DLM classifies incorrectly
CNN_1	67.54%	0.00503	59.47%	0.00687	22.25%
CNN_2	74.24%	0.00276	67.71%	0.00413	12.31%
CNN_3	68.15%	0.00867	60.40%	0.01078	19.16%
ResNet-20	78.43%	0.00757	74.64%	0.00716	7.10%
ResNet-32	79.04%	0.00683	75.03%	0.00648	7.84%

256GB SSD, and a single NVIDIA GeForce 2080-Ti GPU with the VRAM size of 11GB.

Dataset. The experiments were conducted using the entire CIFAR-10 dataset [26] that is widely used for image classification training and testing research.

Deep learning models. Five state-of-the-art DLMs were implemented (three CNN models and two ResNet models). We applied APIs provided by Keras for training models. Settings of the training process are as follows: for CNN_1, we set epochs to 20 and batch size to 256. In the CNN_2 training process, we set epochs to 125 and batch size to 64. Besides, an image data generator (provided by Keras) [21] is used to enrich the training dataset. For CNN_3, we set epochs and batch size to 50 and 128, respectively. For ResNet-20 and ResNet-32, epochs and batch size are set to 120 and 128. Besides, an image data generator is applied in the training process of ResNet models. We set activation functions to ReLU [24] for all CNNs.

We applied the same settings except for epochs for training rDLMs. We set epochs to 30 for training rDLMs of CNN_1 and 60 for training other rDLMs. For each DLM, we trained 20 rDLMs. The value of 20 was arbitrarily chosen without any pre-experimental trial. The total numbers of test cases in the CIFAR-10 training dataset ($T_{\text{CIFAR-10}}$) and test dataset were 50,000 and 10,000, respectively. The size of the subset for training rDLMs was set to 10,000. Each image in the training dataset was indexed. The separation of training dataset into batches in Algorithm 1 is as follows. The first, second, and third subsets are the images with index from 0 to 10,000, from 2,000 to 12,000, and from 4,000 to 14,000, respectively. Other subsets are similarly defined. Note that the epoch and dataset for training an rDLM are smaller than that for training original DLM, thus training an rDLM is much faster than training a DLM.

We used three representative structures for the three CNN models [47], which are summarized in TABLE I. These models had different numbers of convolutional layers and hyperparameters. CNN_1 contained two blocks: The first block consisted of two convolutional layers with 64 kernels, followed by a max-pooling layer. The second block was similar to the first block, but the number of kernels was set to 128. The output part consisted of three fully connected layers with 256, 256, 10 neurons, respectively. CNN_2 and CNN_3 had similar structures compared to CNN_1. In CNN_2, *BatchNormalization* layers [48] were applied. In CNN_3, a global average-pooling layer was applied.

For ResNet-20 and ResNet-32, we used the existing implementations [6], which represented the ResNet [30] with 20 and 32 residual layers, respectively.

The summary of these DLMs is shown in TABLE II. Column 1 shows the DLMs we implemented in our experiment. Column 2 is the number of parameters for each DLM. Column 3 and Column 4 are the average training accuracy and average test accuracy of each DLM on the CIFAR-10 dataset in five runs. During the training process, only models with the highest test accuracy were saved.

For Algorithm 2, the batch size that each time the function GetBatch() (line 3) retrieves was set to 20. These batches were mutually disjointed. Again, this value of 20 was chosen arbitrarily. Since the dataset contains 50,000 images, and so, there were 2500 batches in total. Besides, for the function AdjustWeights(), we set the learning rate to 0.001 for every DLM. The reason for choosing this value was that the weights in these DLMs might be small, and it would be difficult to find an optimum if the learning step is too large. For the training step in Algorithm 2 (lines 28 and 32), we set epochs to 20 and batch size to 256 and 128 for the three CNNs and the two ReNets, respectively.

We ran the above procedure five times to evaluate to what extent Apricot can improve the accuracy of the original DLMs. In each iteration, the time for training models and the accuracies of DLMs and rDLMs are collected for further analysis.

B. Results

The time consumed for training the original DLM and the corresponding rDLMs are summarized in TABLE III. Columns 2 to 5 present the training time of DLM, the total training time

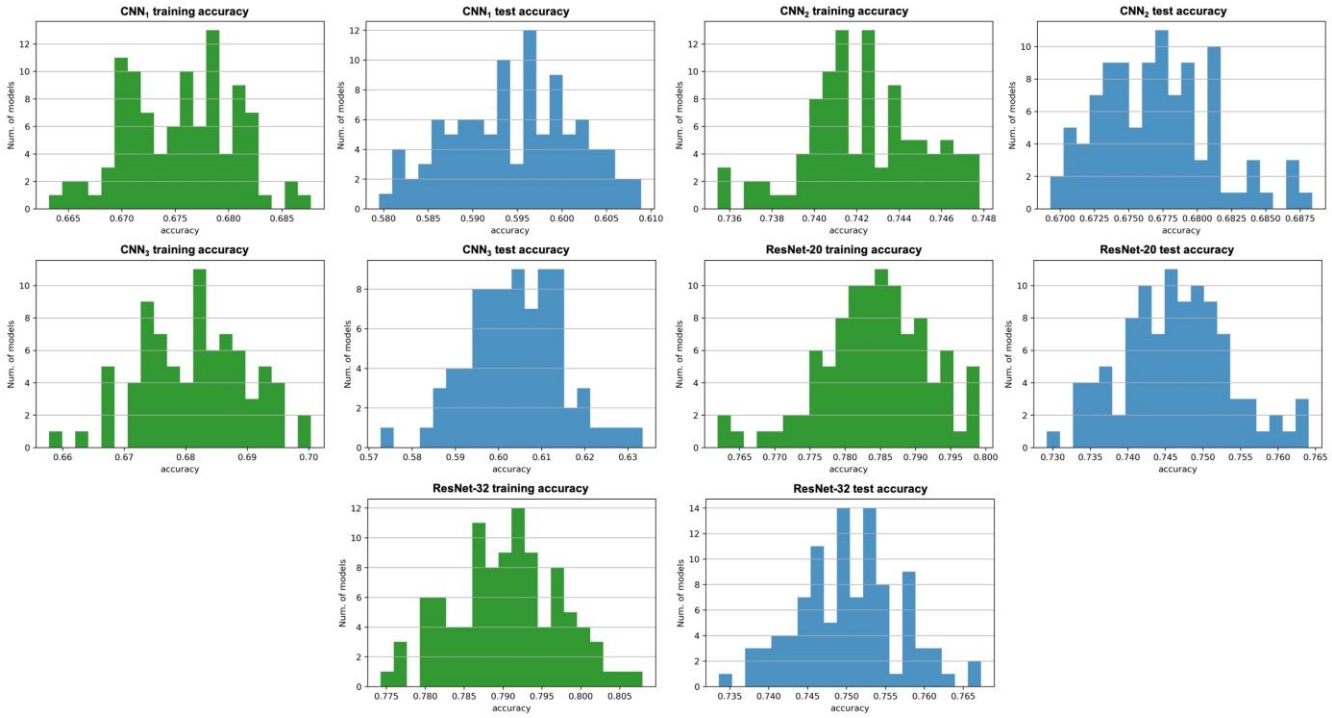

Fig. 2 The training accuracy and test accuracy achieved by the reduced DLMs in the experiment. For each model, there is a pair of plots. The plot on the left (green bars) is the result on training accuracy, and the plot on the right (blue bars) is the result on test accuracy.

of corresponding rDLMs, the number of rDLMs, and average training time of rDLMs, respectively. The time of training a DLM is much longer than that of training an rDLM. Training DLMs need 1min46s, 31min39s, 5min47s, 28min30s, and 31min21s for CNN_1, CNN_2, CNN_3, ResNet-20, and ResNet-32, respectively. In our experiment, we trained rDLMs sequentially, and it is worth noticing that these rDLMs can be trained and collected in a distributed and paralleled way. Column 6 shows percentages of time reduction for training rDLMs compared to training DLM. On average, training an rDLM reduces 71.0% in time compared to training a corresponding DLM.

The results of the training process of rDLMs are shown in TABLE IV. Columns 2 to 5 represent the average training accuracy, test accuracy, and their corresponding standard deviations, respectively. The training accuracies of rDLMs are close to each other, as evidenced by their relatively small standard deviations, so are test accuracies.

In Column 6, we show the percentage of test cases that at least one rDLM classifies each of them correctly, but the original DLM classifies them incorrectly. The result is consistent with the intuition that rDLMs have learned some features that the original DLM cannot easily get correctly, which gives Apricot some insights for weight adaptation. In general, the results show that exploring features and behaviors of rDLMs has a potential good effect in improving the accuracy of a DLM.

The distribution of the accuracy of rDLMs is shown in Fig. 1. Recall that we ran the experiment five times and for each DLM, we trained 100 rDLMs in total. Each subplot summarizes the accuracies of the 100 rDLMs obtained in five runs. The x-axis represents the training/test accuracy of the rDLMs produced in the experiment, and the y-axis represents the number of rDLMs with the corresponding accuracy. Bars in green represent the training accuracy, and bars in blue represent test accuracy. In general, the test accuracy of rDLMs is significantly lower than the original DLM by 20% or more. It appears that rDLM has been trained on a much smaller subset of the training dataset, and thus less information and features have been obtained and learned.

The experimental results of the adaptation phase are shown in TABLE V and Fig. 3. Columns 2 to 10 are average training accuracy, average test accuracy, and maximum gain in test accuracy (i.e., the highest increase in test accuracy that Apricot has ever achieved in the experiment) after the adaptation phases using Strategies 1, 2, and 3, respectively have been completed. Numbers in bold refer to the highest average test accuracy or highest increase in test accuracy in the experiment. Columns 11-12 are the overall average gain and maximum gain across three strategies, respectively. Fig. 3 presents the detailed training and test accuracies achieved in five runs. For each DLM, two subplots are reporting the training and test accuracies, respectively. These four bars are respectively arranged from left to right in each plot. Boxes in blue represent training and test accuracies achieved by original DLMs, and boxes in orange, green and red represent training and test accuracies achieved by DLMs after applying strategies 1, 2 and 3, respectively. In general, the training accuracy and test accuracy of DLMs after fixing are much higher than the original ones. Besides, all three strategies can improve the accuracy of DLMs.

The experimental results showed that for Strategy 2, Apricot consistently increases the accuracy on the three CNN models by 0.89%-1.55% with an average of 1.19% and on the two ResNet

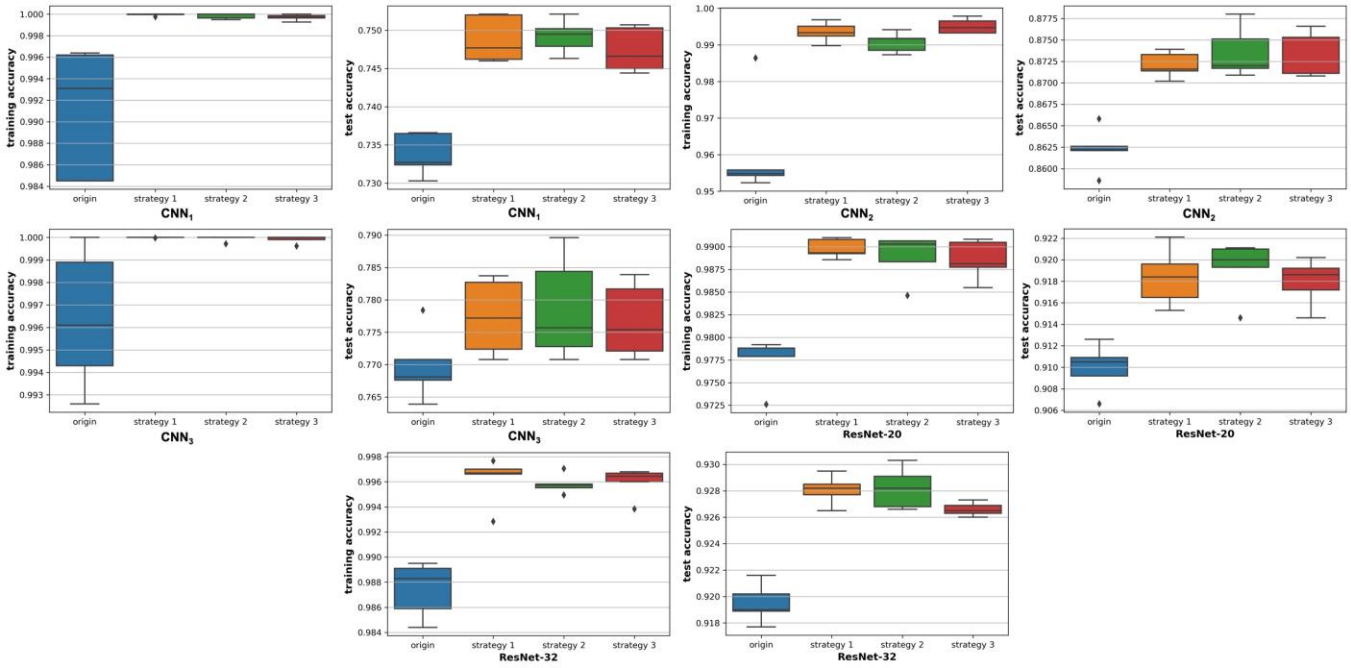

Fig. 3 The training accuracy and test accuracy achieved by DLMs in the experiment. For each DLM, there is a pair of plots. The plot on the left is the result on training accuracies of original DLM (blue boxes) and DLMs fixed by applying Strategies 1 (orange boxes), 2 (green boxes), and 3 (red boxes), respectively. The plot on the right is the result of test accuracy.

TABLE V EXPERIMENTAL RESULTS OF ADAPTATION PROCESS

Model	strategy 1			strategy 2			strategy 3			Overall average gain	Overall Max. gain
	Avg. training accuracy	Avg. test accuracy	Max. gain	Avg. training accuracy	Avg. test accuracy	Max. gain	Avg. training accuracy	Avg. test accuracy	Max. gain		
CNN_1	99.99%	74.88%	**2.18%**	99.98%	**74.92%**	1.99%	99.97%	74.74%	2.04%	1.55%	2.18%
CNN_2	99.35%	87.21%	**1.53%**	99.06%	87.35%	1.52%	99.50%	**87.38%**	1.45%	1.15%	1.53%
CNN_3	99.99%	77.77%	1.51%	99.99%	**77.87%**	**1.68%**	99.99%	77.68%	1.63%	0.89%	1.68%
ResNet-20	98.98%	91.84%	**1.29%**	98.89%	**91.92%**	1.19%	98.85%	91.80%	1.10%	0.92%	1.29%
ResNet-32	99.62%	92.81%	**1.00%**	99.58%	**92.82%**	0.93%	99.60%	92.66%	0.92%	0.87%	1.00%
Average improvement										1.08%	1.536%

models by 0.87%-0.92% with an average of 0.895%. Apricot achieves the highest increase in test accuracy on all DLMs by 1.00%-2.18% by applying Strategy 1. We note that Apricot does not require additional inputs to train the given model, and the accuracies achieved by the two ResNet before applying Apricot are 91.00% and 91.95%, and Apricot has achieved a significant improvement. By combining the results of the three strategies, an average improvement across all models is 1.08%, and an average maximum gain in accuracy is 1.536%. It appears to us that Apricot delivers a promising result and represents a feasible direction to produce DLMs with repaired weights in the experiment.

C. Further Analysis

Apricot achieved more significant improvements on the three CNN models than on ResNet-20 and ResNet-32. Apart from the reason that these two models have achieved high accuracy on the test dataset, we believe that another reason is the more complex structure of ResNet.

Recall that Apricot's improvement on ResNet is 0.895% on average using Strategy 2. On ResNet-20 and ResNet-32, they are 0.92% and 0.87%, respectively. As to be discussed below, by comparing with the results in the literature, we consider that Apricot is effective in improving ResNet.

In [16], ResNet-20, ResNet-32, ResNet-44, ResNet-56, ResNet-110 achieved the test accuracy of 91.25%, 92.49%, 92.83%, 93.03% and 93.57%, respectively. The differences in test accuracy between consecutive pairs of ResNet were 1.24%, 0.34%, 0.20%, and 0.54%, respectively. It showed that the increase in test accuracy of ResNet strongly relied on adding more residual layers, and yet the margin of the increase was still minimal. For instance, by increasing the number of residual layers from 20 to 110 (represented by ResNet-20 and ResNet-110, respectively), the difference in test accuracy was less than 2.0%. A similar evidence was presented on the official website of CIFAR10-ResNet [53], on which ResNet-20 and ResNet-32 achieved the test accuracy of 92.16% and 92.46%, respectively.

The difference was 0.30% between the two models on this dataset, and that between ResNet-30 and ResNet-110 is 0.19%. From TABLE V, the improvements from ResNet-20 to ResNet-32 produced by Apricot is in fact large.

Compared to Strategy 1 and Strategy 3, Strategy 2 achieved more significant improvements except on CNN_2, but the results on CNN_2 achieved by Strategy 2 was 1.12%, which was relatively high. The results show that rDLMs were useful for adjusting weights and optimizing the original DLM in the experiment.

Strategy 1 achieved a higher increase in test accuracy except on CNN_3 by 1.00%-2.18%. The results indicate that although Strategy 1 did not produce a higher average performance compared to Strategy 2, it was likely to fix more ill-trained weights and provide insightful information for finding a global optimum.

In general, if rDLMs classify the same input correctly, there is something in common, such as capturing similar features (realized as weights), and the weights of these rDLMs can provide insightful information for adjusting weights of the original DLM. Besides, those incorrectly classified rDLMs can help search for a global optimum. As the experiment shows, Apricot achieved the highest average test accuracy by applying Strategy 2 in four DLMs and the highest gains in test accuracy by applying Strategy 1 in four DLMs. The overall results seem indicating that both correct and incorrect rDLMs can provide useful search directions. These correct rDLMs can achieve a relatively stable increase in accuracy, and incorrect rDLMs can help perturb weights and avoid local optimums. However, further studies should be made to confirm this conjecture.

From our experience in the experiment, we also found it challenging to extract useful information precisely from rDLMs. That is, it is challenging to adjust weights in more detailed with a clear trend. Besides, recognizing which weights need to be modified and how to modify them are still unclear to us. Through Apricot, we have presented a novel and effective approach to utilize different kinds of DLMs. There is still a large room of improvement in formulating effective synthesis strategies and extracting features from rDLMs more precisely.

Some optimization methods, like regularization, are also helpful for improving the accuracy of DLM. The difference between our work and neural network regularization is that our objective is to utilize rDLMs to further optimize the original DLM by adjusting its weights guided by rDLMs, but regularization methods aim to increase the accuracy or enhance the robustness by making weight matrices sparser (by adding L_0 norm) or making weights smaller (by adding L_2 norm), causing a significant change in weights. Besides, it is worth exploring if our approach can further increase the accuracy of a DLM that has been regularized.

D. Threats to Validity

We only evaluated Apricot on the CFIAR-10 dataset on two kinds of DLMs. Apricot is general, which its weight-adaptation approach can be applied to different kinds of DNN architectures like RNN and GAN. Evaluating Apricot on more datasets and models can strengthen the generalization of this study, and our future work will evaluate Apricot on different state-of-the-art DLMs and large datasets.

During our experimentation, we found that if the original DLM got overfitted, i.e., the training accuracy was close to 100%, it was difficult for Apricot to repair the DLM. It is because that during the adaptation process, Apricot only focuses on inputs that the original DLM predicts incorrectly. If the training accuracy is close to 100%, no or very few adaptation processes could likely be triggered.

For example, on the MNIST dataset [29], the test accuracy of state-of-the-art DLMs is nearly 99%, and the training accuracy is 100%. We have experimented with it on a smaller scale but found that Apricot cannot be effectively triggered. That is why we omit the evaluation on MINST dataset. It also reveals that such scenarios are *not* the current use cases of Apricot.

Apricot starts with failing test cases with respect to the original DLM. In our future work, we will study whether it is practical to trigger Apricot with non-failing test cases with respect to the original DLM. We also note that for most state-of-the-art DLMs and large real-world datasets, it is unlikely to reach 100% accuracy for both training and test processes. Thus, we believe that Apricot is applicable in a majority of scenarios.

Algorithm 2 presents a basic algorithm to update the weights of a DLM (lines 20-21). Owing to this issue, we believe that the performance of Apricot can be further improved if this basic algorithm is replaced by a more advanced one, such as a search-based algorithm. A study on using different weight adjustment schemes, both linear and non-linear ones, is worthy of study in the future.

A possible issue is that in Apricot, our weight adjustments rely on the original weights of the generated rDLMs, i.e., the overall adjustment result relies on the performance of rDLMs. If these rDLMs are not sufficiently trained, it will be more challenging to find a better solution. To overcome this problem, more rDLMs (which currently the experiment used 20 such rDLMs) can be used in Apricot, reducing the effect of potential bias and likely further improving the accuracy performance achieved by Apricot, but training more models can be time-consuming in some cases. Deep learning is GUP-heavy by nature. With success in obtaining impressive classification results, it seems to us that the time and cost budgets for fixing a model can be higher.

Another possible issue is that rDLMs may retain different features compared to the original DLM, or rDLMs and DLM may learn similar features, but these features are kept in a different permutation with respect to the node order in the architecture. This issue can be alleviated by adding an initialization step at the beginning, i.e., DLM is firstly trained after several epochs, and then both architecture and coarsely trained weights are copied to rDLMs. In practice, it is common to start building a learning model with a pre-trained one [3] provided by companies (e.g., [50]). This initialization step has similar concepts. It ensures that some features can be kept in weights and a later training process, ensuring that neurons and weights of DLM and rDLMs in the same position perform the same tasks. The effectiveness of such initialization steps needs further study, and we leave it as our future work.

The experimental results show that both correctly and incorrectly classified rDLMs are complementary to the original DLM, and the effects of these two kinds of rDLMs are different: both rDLMs can help improve the accuracy of DLMs. Those correct rDLMs can provide an insightful search direction, and incorrect rDLMs can help perturb weights to find global optimums and avoid local optimums. To fix ill-trained weights in DLMs, Apricot provides different strategies for different considerations and preferences.

We ran the experiment five times to measure the effectiveness of Apricot in improving the accuracy of DLMs. As shown in Fig. 3, Apricot has presented consistent improvement in accuracy across all five models, but the differences between these three strategies are relatively small. It may worth exploring a time-efficient strategy to reduce the cost incurred by Apricot.

To alleviate this problem, strengthening the experiment by replicate it can help present a more apparent trend on the effectiveness of these three strategies.

In Apricot, the differences between DLM and rDLMs are utilized for adjusting weights. Unlike the use of differential analysis in MODE to identify ill-trained weights or buggy neurons, Apricot does not aim to identify ill-trained weights or problematic neurons, and such weight differences are applied to guide searching weights in a coarse level, which is one of the main differences between Apricot and other differential analysis techniques like MODE.

V. Related Work

A. Deep Neural Network Testing

In recent years, there are several proposals for testing deep neural networks. The first category is about test adequacy criteria and test case generation.

Pei et al. [39] firstly proposed the neuron coverage to evaluate the test sufficiency of a DLM. In neuron coverage, each neuron in each hidden layer is measured on whether it has been activated during the testing phase, and if this is the case, the neuron is said to be covered. They propose the notion of neuron coverage based on this idea and measures the adequacy of testing efforts. Their work also proposed *DeepXplore* to explore corner cases of the input space, searching potential adversarial examples that may lead to severe problems. Nonetheless, previous experimental results showed that even a small set of test data can achieve a high neuron coverage rate. Ma et al. [31] proposed a set of multi-granularity testing criteria for DLMs. The main idea of *k*-multisection coverage is that the ranges of value of outputs for each neuron are collected during the training phase and then are divided into *k* sections. A neuron is said being covered if all these sections of the neuron are covered at least once. The neuron boundary coverage is to count how many cases that the outputs of neurons exceed or below a given bound. Sun et al. [47] further proposed four testing criteria, Sign-Sign (SS) coverage, Value-Sign (VS) coverage, Sign-Value (SV) coverage, and Value-Value (VV) coverage, inspired by the MC/DC coverage criterion. These criteria give us some insights into neuron behaviors of DNN models, and the experimental results showed that these criteria can help debug neural networks and measure test sufficiency.

Some studies explored different methods to test deep learning models. Tian et al. [51] proposed a method, called DeepTest, to generate new inputs for self-driving cars, and maximize neuron coverage at the same time. Experimental results showed that only a small image transformation could lead to misclassifications and erroneous behaviors.

Inspired by mutation testing, Ma et al. [32] proposed DeepMutation to test deep learning models. Like mutation testing in traditional software, a set of mutation operators at both code level and model level are predefined like weight shuffling, layer deactivation, and layer addition. The mutation score used in their work is a metric to evaluate the quality of a test dataset.

Ma et al. [33] proposed MODE for debugging and fixing deep learning models. Two types of common problems are analyzed: under-fitting problems and overfitting problems. Given a buggy model M_b that contains an under-fitting or overfitting problem, a hidden layer is selected where its outputs (that are viewed as features) are most representative. Then a feature model is constructed and trained by extracting layers of M_b from the input layer to the selected hidden layer followed by an extra full-connected layer. Weights between the selected hidden layer and the constructed output layer are viewed as the degree of importance of the corresponding features. Given a test case x, a heat map is constructed for recognizing the degree of importance of features. For each category, MODE recognizes features responsible for correct/incorrect classification by implementing a differential analysis on heat maps that are generated by correctly/incorrectly classified inputs. Then training samples to which these features are most sensitive to them are selected for training M_b. MODE provides a detailed and specific method for recognizing buggy features (or neurons) in a specified layer, but other layers are not analyzed. In Apricot, we compare the differences between the original DLM and correct/incorrect rDLMs. Unlike MODE, Apricot uses these differences to adjust weights rather than recognize ill-trained weights or neurons.

B. Exposing Vulnerabilities of Deep Neural Networks

Deep neural networks are vulnerable to adversarial examples [48]. As we have discussed, approaches of neural network testing and adversarial attack algorithms can give us some insights about erroneous behaviors of DNN models and expose potential vulnerabilities, providing useful information for repairing neural networks.

There is also a large body of work on adversarial attacks. Szegedy et al. [48] proposed box-constrained L-BFGS to find adversarial examples with high success rates. L_2 norm is used to measure the distance between original inputs and generated inputs. They also found that training the original DNN model using generated adversarial examples can regularize the model, but it can be time-consuming and impractical for actual usage.

Goodfellow et al. [13] proposed the Fast Gradient Sign Method (FGSM) to generate adversarial examples. FGSM uses L_∞ to measure distances and calculates gradients of the loss function to guide its search direction. The experimental results showed that FGSM is effective in generating adversarial examples, and its idea inspired many other adversarial attack algorithms.

DeepFool [35] is another algorithm for generating adversarial examples. This work proposed to find hyperplanes separating different classes. In this regard, developers can find some misleading examples near these hyperplanes. Using this assumption simplifies the DNN model for generating adversarial examples.

Xiao et al. [55] proposed to use generative adversarial networks (GANs) to generate adversarial examples. The basic idea is that using a generator to generate adversarial inputs and using a discriminator to distinguish between the original inputs and the generated inputs. The prediction result of the discriminator is used as a part of the loss function of the generator to improve its performance.

Su et al. [46] proposed a new attack algorithm that only changes one pixel to generate adversarial examples, called "One Pixel Attack". For the image classification task, many adversarial attack algorithms mutate all pixels and use L_1, L_2 and L_∞ as constraints, ensuring the perturbation is acceptable. Their paper uses L_0 to restrict that only one pixel can be modified without limiting its value. It is acceptable for humans that the meaning of an image will not change if only one pixel changed, especially for those high-resolution images. Although it could be challenging to find adversarial examples by changing only one pixel and the success rate may not as high as other adversarial attack algorithms like LSA [36] and FGSM [1], the proposed method gives us insights into how one pixel affects final classification results.

C. Neural Network Regularization

Neural network regularization is a useful method to enhance the robustness of deep neural networks without losing accuracy. A common way of regularizing neural networks is to add a term called L_p norm to a loss function.

Han et al. [15] proposed a new training process that contains regularization steps. The overall process is that given a converged DNN, for each layer, it finds a threshold and removes all connections where their weights are below the threshold. After obtaining the sparse DNN, it freezes the remaining weights and retrains the model. Then, it retrains the whole model. The reason why it works is that after the Dense-Sparse-Dense training process, the weight distribution becomes "sharper", which means that most weights are close to 0, reducing the possibility of erroneous behaviors caused by those weights.

Chang et al. [5] proposed a regularization approach to improve the robustness of deep neural networks and discussed the impacts of different norms on regularizing deep neural networks. The original $L_{1/2}$ norm is $\sum_{i=1}^{n} \sqrt{w_i}$. A possible issue using $L_{1/2}$ norm is that when w_i is small, the gradient of w_i can be extremely large. To overcome this problem, the penalty is modified from $L_{1/2}$ norm to L_2 norm when $|w_i|$ is smaller than a constant c, which ensures that the gradients are not too large. From their experimental results, Chang et al. concluded that the modified $L_{1/2}$ norm can significantly reduce the complexity of deep neural networks, and the accuracy can be preserved.

VI. Conclusion

In this paper, we have presented Apricot, a novel weight-adaptation approach to evolve DLMs iteratively. Our insight is that DLMs trained on smaller datasets can provide useful information on the weight adjustment and adaptation, thus optimizing a larger DLM further. Apricot generates a set of rDLMs, adjusts the weights of the original DLM toward the average weights of these rDLMs that each classifies an input correctly, and/or away from the average weights of these rDLMs that each classifies the same input incorrectly, where the input in question is a failing test case experienced by the original DLM. If a higher test accuracy is achieved after weight adjustment, a training process is triggered for further improvement. The experimental results on the CIFAR-10 dataset and five DLMs have shown that Apricot can increase the test accuracy of these DLMs by 0.87%-1.55% with an average of 1.08%, and the highest increases in test accuracy that Apricot has ever achieved in the experiments were 1.00%-2.18% with an average of 1.536%.

One line of future work is to debug and repair deep learning models. There are some challenging problems like evaluating the impacts of a small set of weights on the overall DLM and repairing DLMs. We observe that there is still a huge gap between understanding the behaviors of deep neural networks and the needs underlying the repair work of these networks. Besides, evaluating the performance of deep neural models only by test accuracy or robustness may still not be enough in some cases. We would like to study these issues in the future.

References

[1] F. Ahmadizar, K. Soltanian, F. AkhlaghianTab, L. Tsoulos, "Artificial neural network development by means of a novel combination of grammatical evolution and genetic algorithm," *Engineering Applications of Artificial Intelligence*, vol. 39, pp.1-13, 2015.

[2] B. Baker, O. Gupta, N. Naik, R. Rakar, "Designing Neural Network Architectures using Reinforcement Learning," arXiv: 1611.02167, Mar. 2017.

[3] Building powerful image classification models using very little data [Online]. Available: https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html.

[4] E. Cantú-Paz, C. Kamath, "An empirical comparison of combinations of evolutionary algorithms and neural networks for classification problems," IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), vol.35, no.5, pp.915-927, 2005.

[5] J. Chang, J. Sha, "Prune Deep Neural Networks With the Modified $L_{1/2}$ Penalty," IEEE Access, vol. 7, pp.2273-2280, 2019.

[6] Convolutional Neural Networks for CIFAR-10, https://github.com/BIGBALLON/cifar-10-cnn.

[7] K. Deb, A. Anand, D. Joshi, "A computationally efficient evolutionary algorithm for real-parameter optimization," Evolutionary computation, vol.10, no.4, pp.371-395, 2002.

[8] J. Deng, W. Dong, R. Socher, L. Li, K. Li, L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," In Proc. IEEE conference on computer vision and pattern recognition, 2009, pp.248-255.

[9] J. Duchi, E. Hazan, Y. Singer, "Adaptive subgradient methods for online learning, and stochastic optimization," Journal of Machine Learning Research, Vol. 12, pp.2121-2159, 2011.

[10] C. L. Giles, G. M. Kuhn, and R. J. Williams, "Dynamic recurrent neural networks: Theory and applications," IEEE Trans. Neural Networks, vol.5, pp.153–156, 1994.

[11] X. Glorot, Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," In Proc. the thirteenth international conference on artificial intelligence and statistics, 2010, pp.249-256.

[12] I. Goodfellow, J. Pouget-Abadie, M. Mirza, et al., "Generative Adversarial Nets," In Proc. Advances in neural information processing systems, 2014, pp. 2672-2680.

[13] I. J. Goodfellow, J. Shlens, C. Szegedy, "Explaining and Harnessing Adversarial Examples," arXiv:1412.6572v3 [cs], Mar. 2015.

[14] S. Gu, L. Rigazio, "Towards deep neural network architectures robust to adversarial examples," arXiv: 1412.5068 [cs], Dec. 2014.

[15] S. Han, J. Pool, S. Narang, et al., "DSD: regularizing deep neural networks with dense-sparse-dense training flow," arXiv: 1607.04381 [cs], 2016.

[16] K. He, X. Zhang, S. Ren, J. Sun, "Deep residual learning for image recognition," In Proc. IEEE conference on computer vision and pattern recognition, 2016, pp.770-778.

[17] V. Heidrich-Meisner, C. Igel, "Neuroevolution strategies for episodic reinforcement learning," Journal of Algorithms, vol.64, no.4, pp.152-168, 2009.

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

[19] H. X. Huang, J. C. Li, C. L. Xiao, "A proposed iteration optimization approach integrating backpropagation neural network with genetic algorithm," *Expert Systems with Applications*, vol. 42, no. 2, pp.146-155, 2015.

[20] C. Igel, "Neuroevolution for reinforcement learning using evolution strategies," In Proc. Congress on Evolutionary Computation, 2003, pp. 2588-2595.

[21] Image Preprocessing [Online]. Available: https://keras.io/preprocessing/image.

[22] S. Ioffe, C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," arXiv: 1502.03167 [cs], 2015.

[23] Keras Documentation [Online]. Available: https://keras.io/examples/cifar10_cnn/.

[24] Keras Documentation: Usage of activations [Online]. Available: https://keras.io/activations/.

[25] D. Kingma, J. Ba, "Adam: A Method for Sthochastic Optimization," arXiv: 1412.6980, Jan, 2017.

[26] A. Krizhevsky, G. Hinton, "Learning multiple layers of features from tiny images," Technical report, University of Toronto, 2009.

[27] A. Krizhevsky, I. Sutskever, G. E. Hinton, "Imagenet classification with deep convolutional neural networks," In Proc. Advances in neural information processing systems, 2012, pp.1097-1105.

[28] Y. LeCun, Y. Bengio, G. Hinton, "Deep learning," Nature, vol.521, pp.436-444, 2015.

[29] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998.

[30] F. Li, J. M. Zurada, Y. Liu, W. Wu, "Input Layer Regularization of Multilayer Feedforward Neural Networks," IEEE Access, vol.5, pp.10979-10985, 2017.

[31] L. Ma, F. Juefei-Xu, F. Zhang, et al., "Deepgauge: Multi-granularity testing criteria for deep learning systems," In Proc. ACM/IEEE International Conference on Automated Software Engineering, 2018, pp.120-131.

[32] L. Ma, F. Zhang, J. Sun, et al., "DeepMutation: Mutation testing of deep learning systems," In Proc. IEEE 29th International Symposium on Software Reliability Engineering, 2018, pp.100-111.

[33] S. Ma, Y. Liu, W. C. Lee, X. Zhang, A. Grama, "MODE: automated neural network model debugging via state differential analysis and input selection," In Proc. 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp.175-186.

[34] D. J. Montana, L. Davis, "Training Feedforward Neural Networks Using Genetic Algorithms," In Proc. Joint Conference on Artificial Intelligence, 1989, pp.762-767.

[35] S. M. Moosavi-Dezfooli, A. Fawzi, P. Frossard, "DeepFool: a simple and accurate method to fool deep neural networks," In Proc. IEEE conference on computer vision and pattern recognition, 2016, pp. 2574-2582.

[36] N. Narodytska, S. Kasiviswanathan, "Simple black-box adversarial attacks on deep neural networks," In Proc. IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2017, pp.1310-1318.

[37] A. Odena, I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," arXiv: 1807.10875 [cs], Jul. 2018.

[38] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, A. Swami, "Practical Black-Box Attacks against Machine Learning," In Proc. ACM on Asia Conference on Computer and Communications Security, 2017, pp.506-519.

[39] K. Pei, Y. Cao, J. Yang, S. Jana, "DeepXplore: Automated Whitebox Testing of Deep Learning Systems," In Proc. Symposium on Operating Systems Principles '17, 2017, pp.1-18.

[40] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, A. Kurakin, "Large-scale evolution of image classifiers," In Proc. the 34th International Conference on Machine Learning, 2017, pp.2902-2911.

[41] D. E. Rumelhart, G. E. Hinton, R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[42] S. Scardapane, D. Comminiello, A. Hussain, A. Uncini, "Group sparse regularization for deep neural networks," Neurocomputing, vol.241, pp.81-89, 2017.

[43] K. Simonyan, A. Zisserman, Very "Deep convolutional networks for large-scale image recognition," arXiv:1409.1556v6 [cs], Apr. 2015.

[44] K. Soltanian, F. A. Tab, F. A. Zar, I. Tsoulos, "Artificial neural networks generation using grammatical evolution," In Proc. Iranian Conference on Electrical Engineering, 2013, pp.1-5.

[45] Stanford University. (2019). CS231n Convolutional Neural Networks for Visual Recognition [Online]. Available: http://cs231n.github.io/ [Accessed: Apr. 27, 2019].

[46] J. Su, D. V. Vargas, K. Sakurai, "One pixel attack for fooling deep neural networks," IEEE Transactions on Evolutionary Computation, 2019.

[47] Y. Sun, X. Huang, D. Kroening, "Testing deep neural networks," arXiv: 1803.04792v3 [cs], Mar. 2018.

[48] C. Szegedy, W. Zaremba, I. Sutskever. J. Bruna, D. Erhan, I. Goodfellow, R. Fergus, "Intriguing properties of neural networks," arXiv: 1312.6199v4 [cs], Feb. 2014.

[49] TensorFlow: An end-to-end open source machine learning platform [Online]. Available: https://www.tensorflow.org/.

[50] TensorFlow-Slim image classification model library [Online]. Available: https://github.com/tensorflow/models/tree/master/research/slim.

[51] Y. Tian, K. Pei, S. Jana, B. Ray, "DeepTest: Automated testing of deep-neural-network-driven autonomous cars," In Proc. 40th international conference on software engineering, 2018, pp.303-314.

[52] A. P. Topchy, O. A. Lebedko, "Neural network training by means of cooperative evolutionary search," Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, vol.389, pp.240-241, 1997.

[53] Trains a Resnet on the CIFAR10 dataset [Online]. Available: https://keras.io/examples/cifar10_resnet.

[54] L. Wan, M. Zeiler, S. Zhang, Y. LeCun, R. Fergus, "Regularization of Neural Networks using DropConnect," In Proc. International conference on machine learning, 2013, pp.1058-1066.

[55] C. Xiao, B. Li, J. Y. Zhu, W. He, M. Liu, D. Song, "Generating adversarial examples with adversarial networks," arXiv: 1801.02610 [cs], Feb. 2019.

[56] S. H. Yang, Y. P. Chen, "An evolutionary constructive and pruning algorithm for artificial neural networks and its prediction applications," Neurocomputing, vol.86, pp.140-149, 2012.

[57] W. Zaremba, I. Sutskever, O. Vinyals, "Recurrent Neural Network Regularization." arXiv:1409.2329 [cs], Sep. 2014.

[58] M. Zeiler, "ADADELTA: An Adaptive Learning Rate Method," arXiv: 1212.5701, Dec. 2012.

[59] B. Zhou, A. Lapedriza, A. Khosla, A, Oliva, A. Torralba, "Places: A 10 million image database for scene recognition," vol.40, no.6, pp.1452-1464, 2018.

[60] B. Zoph, Q. Le, "Neural Architecture Search with Reinforcement Learning," arXiv: 1611.01578, 2016.