AutoDA: Automated Decision-based Iterative Adversarial Attacks

Qi-An Fu, *Dept. of Comp. Sci. and Tech., Institute for AI, Tsinghua-Bosch Joint ML Center, THBI Lab, BNRist Center, Tsinghua University, Beijing, China;* Yinpeng Dong, *Dept. of Comp. Sci. and Tech., Institute for AI, Tsinghua-Bosch Joint ML Center, THBI Lab, BNRist Center, Tsinghua University, Beijing, China; RealAI;* Hang Su, *Dept. of Comp. Sci. and Tech., Institute for AI, Tsinghua-Bosch Joint ML Center, THBI Lab, BNRist Center, Tsinghua University, Beijing, China; Peng Cheng Laboratory; Tsinghua University-China Mobile Communications Group Co., Ltd. Joint Institute;* Jun Zhu, *Dept. of Comp. Sci. and Tech., Institute for AI, Tsinghua-Bosch Joint ML Center, THBI Lab, BNRist Center, Tsinghua University, Beijing, China; RealAI; Peng Cheng Laboratory; Tsinghua University-China Mobile Communications Group Co., Ltd. Joint Institute;* Chao Zhang, *Institute for Network Science and Cyberspace / BNRist, Tsinghua University*

https://www.usenix.org/conference/usenixsecurity22/presentation/fu-qi

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

AutoDA: Automated Decision-based Iterative Adversarial Attacks

Qi-An Fu[1], Yinpeng Dong[1,2], Hang Su[1,3,4], Jun Zhu[1,2,3,4]*, Chao Zhang[5]
[1] *Dept. of Comp. Sci. and Tech., Institute for AI, Tsinghua-Bosch Joint ML Center, THBI Lab, BNRist Center, Tsinghua University, Beijing, China;* [2] *RealAI;* [3] *Peng Cheng Laboratory*
[4] *Tsinghua University-China Mobile Communications Group Co., Ltd. Joint Institute*
[5] *Institute for Network Science and Cyberspace / BNRist, Tsinghua University*
fqa19@mails.tsinghua.edu.cn, {dongyinpeng, suhangss, dcszj, chaoz}@tsinghua.edu.cn

Abstract

Adversarial attacks can fool deep learning models by imposing imperceptible perturbations onto natural examples, which have provoked concerns in various security-sensitive applications. Among them, decision-based black-box attacks are practical yet more challenging, where the adversary can only acquire the final classification labels by querying the target model without access to the model's details. Under this setting, existing works usually rely on heuristics and exhibit unsatisfactory performance in terms of query efficiency and attack success rate. To better understand the rationality of these heuristics and further improve over existing methods, we propose AutoDA to automatically discover decision-based iterative adversarial attack algorithms. In our approach, we construct a generic search space of attack algorithms and develop an efficient search algorithm to explore this space. Although we adopt a small and fast model to efficiently evaluate and discover qualified attack algorithms during the search, extensive experiments demonstrate that the discovered algorithms are simple yet query-efficient when attacking larger models on the CIFAR-10 and ImageNet datasets. They achieve comparable performance with the human-designed state-of-the-art decision-based iterative attack methods consistently.

1 Introduction

Deep learning models have obtained impressive performance improvements on various pattern recognition tasks [1–4]. However, these models are vulnerable to adversarial examples [5, 6], which are maliciously crafted by adding small adversarial perturbations to natural examples but can fool the target model to make incorrect predictions. The existence of adversarial examples to deep learning models raises potential risks in security-sensitive applications that adopt deep learning techniques, such as face recognition [7], autonomous driving [8], and medical image analysis [9]. Thus, the study of adversarial robustness has attracted increasing attention. A number of adversarial attack methods have been developed [6, 10–12] to generate adversarial perturbations under various threat models, which help to identify the vulnerabilities and serve as an indispensable surrogate to evaluate adversarial robustness [13, 14].

Along with adversarial attacks, there is also a rapid development of defense methods. To understand their effects and identify the real progress of the field, it is of great importance to evaluate the existing defense methods correctly and reliably [14–16]. It sometimes needs carefully designed adaptive attacks to evaluate the worst-case robustness of a particular defense, e.g., gradient obfuscation [17, 18]. Those attack methods were typically designed by experts case by case, which often requires a considerable amount of manual trial-and-error efforts. One may hope to automatically discover attack methods to ease this burden, which can not only serve as a reasonable baseline for measuring the strength of expert-designed attack methods but also examine the rationality of heuristics or assumptions used by them.

The desire of automated attacks becomes even more urgent in the practical yet challenging setting of decision-based black-box attack. Although various decision-based attack methods have been proposed [7, 10, 12, 19–24], many of them are based on heuristics and exhibit unsatisfactory performance, as compared to gradient-based white-box attack methods which could be optimal in some sense [25]. Automating the process of discovering decision-based attacks can help to understand the rationality of these heuristics, and to discover new methods.

The problem of automatically discovering decision-based attack algorithms falls into the general direction of program synthesis, which aims to automatically discover a program satisfying a user intent specification [26]. Many generic methodologies and techniques have been developed for program synthesis, with the majority focusing on software problems [27–29]. On the other side, the task of automating the process of solving machine learning problems is known as automated machine learning (AutoML) [30]. One most attractive direction of AutoML is neural architecture search (NAS),

*Jun Zhu is the corresponding author.

which aims to automatically discover good architectures of deep networks [31]. Unlike existing works that often start with expert designed layers, a recent work AutoML-Zero [32] moves one step further and shows promise to search for a complete classification algorithm (e.g., two-layer neural networks) from scratch with minimal human participation. However, the discovered classification algorithms are still far behind the current practice. To the best of our knowledge, none of existing program synthesis or AutoML solutions could automatically discover decision-based black-box attack algorithms yet.

In this work, we propose to solve the practical yet challenging problem of decision-based adversarial attack by automatically searching for qualified attack algorithms. We call our approach as **Auto**mated **D**ecision-based **A**ttacks (**AutoDA**). Technically, inspired by program synthesis works, we choose an algorithm template [33] first proposed in the Boundary attack [12] to alleviate the difficulty of the search problem, and apply pruning techniques based on constraints imposed by the algorithm template and the adversarial attack problem to further reduce the search space.

Specifically, for the missing components in the algorithm template, we design a search space constructed from basic mathematical operations, which provides sufficient expressiveness for the decision-based adversarial attack problem with affordable complexity. The way we construct the search space is similar to the aforementioned solution of AutoML-Zero [32]. However, due to the theoretical and practical differences between our problem and AutoML-Zero's, AutoDA settles on quite different design choices and implementations. For instance, we use the static single assignment (SSA) form instead of the three-address code (TAC) form (used in AutoML-Zero) to define the search space for better sample efficiency and computational performance in our use case of generating random programs, as detailed in Section 3.3.

Furthermore, we propose several mechanisms to efficiently explore this search space. First, we develop a simple random search algorithm that can be executed distributively. By combining with several pruning techniques and intuitive priors, this search algorithm shows great efficiency. Second, we utilize a small and fast model for evaluating and filtering attack algorithms during the search to further reduce computational cost. Although our search space is inspired by and based on the Boundary attack, it turns out that the discovered algorithms are simpler and can perform much better than the Boundary attack. They are query-efficient when used to attack larger models, and can be geometrically interpreted, which illustrates the rationality of some heuristics in existing works. In this work, the search space is relatively restricted in a sense that it is not able to express more recent state-of-the-art expert-designed attacks [22, 23]. Our discovered algorithms consistently demonstrate comparable performance with the state-of-the-art baselines, suggesting these expert-designed attacks are near-optimal given this restricted search space.

In summary, we make the following contributions:

- We present AutoDA, the first solution to automatically discover decision-based iterative adversarial attacks.
- We propose a way to construct a search space of decision-based iterative attack algorithms, providing sufficient expressiveness and affordable complexity.
- We propose an effective random search algorithm, together with several pruning techniques and intuitive priors, to efficiently explore the search space.
- We implement a prototype of AutoDA and run comprehensive experiments to evaluate its effectiveness. The discovered algorithms are simple and geometrically interpretable; and they show comparable performance than the state-of-the-art expert-designed decision-based iterative attacks on various models and datasets consistently, suggesting these expert-designed attacks are near-optimal in our search space.

2 Background

Recent advances in deep learning techniques have achieved impressive results on a number of pattern recognition tasks, including image classification [34], face recognition [35], and speech recognition [4, 36]. Thus deep learning models have been integrated into various security-critical applications, e.g., autonomous driving, healthcare, and finance.

However, deep learning classifiers are vulnerable to adversarial examples, which brings potential risks to these security-critical applications. As a result, the study of adversarial attack and defense methods has attracted increasing attention. To measure the quality of an adversarial example, researchers usually use the ℓ_2 or ℓ_∞ norm distance between the natural example and the adversarial example — a smaller distance means a better adversarial example. To generate good adversarial examples and defend against adversarial examples under different threat models, an enormous number of adversarial attack and defense methods have been proposed.

Threat models. Adversarial attack and defense methods are usually designed for some specific threat models. Therefore, clearly defining threat models is essential to understand these methods [15]. According to the adversary's goal, we have the *untargeted* and *targeted* threat models. An *untargeted* adversary aims to cause misclassification of the target model, while a *targeted* adversary goes one step further and aims to cause misclassification as an adversary-desired class. According to the attacker's knowledge of the target model, attacks can be categorized into the *white-box* [6, 13, 25]. and *black-box* threat models [11, 12, 12].

Defense methods. Due to the security threat brought by adversarial examples, various defense methods have been proposed to defend against adversarial attacks [25]. However, many of them cause obfuscated gradients and can be defeated by adaptive attacks [17]. Currently, the most effective defense methods are based on adversarial training [14, 25], which adds adversarial examples to the training process. For instance, the

PGD-based adversarial training [25] improves the robustness of classifiers by adding adversarial examples generated by the PGD attack to the training set. The downside of adversarial training is reducing the model's accuracy on natural examples [14].

Decision-based black-box attacks. Black-box attacks can be further divided into score-based and decision-based ones. In score-based attacks, the adversary has access to the full probability (or score) output for each class [11, 37–39]. In decision-based attacks, the adversary has only the hard-label output (i.e., the final prediction label), and thus has more challenges to generate adversarial examples. Jacobian-based attacks [10, 19] are early work in this field. Later decision-based attacks [7, 21] adopt the random walk framework first proposed in the Boundary attack [12]. Recent iterative attacks might combine with other techniques, e.g., zeroth-order optimization [22, 23]. In this work, we focus on the challenging but practical decision-based black-box attack problem.

3 Design

In this section, we explain the design of AutoDA in detail. For simplicity, we particularly focus on untargeted attacks in this work, where the adversary aims to cause misclassification of the victim classifier. Nevertheless, our approach can be extended to targeted attacks straightforwardly.

3.1 Intuition

Discovering an algorithm containing control flow that satisfies an intent specification is undecidable in general [26], thus it is extremely hard. After investigating the implementations of the existing Boundary attack [12] and Evolutionary attack [7], we found that they share a quite similar control flow, while their main difference lies in a loop-free code segment. This loop-free code segment in both attacks uses only a few dozens of basic scalar and vector mathematical operations. This observation suggests the possibility of fixing the control flow of the algorithm (i.e., in a unified framework) and only searching for the loop-free code segment.

3.2 Overview

One possible approach to reducing the difficulty of searching for algorithms is providing a template for the algorithm, which reduces the problem down to searching for the missing components in the template [33]. Inspired by this approach, we choose the random walk framework for decision-based iterative attacks under the ℓ_2 norm as our algorithm template. This framework is first proposed by the Boundary attack [12] and used by many later decision-based attacks [7, 21].

As outlined in Alg. 1, the random walk process starts at an adversarial starting point $\mathbf{x}_1$, which could be obtained by keeping adding different large random noises to the original example $\mathbf{x}_0$ until finding one that causes misclassification [12]. In each iteration of the random walk, the adversary executes the `generate()` function to generate the next random point $\mathbf{x}'$ to walk to based on the original example $\mathbf{x}_0$ and the best adversarial example $\mathbf{x}$ already found. $\mathbf{x}'$ is usually generated by applying transformations to a Gaussian noise. If $\mathbf{x}'$ is adversarial and is closer to $\mathbf{x}_0$ than the old adversarial example $\mathbf{x}$, we update the adversarial example $\mathbf{x}$ to $\mathbf{x}'$ since we found a better adversarial example with a smaller perturbation. Existing random walk based attacks *guarantee* the inequality $\|\mathbf{x}' - \mathbf{x}_0\|_2 < d_{\min}$ to hold by properly designing the `generate()` function. There are also some hyperparameters inside the `generate()` function controlling the step size of the random walk process. After each iteration, the framework collects the success rate of whether $\mathbf{x}'$ is adversarial and adjusts the hyperparameters according to the success rate of several past trials.

Algorithm 1 Random walk framework for decision-based iterative attacks under the ℓ_2 norm.

Data: original example $\mathbf{x}_0$, adversarial starting point $\mathbf{x}_1$;
Output: adversarial example $\mathbf{x}$ such that the ℓ_2 distortion $\|\mathbf{x} - \mathbf{x}_0\|_2$ is minimized;
Initialization: $\mathbf{x} \leftarrow \mathbf{x}_1$; $d_{\min} \leftarrow \|\mathbf{x} - \mathbf{x}_0\|_2$;
while query budget is not reached **do**
  $\mathbf{x}' \leftarrow$ `generate`$(\mathbf{x}, \mathbf{x}_0)$;
  if $\mathbf{x}'$ is adversarial **and** $\|\mathbf{x}' - \mathbf{x}_0\|_2 < d_{\min}$ **then**
    $\mathbf{x} \leftarrow \mathbf{x}'$; $d_{\min} \leftarrow \|\mathbf{x} - \mathbf{x}_0\|_2$;
  end if
  Update the success rate of whether $\mathbf{x}'$ is adversarial;
  Adjust hyperparameters according to the success rate;
end while

There are two missing components in this template — the `generate()` function and the hyperparameter adjustment strategy. The main difference between existing attacks lies in their `generate()` functions, while they all use similar negative feedback hyperparameter adjustment strategies. Without loss of generality, we only search for the `generate()` function to make our problem easier, and settle on a predefined negative feedback strategy for adjusting hyperparameters similar to existing works, as detailed in Appendix A.

To solve our problem, we adopt the following generic methodology: define a search space for the `generate()` function, design a search method, and search for programs with top performance under some designed program evaluation metrics. Before diving into the details, we provide an overview of AutoDA first: (1) We choose a generic search space constructed from basic scalar and vector mathematical operations which provides sufficient expressiveness for our problem (Section 3.3); (2) We use random search combined with several pruning techniques and intuitive priors to efficiently explore

Table 1: List of available operations in the AutoDA DSL. The suffix of each operation's notation indicates its parameters' type, where `S` denotes scalar type, and `V` denotes vector type. For example, the `VS` suffix means the operation's first parameter is a vector and second parameter is a scalar. In the parameter(s) column, a or $\mathbf{a}$ stands for the first parameter, a for scalar and $\mathbf{a}$ for vector; b or $\mathbf{b}$ stands for the second parameter, b for scalar and $\mathbf{b}$ for vector. In the output column, r for scalar output, and $\mathbf{r}$ for vector output. In the mathematical expression column, the subscript $\cdot_i$ on vector variable means the i-th component of the vector, $\forall i$ means for all dimension of the vector.

ID	Notation	Description	Parameter(s)	Output	Mathematical Expression
1	`ADD.SS`	scalar-scalar addition	a, b	r	$r = a + b$
2	`SUB.SS`	scalar-scalar subtraction	a, b	r	$r = a - b$
3	`MUL.SS`	scalar-scalar multiplication	a, b	r	$r = ab$
4	`DIV.SS`	scalar-scalar division	a, b	r	$r = a/b$
5	`ADD.VV`	vector-vector element-wise addition	$\mathbf{a}, \mathbf{b}$	$\mathbf{r}$	$\mathbf{r} = \mathbf{a} + \mathbf{b}$
6	`SUB.VV`	vector-vector element-wise subtraction	$\mathbf{a}, \mathbf{b}$	$\mathbf{r}$	$\mathbf{r} = \mathbf{a} - \mathbf{b}$
7	`MUL.VS`	vector-scalar broadcast multiplication	$\mathbf{a}, b$	$\mathbf{r}$	$r_i = a_i b, \forall i$
8	`DIV.VS`	vector-scalar broadcast division	$\mathbf{a}, b$	$\mathbf{r}$	$r_i = a_i / b, \forall i$
9	`DOT.VV`	vector-vector dot product	$\mathbf{a}, \mathbf{b}$	r	$r = \mathbf{a} \cdot \mathbf{b}$
10	`NORM.V`	vector ℓ_2 norm	$\mathbf{a}$	r	$r = \Vert\mathbf{a}\Vert_2$

the search space (Section 3.4); (3) We evaluate programs with a small and fast model on a small number of samples to reduce computational cost (Section 3.5). We will elaborate important design choices of AutoDA in the rest of this section.

3.3 Search Space

Designing a search space is the art of trading off between expressiveness and complexity [26]. On one hand, the search space should be expressive enough to include useful programs for the target problem. On the other hand, great expressiveness does not come for free — it usually leads to high complexity. Searching over a complex space is both time-consuming and hard to implement. Instead of using a full-featured programming language like Python which provides more-than-needed expressiveness with high complexity, we choose to design a domain specific language (DSL) specialized for our problem that provides sufficient expressiveness with relative low complexity.

We list all the available operations in our AutoDA DSL in Table 1. These operations are basic mathematical operations for scalars and vectors, and all vector operations have geometric meaning in the Euclidean space. Then we construct our search space for the `generate()` function as all valid static-single-assignment (SSA) form programs in this DSL with a given length and a given number of hyperparameters. Each program is just a sequence of assignment statements. Each of these statements applies one of the available operations to one or two variables (depending on whether the operation accepts one or two parameter(s)), and assigns the output of this operation to a variable. In our use case of generating random programs, we choose the SSA form widely used in modern compilers [40] instead of the three-address code (TAC) form used in the AutoML-Zero [32] for better sample efficiency and computational performance. Although the SSA and TAC forms are equivalent in the sense that they can be converted to each other, when generating random programs in the SSA form, we can enforce many desired properties of these programs explicitly and straightforwardly, e.g., limiting the number of hyperparameters and avoiding unused inputs and statements. In contrast, for the TAC form, we usually need to generate programs first, then check their properties and reject the failed ones. Moreover, checking a TAC form program requires almost as much work as converting it into an equivalent SSA form program. Consequently, this generate-then-check process hurts sample efficiency and computational performance.

It is worth noting that the AutoDA DSL only requires all vector variables to have the same dimension but does not restrict them to a specific dimension. This property of our DSL preserves the possibility for transferring the discovered programs to other datasets with different image shapes without modification, though it may need extra tuning on the initial values of hyperparameters after changing the input dimension.

We design the program to accept three parameters: $\mathbf{x}$ and $\mathbf{x}_0$ as in the `generate(`$\mathbf{x}, \mathbf{x}_0$`)` function from Alg. 1, as well as a random noise $\mathbf{n}$ sampled from the standard Gaussian distribution $\mathcal{N}(\mathbf{0}, \mathrm{I})$. Instead of providing operations for generating random noise in the AutoDA DSL, we provide the random noise as a parameter $\mathbf{n}$, thus the program itself would be pure and more handy to do property testing efficiently.

The above designed AutoDA DSL should have sufficient expressiveness for the decision-based adversarial attack problem under the ℓ_2 norm. For example, we can implement the Boundary attack's `generate()` function with our AutoDA DSL. We provide one possible implementation of it in Ap-

pendix C. On the other hand, the AutoDA DSL does not have high complexity since it has no control flow with only ten unary and binary operations. However, this search space is still huge, because its size grows at least exponentially with the length of the program. As a result, we need to design and implement an efficient search method.

3.4 Search Method

Searching for programs is a combinatorial optimization problem, because the search space is discrete and finite when ignoring the initial values of hyperparameters. In this work, we develop a random search based method combined with several pruning techniques and intuitive priors. We choose random search due to several reasons. First, from a theoretical perspective, the no free lunch theorems for optimization [41] imply that random search is on average a reasonable baseline method for combinatorial optimization. For example, random search based methods are shown to be competitive baselines in NAS [42]. Second, from a practical perspective, random search is much simpler to implement efficiently and correctly than other methods, e.g., evolutionary search, but it is surprisingly effective [29] when combined with other techniques, e.g., pruning techniques. Finally, random search can run in parallel by its nature, which helps us easily distribute tasks to multiple machines. For the hyperparameters, since the framework would adjust them automatically during the random walk process, we initialize them as a given fixed value to reduce implementation complexity and computational cost.

Unlike NAS works, where the search spaces are usually constructed from expert-designed layers such that good architectures are dense in them, AutoDA's search space is constructed from a more generic DSL such that good programs should be quite sparse. Naive random search would waste most computation on meaningless programs. We mitigate this issue by introducing four techniques specialized for the decision-based attack problem from two aspects — the random program generating process and the search process. We will conduct an ablation study on these four techniques to show their effectiveness in Section 5.4.

For the random program generating process, we apply two intuitive priors (or biases) to improve the quality of the generated programs: (1) *Compact program*: We use a program generating algorithm that prefers to generate programs with less unused statements[1]. It is noted that this algorithm should still generate programs with many unused statements, but with a lower probability, so that the search space size is not reduced. We describe the detailed random program generating algorithm used by this work in Section 4.1, which is one possible way to implement this prior. (2) *Predefined statements*: We add three predefined statements $\mathbf{v} = \mathbf{x}_0 - \mathbf{x}$ ($\mathbf{v}$ = `SUB.VV(`$\mathbf{x}_0$`,`$\mathbf{x}$`)`), $d = \|\mathbf{v}\|_2$ (d = `NORM.V(`$\mathbf{v}$`)`), and $\mathbf{u} = \mathbf{v}/d$ ($\mathbf{u}$ = `DIV.VS(`$\mathbf{v}$`,`d`)`) to the program before randomly generating the remaining statements. These predefined statements are common for decision-based attacks under the ℓ_2 norm, because the program needs to minimize the distance between $\mathbf{x}_0$ and $\mathbf{x}$. Thus the distance d and the angle $\mathbf{u}$ between $\mathbf{x}$ and $\mathbf{x}_0$ should be useful. These statements all appear at the very beginning of many existing methods, including the Boundary attack [12], the Evolutionary attack [7], and the state-of-the-art Sign-OPT attack [22]. Again, programs left these predefined statements unused could still be generated, but with a lower probability. Without reducing the size of our search space, both techniques just add priors to the generating process and increase the probability of generating better programs.

For the search process, we apply two pruning techniques to filter out trivially meaningless programs based on constraints imposed by the decision-based attack problem and the random walk algorithm template, including: (1) *Inputs check*: We filter out programs that do not make use of all inputs, because they would be meaningless for the decision-based attack problem. This property can be checked formally. (2) *Distance test*: We filter out programs that generate $\mathbf{x}'$ violating the inequality $\|\mathbf{x}' - \mathbf{x}_0\|_2 < d_{\text{min}}$ required by the framework in Alg. 1. However, formally checking this property is extremely hard. We informally test this property instead, as detailed in Section 4.1.

3.5 Program Evaluation Method

The last step is to define evaluation metrics for programs in the search space such that we can distinguish good programs from bad ones. When evaluating the performance of decision-based attacks, we usually run them against many large deep models to generate adversarial example for each sample in the test set. However, as running large models and attacking all samples in the test set are computationally expensive, this kind of evaluation is time-consuming and impractical for our problem with a huge search space.

To address this issue, we leverage two strategies to make the evaluation fast and cheap. First, we consider the CIFAR-10 dataset [43] and adopt a modified version of EfficientNet[2] [44] for program evaluation during the search process. CIFAR-10 is an image dataset of shape $32 \times 32 \times 3$ widely used in existing adversarial robustness works. We choose it mainly for its low dimension, such that better computational performance and smaller memory footprint can be achieved. Since effective attacks should be effective against both small and large models, attacks discovered using small models should (hopefully) be able to transfer to larger models. However, this further shrunk version of EfficientNet-B0 we used is too small

[1] When a variable is assigned but unused in the return value of the program, we call it "unused variable", and call the corresponding assignment statement "unused statement".

[2] EfficientNets are small and fast deep models that achieve high accuracy on various benchmarks. We shrunk the EfficientNet-B0, the smallest variant of EfficientNets, by a factor of 0.5 to make it run even faster.

to achieve reasonable accuracy on the complete CIFAR-10 dataset, while for some pair of labels in CIFAR-10 it can. So we train a binary classifier on two selected classes achieving reasonable classification accuracy. We trained the classifier normally without any defensive techniques like adversarial training. This classifier can process more than 60,000 images per second on a single GTX 1080 Ti GPU. Second, we evaluate programs on a handful of examples and take an average over the evaluation metrics to save GPU time. Instead of using an absolute ℓ_2 distance between the original example $\mathbf{x}_0$ and the best adversarial example $\mathbf{x}$ the program found, we use a relative distance $\|\mathbf{x}-\mathbf{x}_0\|_2/\|\mathbf{x}_1-\mathbf{x}_0\|_2$ as the metric where $\mathbf{x}_1$ is the adversarial starting point as in Alg. 1, which is called ℓ_2 *distortion ratio*. A lower ℓ_2 distortion ratio means a better program.

Even with this small and fast classifier, running programs for tremendous random walk iterations is still computationally expensive. However, adopting lots of iterations is necessary for hyperparameters adjustment strategies to take effect in existing methods [7, 12]. To mitigate this issue, we first evaluate programs for a small number of iterations and select several top performing programs according to the program evaluation metric. Then we perform a second round of evaluation of these programs for a larger number of iterations. This evaluation strategy would also prefer choosing programs that achieve relatively high query efficiency within few iterations. At the initial stage of the random walk process, the success rate is usually high, and thus the hyperparameters adjustment strategy tends to overshoot and harm the performance. So we practically disable hyperparameters adjustment in the small evaluation.

4 Implementation

We implemented a prototype of AutoDA from scratch, since the task is performance-critical while there is no existing system being suitable for our task to the best of our knowledge. We implemented the main functionality of searching for top performing programs as described in Section 3 with about 4,000 lines of C++. About 2,000 lines of Python are used to pre-process datasets, prepare the classifier, and benchmark the discovered algorithms against baseline attack methods. Figure 1 illustrates the implementation overview of our proposed AutoDA.

We train the small and fast classifier described in Section 3.5 with Python using a keras [45] implementation [46] of EfficientNets [44]. Since the main functionality of AutoDA is implemented in C++ for performance consideration, we save this classifier as the SavedModel format, so that we can use the TensorFlow C API to easily load it as well as run it on GPU in C++. Though small and fast, this classifier requires a large batch size ($> 1,000$) to achieve acceptable GPU utilization, which brings extra difficulty to the implementation of AutoDA (See Section 4.2).

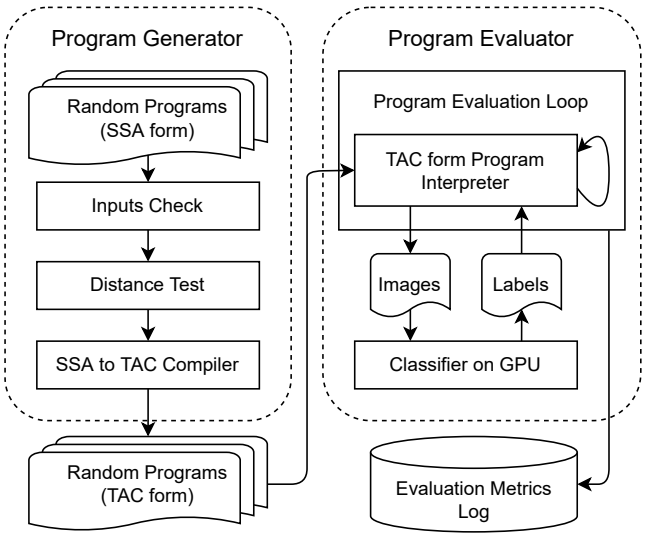

Figure 1: Implementation overview of AutoDA.

4.1 Program Generator

The program generator generates random SSA form programs and filters out trivially bad ones, then translates them into their equivalent TAC form programs and submits them in batches to the program evaluator. The reason for batching here is to satisfy the large batch size requirement of the classifier. We found that the program generator is CPU intensive, so our implementation runs it in a dedicated thread pool.

Random program generation. As mentioned in Section 3.4, we generate SSA form random programs with two priors: (1) *compact program*, and (2) *predefined statements*. The implementation of the *predefined statements* prior is straightforward. As for the *compact program* prior, we use a simple heuristic to implement it — while the program has not reached the max length, we randomly and uniformly select one new operation from all available operations in the DSL, then randomly select parameter(s) for this operation to construct a new assignment statement and append this statement to the program. During this process, we keep a list of all currently unused variables, so that when choosing parameter(s) for the new operation, we choose these unused variables with higher probability than other variables. When a currently unused variable is chosen as the new operation's parameter, the number of unused variables is reduced.

Inputs check and distance test. The *inputs check* and *distance test* are two pruning techniques we introduced in Section 3.4. The *inputs check* is implemented using some basic static analysis techniques [47]. For the *distance test*, since the inequality in Alg. 1 is extremely hard to be checked formally, we test this inequality on ten different inputs for each program, and filter out programs failed in any of these tests. Though the *inputs test* does not guarantee the inequality to hold for all inputs, it works well in practice. The *inputs*

check and *distance test* are both done on CPU cores. By filtering out bad programs before they reach the classifier on GPU, many fewer programs need to be evaluated on GPU. We will show that they save lots of expensive GPU computational cost for us in Section 5.1.

SSA to TAC compiler. SSA form programs are easy to analyze but slow and memory-inefficient to run. After SSA form programs passed the *inputs check* and *distance test*, they are translated into their equivalent TAC form programs using a simple SSA to TAC compiler we implemented from scratch. During the translation, we first discard unused statements then allocate memory slots. Both can be solved with some simple compiler techniques [47]. The output TAC form programs are usually shorter and run faster with less memory usage than the original SSA form programs (See Appendix B and C for samples).

4.2 Program Evaluator

The program evaluator evaluates TAC form programs submitted by the program generator in batches against the classifier on GPU. In the program evaluation loop, for each batch of programs, we first run these programs for 100 iterations without tuning hyperparameters and only keep the best program (with the lowest ℓ_2 distortion ratio) in each batch. Then we collect and run these best programs in batches for 10,000 iterations to finally get their evaluation metrics as explained in Section 3.5. The evaluator logs ℓ_2 distortion ratios of these programs as well as these programs themselves, so that the best performing programs can be extracted from the log file.

We implement an interpreter to execute TAC form programs on CPU using the Eigen linear algebra C++ template library [48] for vector operations. One CPU thread is usually not enough to fully utilize one GPU. For performance, the evaluator evaluates multiple batches of programs in a thread pool. In each random walk iteration in Alg. 1, the program generates one image and queries the classifier on GPU for its label to know whether it is adversarial. However, we need to query the classifier in large batches to achieve acceptable throughput. To alleviate this issue, these interactions between the programs on CPU and the classifier on GPU are done asynchronously in large batches. As a result, with a proper thread pool size, the evaluator can fully utilize both these threads and the classifier on GPU.

5 Experiments

In this section, we first evaluate AutoDA to search for top performing programs under the program evaluation metric described in Section 3.5 on the small CIFAR-10 classifier (Section 5.1). Then we run benchmarks to show that, under the intended setting of attacking normal CIFAR-10 models, the discovered algorithms can achieve comparable or better performance than existing human-designed decision-based attacks (Section 5.2). To further understand the strength and weakness of the discovered algorithms, we also run benchmarks with defended models on CIFAR-10 and models on a larger dataset — ImageNet (Section 5.3). When attacking these models, these algorithms discovered by AutoDA using the small CIFAR-10 binary classifier can also achieve competitive performance. Finally, we conduct an ablation study for the four techniques used in the search method of AutoDA proposed in Section 3.4 to show their effectiveness (Section 5.4).

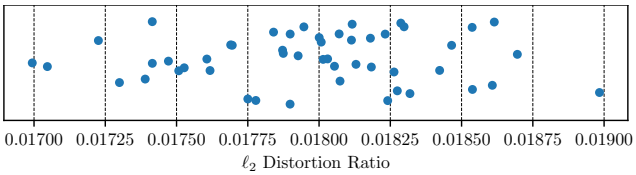

Figure 2: Distribution of the lowest ℓ_2 distortion ratios found in each of the 50 runs of the searching-for-programs experiment.

```
def AutoDA_1st(s0, v1, v2, v3):
    v4 = SUB.VV(v1, v2)
    s5 = NORM.V(v4)
    v6 = DIV.VS(v4, s5)
    v8 = MUL.VS(v3, s0)
    v11 = MUL.VS(v8, s5)
    v17 = ADD.VV(v8, v6)
    s18 = DOT.VV(v17, v8)
    v21 = MUL.VS(v4, s18)
    v22 = ADD.VV(v21, v2)
    v23 = SUB.VV(v22, v11)
    return v23
```

```
def AutoDA_2nd(s0, v1, v2, v3):
    v4 = SUB.VV(v1, v2)
    s5 = NORM.V(v4)
    v6 = DIV.VS(v4, s5)
    v7 = MUL.VS(v3, s0)
    v8 = ADD.VV(v7, v6)
    s9 = NORM.V(v2)
    v10 = MUL.VS(v8, s5)
    s11 = DOT.VV(v10, v7)
    v12 = DIV.VS(v6, s9)
    v17 = ADD.VV(v6, v12)
    v20 = MUL.VS(v17, s11)
    v21 = ADD.VV(v1, v20)
    v23 = SUB.VV(v21, v10)
    return v23
```

Figure 3: The SSA form programs of AutoDA 1st and 2nd, where `s0` is the hyperparameter, `v1` is the original example $\mathbf{x}_0$, `v2` is the adversarial example $\mathbf{x}$ the random walk process already found, and `v3` is the standard Gaussian noise $\mathbf{n}$. The return value of these programs is $\mathbf{x}'$, the next random point to walk to. The `s`-prefix in variable's name means the variable is a scalar, and `v`-prefix for vector. Unused statements are discarded for clarity. The original programs as well as the compiled TAC form programs are provided in Appendix B.

5.1 Searching for Programs

We first introduce the detailed settings of running AutoDA to search for top performing programs. For the search space, we limit the maximum length of the program to 20 (i.e., the length of the Boundary attack's `generate()` function in AutoDA DSL). We allow one scalar hyperparameter and set it to 0.01 initially. We use the binary classifier for class 0 (airplane) and class 1 (automobile) of the CIFAR-10 dataset described in Section 3.5. For the search and evaluation methods, we first generate programs randomly with all the four techniques

introduced in Section 3.4, then evaluate these programs in batches for 100 iterations to calculate their ℓ_2 distortion ratios. For each batch, we select one example from the test set randomly and use it to evaluate all programs in this batch, such that the ℓ_2 distortion ratios of these programs in the same batch can be compared with each other. The batch size here is 150, which achieves acceptable performance on our hardware. Second, we select the best programs with the lowest ℓ_2 distortion ratios from each batch of programs and evaluate them for 10,000 iterations on ten fixed examples from the CIFAR-10 test set to obtain their final ℓ_2 distortion ratios. Since the ten examples are fixed, these final ℓ_2 distortion ratios can be compared with each other.

We run this experiment for 50 times in parallel. Each run allows a maximum number of 500 million queries to the classifier, which takes about two hours on one GTX 1080 Ti GPU. In all 50 runs, we generate about 125 billion random programs. 45.475% of these programs failed in the *inputs check*, 54.497% of them failed in the *distance test*, and only 0.028% of them survived both and continued to be evaluated against the classifier on GPU. These results show that the *inputs check* and *distance test* techniques save a lot of expensive GPU computational cost for us. We achieve a throughput of 294k programs per second per GTX 1080 Ti GPU.

Results. We plot the lowest ℓ_2 distortion ratios found on the ten fixed examples in each run in Figure 2. They average at 0.01797 with a standard deviation of 0.00043. The top one ℓ_2 distortion ratio is 0.01699, the second place is 0.01705, while the third place is 0.01723. The first and second place programs have very close evaluation metrics and we choose both of them to compare with human-designed attacks. We call them *AutoDA 1st* and *AutoDA 2nd* respectively.

We show the SSA form programs of AutoDA 1st and 2nd in Figure 3. We are surprised that they are quite short after discarding unused statements — the length of AutoDA 1st and 2nd are 10 and 13 respectively, while the Boundary attack's `generate()` function uses 20 statements when expressed in the AutoDA DSL, and the more recent Sign-OPT attack [22] and HopSkipJumpAttack [23]'s official implementations [49, 50] both have several hundred lines of Python code.

Geometric interpretation (See Appendix B for details). It turns out that the AutoDA 2nd is approximately the same algorithm as AutoDA 1st, and the return value $\mathbf{x}'_1$ of the AutoDA 1st program which represents the next random point $\mathbf{x}'$ to walk to can be decomposed as $\mathbf{x}'_1 = \mathbf{x} + \mathbf{d}_\parallel + \mathbf{d}_\perp$, where

$$\mathbf{d}_\parallel = s^2\|\mathbf{n}\|_2^2\mathbf{d}, \quad \mathbf{d}_\perp = s\left(\frac{\mathbf{d}\cdot\mathbf{n}}{\|\mathbf{d}\|_2}\mathbf{d} - \|\mathbf{d}\|_2\mathbf{n}\right), \tag{1}$$

$\mathbf{d} = \mathbf{x}_0 - \mathbf{x}$ is the vector pointing from the adversarial example $\mathbf{x}$ to the original example $\mathbf{x}_0$, and $\mathbf{n}$ is the standard Gaussian noise described in Section 3.3. We can then give AutoDA 1st a geometric interpretation based on this decomposition: the $\mathbf{d}_\parallel$ component makes movement towards the original example, and the $\mathbf{d}_\perp$ component tries to move along the local classification boundary similar to the Boundary attack's orthogonal perturbation. These similarities qualitatively suggest the rationality of some heuristics used in existing attacks. Eq. (1) also shows that the hyperparameter s controls the step size of both components. Especially, since the $\mathbf{d}_\parallel$ component makes movement towards the original example $\mathbf{x}_0$, it should at least avoid walking past $\mathbf{x}_0$, thus the expectation of its coefficient $\mathbb{E}(s^2\|\mathbf{n}\|_2^2) = s^2\dim(\mathbf{n})$ should be smaller than 1.

5.2 Benchmark Results on Normal CIFAR-10 Models

In this subsection, we benchmark the AutoDA 1st and 2nd for attacking various normal CIFAR-10 models under the ℓ_2 norm untargeted decision-based threat model, and compare them with four existing human-designed baseline attack methods. We run this benchmark to empirically show that, though we use a small and fast CIFAR-10 binary classifier during the search, the discovered algorithms perform well under our intended setting of attacking larger normal CIFAR-10 models.

Benchmark metrics. (1) We adopt the ℓ_2 *distortion vs. queries* curve, which is widely used in previous decision-based attack works [12, 22]. (2) We also use the *attack success rate vs. queries* curve to show the effectiveness and efficiency of these attack methods following Dong et al.'s benchmark [14] methodology. We consider one attack to be successful after it finds adversarial example whose ℓ_2 distance w.r.t. the original example is smaller than a given threshold ε. We demonstrate benchmark results with $\varepsilon = 1.0$ and $\varepsilon = 0.5$. It is common practice to choose $\varepsilon = 1.0$ in existing works. A smaller $\varepsilon = 0.5$ makes the attack problem harder, and makes the benchmark more comprehensive. As for the number of iterations (or queries), we choose 20,000 to keep consistent with Dong et al.'s benchmark [14]. It is worth noting that during the search we only allow 10,000 iterations.

Models and data. We choose four models of varied architectures for this benchmark: (1) the ResNet50 [2] model, (2) the DenseNet [51] model, (3) the DPN [52] model, and (4) the DLA [53] model. For the ResNet50 model, we use the pre-trained checkpoint provided by Engstrom et al. [54], whose test accuracy is 94.4%. For the other three models, we trained them by ourselves, and they achieve test accuracy of 95.7%, 95.6%, and 95.1% respectively. All these models achieve close to state-of-the-art test accuracy on the CIFAR-10 dataset. We select the first 1,000 images from the CIFAR-10 test set to run the benchmark. All pixels are scaled to be in the range $[0, 1]$. We clip the perturbed image into this valid range $[0, 1]$ for all attacks by default.

Baselines. We compare AutoDA 1st and 2nd with the following four decision-based attacks[3]: (1) the Boundary at-

[3] In this work, we particularly focus on decision-based attacks under the ℓ_2 norm, thus we search for good programs using the ℓ_2 distortion ratio as the evaluation metric as described in Section 3.5. There are a few decision-

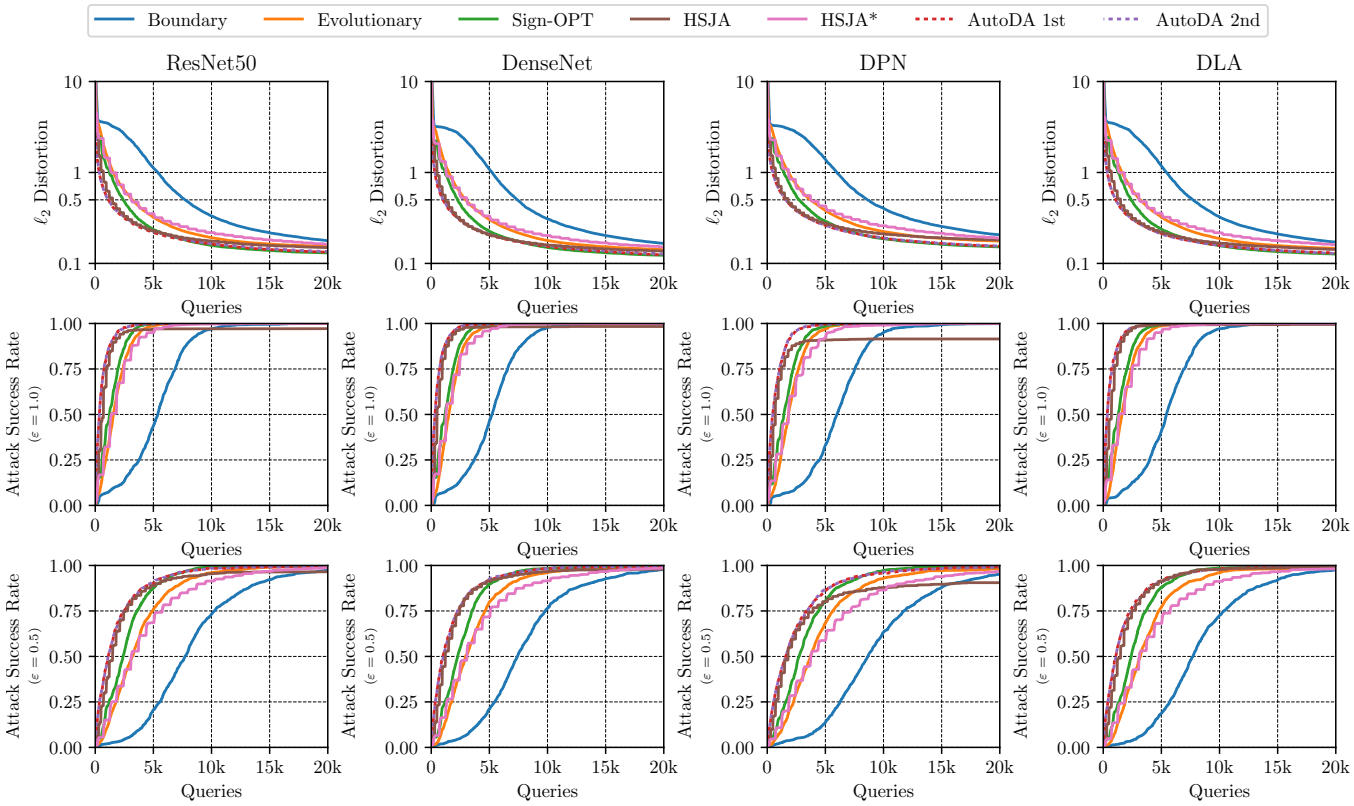

Figure 4: The *median ℓ_2 distortion (on a log scale) vs. queries* and *attack success rate* ($\varepsilon = 1.0$ *and* $\varepsilon = 0.5$) *vs. queries* curves for the six attack methods on the four normal CIFAR-10 models. As mentioned in the 5th paragraph of Section 5.2, the HSJA does not perform well for the DPN model, while the HSJA* does.

tack [12], (2) the Evolutionary attack [7], (3) the Sign-OPT attack [22], and (4) the HopSkipJumpAttack (HSJA) [23]. The first two attacks are both based on the random walk framework described in Alg. 1. These two attacks are included in Dong et al.'s benchmark [14], so we adapt their implementations. The third one and fourth one are two recently proposed state-of-the-art attacks based on zeroth-order optimization. We adapt the implementations from their official repositories [49, 50]. The implementations of the Boundary attack and Evolutionary attack from Dong et al.'s benchmark supports attacking images in a batch, which improves GPU utilization and reduces overall runtime by a lot. However, the official implementations of the Sign-OPT attack and HSJA do not support batching. To support this feature, we make necessary modifications to their official implementations. The Sign-OPT attack and HSJA might spend up to several hundreds of queries for finding starting points (since they use complex methods for finding starting points), while other attacks do not. Therefore, for all attack methods (including ours), we include these queries for finding the starting points in the total queries to make the comparison fairer. Attacks might spend an excessive number of trails to find a starting point, especially the Sign-OPT and HSJA's original implementations. To alleviate this problem, when attacks failed to find a starting point after a limited number of queries, we let them fallback to starting points selected from the test set.

Hyperparameters. The hyperparameters of AutoDA 1st and 2nd are initialized to 0.01, the same value used in the search as described in Section 5.1. For the four baseline attacks, we leave their default hyperparameters unmodified. The HSJA has a hyperparameter to switch between using the default *geometric progression* or *grid search* for step-size search. In our experiments, we found that the default *geometric progression* does not perform well when attacking some models. So we also include HSJA using *grid search* in the benchmark, and denote it as HSJA*.

Results. We plot the *median ℓ_2 distortion (on a log scale) vs. queries* and *attack success rate* ($\varepsilon = 1.0$ *and* $\varepsilon = 0.5$) *vs. queries* curves for the six attacks on the four normal CIFAR-10 models in Figure 4. We also provide *attack success rate* ($\varepsilon = 1.0$) at different number of *queries* in Table 2. Extra results for numerical comparisons are provided in Appendix D. We also provide visualization for the generated adversarial

based attacks specifically designed for and supporting only the ℓ_∞ norm, e.g. RayS [24]. These ℓ_∞ only attacks are not included in our benchmark, since they are designed for a different threat model.

Table 2: The *attack success rate* ($\varepsilon = 1.0$) given different number of *queries* for the six attack methods on the four normal CIFAR-10 models.

Model	ResNet50			DenseNet			DPN			DLA		
Queries	2,000	4,000	20,000	2,000	4,000	20,000	2,000	4,000	20,000	2,000	4,000	20,000
Boundary	10.7%	28.4%	100.0%	10.6%	28.5%	100.0%	7.5%	20.9%	99.9%	9.4%	25.4%	100.0%
Evolutionary	64.9%	96.3%	100.0%	66.9%	95.8%	100.0%	56.0%	93.3%	100.0%	63.9%	95.5%	100.0%
Sign-OPT	76.1%	98.8%	100.0%	77.8%	98.9%	100.0%	67.8%	96.6%	100.0%	74.2%	98.1%	100.0%
HSJA	91.9%	96.6%	97.1%	94.2%	97.9%	98.3%	85.5%	90.6%	91.5%	93.8%	99.0%	99.5%
HSJA*	67.4%	92.6%	100.0%	71.7%	92.9%	100.0%	60.1%	86.5%	100.0%	65.1%	91.7%	100.0%
AutoDA 1st	**95.9%**	**99.7%**	100.0%	96.4%	99.5%	100.0%	92.4%	98.7%	100.0%	**95.8%**	**99.3%**	100.0%
AutoDA 2nd	95.6%	99.5%	100.0%	**96.5%**	**99.7%**	100.0%	**92.7%**	**99.1%**	100.0%	95.4%	99.3%	100.0%

examples of our AutoDA 1st and the Boundary attack given different number of queries in Appendix D.

From these results, we can observe that our attacks consistently outperform the Boundary attack by a lot under all benchmark metrics on all four models. Since AutoDA's search space are inspired by and based on the Boundary attack, these results are quite impressive — though the AutoDA DSL includes only operations used by the Boundary attack (The Sign-OPT attack and HSJA are not included in the search space), the search space contains much better and simpler algorithms than the Boundary attack, illustrating that the expressiveness of the search space is adequate.

When compared with other baselines, the AutoDA 1st and 2nd also achieve competitive performance consistently on all four models. It is worth noting that our algorithms achieve relatively higher attack success rates and lower ℓ_2 distortion under a relatively small number of queries ($< 5,000$), demonstrating their *high query efficiency*. Though the HSJA* and Evolutionary attack perform better than the Boundary attack, our algorithms still outperform them under any query budget. HSJA achieves high query efficiency and is only slightly worse than ours for small number of queries. However, the HSJA does not perform well as the number of queries growing, especially on the DPN model. The Sign-OPT attack's query efficiency is worse than ours for small number of queries. When the number of queries growing larger, Sign-OPT converges to slightly better adversarial examples. For example, on the ResNet50 model at 20,000 queries, the median ℓ_2 distortion is 0.131 for Sign-OPT, while 0.133 and 0.135 for AutoDA 1st and 2nd, and the fourth place is HSJA achieving 0.149 (See Appendix D for more numerical results).

5.3 Benchmark Results on Adversarially Trained Models and ImageNet Models

To further understand the strength and weakness of the AutoDA 1st and 2nd, we also benchmark them against defended models and models on a larger dataset. For defended models, we consider adversarially trained models. For larger dataset, we consider the ImageNet dataset.

Benchmark metrics. We use the same benchmark metrics as in Section 5.2. For ImageNet models, instead of the plain ℓ_2 distance, existing works usually use the normalized ℓ_2 distance to measure distortion, which is defined as $\|\cdot\|_2/\sqrt{d}$ where d is the dimension of the input to the model, because different ImageNet models might require different input dimensions (e.g., $224 \times 224 \times 3$). We use the normalized ℓ_2 distance for ImageNet models, too. For this special distance measurement for ImageNet models, we choose an attack success threshold of $\varepsilon = \sqrt{0.001}$, which is used by Dong et al.'s benchmark [14], as well as a smaller threshold of $\varepsilon = \sqrt{0.001}/2$ to make the benchmark more comprehensive.

Models and data. For defended models, we choose two adversarially trained models: (1) the pre-trained ℓ_2 adversarially trained ResNet50 [2] model provided by Engstrom et al. [54], (2) the pre-trained ℓ_∞ adversarially trained WRN [55] model provided by Madry et al. [25]. Their test accuracy is 82.4% and 87.3% respectively. For models on larger datasets, we consider the widely used ImageNet dataset, and choose two pre-trained ImageNet models provided by the torchvision package [56]: (1) the WRN [55] model, and (2) the ResNet101 [2] model. Their test accuracy is 78.5% and 77.4% respectively. They both accept inputs of dimension $224 \times 224 \times 3$. Similar to CIFAR-10 models, we also scale all pixels into the range $[0, 1]$ for the two ImageNet models. For CIFAR-10 data, we select the first 1,000 images from the CIFAR-10 test set same as in Section 5.2. For ImageNet data, we select the first 1,000 images from the ImageNet test set.

Baselines. For the two defended CIFAR-10 models and the two normal ImageNet models, we use the same baselines as in Section 5.2. The implementations of Boundary attack and Evolutionary attack from Dong et al.'s benchmark enable the dimension reduction trick [37], which is a generally useful trick for the acceleration of black-box attacks [7]. Though this trick can also be integrated into our AutoDA 1st and 2nd easily, the other two baseline attacks' implementations do

not support this trick. To make the comparison fairer and to understand the original attack's strength, we disable this trick for all attacks in this benchmark including our AutoDA 1st and 2nd.

Hyperparameters (See Appendix B for more explanations). As concluded in Section 5.1, the hyperparameter s should satisfy $s^2 \dim(\mathbf{n}) < 1$. However, for the ImageNet dataset where $\dim(\mathbf{n}) = 224 \times 224 \times 3$, when $s = 0.01$, $s^2 \dim(\mathbf{n}) \approx 15 \gg 1$. In order to satisfy this inequality, we decrease hyperparameters' initial value to 0.001 when attacking ImageNet models.

Results. We plot the *median ℓ_2 distortion (on a log scale) vs. queries* and *attack success rate ($\varepsilon = 1.0$ and $\varepsilon = 0.5$) vs. queries* curves for the six attacks on the two adversarially trained CIFAR-10 models in Figure 5(a), as well as the *normalized ℓ_2 distortion vs. queries* and *attack success rate ($\varepsilon = \sqrt{0.001}$ and $\varepsilon = \sqrt{0.001}/2$) vs. queries* curves for the five attacks on the two normal ImageNet models in Figure 5(b).

For the two adversarially trained CIFAR-10 models, the Boundary attack still falls far behind the others. Our algorithms are competitive with HSJA and Sign-OPT attack for these two defended models. Our algorithms have slightly worse query efficiency than HSJA for a small number of queries, while HSJA falls behind our algorithms after about 10,000 queries. Though Sign-OPT performs slightly better when the number of queries grows larger than about 7,000, our algorithms achieve better query efficiency under the setting with a smaller number of queries.

To our surprise, our discovered algorithms can actually work on the ImageNet models, whose input dimension is 49 times larger than that of the CIFAR-10 models. The Boundary attack falls far behind the other attacks. Our algorithms still show better query efficiency than Sign-OPT for about 7,000 queries. For a larger number of queries, Sign-OPT converges to slightly better adversarial examples than ours. From both the *distortion vs. queries* and the *attack success rate vs. queries* curves, we observe that our algorithms have better query efficiency than HSJA under a small number of queries, too. For a larger number of queries, our algorithms have a little worse or similar attack success rate with HSJA, while have better distortion results than HSJA.

There are many widely-used metrics to benchmark attacks. As a result, even state-of-the-art attacks cannot consistently achieve better performance than others, e.g., HSJA converges faster than Sign-OPT, while Sign-OPT converges better than HSJA. To conclude, benchmark results in Section 5.2 and Section 5.3 suggest that our attacks are in the middle of these two state-of-the-art attacks, and much better than the Boundary attack. In this work, our search space is based on the Boundary attack and kept relatively restricted, e.g., our DSL is not able to express Sign-OPT or HSJA. These experiment results suggest that this search space based on the Boundary attack contains much stronger attacks than the Boundary attack, and existing state-of-the-art attacks are near-optimal given this restricted search space.

5.4 Ablation Study on Search Method

As described in Section 3.4, we apply four techniques to our search method, which are (1) *inputs check*, (2) *distance test*, (3) *compact program*, and (4) *predefined statements*. To illustrate their effectiveness, we conduct an ablation study on ten variants of our search method in this section, including (a) four isolated variants, where each variant is applied with only one of the four techniques, (b) four exclude-one variants, where each variant is applied with three techniques, i.e., one of the four techniques is excluded, (c) base variant, which is just naive random search, and (d) full variant, which includes all four techniques. We cannot run the full experiments described in Section 5.1 for each of the ten variants, since they are too time consuming. Instead, for each of these ten variants, we run it to evaluate 100,000 programs against the classifier for 100 iterations on five fixed examples and calculate the ℓ_2 distortion ratio for each program. We plot the top 200 lowest ℓ_2 distortion ratios that each variant found in Figure 6.

As we can see from the results, the four isolated variants overall perform better than the base variant, and the full variant performs better than the four exclude-one variants. Both facts suggest the effectiveness of the four techniques. The third column (isolated variant for *distance test*) seems to be an exception — when compared to the base variant, it provides better lowest ℓ_2 distortion ratio (< 0.3) and better overall performance, but it performs worse in the 0.3 to 0.5 range. One possible reason for this phenomenon is that, there are algorithms failed in the *distance test* which are able to reduce the ℓ_2 distortion ratio after 100 steps. For example, an algorithm might sometimes reduce the distance but sometimes increase the distance. These algorithms do not meet the requirement of Alg. 1, thus the *distance test* filters them out.

It worth noting that these results are *qualitative*. On the one hand, these techniques interfere with each other. On the other hand, there is no straightforward qualitative measurement for a set of programs' overall quality. As a result, the absolute improvement shown in Figure 6 does not imply the effectiveness of each technique.

6 Discussion

We have demonstrated AutoDA's design and implementation, as well as the competitive performance of the attack algorithms discovered by it. In this section, we discuss its limitations and possible future extensions.

Threat models. In this work, we particularly focus on the automated discovery of *untargeted* decision-based adversarial attacks under *the ℓ_2 norm*. Since the random walk framework Alg. 1 also supports doing targeted attack under the ℓ_2 norm, our approach can be extended to the *targeted* threat model

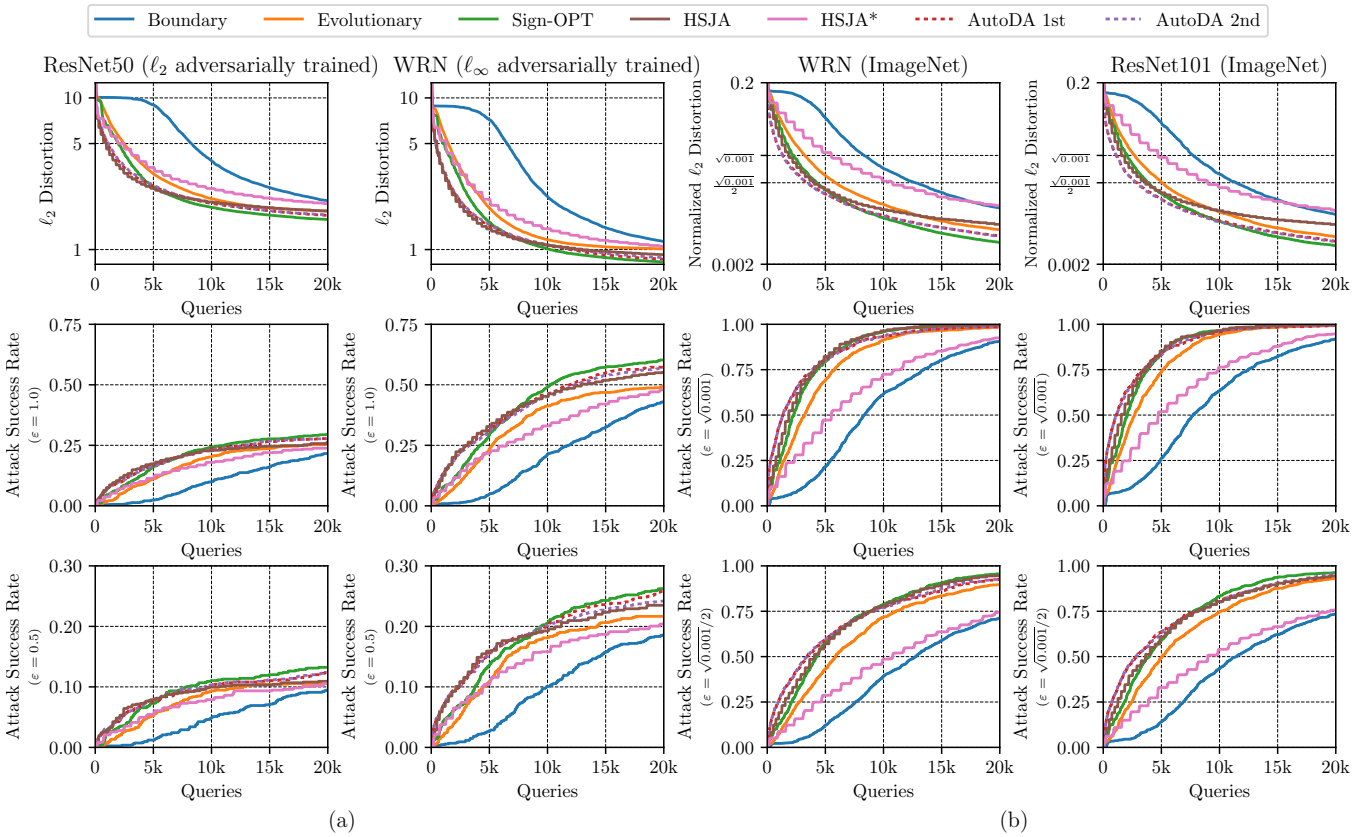

Figure 5: (a) The first two columns are the *median ℓ_2 distortion (on a log scale) vs. queries* and *attack success rate ($\varepsilon = 1.0$ and $\varepsilon = 0.5$) vs. queries* curves for the six attack methods on the two adversarially trained CIFAR-10 models. (b) The last two columns are the *median normalized ℓ_2 distortion (on a log scale) vs. queries* and *attack success rate ($\varepsilon = \sqrt{0.001}$ and $\varepsilon = \sqrt{0.001/2}$) vs. queries* curves for the six attack methods on the two normal ImageNet models.

straightforwardly. However, extending to *the ℓ_∞ norm* might need more efforts. Though the random walk algorithm template shown in Alg. 1 does not limit itself to the ℓ_2 norm, existing random-walk-based iterative attacks are usually designed for the ℓ_2 norm only but not ℓ_∞ [7, 12, 21]. Other approaches [22, 23] based on zeroth-order optimization provide extensions to the ℓ_∞ norm threat model, but usually exhibiting unsatisfactory performance [24]. There are a few attack methods specially designed for the ℓ_∞ norm, e.g., RayS [24]. However, RayS contains relative complex control flow. These facts suggest that designing another search space or even designing another algorithm template might be necessary for the ℓ_∞ norm threat model.

Search space. To reduce computational cost, we limit the search space to be relatively small in this work. Though the experiments show that there are good attack algorithms in this search space, a larger thus more powerful search space might lead to even better algorithms. More mathematical operations could be added to the AutoDA DSL. Also, carefully integrating some high-order operations (e.g., binary search used in many existing attacks [22–24]) into the DSL might lead to a much more powerful search space. However, a larger search space with higher complexity would always lead to higher implementation complexity and higher computational cost, thereby always a tradeoff to consider in practice.

Search method. In this work, we use random search combined with several pruning techniques and intuitive priors. This search method is quite simple and works quite well for our search space. Though random search based method should be treated as a reasonable baseline as discussed in Section 3.4, advanced search methods might help to explore a more powerful search space, thus they are worth trying.

Interpretability. The downside of automating is lacking interpretability in general. Due to the black-box nature of deep learning models, it is highly nontrivial to carry out theoretical analysis about decision-based attacks including many existing expert-designed ones [7, 12]. We are quite lucky to discover geometrically interpretable attacks (See Appendix B). However, this kind of theoretical analysis can only be carried out by domain experts in a case-by-case manner. There is still

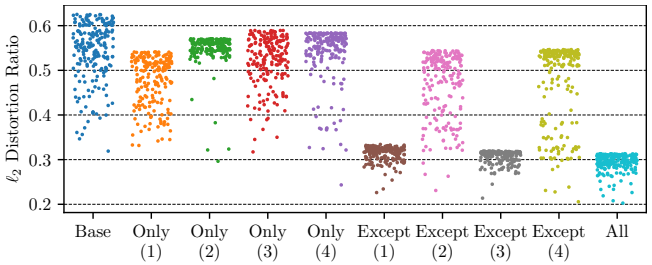

Figure 6: Search method ablation study experiment results. Each column shows the top 200 lowest ℓ_2 distortion ratios found by each search method. The first column is the base variant. The last column is the full variant. The (1), (2), (3), and (4) in the labels of x-coordinate are for *inputs check*, *distance test*, *compact program*, and *predefined statements* respectively. The second to fifth columns are the four isolated variants. The sixth to ninth columns are the four exclude-one variants.

something we can do about this in the future, e.g., we can do automatically simplification and equivalence checking for the discovered algorithms.

Parameters and classifier selection. Though we have done an ablation study on our search method in Section 5.4, how other design choices impact the system is still not clear and can be investigated in the future, e.g., the number of queries allowed for one program during the search (it is set to 10,000 in this work), the max length of programs when random generating them (it is set to 20 in this work), and the noise distribution (it is set to the standard Gaussian distribution in this work). Moreover, selecting classifiers for evaluating programs during the search is worth further investigating, too. For example, using classifiers with higher input dimension during the search might lead to better algorithms for ImageNet models. Adversarially training the classifier might lead to better algorithms on defended models.

Practicality. Decision-based black-box setting is one of the most practical settings for adversarial attacks. For example, the Boundary attack paper [12] attacked a cloud-based computer vision API, and the Evolutionary attack paper [7] attacked an online face recognition system. Since our AutoDA 1st and 2nd show better performance than these two attacks on various models with different network architectures (many of these architectures have real-world usage) in our benchmark, we believe our AutoDA 1st and 2nd can also be used to attack these online services. However, as running attacks against online services can be time-consuming and expensive, it is not our current focus to run our benchmark on these online services. Moreover, there are defensive methods for detecting black-box adversarial attacks for online services [57]. Investigating how the automatically generated attacks interact with these defended real-world applications, and even to automatically find better attacks to evade the detection are both possible future work.

7 Related Work

Our AutoDA relates to program synthesis, as well as automated machine learning (AutoML). In this section, we briefly review some prior works in these two fields.

Program synthesis. Our approach relates to program synthesis aforementioned in Section 1, whose core problem is to generate a program that meets an intent specification [26]. Program synthesis works usually use non-machine-learning techniques, e.g., SKETCH [27] solves the programming by sketching problem using SAT solver and Brahma [28] can efficiently discover highly nontrivial up to 20 lines loop-free bitvector programs. Recent works may use machine learning techniques, e.g., DeepCoder [29] solves the programming by example problem using neural-guided search. These works mainly focus on software problems instead of machine learning problems.

Automated machine learning. Our work also shares the goal with automated machine learning (AutoML), which aims to automate the process of solving machine learning problems [30]. One most relevant direction of AutoML is neural architecture search (NAS), which aims to discover good architectures of deep networks [31]. Existing NAS works usually construct their search space from expert-designed layers, thus good architectures in them are dense. Advanced search methods are often used in NAS works, including reinforcement learning [31] and even gradient-based optimization [58]. However, random search based methods are shown to be simple yet competitive baselines [42, 59].

8 Conclusions

In this work, we propose to automate the process of discovering decision-based iterative attack algorithms. Starting from the random walk framework as the algorithm template, we construct our generic search space from the AutoDA DSL, explore this search space using random search integrated with several pruning techniques and intuitive priors, and evaluate programs in the search space using a small and fast model. The discovered attack algorithms are simple and geometrically interpretable, and meanwhile they consistently achieve competitive performance especially high query efficiency when attacking models on the CIFAR-10 and ImageNet datasets.

Acknowledgments

We would like to thank all the anonymous reviewers and our shepherd, Dr. Christian Wressnegger, for their valuable feedback that greatly helped us improve this paper.

Besides, we would like to thank Min Zhou for providing advice on program synthesis. This work was supported by the National Key Research and Development Program of China (2020AAA0106000, 2020AAA0104304, 2020AAA0106302, 2021YFB2701000), NSFC Projects (Nos. 62061136001, 62076147, U19B2034, U1811461, U19A2081, 61972224), Beijing NSF Project (No. JQ19016), BNRist (BNR2022RC01006), Tsinghua-Alibaba Joint Research Program, Tsinghua Institute for Guo Qiang, Tsinghua-OPPO Joint Research Center for Future Terminal Technology, and the High Performance Computing Center, Tsinghua University.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *NIPS*, 2012, pp. 1106–1114.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016, pp. 770–778.

[3] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, 2019, pp. 4171–4186.

[4] A. Graves, A. Mohamed, and G. E. Hinton, "Speech recognition with deep recurrent neural networks," in *ICASSP*, 2013, pp. 6645–6649.

[5] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *ICLR*, 2014.

[6] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *ICLR*, 2015.

[7] Y. Dong, H. Su, B. Wu, Z. Li, W. Liu, T. Zhang, and J. Zhu, "Efficient decision-based black-box adversarial attacks on face recognition," in *CVPR*, 2019, pp. 7714–7722.

[8] P. Jing, Q. Tang, Y. Du, L. Xue, X. Luo, T. Wang, S. Nie, and S. Wu, "Too good to be safe: Tricking lane detection in autonomous driving with crafted perturbations," in *USENIX Security 2021*, 2021, pp. 3237–3254.

[9] S. G. Finlayson, J. D. Bowers, J. Ito, J. L. Zittrain, A. L. Beam, and I. S. Kohane, "Adversarial attacks on medical machine learning," *Science*, vol. 363, pp. 1287–1289, 2019.

[10] N. Papernot, P. D. McDaniel, I. J. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *AsiaCCS*, 2017, pp. 506–519.

[11] A. Ilyas, L. Engstrom, A. Athalye, and J. Lin, "Black-box adversarial attacks with limited queries and information," in *ICML*, 2018, pp. 2142–2151.

[12] W. Brendel, J. Rauber, and M. Bethge, "Decision-based adversarial attacks: Reliable attacks against black-box machine learning models," in *ICLR*, 2018.

[13] N. Carlini and D. A. Wagner, "Towards evaluating the robustness of neural networks," in *IEEE Symposium on Security and Privacy*, 2017, pp. 39–57.

[14] Y. Dong, Q. Fu, X. Yang, T. Pang, H. Su, Z. Xiao, and J. Zhu, "Benchmarking adversarial robustness on image classification," in *CVPR*, 2020, pp. 318–328.

[15] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. J. Goodfellow, A. Madry, and A. Kurakin, "On evaluating adversarial robustness," *arXiv.org*, 2019.

[16] F. Croce, M. Andriushchenko, V. Sehwag, N. Flammarion, M. Chiang, P. Mittal, and M. Hein, "RobustBench: a standardized adversarial robustness benchmark," *arXiv.org*, 2020.

[17] A. Athalye, N. Carlini, and D. A. Wagner, "Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples," in *ICML*, 2018, pp. 274–283.

[18] F. Tramèr, N. Carlini, W. Brendel, and A. Madry, "On adaptive attacks to adversarial example defenses," in *NeurIPS*, 2020.

[19] N. Papernot, P. D. McDaniel, and I. J. Goodfellow, "Transferability in machine learning: from phenomena to black-box attacks using adversarial samples," *arXiv.org*, 2016.

[20] M. Cheng, T. Le, P. Chen, H. Zhang, J. Yi, and C. Hsieh, "Query-efficient hard-label black-box attack: An optimization-based approach," in *ICLR*, 2019.

[21] T. Brunner, F. Diehl, M. Truong-Le, and A. C. Knoll, "Guessing smart: Biased sampling for efficient black-box adversarial attacks," in *ICCV*, 2019, pp. 4957–4965.

[22] M. Cheng, S. Singh, P. H. Chen, P. Chen, S. Liu, and C. Hsieh, "Sign-OPT: A query-efficient hard-label adversarial attack," in *ICLR*, 2020.

[23] J. Chen, M. I. Jordan, and M. J. Wainwright, "HopSkipJumpAttack: A query-efficient decision-based attack," in *IEEE Symposium on Security and Privacy*, 2020, pp. 1277–1294.

[24] J. Chen and Q. Gu, "RayS: A ray searching method for hard-label adversarial attack," in *KDD*, 2020, pp. 1739–1747.

[25] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *ICLR*, 2018.

[26] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[27] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, 2006, pp. 404–415.

[28] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *PLDI*, 2011, pp. 62–73.

[29] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "DeepCoder: Learning to write programs," in *ICLR*, 2017.

[30] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *NIPS*, 2015, pp. 2962–2970.

[31] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *ICLR*, 2017.

[32] E. Real, C. Liang, D. R. So, and Q. V. Le, "AutoML-Zero: Evolving machine learning algorithms from scratch," in *ICML*, 2020, pp. 8007–8019.

[33] S. Srivastava, S. Gulwani, and J. S. Foster, "Template-based program verification and program synthesis," *Int. J. Softw. Tools Technol. Transf.*, pp. 497–518, 2013.

[34] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception architecture for computer vision," in *CVPR*, 2016, pp. 2818–2826.

[35] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "DeepFace: Closing the gap to human-level performance in face verification," in *CVPR*, 2014, pp. 1701–1708.

[36] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, 2012.

[37] P. Chen, H. Zhang, Y. Sharma, J. Yi, and C. Hsieh, "ZOO: zeroth order optimization based black-box attacks to deep neural networks without training substitute models," in *AISec@CCS*, 2017, pp. 15–26.

[38] J. Uesato, B. O'Donoghue, P. Kohli, and A. v. den Oord, "Adversarial risk and the dangers of evaluating against weak attacks," in *ICML*, 2018, pp. 5032–5041.

[39] S. Cheng, Y. Dong, T. Pang, H. Su, and J. Zhu, "Improving black-box adversarial attacks with a transfer-based prior," in *NeurIPS*, 2019, pp. 10 932–10 942.

[40] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–88.

[41] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Trans. Evol. Comput.*, pp. 67–82, 1997.

[42] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," in *UAI*, 2019, pp. 367–377.

[43] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.

[44] M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *ICML*, 2019, pp. 6105–6114.

[45] F. Chollet. (2015) Keras. [Online]. Available: https://github.com/fchollet/keras

[46] B. B. Pavel Yakubovskiy, Sasha Illarionov. (2019) EfficientNet keras. [Online]. Available: https://github.com/qubvel/efficientnet

[47] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, ser. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[48] G. Guennebaud, B. Jacob *et al.* (2010) Eigen v3. [Online]. Available: http://eigen.tuxfamily.org

[49] M. Cheng. (2018) attackbox. [Online]. Available: https://github.com/cmhcbb/attackbox

[50] J. Chen, M. I. Jordan, and M. J. Wainwright. (2019) HSJA. [Online]. Available: https://github.com/Jianbo-Lab/HSJA

[51] G. Huang, Z. Liu, L. v. der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *CVPR*, 2017, pp. 2261–2269.

[52] Y. Chen, J. Li, H. Xiao, X. Jin, S. Yan, and J. Feng, "Dual path networks," in *NIPS*, 2017, pp. 4467–4475.

[53] F. Yu, D. Wang, E. Shelhamer, and T. Darrell, "Deep layer aggregation," in *CVPR*, 2018, pp. 2403–2412.

[54] L. Engstrom, A. Ilyas, H. Salman, S. Santurkar, and D. Tsipras. (2019) Robustness (Python library). [Online]. Available: https://github.com/MadryLab/robustness

[55] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *BMVC*, 2016.

[56] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *NeurIPS*, 2019, pp. 8024–8035.

[57] S. Chen, N. Carlini, and D. A. Wagner, "Stateful detection of black-box adversarial attacks," *arXiv.org*, 2019.

[58] H. Liu, K. Simonyan, and Y. Yang, "DARTS: differentiable architecture search," in *ICLR*, 2019.

[59] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *CVPR*, 2018, pp. 8697–8710.

Appendix

A Hyperparameter Adjustment Strategies

In this section, we will introduce the hyperparameters adjustment strategies used in the Boundary attack [12], the Evolutionary attack [7], and our AutoDA.

Boundary attack. The Boundary attack [12] has two hyperparameters — δ controlling the step size of the orthogonal perturbation, and ε controlling the step size of moving towards the original example. The original implementation would adjust both hyperparameters during the random walk process. If the success rate for the past several trails is too low, ε would be decreased, and when the success rate is too high, ε would be increased. The attack is converged when ε reaches zero. δ is adjusted according to the orthogonal perturbation's success rate similar to adjusting ε. However, using a fixed δ during the random walk process has negligible performance impact with a proper initial value. In this case, we only need to adjust ε during the random walk process.

Evolutionary attack. The Evolutionary attack [7] has several hyperparameters, while only μ is adjusted during the random walk process. μ is adjusted once for every T iterations as follows:

$$\mu \leftarrow \mu \cdot \exp(p - \bar{p}) \tag{2}$$

where p is the success rate of past T trails, $\bar{p}$ is a predefined target success rate. This is a negative feedback strategy keeping the success rate around $\bar{p}$. The original implementation set T to 30 and $\bar{p}$ to 0.2.

AutoDA. Our AutoDA uses a negative feedback strategy similar to the ones used in the above two attack methods especially the Evolutionary attack. Instead of using the $\exp(p-\bar{p})$ function in the Evolutionary attack, we use a piecewise linear function f that satisfies $f(0) = l$, $f(1) = h$, and $f(\bar{p}) = 1$, where l, h are both predefined constants satisfying $0 < l < 1 < h$. The $\bar{p}$ is the predefined target success rate same as in the Evolutionary attack. For implementation simplicity, instead of adjusting the hyperparameter every T iterations as in the Evolutionary attack, we adjust the hyperparameter in each iteration according to a decayed success rate p defined as follows:

$$p \leftarrow \alpha \cdot p + (1 - \alpha) \cdot k \tag{3}$$

where $k = 1$ if $\mathbf{x}'$ is adversarial, otherwise $k = 0$, α is the decay rate. We adjust the hyperparameter s as follows:

$$s \leftarrow s \cdot [f(p)]^{1/10} \tag{4}$$

where the extra $\cdot^{1/10}$ is to stabilize the hyperparameter adjustment, so that when p is quite close to 0.0 for ten iterations, s would be decreased to at most $l \cdot s$ instead of a much smaller $l^{10} \cdot s$, and when p is quite close to 1.0 for ten iterations, s would be increased to at most $h \cdot s$ instead of a much larger $h^{10} \cdot s$. This negative feedback strategy would keep the success rate around $\bar{p}$, too. We set α to 0.95, l to 0.5, h to 1.5, and $\bar{p}$ to 0.25 in AutoDA.

B The AutoDA 1st and AutoDA 2nd

For the AutoDA 1st and 2nd, we show their original SSA form programs, their SSA form programs after discarding

Table 3: The *attack success rate* ($\varepsilon = 0.5$) given different number of *queries* for the six attack methods on the four normal CIFAR-10 models.

Model	ResNet50			DenseNet			DPN			DLA		
Queries	2,000	4,000	20,000	2,000	4,000	20,000	2,000	4,000	20,000	2,000	4,000	20,000
Boundary	3.3%	10.7%	97.4%	3.9%	12.6%	97.7%	3.9%	8.5%	95.1%	3.1%	11.0%	97.5%
Evolutionary	29.4%	65.4%	99.6%	31.0%	65.9%	99.1%	21.2%	54.6%	98.1%	28.1%	63.2%	98.8%
Sign-OPT	39.2%	79.3%	**99.9%**	43.3%	81.9%	**99.7%**	33.1%	71.6%	**99.3%**	39.1%	78.7%	**99.7%**
HSJA	64.0%	85.8%	96.6%	66.8%	88.2%	98.0%	55.0%	76.8%	90.5%	65.0%	87.2%	99.0%
HSJA*	33.7%	61.4%	98.8%	37.0%	64.9%	98.5%	28.4%	52.9%	96.7%	32.9%	60.8%	98.1%
AutoDA 1st	**69.8%**	87.1%	99.8%	70.8%	**88.6%**	99.5%	58.0%	80.1%	**99.3%**	**70.6%**	**88.3%**	99.4%
AutoDA 2nd	68.6%	**87.4%**	99.7%	**71.0%**	88.1%	99.4%	**58.3%**	**80.2%**	99.2%	70.1%	87.3%	99.3%

Table 4: The *median* ℓ_2 *distortion* given different number of *queries* for the six attack methods on the four normal CIFAR-10 models.

Model	ResNet50			DenseNet			DPN			DLA		
Queries	2,000	4,000	20,000	2,000	4,000	20,000	2,000	4,000	20,000	2,000	4,000	20,000
Boundary	3.000	1.636	0.178	2.847	1.579	0.166	2.962	1.911	0.206	3.029	1.711	0.172
Evolutionary	0.793	0.399	0.154	0.754	0.378	0.142	0.914	0.468	0.176	0.812	0.401	0.146
Sign-OPT	0.611	0.288	**0.131**	0.586	0.273	**0.123**	0.707	0.352	**0.152**	0.619	0.289	**0.127**
HSJA	0.399	0.252	0.149	0.361	0.228	0.137	0.460	0.301	0.181	0.378	0.243	0.142
HSJA*	0.732	0.402	0.162	0.680	0.376	0.152	0.842	0.482	0.190	0.750	0.403	0.161
AutoDA 1st	**0.356**	**0.245**	0.133	**0.338**	**0.231**	0.124	**0.434**	0.295	0.154	**0.343**	0.239	0.130
AutoDA 2nd	0.364	0.254	0.135	0.344	0.236	0.127	0.436	**0.294**	0.155	0.347	**0.237**	0.129

unused statements, and their compiled TAC form programs in Figure 8.

Though this work focuses on automatically generating attack algorithms but not theoretically understanding them, it turns out that the two discovered programs are related, and there is a geometric interpretation for them. The return value of the AutoDA 1st and 2nd which represents the next random point $\mathbf{x}'$ to walk to can be simplified into the following form ($\mathbf{x}'_1$ for the AutoDA 1st, $\mathbf{x}'_2$ for the AutoDA 2nd),

$$\mathbf{x}'_1 = \mathbf{x} + s\left[\left(\frac{\mathbf{d}\cdot\mathbf{n}}{\|\mathbf{d}\|_2} + s\|\mathbf{n}\|_2^2\right)\mathbf{d} - \|\mathbf{d}\|_2\mathbf{n}\right] \tag{5}$$

$$\mathbf{x}'_2 = \mathbf{x} + s\left[\left(\frac{\mathbf{d}\cdot\mathbf{n}}{\|\mathbf{d}\|_2} + s\|\mathbf{n}\|_2^2\right)\left(1 + \frac{1}{\|\mathbf{x}\|_2}\right)\mathbf{d} - \|\mathbf{d}\|_2\mathbf{n}\right] \tag{6}$$

where $\mathbf{x}$ is the best adversarial example already found and $\mathbf{x}_0$ is the original example (both are defined in Alg. 1), $\mathbf{n}$ is the standard Gaussian noise described in Section 3.3, s is the scalar hyperparameter, and $\mathbf{d} = \mathbf{x}_0 - \mathbf{x}$. s, $\mathbf{x}_0$, $\mathbf{x}$, and $\mathbf{n}$ are `s0`, `v1`, `v2`, and `v3` in Figure 3 respectively.

Relation between the AutoDA 1st and 2nd. In Eq. (5) and Eq. (6), we decompose the noise added to $\mathbf{x}$ into a $\mathbf{d}$ component and a $\mathbf{n}$ component. In this form, the only difference between $\mathbf{x}'_1$ and $\mathbf{x}'_2$ is that $\mathbf{x}'_2$ have an extra $1 + 1/\|\mathbf{x}\|_2$ factor in the $\mathbf{d}$ component's coefficient. $\|\mathbf{x}\|_2$ should have similar magnitude to $\sqrt{\dim(\mathbf{x})}$, which is $\sqrt{32\times32\times3} \approx 55$ for CIFAR-10 and $\sqrt{224\times224\times3} \approx 388$ for ImageNet. Hence $1 + 1/\|\mathbf{x}\|_2$ should be quite close to 1. Thus the AutoDA 2nd is approximately the same as the AutoDA 1st, which explains their benchmark results' similarity.

Geometric interpretation. We can do an orthogonal decomposition of $\mathbf{x}'_1 - \mathbf{x}$ (the noise added to $\mathbf{x}$ to get $\mathbf{x}'_1$) with respect to $\mathbf{d}$,

$$\mathbf{x}'_1 - \mathbf{x} = \underbrace{s^2\|\mathbf{n}\|_2^2\mathbf{d}}_{\mathbf{d}_\parallel} + \underbrace{s\left(\frac{\mathbf{d}\cdot\mathbf{n}}{\|\mathbf{d}\|_2}\mathbf{d} - \|\mathbf{d}\|_2\mathbf{n}\right)}_{\mathbf{d}_\perp} \tag{7}$$

where the $\mathbf{d}_\parallel$ component is parallel to $\mathbf{d}$, and the $\mathbf{d}_\perp$ component is orthogonal to $\mathbf{d}$. With the decomposition in Eq. (7), we can give AutoDA 1st a geometric interpretation. (1) The $\mathbf{d}_\parallel$ component makes movement towards the original example and reduce the distance between the adversarial example and the original example. (2) The $\mathbf{d}_\perp$ component is just the projection of $-\|\mathbf{d}\|_2\mathbf{n}$ onto orthogonal complement of $\mathbf{d}$, which

tries to move along the local classification boundary similar to the Boundary attack's orthogonal perturbation. (3) s controls the step size, since $\mathbf{d}_{\parallel}$ has a s^2 coefficient and $\mathbf{d}_{\perp}$ has a s coefficient. Smaller s leads to smaller step size.

Hyperparameter. Considering the geometric interpretation for $\mathbf{d}_{\parallel}$ in previous paragraph, we can also know why we need to set s to a smaller value when attacking ImageNet models. Since this component makes movement towards the original example to reduce the distance between the adversarial example and the original example, it should at least avoid walking past $\mathbf{x}_0$, thus its coefficient $s^2\|\mathbf{n}\|_2$ should be smaller than 1. We can calculate the expectation of $s^2\|\mathbf{n}\|_2^2$ as $\mathbb{E}(s^2\|\mathbf{n}\|_2^2) = s^2\dim(\mathbf{n})$. For ImageNet models, $\dim(\mathbf{n}) = 224\times224\times3 = 150528$. When $s = 0.01$, $\mathbb{E}(s^2\|\mathbf{n}\|_2^2) = 15.0528 \gg 1$. Thus we need to set s to a smaller value when attacking ImageNet models.

Relation to the Boundary attack. As mentioned above, the $\mathbf{d}_{\perp}$ component is similar to the Boundary attack's orthogonal perturbation, illustrating rationality of this heuristic design. However, there are other important differences between AutoDA 1st and the Boundary attack. (1) Our attacks have only one hyperparameter s, which controls both $\mathbf{d}_{\perp}$ and $\mathbf{d}_{\parallel}$. As s decreasing during the random walk process, the magnitude of the $\mathbf{d}_{\parallel}$ component would decrease faster than the $\mathbf{d}_{\perp}$ component, since the s^2 coefficient in $\mathbf{d}_{\parallel}$ decreases faster than the s coefficient in $\mathbf{d}_{\perp}$. In contrast, the Boundary attack has two hyperparameters. They are tuned separately during the random walk process as described in Section A. (2) The Boundary attack's orthogonal perturbation is obtained by projecting a random noise onto the sphere centered at $\mathbf{x}_0$ and then normalizing its length to δ. Our attacks take a simpler approach of directly projecting a random noise onto the orthogonal complement of $\mathbf{d}$ without normalization. Besides, a fixed δ has negligible performance impact as mentioned in Section A. This might suggest using a hyperparameter δ to normalize the projection of the random noise is not effective enough, while our approach of projection without normalization performs better.

C Implement the Boundary Attack with the AutoDA DSL

To show the expressiveness of our DSL, we provide one possible implementation of the Boundary attack [12] using our AutoDA DSL in Figure 7.

D Extra Benchmark Results

We provide extra benchmark results for Section 5.2. Table 3 and Table 4 show the *attack success rate* ($\varepsilon = 0.5$) and the ℓ_2 *distortion* given different number of *queries* for the six attack methods on the four normal models on the CIFAR-10 dataset respectively. Moreover, Figure 9 provides intuitive

```
def Boundary_generate(
      s0, s1, s2, v3, v4, v5):
    v6 = SUB.VV(v3, v4)
    s7 = NORM.V(v6)
    v8 = DIV.VS(v6, s7)
    s9 = DOT.VV(v5, v8)
    v10 = MUL.VS(v8, s9)
    v11 = SUB.VV(v5, v10)
    s12 = NORM.V(v11)
    s13 = MUL.SS(s1, s7)
    s14 = DIV.SS(s13, s12)
    v15 = MUL.VS(v11, s14)
    v16 = SUB.VV(v6, v15)
    v17 = DIV.VS(v16, s2)
    s18 = NORM.V(v17)
    v19 = SUB.VV(v3, v17)
    s20 = MUL.SS(s0, s7)
    s21 = SUB.SS(s18, s7)
    s22 = ADD.SS(s20, s21)
    s23 = DIV.SS(s22, s18)
    v24 = MUL.VS(v17, s23)
    v25 = ADD.VV(v19, v24)
    return v25
```

(a)

```
def Boundary_generate(
      s0, s1, s2, v0, v1, v2):
    v3 = SUB.VV(v0, v1)
    s3 = NORM.V(v3)
    v4 = DIV.VS(v3, s3)
    s4 = DOT.VV(v2, v4)
    v4 = MUL.VS(v4, s4)
    v4 = SUB.VV(v2, v4)
    s4 = NORM.V(v4)
    s5 = MUL.SS(s1, s3)
    s4 = DIV.SS(s5, s4)
    v4 = MUL.VS(v4, s4)
    v3 = SUB.VV(v3, v4)
    v3 = DIV.VS(v3, s2)
    s4 = NORM.V(v3)
    v4 = SUB.VV(v0, v3)
    s5 = MUL.SS(s0, s3)
    s3 = SUB.SS(s4, s3)
    s3 = ADD.SS(s5, s3)
    s3 = DIV.SS(s3, s4)
    v3 = MUL.VS(v3, s3)
    v3 = ADD.VV(v4, v3)
    return v3
```

(b)

Figure 7: (a) One possible implementation of the Boundary attack's `generate()` function as a SSA form program in the AutoDA DSL. `s0` and `s1` are the ε and δ hyperparameters in the Boundary attack respectively, `s2` is derived from the δ as $\sqrt{1+\delta^2}$. The Boundary attack would adjust both the ε and the δ during the random walk process. However, a fixed δ has negligible performance impact as mentioned in Section A. For simplicity, we fix the δ so that both `s1` and `s2` can be considered as fixed hyperparameters. As a result, we do not need to add the extra $\sqrt{\cdot}$ operation to our DSL for implementing the Boundary attack. `v3` is the original example $\mathbf{x}_0$, `v4` is the adversarial example $\mathbf{x}$ the random walk process already found, and `v5` is the standard Gaussian noise $\mathbf{n}$. (b) The compiled TAC form version of the same program. The `s0`, `s1`, `s2`, `v0`, `v1` and `v2` are compiled from the `s0`, `s1`, `s2`, `v3`, `v4` and `v5` in the SSA form program respectively.

impressions about our algorithm's performance by visualizing adversarial examples generated by the AutoDA 1st and Boundary attack under different number of queries against the ResNet50 model on CIFAR-10.

Discard Unused Statements

```
def AutoDA_1st(s0, v1, v2, v3):
    v4 = SUB.VV(v1, v2)
    s5 = NORM.V(v4)
    v6 = DIV.VS(v4, s5)
    v7 = MUL.VS(v4, s0)
    v8 = MUL.VS(v3, s0)
    s9 = DIV.SS(s0, s0)
    s10 = SUB.SS(s9, s5)
    v11 = MUL.VS(v8, s5)
    s12 = ADD.SS(s10, s9)
    s13 = MUL.SS(s12, s0)
    s14 = ADD.SS(s9, s12)
    v15 = DIV.VS(v11, s13)
    s16 = DOT.VV(v11, v11)
    v17 = ADD.VV(v8, v6)
    s18 = DOT.VV(v17, v8)
    v19 = SUB.VV(v7, v15)
    s20 = DOT.VV(v19, v4)
    v21 = MUL.VS(v4, s18)
    v22 = ADD.VV(v21, v2)
    v23 = SUB.VV(v22, v11)
    return v23
```

```
def AutoDA_1st(s0, v1, v2, v3):
    v4 = SUB.VV(v1, v2)
    s5 = NORM.V(v4)
    v6 = DIV.VS(v4, s5)
    v8 = MUL.VS(v3, s0)
    v11 = MUL.VS(v8, s5)
    v17 = ADD.VV(v8, v6)
    s18 = DOT.VV(v17, v8)
    v21 = MUL.VS(v4, s18)
    v22 = ADD.VV(v21, v2)
    v23 = SUB.VV(v22, v11)
    return v23
```

Allocate Memory Slots

```
def AutoDA_1st(s0, v0, v1, v2):
    v3 = SUB.VV(v0, v1)
    s1 = NORM.V(v3)
    v4 = DIV.VS(v3, s1)
    v5 = MUL.VS(v2, s0)
    v6 = MUL.VS(v5, s1)
    v4 = ADD.VV(v5, v4)
    s1 = DOT.VV(v4, v5)
    v3 = MUL.VS(v3, s1)
    v3 = ADD.VV(v3, v1)
    v3 = SUB.VV(v3, v6)
    return v3
```

(a)

Discard Unused Statements

```
def AutoDA_2nd(s0, v1, v2, v3):
    v4 = SUB.VV(v1, v2)
    s5 = NORM.V(v4)
    v6 = DIV.VS(v4, s5)
    v7 = MUL.VS(v3, s0)
    v8 = ADD.VV(v7, v6)
    s9 = NORM.V(v2)
    v10 = MUL.VS(v8, s5)
    s11 = DOT.VV(v10, v7)
    v12 = DIV.VS(v6, s9)
    s13 = DOT.VV(v12, v8)
    s14 = MUL.SS(s13, s13)
    s15 = MUL.SS(s14, s14)
    s16 = MUL.SS(s11, s15)
    v17 = ADD.VV(v6, v12)
    s18 = SUB.SS(s9, s11)
    s19 = SUB.SS(s9, s18)
    v20 = MUL.VS(v17, s11)
    v21 = ADD.VV(v1, v20)
    s22 = MUL.SS(s16, s19)
    v23 = SUB.VV(v21, v10)
    return v23
```

```
def AutoDA_2nd(s0, v1, v2, v3):
    v4 = SUB.VV(v1, v2)
    s5 = NORM.V(v4)
    v6 = DIV.VS(v4, s5)
    v7 = MUL.VS(v3, s0)
    v8 = ADD.VV(v7, v6)
    s9 = NORM.V(v2)
    v10 = MUL.VS(v8, s5)
    s11 = DOT.VV(v10, v7)
    v12 = DIV.VS(v6, s9)
    v17 = ADD.VV(v6, v12)
    v20 = MUL.VS(v17, s11)
    v21 = ADD.VV(v1, v20)
    v23 = SUB.VV(v21, v10)
    return v23
```

Allocate Memory Slots

```
def AutoDA_2nd(s0, v0, v1, v2):
    v3 = SUB.VV(v0, v1)
    s1 = NORM.V(v3)
    v3 = DIV.VS(v3, s1)
    v4 = MUL.VS(v2, s0)
    v5 = ADD.VV(v4, v3)
    s2 = NORM.V(v1)
    v5 = MUL.VS(v5, s1)
    s1 = DOT.VV(v5, v4)
    v4 = DIV.VS(v3, s2)
    v3 = ADD.VV(v3, v4)
    v3 = MUL.VS(v3, s1)
    v3 = ADD.VV(v0, v3)
    v3 = SUB.VV(v3, v5)
    return v3
```

(b)

Figure 8: (a) and (b) show the original SSA form programs, the SSA form programs after discarding unused statements, and the compiled TAC form programs after allocating memory slots for the SSA form programs, for the AutoDA 1st and AutoDA 2nd respectively. In the original SSA form programs and the SSA form programs after discarding unused statements, `s0` is the hyperparameter, `v1` is the original adversarial example $\mathbf{x_0}$, `v2` is the adversarial example $\mathbf{x}$ the random walk process already found, and `v3` is the standard Gaussian noise $\mathbf{n}$. In the TAC form programs, `s0`, `v0`, `v1` and `v2` is compiled from the `s0`, `v1`, `v2`, `v3` in the SSA form programs respectively.

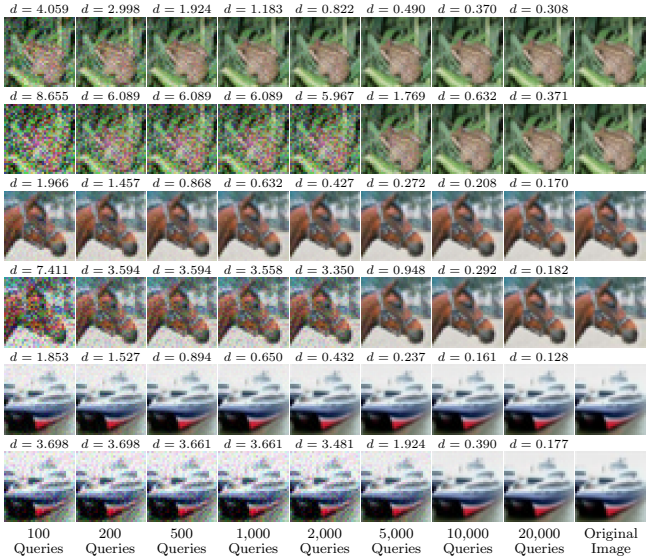

Figure 9: Adversarial examples generated by the AutoDA 1st and Boundary attack under different number of queries (the first eight columns) as well as their corresponding original natural images (the last column) when attacking the normal ResNet50 model on CIFAR-10. The 1st, 3rd and 5th rows are adversarial examples generated by AutoDA 1st, while the 2nd, 4th and 6th rows are for Boundary. The ℓ_2 distortion d is provided above each image.