## Search Based Repair of Deep Neural Networks

Jeongju Sohn School of Computing, KAIST Daejon, Republic of Korea kasio555@kaist.ac.kr Sungmin Kang School of Computing, KAIST Daejon, Republic of Korea stuatlittle@kaist.ac.kr Shin Yoo School of Computing, KAIST Daejon, Republic of Korea shin.yoo@kaist.ac.kr

#### **ABSTRACT**

Deep Neural Networks (DNNs) are being adopted in various domains, including safety critical ones. The wide-spread adoption also calls for ways to guide the testing of their accuracy and robustness, for which various test adequacy criteria and input generation methods have been recently introduced. In this paper, we explore the natural subsequent step: given an input that reveals unexpected behaviour in a trained DNN, we propose to repair the DNN using input-output pairs as a specification. This paper introduces Arachne, a novel program repair technique for DNNs. Arachne first performs sensitivity based fault localisation to limit the number of neural weights it has to modify. Subsequently, Arachne uses Particle Swarm Optimisation (PSO) to directly optimise the localised neural weights until the behaviour is corrected. An empirical study using three different benchmark datasets shows that Arachne can reduce the instances of the most frequent misclassification type committed by a pre-trained CIFAR-10 classifier by 27.5%, without any need for additional training data. Patches generated by Arachne tend to be more focused on the targeted misbehaviour than DNN retraining, which is more disruptive to non-targeted behaviour. The overall results suggest the feasibility of patching DNNs using Arachne until they can be retrained properly.

#### **KEYWORDS**

Automated Program Repair, Deep Neural Network

#### 1 INTRODUCTION

Deep Neural Networks (DNNs) have been rapidly being adopted in many application areas [19], ranging from image recognition [17, 31], speech recognition [11], machine translation [13, 30], to safety critical domains such as autonomous driving [2, 3] and medical imaging [21]. With the widening application areas, there has been a growing concern that these DNNs should be *tested*, both in isolation and as a part of larger systems, to ensure dependable performance. The need for testing resulted in two major classes of techniques. First, to evaluate sets of test inputs, many test adequacy criteria have recently been proposed [15, 24, 32]. Second, ways to synthesise new inputs by applying small perturbations (such as emulation of different lighting or weather conditions) to given inputs have been introduced [22, 38]. Newly synthesised inputs can not only improve the proposed test adequacy due to the increased input diversity, but also actually reveal unexpected behaviour of DNNs under test.

Compared to the traditional software developed by human engineers, however, the stages after the detection of unexpected behaviour remain relatively unexplored for DNNs. This is due to the major difference between the way DNNs and code are developed: one is trained by algorithms based on training data, while the other is written by human engineers based on specifications.

Consequently, existing efforts to *repair* the unexpected behaviour mostly focus on how to efficiently and effectively retrain the DNNs. For example, Ma et al. [23] try to identify the features that are the most responsible for the unexpected behaviour, and synthesise inputs for retraining based on their relevance to these features.

Given that DNNs are initially trained by algorithms, retraining as repair can be considered as a natural solution. One weakness of retraining as a repair technique, however, is its stochastic nature. Unlike human debugging, there is no guarantee that, at the end of the retraining process, the unexpected behaviour will be removed while all correct behaviour are retained. The retraining may, or may not, correct the misbehaviour, and it may do so at the expense of other, initially correct, behaviour. Even when compared to stochastic Automated Program Repair (APR) techniques [10, 27, 37], retraining is less *focused* on the misbehaviour as the loss function usually focuses only on the overall accuracy improvement. We argue that a need for focused repairs can be realistic in real world use cases: for example, users may prioritise higher accuracy for a specific label of a DNN classifier in safety critical applications, even if it requires a trade-off with accuracy for lower priority labels.

This paper introduces Arachne, a search-based automated program repair technique for DNNs. Instead of retraining, Arachne directly manipulates the neural weights and searches the space of possible DNNs, guided by a specifically designed fitness function following Generate and Validate APR techniques [10, 27, 37]. Arachne resembles APR techniques for traditional code in many aspects: it adopts a fault localisation technique, and uses both positive and negative inputs to retain correct behaviour and to generate a patch, respectively. Its internal representation of a patch is even simpler than the representation for code, as a set of neural weights can be represented as a vector of real numbers. Consequently, Arachne uses Particle Swarm Optimisation (PSO) [14, 35] as its search algorithm. During a fitness evaluation, Arachne updates the chosen neural weights of the DNN under repair with values from the PSO candidate solution, executes the inputs, and computes the fitness value based on the outcomes.

We have empirically evaluated Arachne on three image classification benchmark datasets and three corresponding DNN classifiers: Fashion MNIST [36], CIFAR-10 [16], and German Traffic Sign Recognition Benchmark (GTSRB) [29]. Initially, to study the impact that fault localisation and a particular choice of misbehaviour has on the success rate of the repair, we evaluate the feasibility of Arachne against faults that are artificially induced via under-training of DNNs. Subsequently, we also show that Arachne can repair a fully trained DNN as newly incoming inputs reveal previously unknown unexpected behaviour. This adaptive repair process does not require any additional data, apart from the new inputs that revealed the unexpected behaviour.

The technical contributions of this paper are as follows:

- The paper introduces Arachne, a novel search based repair technique for DNNs. Unlike existing approaches that retrain a DNN model using more inputs, Arachne aims to directly repair a pre-trained model by adjusting neural weights.
- We empirically evaluate Arachne against three widely studied image classification benchmarks, using under-training to induce unexpected behaviour. Arachne can produce repairs more focused on targeted misbehaviour, while only minimally perturbing other behaviour. Retraining, on the other hand, can significantly change untargeted behaviour.
- We show how Arachne can be used, in practice, to repair unexpected behaviour revealed by new incoming inputs. Arachne can repair misclassification results, while minimising disruptions to existing behaviour of a DNN.

The remainder of this paper is organised as follows. Section 2 describes the components of our DNN repair technique, Arachne. Section 3 sets out the research questions and describes experimental protocols. Section 4 outlines the set-up of the empirical evaluation, the results of which is discussed in Section 5. Section 6 discusses the implications and the future work. Section 7 presents the related work and Section 8 concludes.

# 2 ARACHNE: SEARCH BASED REPAIR FOR DEEP NEURAL NETWORKS

This section motivates the development of Arachne and describes its internal components.

#### 2.1 Motivation

Arachne aims to *repair* an existing DNN. By repairing a DNN, we mean improving its performance with respect to the accuracy of decisions it makes. Given that DNNs are machine learning models, the natural way of improving its performance would be either additional or better prepared learning. Improvement by learning would typically involve a larger volume of trained data, curated more carefully to avoid the unexpected behaviour. Unfortunately, preparing training data is known to be the major bottleneck to practical application of machine learning due to the high cost [1, 26].

Our motivation for Arachne is not to replace well designed learning process as a means of improving general learning capability and model accuracy. Rather, we want to introduce an alternative repair technique that can introduce a direct and focused improvement over a small set of unexpected behaviour, without requiring larger and better curated training datasets. In this sense, we expect Arachne to complement other training based techniques. Note that Arachne targets unexpected behaviour that can be repaired by neural weight manipulations alone: if a DNN model underperforms due to its inappropriate neural network model architecture, we do not expect Arachne to be effective at repairing it.

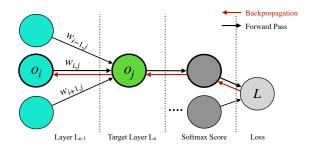
#### 2.2 Overview

Arachne performs two main operations: localisation and patch generation. In the localisation phase, given a set of inputs that cause unexpected behaviour, Arachne identifies a set of neural weights that are likely to be related to the observed misbehaviour. This phase is called localisation; the intuition is that changing the

values of these neural weights is likely to affect the model behaviour, possibly to the correct direction.

Subsequently, in the patch generation phase, Arachne augments the negative inputs (i.e., inputs that lead to the unexpected behaviour) with a fixed number of positive inputs (i.e., inputs that are processed correctly). Similarly to Generate and Validate (G&V) Automated Program Repair techniques such as GenProg [33] and Arja [37], the positive inputs are used by the fitness function of Arachne to retain the initially correct behaviour of the DNN under repair. Following a fitness function based on execution of both positive and negative inputs, Arachne uses Particle Swarm Optimisation [14] to search for the set of neural weights that would correct the behaviour of the DNN under repair. Details of localisation and patch generation phases are discussed in the following subsections.

Figure 1: Layers and Neural Weights Considered by Arachne



## 2.3 Localisation Phase

Attempting to adjust all neural weights of a DNN model for repair would be too costly, as even relatively simple DNNs consist of thousands of weight parameters. To reduce the search space, Arachne only considers the neural weights connected to the final output layer, as depicted in Figure 1. In addition, Arachne adopts a simple localisation method to identify weights that are likely to be responsible for the targeted misbehaviour.

Algorithm 1 presents pseudo-code of the localisation method. First, Arachne sorts the neural weights according to the gradient loss of the faulty input backpropagated to the corresponding neuron (Line 1 to 6), and subsequently only considers top  $N_g$  neural weights to narrow down the number of candidate weights (Line 8). In Figure 1, for the neural weight  $w_{i,j}$  that connects the ith neuron in layer  $L_{n-1}$  to the jth neuron in our target layer, the gradient loss backpropagated from the final loss, L, is computed as  $\frac{\partial L}{\partial w_{i,j}} = \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial w_{i,j}}$  ( $o_j$  is the output activation value of the jth neuron in  $L_n$ ).

In addition to gradient loss, Arachne also considers how much impact each neural weight has on the final outcome, when forward-propagating the faulty input (Line 8 to 12). The forward impact of a neural weight on the final outcome,  $fwd_imp$ , is computed simply as the product of the given weight and the activation value of the corresponding neuron in the previous layer: for example,  $o_iw_{i,j}$  for the neural weight  $w_{i,j}$  in Figure 1. This computation is performed by ComputeForwardImpact (Line 10).

Note that, ideally, we want to target neural weights that are both responsible for the unexpected behaviour (i.e., large gradient loss) and the most influential to the outcome (i.e., high impact). However, since a weight value that has the highest impact is not necessarily the cause of the unexpected behaviour, we treat these two factors separately. Consequently, Arachne considers gradient loss and forward impact as two competing objectives to be maximised. The result of localisation is the set of weights that constitutes the Pareto front for these two objectives (Line 13).

## Algorithm 1: Localisation method of Arachne

```
input: A DNN model to be repaired, M, a set of neural weights, W, a
           set of inputs that reveal the fault, I_{neq}, a loss function, \mathcal{L},
           and the number of neural weight candidates to choose based
           on gradient loss, N_q
   output: a set of neural weights to target for repair, W_t
1 candidates ← [];
2 for weight in W do
       grad\_loss \leftarrow ComputeGradient(weight, M, I_{neg}, \mathcal{L});
       Add tuple (weight, grad_loss) to candidates;
5 end
6 Sort candidates w.r.t. gradient loss;
7 pool ← {};
s for i = 0 to N_q do
       (weight, grad\ loss) \leftarrow candidates[i];
       fwd_{imp} \leftarrow ComputeForwardImpact(weight, M, I_{neg});
       Add tuple (weight, grad loss, fwd imp) to pool;
11
12 end
13 W_t \leftarrow ExtractParetoFront(pool);
14 return W_t
```

## 2.4 Patch Generation

Arachne uses Particle Swarm Optimisation (PSO) algorithm to generate patches that repair the misbehaviour of a DNN model. Here, we describe how PSO is configured for DNN repair.

2.4.1 Particle Swarm Optimisation (PSO). PSO is known to be particularly effective for optimisation in continuous domains [25] and, consequently, a suitable choice for repairing DNNs by modifying their neural weights. A patch generated by Arachne is essentially a set of new neural weight values to be applied to specified weights. Arachne represents this set of new weight values as a vector, which in turn represents the location of a particle in PSO.

PSO updates the location vector,  $\vec{x}$ , at timestep t+1, using a velocity vector (Equation 1). The velocity vector itself is computed based on current velocity,  $\vec{v}_t$ , the location of the best fitness value the particle has locally observed so far,  $\vec{p}_l$ , and the location of the best fitness value the entire swarm has globally observed so far,  $\vec{p}_g$ , each multiplied with uniform random bias vectors. The parameter  $\chi$  is a constant multiplier computed from weights to the local and global components,  $\phi_1$  and  $\phi_2$  (Equation 3): these parameters, collectively called "constriction coefficients", control the convergence of particles in a swarm without setting an explicit boundary of velocity [5]. We use the same value for  $\phi_1$  and  $\phi_2$ .

$$\vec{x}_{t+1} \leftarrow \vec{x}_t + \vec{v}_{t+1} \tag{1}$$

$$\vec{v}_{t+1} \leftarrow \chi(\vec{v}_t + \vec{U}(0, \phi_1)(\vec{p}_l - \vec{x}_t) + \vec{U}(0, \phi_2)(\vec{p}_g - \vec{x}_t))$$
 (2)

$$\chi \leftarrow \frac{2}{\phi - 2 + \sqrt{\phi^2 - 4\phi}}, \text{ where } \phi = \phi_1 = \phi_2 \tag{3}$$

2.4.2 Initialisation. PSO is a population based global search algorithm, and requires the location of each particle to be initialised at the beginning. While uniform random initialisation is a widely accepted method, Arachne attempts to repair an already trained DNN model and may benefit from exploiting the learnt weight values. Consequently, Arachne initialises the location of each particle based on the distribution of the learnt weights. Each element of a particle vector is sampled from a normal distribution defined by the mean and the standard deviation of all sibling neural weights. By sibling weights, we refer to all weights corresponding to connections between our target layer and the one preceding it, i.e., all weights between  $L_n$  and  $L_{n-1}$  in Figure 1. The initial velocity of each particle is set to zero, following the literature [8] as well as our own experimental results.

2.4.3 Fitness Function. PSO updates each particle based on the best known location, which is decided by a fitness function. Similarly to other G&V techniques, Arachne uses a fitness function that consists of two main parts: fixing the unexpected behaviour, and retaining correct behaviour. The fitness function is defined as follows:

$$fitness = \frac{N_{patched} + 1}{Loss(I_{neg}) + 1} + \frac{N_{intact} + 1}{Loss(I_{pos}) + 1}$$
(4)

The set  $I_{neg}$  contains inputs that reveal the unexpected behaviour, whereas the set  $I_{pos}$  contains inputs that are processed correctly: both sets are fixed by Arachne when beginning the repair attempt. Section 4.3 describes the details of how these sets are composed. On the contrary,  $N_{patched}$  is the number of inputs in  $I_{neg}$  whose output is corrected by the current patch, whereas  $N_{intact}$  is the number of inputs in  $I_{pos}$  whose output is still correct. The model loss values obtained from these input sets,  $Loss(I_{neg})$  and  $Loss(I_{pos})$ , are used to reflect improvements and damages that do not amount to changes to  $N_{patched}$  and  $N_{intact}$ . Note that we add one to both numerators and denominators so that one can provide guidance when the other becomes zero, and vice versa.

#### 3 EXPERIMENTAL PROTOCOLS

This section outlines the experimental protocols for the empirical study, and presents the research questions.

#### 3.1 Faults for DNNs

This paper uses the term *repair* to refer to the process of correcting the unexpected behaviour of a DNN, for the sake of convenience. The nomenclature of the cause of the unexpected behaviour can, however, lead to a philosophical question: is it possible to state that a DNN contains a *fault* when it misbehaves? Faults in code are committed by humans, but the misbehaviour of DNNs simply emerge and are actually anticipated. We expect this difference to have a significant impact on future work on DNN testing and repair.

The more pressing and practical issue is that it is not possible to curate, document, and create a benchmark of the causes of DNN

misbehaviour, as they are not explicitly committed. Consequently, in lieu of external fault benchmarks, the empirical evaluation of Arachne uses two classes of faults: those artificially induced by under-training, and those naturally emerging after full training. Note that the use of the term fault and faulty input is for the sake of convenience, and is not intended to answer the question about the nature of faults in DNNs.

3.1.1 Artificially Induced Faults. Code is written, while DNNs are trained. Extending this parallel, we posit that: faults are committed when code is poorly written, and emerge when DNNs are poorly trained. To create a sufficient number of unexpected behaviour of DNNs under a controlled setting, we simply under-train a DNN using a given training dataset by reducing the number of epochs spent for training. Inspired by the competent programmer hypothesis [6], the margin of under-training is set to be relatively small: we constrain the maximum accuracy of under-trained models to be under 90%. We test the feasibility of Arachne, as well as the impact of fault localisation and target input selection, using models that are more likely to contain faults, i.e., models with lower accuracy.

3.1.2 Naturally Emerging Faults. Fully trained DNNs will still show some level of unexpected behaviour. We collect and call these naturally emerging faults. There are fewer of these faults, and we expect these faults to be harder to repair without disrupting already correct behaviour. We show that Arachne can be used to repair a DNN model as new incoming inputs reveal unexpected behaviour, using naturally emerging faults.

## 3.2 Use Case Scenarios And Their Evaluations

We envision and evaluate the following two distinct use case scenarios for Arachne.

3.2.1 Corrective Repair. This scenario corresponds to repairing unexpected behaviour revealed by the training dataset at the end of training. In this case, Arachne can use an input in the training dataset to repair its outcome. We use the term *corrective* in the sense that Arachne patches a fault that occurred despite the information about the correct outcome being available in the training dataset. RQ1 to RQ4 concern the corrective repair scenario with under-trained models and artificially induced faults described in Section 3.1.1. For the evaluation of corrective repair, we only need to collect fault inducing inputs from the training dataset.

3.2.2 Adaptive Repair. Reflecting a more realistic use case, this scenario corresponds to repairing unexpected behaviour revealed by a previously unseen input, possibly after the deployment of a DNN model. We use the term adaptive in the sense that Arachne adapts the model regarding how to handle a previously unseen input. Note that our intention is not to improve the general accuracy of the model: rather, the aim of our repair is to make a trade-off between specific behaviour. RQ5 concerns this scenario, using the naturally emerging faults described in Section 3.1.2.

The evaluation of the adaptive repair scenario is as follows. We use the entire training dataset in the benchmark to fully train a DNN model (Section 4.4 describes how fully trained models are defined). After training, the test dataset is divided into two halves. From one half, we randomly select 20 inputs that reveal unexpected

behaviour (i.e., naturally emerging faults). We repair these one by one in a random order, replacing the model with the repaired version if Arachne succeeds at generating a patch. The second half of the test dataset is used as a validation set, to measure validation accuracy of each repaired model. We also report the model accuracy measured using the original training dataset. Finally, we look at the model behaviour closely to check whether the patches are actually changing the unexpected behaviour. This experimental scenario is designed to measure the accumulative impact of adaptive repairs on the model accuracy.

## 3.3 Research Questions

We investigate the following research questions to evaluate the effectiveness of Arachne. RQ1 to RQ4 evaluate Arachne on artificially induced faults using under-trained models, while RQ5 evaluates Arachne on naturally emerging faults using fully-trained models.

**RQ1. Feasibility:** Can Arachne repair unexpected behaviour of a DNN model? To show the feasibility of DNN repair via direct neural weight manipulation, we train DNN classifiers and run Arachne on randomly selected inputs from training data that are misclassified. Due to the inherently stochastic nature of PSO algorithm used by Arachne, we repeat Arachne 30 times, attempting to repair a single randomly chosen misclassified input at each attempt. We answer RQ1 by measuring classification accuracy against both training data (to ensure the repair has been done) and unseen test data (to measure the disruption to the generalised behaviour). We also report repair success rates (See Section 4.5 for details).

RQ2. Localisation Effectiveness: How effective is the localisation at identifying neural weights to patch? To evaluate the effectiveness of the proposed localisation method, we compare the performance of Arachne with and without the localisation. We use two alternative methods as the baseline: random selection of weights and selection based only on gradient loss. When using only gradient loss, we sort neural weights in the target weight variable in decreasing order of their gradient loss values over the misclassified inputs. Then, we select the same number of neural weights as the number of faulty inputs to patch for. Random selection simply chooses the same number of neural weights randomly. Both alternatives still choose neural weights only from the last layer of neurons, similar to the full localisation method used by Arachne. We report model accuracy after patching as well as success rates for each method; we expect Arachne's localisation to outperform both alternatives.

**RQ3. Multiple Faults Repair:** Can Arachne successfully repair multiple faults simultaneously? In code based APR, repairing multiple faults remains a significant challenge due to masking between faults as well as the need to automatically modify multiple locations in the code. We investigate whether a similar trend is observed when attempting to repair DNN models. We define a single fault as a specific type of misbehaviour, such as misclassifying label A to label B. To investigate the difference between single and multiple faults, we compare running Arachne using negative inputs from a single fault against running Arachne using negative inputs from a random choice of faults. For each DNN model, three single fault

types that have the most corresponding input instances in the training dataset have been chosen. We expect single fault patches to be more focused and less disruptive, as they would modify weights related to fewer features compared to multiple faults patches. Similar to RQ1 and RQ2, we report model accuracy before and after repair, as well as repair success rates (see Section 4.5 for details). In addition, we also investigate whether the localisation result is affected by the selection of faults: we expected that localisation will be more focused when Arachne targets more specific faults.

**RQ4.** Comparison to Retraining: How is Arachne different from simply retraining the model for repair? We compare Arachne to the retraining approach. We use the same set of positive and negative inputs used for RQ3 as the data for retraining. We do not use the single faulty input dataset (as in RQ1) to avoid overfitting to the single faulty input. We repeat the retraining 30 times, to match the results of Arachne in RQ3, and compare the results.

**RQ5.** Adaptive Repair: Can Arachne be used to repair a deployed DNN model incrementally as unexpected behaviour is gradually revealed? We answer RQ5 by evaluating Arachne under the adaptive repair use case scenario, described in Section 3.2.2. For each DNN model studied, we collect 20 naturally emerging faults, and attempt to repair these faults one by one consecutively, following the steps in Section 3.2.2.

## 4 EXPERIMENTAL SETUP

This section describes the details of experimental setup.

## 4.1 Subjects

We use three well-known datasets to test the effectiveness of Arachne: Fashion MNIST (FM), German Traffic Sign Recognition Benchmark (GTSRB) and CIFAR-10 (C10).

- 4.1.1 Fashion MNIST (FM). FM has been introduced to overcome the weakness of the widely studied image classification benchmark, MNIST [20]: Xiao et al. argue that FM is more challenging and more relevant to the latest machine learning algorithms, than MNIST [36]. Instead of hand-written digits, FM contains 60,000 training input images and 10,000 test input images of various types of fashion items. Each image is a grey-scale image of size (28,28), associated with one of ten labels. For this dataset, we train a neural network composed of two convolutional layers, (CONV(32), CONV(64)), and one fully connected layer (DENSE (256)), followed by a softmax layer. We use cross-entropy loss function.
- 4.1.2 German Traffic Sign Recognition Benchmark (GTSRB). The GTSRB benchmark contains 51,839 images (39,209 for training, 12,630 for test), each of which is a 48 by 48 RGB image of real-world German traffic signs labelled with 43 different classes [29]. For this dataset, we retrain a VGG16 [28] model that has been pre-trained on the ImageNet dataset [7]. The pre-trained weights of the VGG16 model have been obtained from Keras [4]. After the VGG16, we added two dense layers (DENSE(4096), DENSE(4096)), followed by the final layer for labels (DENSE(43)).
- 4.1.3 CIFAR-10 (C10). : CIFAR-10 contains 50,000 training data samples and 10,000 test data samples with 10 different classes [16].

Similar to GTSRB, we retrain a pre-trained VGG16 model on this dataset. The rest of the architecture is the same as the one used for GTSRB, except that the final layer is (DENSE(10)) for labels.

## 4.2 Algorithm Configuration

The PSO algorithm in Arachne requires  $\phi_1$  and  $\phi_2$  parameter values (see Section 2.4.1, Equation (2) and Equation (3)). We follow the general recommendation in the literature and set both to 4.1 [25]. To further ensure convergence of particles, we additionally set velocity bounds for Arachne, vb, as follows:

$$wb \leftarrow max(W) - min(W) \tag{5}$$

$$vb \leftarrow (\frac{wb}{5}, wb \times 5)$$
 (6)

W is the set of all neural weights between our target layer and the preceding one (i.e., between layer  $L_n$  and  $L_{n-1}$  in Figure 1): wb is the difference between the maximum and minimum weight values. The lower and upper velocity bounds are set as one fifth and five times of wb: factor five has been configured empirically. In addition, PSO uses a population size of 100, and the maximum number of iterations is 100. To further reduce the time spent on the search, we allow Arachne to stop earlier if it fails to find a better patch than the current best during ten consecutive iterations.

The number of neural weights to be first localised by Arachne,  $N_g$ , is set to be the number of negative inputs to repair multiplied by 20 (see Section 2.3 for details).

#### 4.3 Fitness Evaluation

Arachne requires both positive and negative input sets to compute fitness for PSO. A set of positive inputs,  $I_{pos}$ , is sampled from the correctly processed inputs in the training dataset. We sample 200 positive inputs for all research questions (RQ1 to 5). A set of negative inputs,  $I_{neg}$ , is sampled from the faulty inputs. For RQ1 and RQ2,  $I_{neg}$  contains a single faulty input from the training dataset. For RQ3 and RQ4,  $I_{neg}$  contains five faulty inputs from the training dataset, sampled either randomly or from a specific class of faults (i.e., a pair of ground truth and incorrectly predicted label). For RQ5,  $I_{neg}$  contains a single negative input from the part of the test dataset we use to emulate the real workload.

## 4.4 Model Training

Section 3.1 introduces two different classes of faults: artificially induced faults and naturally emerging faults. To collect these faults, we use under-trained and fully-trained models, as follows. For training VGG16 models on GTSRB and CIFAR-10, we use 20% of the training data as the validation set.

- Under-trained model: We manually controlled the training so that the training accuracy does not exceed 90%. The resulting under-trained model accuracy values for FM, GT-SRB, and C10 are 0.8526, 0.8749, and 0.8495, respectively.
- Fully-trained model: For fully-trained models, we reserve 20% of the training data as a validation dataset, and stop the training only when the validation accuracy decreases in five consecutive epochs. The resulting final training accuracy for fully trained models for FM, GTSRB, and C10 are 0.9981, 0.9916, and 1.0, respectively.

We also use retraining as the baseline for RQ4. Our retraining approach uses the same sets of inputs as those used in RQ3. Since these sets of inputs are much smaller than the original training datasets, it is possible that the retrained model overfits to them. To avoid overfitting, we stop the retraining once the test dataset accuracy begins to drop consecutively. Consequently, we retrain for 20, five, and five epochs for FM, GTSRB, and C10, respectively.

#### 4.5 Evaluation metric

To evaluate how much of the given faulty inputs Arachne can successfully repair, we define and measure the repair rate as in Equation (7). For n repeated runs, we report the average repair rate.

$$repair\_rate(RR) = \frac{[\text{\# of patched input from }I_{neg}]}{|I_{neg}|} \tag{7}$$
 We are also interested in how successfully Arachne retains the

We are also interested in how successfully Arachne retains the initially correct behaviour of the model under repair, represented by the inputs in  $I_{pos}$ . To this end, we define and measure the break rate as in Equation (8), and report the average.

$$break\_rate(BR) = \frac{[\# \text{ of broken input from } I_{pos}]}{|I_{pos}|} \tag{8}$$

Since Arachne is a stochastic repair technique, we report the success rate (SR) computed across multiple runs as well. A repair attempt is considered to be successful if its repair rate is greater than zero while break rate is zero. However, to ensure that the repair has not been achieved at the cost of general accuracy, we also report the model accuracy before and after the patch.

## 4.6 Implementation & Environment

Arachne is implemented in Python version 3.6; our implementation of PSO is extended from the example provided by DEAP [9]. DNN models, as well as our retraining methods, are implemented using TensorFlow version 1.12.0. Code and model data are publicly available from https://blinded. All experiments have been performed on machines equipped with Intel Core i7 CPU, 32GB RAM, and NVidia GTX1080 GPU.

#### 5 RESULTS AND ANALYSIS

This section reports and discusses the results of the empirical evaluation, and answers the research questions described in Section 3.3.

## 5.1 RQ1: Feasibility

Table 1 reports the average repair rate (RR) and the average break rate (BR) computed from 30 repeated runs of Arachne, as well as the success rate (SR) across these runs. RQ1 concerns the column LOC, which contains the results obtained using the full localisation method. Since RQ1 uses a single faulty input, the repair rate (RR) is effectively the ratio of runs in which the faulty input was patched. Arachne succeeds in 20%, 63%, and 37% of runs for FM, GTSRB, and C10, respectively. In case of FM and GTSRB, a single run resulted in a non-zero break rate. However, the overall success rate for these runs are the same as their repair rate, as the runs with non-zero BR also failed to patch the faulty input and consequently were not counted as successful runs.

Table 2 and Figure 2 reveal interesting details about patches generated by Arachne (RQ1 concerns the violinplots and the column

Table 1: Average Repair Rate (RR), Break Rate (BR), and Success Rate of Arachne using the full localisation method (LOC), gradient loss (GL), and random selection (RS)

		LOC			GL		RS RR BR SR			
Sub.	RR	BR	SR	RR	BR	SR	RR	BR	SR	
					0.0000					
GTS	0.6333	0.0002	0.63	0.3333	0.0003	0.27	0.0000	0.0000	0.00	
C10	0.3667	0.0000	0.37	0.2333	0.0002	0.20	0.0000	0.0000	0.00	

labelled LOC). Figure 2 contains the accuracy of three models, measured against training and test datasets respectively. For GTSRB and C10, the patch not only improves the training accuracy but also the test accuracy. Improvements of model accuracy can be observed more clearly in Figure 2, in which the model accuracy before patch is marked with the black dashed line. The blue horizontal lines in the violinplots denote the median model accuracy after repair. The trend suggests that the patches generated by Arachne positively affect not only the faulty input but other inputs as well. Our interpretation is that the under-training leaves room for accuracy increase, from which the patch generated by Arachne benefits.

Table 2: Model Accuracy of Arachne using full localisation (LOC), gradient loss (GL), and random selection (RS) as the localisation method

	Initial	Lo	OC .	0	GL.	F	2S
Subject		Mean	Med.	Mean	Med.	Mean	Med.
FM Train	0.8526	0.8512	0.8518	0.8525	0.8526	0.8516	0.8525
Test	0.8391	0.8379	0.8385	0.8394	0.8393	0.8382	0.8390
GTS Train	0.8749	0.8755	0.8755	0.8755	0.8751	0.8748	0.8749
Test	0.8096	0.8098	0.8100	0.8099	0.8097	0.8093	0.8096
C10 Train	0.8495	0.8503	0.8506	0.8506	0.8502	0.8495	0.8495
	0.6999	0.7003	0.7005	0.7004	0.7001	0.7001	0.6999

**Answer to RQ1:** Based on observed repair, break, and success rates, we answer RQ1 positively. Arachne successfully can generate patches that correct faulty inputs. Moreover, the patches can improve not only the training accuracy but also the test accuracy. The success rate of a repair attempt ranges from 0.2 to 0.63.

## 5.2 RQ2: Localisation Effectiveness

The results of alternative fault localisation methods are shown in Table 1 and Table 2: GL denotes using gradient loss only, and RS denotes using random selection of neural weights. In terms of repair rate and success rate, full localisation (LOC) clearly outperforms GL. However, GL produces slightly higher mean accuracy for all three models compared to LOC. We explain this with the fact that GL points to neural weights that would minimise the overall loss the most, whereas LOC also considers the direct impact each neural weight has on the model outcome for the faulty input. Consequently, LOC obtains higher RR, but can also result in lower model accuracy if patching the faulty input leads to disrupting existing correct behaviour. In comparison, RS cannot successfully repair any models, despite not breaking any correct behaviour (zero BR for all models).

Figure 2 presents violinplots of the accuracy values of the 30 repetitions of Arachne. Model accuracy of GL shows smaller variance for all three models when compared to both LOC and RS,

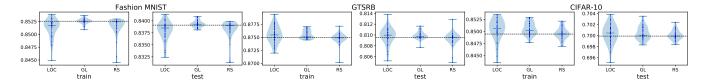


Figure 2: Model accuracy using full localisation (LOC), gradient loss (GL), and random selection (RS) as localisation method. The dashed lines denote the initial accuracy of the models. The blue horizontal lines and the dots in the violinplots indicate the median and the individual model accuracy of after repair

adding support to our explanation of GL producing the highest mean accuracy: GL prioritises neural weights that are correlated with the largest loss reduction, hence the better results in terms of overall model accuracy. However, the results also suggest that a successful patch, i.e., a focused manipulation of neural weights that correct the unexpected behaviour, may be generated at the cost of disrupting existing correct behaviour, as seen in the higher RR and SR but lower model accuracy of LOC.

Answer to RQ2: Different localisation methods have different impact on the resulting patches. In terms of changing the unexpected behaviour in question, LOC shows the highest rate of successful repair, as it identifies neural weights that have the highest impact on the relevant outcome. However, in terms of improving the overall model accuracy, GL is more effective. The results also show that localisation of neural weights is a necessary step for repairing DNN models: RS fails to produce any patches.

## 5.3 RQ3: Multiple Faults Repair

Table 3 shows the three single fault types that we use to answer RQ3. Frequent Fault 1, 2, and 3 refer to the the most, the second most, and the third most frequent misclassifications regarding the training dataset, respectively. For example, misclassifying label six (shirts) to zero (T-shirts) is the most frequent in FM, as 818 inputs that correspond to label six are classified as label zero by the model. We construct  $I_{neg}$  by randomly choosing five inputs from each type. We compare the results of Arachne against these faulty inputs to the results against five random faulty inputs. All results for RQ3 are obtained using full localisation method (LOC).

Table 3: Top 3 misclassification types in FM, GTSRB, and C10

Туре	FM	GTSRB	CIFAR-10
		26 → 18 (196)	
Freq. Fault 2	$6 \to 2 (687)$	$0 \to 1 (171)$	$2 \to 4 (517)$
Freq. Fault 3	$2 \to 4 (667)$	24 → 31 (143)	$5 \to 3 (347)$

Table 4 shows the repair, break, and success rates for single and multiple faults scenarios. It shows that Arachne performs better when trying to patch faulty inputs that are of a single type, rather than faulty inputs of random multiple types. The highest success rate is 0.83 for GTSRB. For all three models, repairing a random set of faulty inputs is more difficult, yielding lower RR and SR.

Table 5 and Figure 4 show the changes in model accuracy after fixing a single and multiple faults. There is no significant difference between single and multiple faults: Arachne performs better for single faults in some cases (e.g., Frequent Fault 1 for FM), while

Table 4: Repair, break, and success rate of Arachne on four sets of faulty inputs: sets of five faulty inputs sampled from the three most frequent faults, and a random set of faults

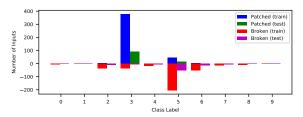
				Freq. Fault 2								
Sub.	RR	BR	SR	RR	BR	SR	RR	BR	SR	RR	BR	SR
FM	0.36	0.00	0.83	0.19	0.00	0.36	0.24	0.00	0.60	0.17	0.00	0.67
		0.00										
C10	0.49	0.00	0.80	0.44	0.00	0.80	0.34	0.00	0.57	0.18	0.00	0.67

the opposite is observed in other cases (e.g., Frequent Fault 1 for GTSRB). However, compared to results for RQ1, both mean and median model accuracy after the repair decrease in many cases. This may be partly explained by the fact that  $I_{neg}$  now contains five, not one, faulty inputs: accordingly, Arachne is forced to make more pervasive changes to relevant neural weights, which also results in larger changes in overall model accuracy.

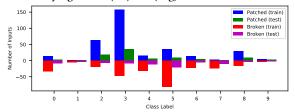
Table 5: Model accuracy after multiple faults patches

		Initial			Freq. 1	ault 2	Freq. 1	Fault 3		
Sub.			Mean	Med.	Mean	Med.	Mean	Med.	Mean	Med.
FM	Train	0.8526	0.8504	0.8514	0.8506	0.8514	0.8499	0.8510	0.8501	0.8512
FM	Test	0.8391	0.8370	0.8376	0.8379	0.8385	0.8369	0.8378	0.8364	0.8370
GTS	Train	0.8749	0.8745	0.8747	0.8742	0.8749	0.8760	0.8764	0.8757	0.8755
GIS	Test	0.8096	0.8084	0.8087	0.8073	0.8088	0.8101	0.8105	0.8091	0.8095
C10	Train	0.8495	0.8508	0.8512	0.8510	0.8514	0.8486	0.8491	0.8507	0.8508
C10	Test	0.6999	0.7007	0.7008	0.7006	0.7010	0.7000	0.6998	0.7009	0.7007

To investigate how focused patches are, we look at the changes in classification results more closely. Figure 3a shows the distribution of inputs of C10 that changed their outcomes when repaired by Arachne for Frequent Fault 1. The unexpected behaviour is that some inputs with ground truth label three are classified as label five (see Table 3). The barplots show the average number of patched (i.e., misclassified initially, corrected after the patch) and broken (correct initially, misclassified after the patch) inputs, from training and test datasets respectively. From the plot, the changes made by Arachne can be interpreted as accepting more inputs correctly as label three (high blue and green bars for label three), at the cost of misclassifying some inputs whose ground truth label is five (high red and purple bars for label five): Arachne reduces the instances of misclassifying label three as label five in the training by 27.5%, and by 21.3% in the test data. Changes to other labels are relatively minor. In contrast, Figure 3b shows the distribution of changed inputs of C10 when repaired by Arachne for five random faulty inputs. Note that, while the trend of converting label five to label three is dominant (as it is the most frequent misclassification by the



(a) Targeting the most frequent misclassification in CIFAR-10: classifying label 3 (cat) as 5 (dog)



(b) Targeting five random misclassifications

Figure 3: Average number of patched and broken inputs for single and multiple faults

model), other labels are more visibly affected by the patch, reflecting the fact that the patch targets multiple faults.

Table 6: Number of localised neural weights

Subject	Freq. Fault 1 Mean Total		Freq. 1	ault 2	Freq. 1	ault 3	Random		
FM	4.97	52	4.73 5.17	71	4.97	62	4.57	101	
GTS	4.47	72	5.17	63	4.23	68	5.37	150	
C10	5.43	86	5.17	68	4.43	79	5.60	145	

Finally, we look at the number of neural weights localised for single and multiple fault scenarios. We expect more neural weights to be chosen for patching multiple faults (note that the number of localised weights are not fixed as Arachne uses Pareto optimality to consider both gradient loss and forward impact). Table 6 show how many neural weights have been localised by Arachne for each fault: we report the average number of localised neural weights, as well as the total number of neural weights that have been chosen at least once during the 30 repeated runs.

The average number of chosen weights does not change significantly, even when repairing a set of randomly chosen faulty inputs. However, repairing random inputs does involve a wider range of neural weights, as can be seen from the significantly higher total count for randomly chosen multiple faulty inputs. The comparison between the average and the total count also reveals that certain weights are repeatedly chosen: were the localisation random, the total count would be close to 30 times of the average. The overlaps between chosen neural weights suggest that localisation is indeed effective against single fault types.

Answer to RQ3: Arachne shows success rates over 0.8 when repairing a single type of fault. Compared to patches generated to repair multiple faults, Arachne generates single fault type patches that are more focused on the target faulty behaviour while having less impact on other behaviour. The comparison between single

and multiple fault repairs also suggests that localisation method is effective at focusing on relevant neural weights.

Table 7: Comparison of Arachne and retraining with respect to repair, break, and success rate

		Fre	Freq. Fault 1 RR BR SR			q. Faul	t 2	Freq. Fault 3		
Sub.	Method	RR	BR	SR	RR	BR	SR	RR	BR	SR
FM	Arachne Retrain					0.00 0.01			0.00 0.01	0.60 0.17
GTS	Arachne Retrain				0.52 0.10		0.53 0.10		0.00 0.01	0.70 0.03
C10	Arachne Retrain		0.00 0.01		0.44 0.75	0.00 0.01	0.80 0.20		0.00 0.01	0.57 0.30

## 5.4 RQ4: Comparison to Retraining

Table 7 shows the repair, break, and success rates of Arachne and the retraining method against Frequent Fault 1, 2, and 3, respectively, while Table 8 shows the comparison of mean accuracy between models repaired by Arachne and the retraining method. Comparing two tables reveals an interesting trend. In all cases, retraining accuracy is higher than not only that of Arachne but also the accuracy of the initial model: e.g., accuracy of FM model against the training data changes from the initial 0.8526 to 0.8551 after retraining. However, the improved overall accuracy almost always comes from breaking at least one of the test inputs in  $I_{pos}$ , as can be seen in BR values of the retraining method. Due to the definition of a successful repair, the retraining method shows very low SR.

Table 8: Model Accuracy of the retraining method for repairing Frequent Fault 1, 2, 3 and the random choice of faults

Sub.		Initial	Freq. 1 Arac.	Freq. 1 Arac.	Freq. 1 Arac.	Fault 3 Retr.
FM	Train Test	0.8526 0.8391	0.8504 0.8370	0.8506 0.8379	 0.8499 0.8369	
GTS	Train Test	0.8749 0.8096	0.8745 0.8084	0.8742 0.8073	0.8760 0.8101	
C10	Train Test	0.8495 0.6999	0.8508 0.7007	0.8510 0.7006	0.8486 0.7000	

We investigate the changes in model behaviour more closely by looking at the number of patched and broken inputs per label. Figure 5 shows the distribution of the average number of patched and broken inputs after the retraining based patch is applied to Frequent Fault 1 of CIFAR-10. Compared to Figure 3a, it shows that the patch generated by retraining is much more disruptive. In addition to patching misclassified label three, retraining also alters the classification results of many other labels.

The more disruptive nature of retraining based patches generalises to unseen test inputs as well. Table 9 shows the average number of patched and broken inputs from the unseen test dataset, per label: A and R represent Arachne and the retraining method, while P and B represent Patched and Broken inputs, respectively. The per label analysis for GTSRB has been omitted as the dataset contains 43 labels. Results show the same trend observed in Figure 3a and Figure 5: retraining patches more inputs, but also breaks more inputs with labels that are not targeted for repair. Arachne, on

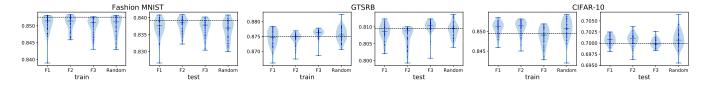


Figure 4: The violinplots of Arachne's model accuracy. F1, F2, F3, and Random denote the results of Arachne on Freq. Fault 1, 2, 3, and the random choice of faults, respectively. The dashed lines indicate the initial model accuracy of the models. The blue lines and dots in the plots denote the median and the individual model accuracy of the repaired model, respectively

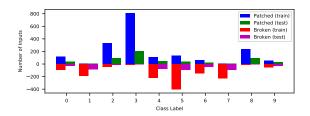


Figure 5: Average number of patched and broken inputs per label in the dataset of CIFAR-10 after retraining

the other hand, patches fewer inputs, but its damage to non-targeted labels is much smaller than that of the retraining method.

Table 9: Average number of patched and broken inputs from the test dataset on Frequent Fault 1 of FM and CIFAR-10

Sub.	M.	T.					Lab	oel					
			0	1	2	3	4	5	6	7	8	9	Total
FM	A	P B	20.7 46.5	0.8 0.7	2.8 5.7	3.2 4.8	1.1 3.0	0.0 0.1	38.1 26.9	0.0	0.3 1.0	0.0	67.0 88.8
	R	P B	2.4 35.9	5.9 3.7	4.5 39.1	10.3 6.4	29.8 8.1	23.1 0.7	57.1 4.3	6.0 7.7	0.3 6.8	1.4 9.4	140.8 122.1
C10	A	P B	0.2	0.0 1.0	0.9 10.9	89.8 5.7	1.0 6.7	13.2 50.9	0.4 14.0	0.6 4.6	0.0 1.8	0.1 1.5	106.2 97.8
-10	R	P B	36.1 30.1	3.1 79.9	89.4 14.3	200.1 3.9	41.3 71.9	33.2 92.0	22.2 40.9	1.4 89.2	89.4 1.5	32.6 28.3	549.0 452.0

Answer to RQ4: The retraining method outperforms Arachne in terms of model accuracy, but produces more disruptive patches that may alter initially correct behaviour. Arachne can produce more focused patches that correct fewer faulty inputs while preserving more of the existing behaviour.

## 5.5 RQ5: Adaptive Repair

Figure 6 shows the results of the adaptive repair scenario from the CIFAR-10 dataset  $^1$ . Figure 6a shows how the training and the validation model accuracy changes across accumulative repairs. Each datapoint correspond to a patched model. Arachne successfully patches the model 15 times out of 20 consecutive attempts. Interestingly, Arachne did not break any input in  $I_{pos}$  during the successful repair attempts, despite the model being fully trained and well fitted to the training dataset. After 15 patches are applied, the

training accuracy decreased from 1.0 to 0.989, while the validation accuracy decreased from 0.723 to 0.7.

There are patches that affect the validation accuracy more notably than the others. The patch from the 10th repair increases the validation accuracy by 0.0028, while the patch from the 18th repair decreases the validation accuracy by 0.0064. Figure 6b and Figure 6c show the per label analysis for these two patches. Patch 10 fixes the misclassification of label (horse) seven as one (automobile): the patched version correctly classifies over 30 unseen inputs from the validation set, while breaking much fewer, resulting in the increased validation accuracy. Note that the better classification of horses comes at the cost of misclassification of deers (label four) though. On the other hand, patch 18 fixes the misclassification of label four (deer) as two (bird), but the patch disrupts many other labels, resulting in lower accuracy. In both cases, however, we observe that both the correction and disruption caused by patches tend to generalise to unseen validation data.

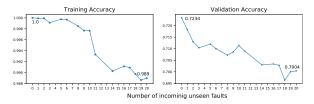
**Answer to RQ5:** The results of the evaluation of the adaptive repair scenario suggest that Arachne can produce patches that generalise to unseen data even against fully trained models. Accumulative patches lead to gradual decrease of accuracy rather than dramatic loss of performance.

#### 6 DISCUSSION AND FUTURE WORK

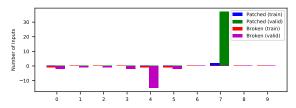
This section discusses some implications of Arachne and lays out related future work.

Do models learn something new by being patched? Arachne attempts to repair pre-trained DNN models without requiring any additional data. The lack of information need, as well as the results we have observed, mean that Arachne patches a DNN model by shifting its internal priority, rather than via learning. Consequently, Arachne is not appropriate for improving the general accuracy of any given DNN model. However, our motivation is to investigate the feasibility of an APR tool for DNN that can create trade-offs between different model behaviour, at least temporarily until proper (re)training is possible with a sufficiently large and diverse dataset. We must properly retrain our models instead of patching. To stress again, our aim is not to replace proper training or more advanced learning algorithms. However, as DNNs are adopted increasingly widely, we think the need to hot-patch a DNN model will rise. Since deep learning is a stochastic process, it is possible that, at the end of training, an end user may prefer higher accuracy for a specific set of behaviour than others. It is also possible that an end user may want to change specific behaviour, without the cost of curating more training data.

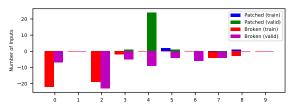
 $<sup>^1\</sup>mathrm{Results}$  for FM and GTSRB, as well as the full results for CIFAR-10, are available online at https://blinded.



(a) Changes of training and validation accuracy across 20 adaptive repair attempts against a fully trained C10 classifier



(b) Number of patched and broken inputs after the adaptive patch number 10 for C10



(c) Number of patched and broken inputs after the adaptive patch number 18 for C10

Figure 6: Adaptive repair of fully trained CIFAR-10 classifier

The retraining method used in the paper is too simplistic. We could not adopt any retraining method that would require additional training data (e.g., MODE [23]) as a fair baseline, since Arachne is designed specifically not to require additional training data. Consequently, the retraining method used in the paper is chosen to show what is possible using only the available training data. A more thorough comparison to a wider range of retraining techniques remains as future work.

Targeting only the last layer appears too constricted. Arachne aims to retain as much existing correct behaviour as possible, which means that any patch it generates has to be as minimal as possible. We posit that manipulating layers that are close to the input is likely to have wider impact on the overall outcomes of the model, because the shallower a layer is, the wider its output values will be used by the remainder of the model. Consequently, we focused on the last layer. However, a more effective localisation technique may be able to identify a mixture of shallow and deep neurons that are collectively suitable for finding a repair. Designing such a localisation technique remains future work.

#### 7 RELATED WORK

While there is a vast body of literature on techniques to improve the learning capability of DNNs [19], research on how to validate the performance and the quality of DNN models is relatively young. Existing literature focus on ways to reveal unexpected behaviour by using the combination of input synthesis and metamorphic oracles [12, 24]. A number of test adequacy criteria have also been introduced and studied [15, 22, 24, 32].

Debugging, or patching, a DNN model has so far been formulated in the context of (re)training, i.e., continuing to learn until the correct behaviour are trained. Ma et al. proposed MODE [23], which uses Generative Adversarial Networks (GANs) to synthesise additional inputs that focus on the features relevant to the unexpected behaviour. These new inputs are used to retrain the DNN model under repair. Arachne, on the other hand, formulates the same problem as a direct Automated Program Repair, and manipulates a DNN without requiring additional data.

Arachne is heavily inspired by a class of APR techniques called Generate and Validate (G&V) [10, 18, 34, 37]. Concepts such as the use of positive and negative input sets, the use of fault localisation, and the use of metaheuristic search as the main driver of the repair are all inherited from existing G&V techniques. Like other G&V techniques, Arachne first *generates* a candidate patch and *validates* the patch by applying it and subsequently executing the patched model against the set of positive and negative inputs. However, due to the nature of DNNs, some fundamental differences exist. Both the program and the patch representation is numerical and continuous for Arachne. Unlike the fitness function of GenProg [10, 18] that only counts discrete step changes in the number of test cases that fail, Arachne can use the model loss as a guidance even when no input changes any model behaviour.

#### 8 CONCLUSION

We present Arachne, an APR technique for repairing unexpected behaviour of DNN models. Arachne uses Particle Swarm Optimisation to directly manipulate neural weight values, which are chosen by a specially designed localisation heuristic. An empirical evaluation using three widely studied DNN benchmark datasets suggests that Arachne can repair unexpected behaviour of DNN models while minimally disrupting existing correct behaviour. Future work will consider more sophisticated search algorithms and localisation heuristics, as well as comparison to various retraining methods.

## REFERENCES

- [1] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2019). IEEE Press, 291–300.
- [2] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. 2015. Deepdriving: Learning affordance for direct perception in autonomous driving. In Proceedings of the IEEE International Conference on Computer Vision. 2722–2730.
- [3] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. 2017. Multi-view 3d object detection network for autonomous driving. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 1907–1915.
- [4] François Chollet et al. 2015. Keras. https://github.com/fchollet/keras.
- [5] M. Clerc and J. Kennedy. 2002. The particle swarm explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation* 6, 1 (Feb 2002), 58–73.
- [6] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on test data selection: Help for the practical programmer. *IEEE Computer* 11 (1978), 31–41.
- [7] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In 2009 IEEE Conference on Computer Vision and Pattern Recognition. 248–255. https://doi.org/10.1109/CVPR.2009.5206848
- [8] A. Engelbrecht. 2012. Particle swarm optimization: Velocity initialization. In 2012 IEEE Congress on Evolutionary Computation. 1–8. https://doi.org/10.1109/CEC. 2012.6256112

- [9] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. Journal of Machine Learning Research 13 (July 2012), 2171–2175.
- [10] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 bugs for \$8 Each. In Proceedings of the 34th International Conference on Software Engineering, 3–13.
- [11] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. 2012. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. IEEE Signal Processing Magazine 29, 6 (Nov 2012), 82–97. https://doi.org/10.1109/MSP.2012.2205597
- [12] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In Computer Aided Verification, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 3–29.
- [13] Sébastien Jean, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. 2015. On Using Very Large Target Vocabulary for Neural Machine Translation. In Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). Association for Computational Linguistics, Beijing, China, 1–10. https://doi.org/10.3115/v1/P15-1001
- [14] J. Kennedy and R. Eberhart. 1995. Particle swarm optimization. In Proceedings of ICNN'95 - International Conference on Neural Networks, Vol. 4. 1942–1948 vol.4. https://doi.org/10.1109/ICNN.1995.488968
- [15] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding Deep Learning System Testing using Surprise Adequacy. In Proceedings of the 41th International Conference on Software Engineering (ICSE 2019). IEEE Press, 1039–1049. https://doi.org/10.1109/ICSE.2019.00108
- [16] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. Technical Report. University of Toronto.
- [17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. Commun. ACM 60, 6 (May 2017), 84–90. https://doi.org/10.1145/3065386
- [18] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. IEEE Transactions on Software Engineering 38, 1 (Jan 2012), 54–72. https://doi.org/10.1109/TSE.2011.104
- [19] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. Nature 521, 7553 (2015), 436.
- [20] Yann LeCun, Corinna Cortes, and CJ Burges. 2010. MNIST handwritten digit database. AT&T Labs [Online]. Available: http://yann.lecun.com/exdb/mnist 2 (2010).
- [21] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen A.W.M. van der Laak, Bram van Ginneken, and Clara I. Sánchez. 2017. A survey on deep learning in medical image analysis. Medical Image Analysis 42 (2017), 60 – 88. https://doi.org/10. 1016/j.media.2017.07.005
- [22] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: Multi-granularity Testing Criteria for Deep Learning Systems. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018). ACM, New York, NY, USA, 120–131.
- [23] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: Automated Neural Network Model Debugging via State Differential Analysis and Input Selection. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018). ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/3236024.3236082
- [24] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17). ACM, New York, NY, USA, 1–18. https://doi.org/10.1145/3132747.3132785
- [25] Riccardo Poli, James Kennedy, and Tim Blackwell. 2007. Particle swarm optimization. Swarm intelligence 1, 1 (2007), 33–57.
- [26] Yuji Roh, Geon Heo, and Steven Euijong Whang. 2018. A Survey on Data Collection for Machine Learning: a Big Data AI Integration Perspective. CoRR abs/1811.03402 (2018). arXiv:1811.03402 http://arxiv.org/abs/1811.03402
- [27] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing Evolution for Multi-hunk Program Repair. In Proceedings of the 41st International Conference on Software Engineering (ICSE '19). IEEE Press, Piscataway, NJ, USA, 13–24.
- [28] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. CoRR abs/1409.1556 (2014). http://arxiv.org/abs/1409.1556
- [29] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. 2012. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks* 0 (2012), -. https://doi.org/10.1016/j.neunet.2012.02.016
- [30] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In Proceedings of the 27th International Conference

- on Neural Information Processing Systems (NIPS'14). MIT Press, Cambridge, MA, USA, 3104–3112.
- [31] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition. 1–9.
- [32] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-neural-network-driven Autonomous Cars. In Proceedings of the 40th International Conference on Software Engineering (ICSE '18). ACM, New York, NY, USA, 303–314. https://doi.org/10.1145/3180155.3180220
- [33] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE '09). IEEE, Vancouver, Canada, 364–374.
- [34] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware Patch Generation for Better Automated Program Repair. In Proceedings of the 40th International Conference on Software Engineering (ICSE '18). ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233
- [35] Andreas Windisch, Stefan Wappler, and Joachim Wegener. 2007. Applying Particle Swarm Optimization to Software Testing. In Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO '07). ACM, New York, NY, USA, 1121–1128. https://doi.org/10.1145/1276958.1277178
- [36] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. arXiv:cs.LG/cs.LG/1708.07747
- [37] Y. Yuan and W. Banzhaf. 2018. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. IEEE Transactions on Software Engineering (2018), 1–1.
- [38] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018). ACM, New York, NY, USA, 132–142. https://doi.org/10.1145/3238147.3238187