One Pixel Adversarial Attacks via Sketched Programs

TOM YUVILER and DANA DRACHSLER-COHEN, Technion, Israel

Neural networks are successful in various tasks but are also susceptible to adversarial examples. An adversarial example is generated by adding a small perturbation to a correctly-classified input with the goal of causing a network classifier to misclassify. In one pixel attacks, an attacker aims to fool an image classifier by modifying a single pixel. This setting is challenging for two reasons: the perturbation region is very small and the perturbation is not differentiable. To cope, one pixel attacks iteratively generate candidate adversarial examples and submit them to the network until finding a successful candidate. However, existing works require a very large number of queries, which is infeasible in many practical settings, where the attacker is limited to a few thousand queries to the network. We propose a novel approach for computing one pixel attacks. The key idea is to leverage program synthesis and identify an expressive program sketch that enables to compute adversarial examples using significantly fewer queries. We introduce OPPSLA, a synthesizer that, given a classifier and a training set, instantiates the sketch with customized conditions over the input's pixels and the classifier's output. OPPSLA employs a stochastic search, inspired by the Metropolis-Hastings algorithm, that synthesizes typed expressions enabling minimization of the number of queries to the classifier. We further show how to extend OPPSLA to compute few pixel attacks minimizing the number of perturbed pixels. We evaluate OPPSLA on several deep networks for CIFAR-10 and ImageNet. We show that OPPSLA obtains a state-of-the-art success rate, often with an order of magnitude fewer queries than existing attacks. We further show that OPPSLA's programs are transferable to other classifiers, unlike existing one pixel attacks, which run from scratch on every classifier and input.

CCS Concepts: • **Software and its engineering → Search-based software engineering**; **Automatic programming**; • **Computing methodologies → Neural networks**.

Additional Key Words and Phrases: program synthesis, adversarial attack, computer vision

ACM Reference Format:
Tom Yuviler and Dana Drachsler-Cohen. 2023. One Pixel Adversarial Attacks via Sketched Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 187 (June 2023), 25 pages. https://doi.org/10.1145/3591301

1 INTRODUCTION

Over the past decade, many works have demonstrated that deep neural networks (DNNs) are susceptible to adversarial example attacks. Most adversarial attacks focus on image classifiers, e.g., Goodfellow et al. [2015]; Kurakin et al. [2017]; Madry et al. [2018]; Szegedy et al. [2014]; Yuan et al. [2019]. In this setting, an adversarial attack adds a small perturbation to a correctly-classified image with the goal of causing the network to misclassify. Su et al. [2017] consider an extremely limited setting where the attacker is allowed to change a single pixel and propose an attack, called a *one pixel attack*. Since then, several works have proposed one pixel attacks as well as few pixel attacks [Alatalo et al. 2022; Croce et al. 2022; Croce and Hein 2019; Nguyen-Son et al. 2021; Quan et al. 2021]. These kinds of attacks are highly challenging because the perturbation region is extremely small and the perturbation is not differentiable. To cope, one and few pixel attacks iteratively

Authors' address: Tom Yuviler, tom.yuviler@campus.technion.ac.il; Dana Drachsler-Cohen, ddana@ee.technion.ac.il, Technion, Haifa, Israel.

2475-1421/2023/6-ART187
https://doi.org/10.1145/3591301

generate candidate adversarial examples and submit them to the network, until finding a successful candidate. However, existing approaches require a very high number of queries, making their attacks very expensive, or even infeasible, in a real-world setting where the network limits the number of queries. For example, many online classifiers allow the user to pose a limited number of queries for free per month, after which users can pay for more queries: Amazon Rekognition[1] and Microsoft Azure Computer Vision API[2] allow 5000 queries for free, while Clarifai moderation-recognition API[3] and Google Cloud Vision API[4] allow 1000 queries for free.

To make few pixel attacks more practical, Croce et al. [2022] propose the Sparse-RS attack, relying on random search, where the attacker is limited in the number of queries posed to the network. For one pixel attacks, Sparse-RS generates candidate adversarial examples by randomly picking a pixel to perturb and a perturbation. Sparse-RS obtains a state-of-the-art success rate (where the success rate is the percentage of perturbed images that the network misclassifies out of all perturbed images) with a few thousand queries. However, in practical settings, this number is still high. To illustrate, our experiments show that Sparse-RS obtains at best a success rate of 51% with 1000 queries (Section 6). Namely, given a 5000 query limit (as in Amazon Rekognition and Microsoft Azure Computer Vision API), Sparse-RS can generate about three adversarial examples for free per month, and given a 1000 query limit (as in Clarifai moderation-recognition API and Google Cloud Vision API), it can generate about one adversarial example for free per month. This raises the question: *Can we compute one pixel attacks with a few hundred queries?*

In this work, we draw inspiration from program synthesis and propose to compute one pixel adversarial programs. Given an image classifier and a training set of images, our goal is to synthesize a program that, given a classifier and an image to attack, generates an adversarial example by dynamically identifying the most prominent pixel locations and perturbations for the attack. Like prior one pixel attacks, an adversarial program generates candidate adversarial examples and submits them to the network. Unlike prior one pixel attacks, the large number of queries is only required for the synthesis process. Afterwards, the synthesized program dynamically identifies the prominent candidate adversarial examples, based on the learned conditions and the network's output for the submitted candidates. Conceptually, the synthesized program provides a query-efficient counterexample-guided attack. Our adversarial programs are more practical than Sparse-RS: the attacker only needs to invest an initial fee for the synthesis, and then she can run the attack on new inputs with an order of magnitude fewer queries.

As in any program synthesis task, computing adversarial programs introduces two main challenges: (1) identifying a small yet expressive domain-specific language (DSL) for the programs, and (2) designing an efficient search algorithm to find a program in this DSL meeting the requirements. Additionally, computing adversarial programs introduces unique challenges: (3) the program does not only depend on the given input (i.e., the classifier and the image to attack), but also on intermediate results (i.e., the candidate adversarial examples) and (4) candidate examples do not adhere to a strict partial order, thus an unsuccessful candidate does not enable to safely prune other candidates.

To cope with the above challenges, we propose a program sketch [Solar-Lezama 2009] expressing *prioritizing programs*. Given a classifier and an image, a prioritizing program enumerates all possible candidate adversarial examples until finding a successful adversarial example. That is, given an unlimited number of queries, it is guaranteed to find a successful candidate, if exists. The goal of a prioritizing program is to dynamically prioritize the candidates, in order to find a successful adversarial example with a minimal number of queries. The dynamic prioritization is determined

[1] https://aws.amazon.com/rekognition/pricing/?nc=sn&loc=4
[2] https://azure.microsoft.com/en-us/pricing/details/cognitive-services/computer-vision/
[3] https://www.clarifai.com/pricing
[4] https://cloud.google.com/vision/pricing

by the synthesized conditions. The condition language is defined over pixel locations, pixel values, and the network's output for the submitted candidates. For example, the conditions can prioritize pixels at the center of the image (i.e., pixels whose L_∞ distance from the center of the image is less than a given number) or dark pixels (i.e., pixels whose RGB average value is lower than a given number). The conditions can also decrease the priority of pixels. For example, a condition may lower the prioritization of pixels that are relatively bright (i.e., pixels whose minimal RGB value is greater than a given number). Prioritizing programs explicitly maintain the ordering over all location-perturbation pairs. Thus, they focus on a finite set of possible location-perturbation pairs. Croce and Hein [2019] and Croce et al. [2022] identify that most one and few pixel adversarial examples use perturbations defined by one of the eight corners of the RGB color cube. Thus, our adversarial programs consider for the location-perturbation pairs the set of pairs consisting of any pixel location and any perturbation defined by one of the corners of the RGB color cube.

To instantiate a sketch with customized conditions, we propose OPPSLA, a synthesizer for *One Pixel Program Sketch for adversariaL Attacks*. OPPSLA synthesizes the conditions given a classifier and a training set. Its main challenge is that the condition language consists of boolean constraints (e.g., $\leq$) as well as real-valued variables and constants. Namely, the condition language is both infinite and inapplicable for standard numerical optimization. Although there are search procedures that can search in such spaces, these are oblivious to the structure of the program and thus may generate conditions that are not grammatically correct. Another challenge is that identifying whether a condition is successful is not trivial. In program synthesis, it is common to prefer conditions that match most input examples. However, this approach may result in a large number of queries for input examples that do not match the conditions. To cope with the challenges, OPPSLA employs a stochastic search inspired by Metropolis-Hastings [Chib and Greenberg 1995]. Our search produces grammatically correct programs (similarly to Schkufza et al. [2013]), minimizing the number of queries. Technically, OPPSLA represents an adversarial program (i.e., an instantiated sketch) by an abstract syntax tree over the conditions. Namely, every node corresponds to a typed component. OPPSLA begins with a random program, determined by a random tree. Given a program, it randomly picks a node in the tree and mutates its subtree by replacing each node with an element of the same type. Thereby, OPPSLA constructs only trees corresponding to well-typed programs. Given a mutated program, OPPSLA evaluates it based on a score function. The higher the score of the mutated program, the higher the probability that OPPSLA replaces the current program with the mutated one and continues the optimization from it. Unlike prior works [Gulwani et al. 2017; Schkufza et al. 2013], OPPSLA cannot rely on a score function based on the number of inputs obtaining the desired output, because all prioritization programs have the same success rate. Instead, its score function is based on the average number of queries required for generating successful adversarial examples. Thereby, OPPSLA prefers programs minimizing the number of queries.

We present two extensions of OPPSLA. First, we extend it to compute few pixel attacks perturbing up to k pixels (for $k > 1$), minimizing the number of perturbed pixels. The idea is to execute an adversarial program and, if no successful adversarial example is found, leverage the dynamic prioritization over the location-perturbation pairs to narrow the search space to a small set of the most prominent location-perturbation pairs. Then, as proposed by Croce and Hein [2019], OPPSLA assigns a score for every pair and performs a weighted sampling, for every $k' \in \{2, \ldots, k\}$, until finding a successful adversarial example. The second extension allows the perturbations to have a finer granularity than the eight corners of the RGB color cube. That is, instead of only considering for every color channel the values 0 and 1, OPPSLA considers intermediate values, in order to expand the space of adversarial examples. The main challenge is that the number of possible perturbations grows exponentially with the number of splits to the interval $[0, 1]$. To

mitigate it, we adapt the sketch by planting a flag in the ordering. When OPPSLA reaches the flag, it considers perturbations of a finer granularity only to the most prominent pixel locations.

We evaluate OPPSLA on several CIFAR-10 and ImageNet networks, consisting of millions of parameters. We show that OPPSLA obtains a state-of-the-art success rate compared to existing one pixel attacks. Further, it requires significantly fewer queries than Sparse-RS, the current state-of-the-art attack that minimizes the number of queries. For example, on our CIFAR-10 classifiers, given 100 queries, OPPSLA obtains a success rate of 43%, while Sparse-RS obtains only 27%. To understand the effectiveness of our conditions, we compare to the constant program that has no conditions and relies on the initial prioritization. Note that this program and OPPSLA's programs have the same success rate. We show that OPPSLA's programs require 3x fewer queries than the constant program. We further study the transferability of adversarial programs. An adversarial program is transferable if, when executed on a network different from the one used during the synthesis process, the program requires a similar number of queries to generate successful adversarial examples. We show that, when transferring an adversarial program to another network, the number of queries increases by 46% compared to the number of queries required for its original network. Namely, the number of queries is still an order of magnitude lower than the number of queries of existing one pixel attacks. This makes our adversarial programs practical, because attackers need not synthesize a unique adversarial program for every network. We further show that the number of queries posed to the classifier during the synthesis is relatively low and that OPPSLA can be integrated in an adversarial training to increase the robustness of classifiers to its attack. Lastly, we evaluate OPPSLA for few pixel attacks allowing up to $k \in \{2, \ldots, 20\}$ perturbed pixels. We compare to CornerSearch [Croce and Hein 2019], the state-of-the-art for computing few pixel attacks minimizing the number of perturbed pixels. We show that OPPSLA requires 4.7x fewer queries.

To conclude, our main contributions are:

- A space of programs for computing one pixel attacks. The program space is defined by a sketch, guaranteeing to find an adversarial example, if exists. The missing parts of the sketch are conditions, whose language consists of constraints over pixel locations, pixel values, and the network's output for the submitted candidates.
- OPPSLA, a program synthesizer that employs a stochastic search, inspired by the Metropolis-Hastings algorithm, that instantiates the sketch with grammatically correct conditions, such that the corresponding program minimizes the number of queries to the network.
- Two extensions of OPPSLA: (1) a few pixel attack minimizing the number of perturbed pixels, and (2) a one pixel attack considering finer granularity perturbations.
- An extensive evaluation over several CIFAR-10 and ImageNet networks. The results show that OPPSLA obtains a state-of-the-art success rate for one pixel attacks with an order of magnitude fewer queries. Results also show that adversarial programs are transferable to other networks and that OPPSLA requires 4.7x fewer queries to compute few pixel attacks.

2 PROBLEM DEFINITION

In this section, we define our problem. We begin with a background on neural network classifiers and L_0 adversarial attacks. We then formally define our problem and discuss existing approaches.

Neural network classifiers. We focus on classifiers for colored two-dimensional images, although our approach can be easily adapted to gray-scaled images. A colored image is a $d_1 \times d_2$ matrix of pixels, where a pixel is an RGB triple in $[0, 1]^3$. Given a set of classes $C = \{1, \ldots, c\}$, a classifier is a function mapping an image to a score vector over the possible classes $N : [0, 1]^{d_1 \times d_2 \times 3} \rightarrow \mathbb{R}^c$. We focus on classifiers implemented by a neural network. A network consists of layers, where the first layer is the input layer, taking an image and passing it to the next layer, and the last

layer is the output layer returning a vector $N(x) \in \mathbb{R}^c$. The output vector assigns a score for each class in C. Given an input x, the classification of the network is the class with the highest score, $c' = \text{argmax}(N(x))$. For example, a classifier for the CIFAR-10 dataset determines for each image the object that is shown (e.g., a deer or a ship), out of ten possible classes. We omit the description of the underlying computation of a network, because OPPSLA computes attacks in a black-box setting and is thus oblivious to it. In particular, OPPSLA is general to any network architecture.

Adversarial example attacks. Over the past decade, adversarial example attacks for image classifiers have gained a lot of attention [Carlini and Wagner 2017; Goodfellow et al. 2015; Kurakin et al. 2017; Madry et al. 2018; Szegedy et al. 2014; Tabacof and Valle 2016; Yuan et al. 2019]. An adversarial attack adds a small perturbation to a correctly-classified input with the goal of causing the network to misclassify. Commonly, the attack model limits the allowed perturbation size with respect to some norm, e.g., L_∞, L_2, L_1 or L_0. Formally, given an image classifier N, an input image $x \in [0,1]^{d_1 \times d_2 \times 3}$ correctly-classified as class c_x, and a bound on the perturbation size ϵ along with a p-norm, the goal of an adversarial attack is to compute $\delta \in [-1,1]^{d_1 \times d_2 \times 3}$ satisfying three requirements: (1) $||\delta||_p \leq \epsilon$, (2) $x+\delta \in [0,1]^{d_1 \times d_2 \times 3}$ and (3) the classification of $x+\delta$ is not c_x, that is: $\text{argmax}(N(x+\delta)) \neq c_x$. Such attack is called an untargeted attack, and it is the kind of attacks we focus on in this paper. We note that a targeted attack is defined similarly only that there is an additional input, the target class $c_t \neq c_x$, and the goal is to compute δ such that $\text{argmax}(N(x+\delta)) = c_t$. We say an adversarial example is *successful* if it is defined by $\delta \in [-1,1]^{d_1 \times d_2 \times 3}$ satisfying all three requirements; otherwise, we say the adversarial example is *unsuccessful*. Existing adversarial attacks can be categorized as white-box attacks or black-box attacks. In a white-box attack, the attacker has a full access to the attacked network, including its architecture and parameters [Carlini and Wagner 2017; Goodfellow et al. 2015; Moosavi-Dezfooli et al. 2016; Szegedy et al. 2014]. In a black-box attack, which is often more realistic, the attacker only submits inputs to the classifier to get their output [Croce et al. 2022; Ilyas et al. 2018; Meunier et al. 2019; Papernot et al. 2016; Qiu et al. 2021].

One pixel and few pixel attacks. In this work, we focus on one pixel and few pixel attacks, which are extreme settings of L_0 attacks, i.e., the norm measuring the size of the perturbation δ is $p = 0$. The L_0 norm of a vector δ is its number of non-zero entries. In other words, since δ is the difference between an image x and a perturbed image $x+\delta$, its L_0 norm is equal to the number of different pixels: $||(x+\delta)-x||_0 = |\{(i,j) \in [d_1] \times [d_2] \mid (x+\delta)_{i,j} \neq x_{i,j}\}|$, where $[d] = \{1, \ldots, d\}$. Because the difference is a discrete value, we denote the limit on the perturbation's size by k: $||\delta||_0 \leq k$. In one pixel attacks, the attacker is allowed to perturb a single pixel, i.e., $k = 1$. Formally, given a classifier N, an image $x \in [0,1]^{d_1 \times d_2 \times 3}$ correctly-classified as class c_x, the goal of a one pixel attack is to compute $\delta \in [-1,1]^{d_1 \times d_2 \times 3}$ such that $||\delta||_0 = 1$, $x+\delta \in [0,1]^{d_1 \times d_2 \times 3}$ and $\text{argmax}(N(x+\delta)) \neq c_x$. Note that the attacker is allowed to arbitrarily perturb the chosen pixel, in particular she can perturb every RGB channel (in its valid range $[0,1]$). In few pixel attacks, the attacker is allowed to perturb up to k pixels, where k is a very small number. We focus on a restrictive setting where the attacker computes an attack minimizing the number of perturbed pixels. Formally, given a classifier N, an image $x \in [0,1]^{d_1 \times d_2 \times 3}$ correctly-classified as class c_x, and a bound k, the goal of a few pixel attack is to compute $\delta \in [-1,1]^{d_1 \times d_2 \times 3}$, such that $||\delta||_0 \leq k$, $x+\delta \in [0,1]^{d_1 \times d_2 \times 3}$, $\text{argmax}(N(x+\delta)) \neq c_x$, and the number of perturbed pixels is minimal.

Problem definition. We address the problems of computing one and few pixel attacks, in a black-box setting, with a minimal number of queries. To define the problems, we introduce a notation. We denote by $Q(A, N, x, k)$ the number of queries posed by attack A to classifier N when computing an adversarial example for input x, with at most k perturbed pixels. We begin with defining the problem of minimal-query minimal-perturbation attacks.

Definition 2.1 (Minimal-Query Minimal-Perturbation Attacks). Given a classifier N, providing a black-box access, an image $x \in [0,1]^{d_1 \times d_2 \times 3}$ correctly-classified as class c_x, and a bound k on the number of perturbed pixels, a minimal-query minimal-perturbation attack computes a perturbation $\delta \in [-1,1]^{d_1 \times d_2 \times 3}$ such that $x + \delta \in [0,1]^{d_1 \times d_2 \times 3}$ and:

(1) $||\delta||_0 \leq k$,
(2) $\text{argmax}(N(x+\delta)) \neq c_x$,
(3) $||\delta||_0$ is minimal among all attacks satisfying (1) and (2), and
(4) the number of queries $Q(A, N, x, k)$ posed by A to N for computing δ is minimal among all attacks computing perturbations satisfying (1), (2), and (3).

In this work, we call a minimal-query minimal-perturbation attack a few pixel attack (for short), and if $k = 1$, we call it a one pixel attack. Note that for one pixel attacks, requirement (3) trivially holds since at least one pixel has to be perturbed.

Existing approaches. To the best of our knowledge, there is no work proposing minimal-query minimal-perturbation attacks. However, there are many works that address closely-related problems. Several works compute one and few pixel attacks in a black-box setting [Alatalo et al. 2022; Croce et al. 2022; Croce and Hein 2019; Narodytska and Kasiviswanathan 2017; Nguyen-Son et al. 2021; Quan et al. 2021; Schott et al. 2019]. Among them, only Sparse-RS [Croce et al. 2022] aims to minimize the number of queries posed to the network, and Croce and Hein [2019]; Narodytska and Kasiviswanathan [2017]; Schott et al. [2019] aim to minimize the number of perturbed pixels in few pixel attacks. For other norms, there are several works that propose adversarial example attacks, in a black-box setting, aiming to minimize the number of queries, e.g., Andriushchenko et al. [2020]; Bai et al. [2020]; Ilyas et al. [2018].

3 KEY IDEA: PROGRAM SYNTHESIS FOR ADVERSARIAL ATTACKS

In this section, we present our key idea for designing one and few pixel attacks: relying on program synthesis. We begin by describing the advantages of leveraging program synthesis for our problem, then discuss the challenges, and finally present adversarial programs.

Advantages. Program synthesis combines several unique advantages that make it suitable for our problem. First, it is a learning algorithm and thus, after the initial training, it can compute the desired output with a lower number of queries, compared to existing attacks. This is novel compared to existing attacks, which do not generalize from previous inputs and rely on other techniques, such as differential evolution [Das and Suganthan 2011; Storn and Price 1997] or random search [Karnopp 1963]. Second, it is suitable for learning concepts that are not differentiable, like one and few pixel attacks. Third, it enforces a rigid structure and thus must generalize within its language. Thereby, it is forced to identify the most vulnerable characteristics of the network. As a by-product of this structure, programs provide a high-level description of the network's weaknesses. For example, a program can identify that, for a certain network, perturbations in the center of the image are more likely to create successful adversarial examples, while for another network, perturbations of pixels with high RGB values are less likely to create successful adversarial examples. An interpretable structure can increase the confidence of users in the network, as well as provide guidelines for retraining the network to be more robust to attacks. Several adversarial attacks provide an easy-to-understand descriptions of a network's weaknesses (e.g., Casper et al. [2021]; Wang et al. [2022]), however, none focuses on one or few pixel attacks. For one pixel attacks, two works propose a post-hoc analysis of the network's weaknesses [Alatalo et al. 2022; Vargas and Su 2020]. While their analysis can draw similar conclusions to those drawn by our adversarial programs, these works do not leverage them to compute new adversarial examples. We note that

the idea of relying on program synthesis to provide an interpretable means to neural networks has been proposed by prior work [Hein et al. 2018; Verma et al. 2018], however, none focuses on adversarial attacks.

Challenges. Despite the advantages, program synthesis is not easily amenable to one and few pixel attacks because of several challenges. Some challenges are common for program synthesis tasks, while others are unique to one and few pixel attacks. The first challenge is to identify a small yet expressive domain-specific language (DSL) for the programs. While this challenge is very common in program synthesis tasks, it is especially challenging for adversarial attacks, because they are not typically computed using programming language constructs. The second challenge is identifying an efficient search algorithm to find a program in this DSL. This is also a common challenge, but because neural networks and their inputs range over real-valued numbers, the challenge is more complex. The third challenge is that the program synthesizer cannot simply search over a standard space of arithmetic expressions, conditions, and loops depending on the input. This is because the synthesized program must generate multiple candidate adversarial examples and submit them to the network until finding a successful candidate. Further, since the program's execution depends on the network's output for the candidate examples, the search space is not easily amenable to pruning as common in standard program synthesis tasks, where programs adhere to a partial-order. The last challenge is general to any algorithm addressing our problem: we aim to balance between conflicting goals: (1) finding a successful adversarial example, (2) minimizing the number of queries, and (3) in few pixel attacks, minimizing the number of perturbed pixels.

Prioritizing programs. To cope with the above challenges, we propose to rely on *prioritizing programs*. A prioritizing program considers every possible candidate adversarial example, and its goal is to quickly identify a successful candidate by relying on a dynamic prioritization of the candidates. A prioritizing program has a predefined structure, called a sketch [Solar-Lezama 2009], and its missing parts are the conditions defining the prioritization. Namely, the task of identifying a suitable DSL and a search algorithm is relevant only for instantiating these conditions. A prioritizing program also simplifies the challenge of handling the conflicting goals: it is guaranteed to find a successful adversarial example (if exists), and its goal is thus to minimize the number of queries. We begin by focusing on one pixel attacks, thereby ignoring the goal of minimizing the number of perturbed pixels. In Section 5, we explain how to extend our approach for few pixel attacks.

Program synthesis of prioritizing programs. Based on this idea, we rephrase our problem as a synthesis task. We are given a sketch of a prioritizing program for one pixel attacks and a DSL of conditions to complete the sketch. The synthesis task is to complete the sketch for a given network, such that when the program is invoked on new images, the number of queries posed to the network is minimal. As common in program synthesis and learning tasks, we assume a training set to guide the synthesizer, consisting of pairs of images and their true classes. We further assume a test set to evaluate the number of queries posed on new inputs. Note that unlike common synthesis tasks, where the synthesized program is checked by a user or a solver, a prioritizing program is correct-by-construction. Instead, its goal is to reduce the number of queries posed to the network. We next define it formally.

Definition 3.1 (Synthesis of Prioritizing Programs). Let P be a program sketch of a prioritizing program for one pixel attacks and $\mathcal{L}$ a DSL to complete P. Given a classifier N and training and test sets $\mathcal{D}_{Tr}$, $\mathcal{D}_{Ts}$, the goal of the synthesis is to complete P based on N and $\mathcal{D}_{Tr}$ such that the synthesized program minimizes the number of queries posed to N when evaluated on $\mathcal{D}_{Ts}$.

4 PRIORITIZING ADVERSARIAL PROGRAMS

In this section, we present the space of prioritizing programs we propose for one pixel attacks. We begin with defining the space of programs by defining a program sketch. We then present the language of conditions to instantiate the sketch.

4.1 A Sketch for One Pixel Attacks

In this section, we present our program sketch for computing one pixel attacks with a minimal number of queries. At a high-level, the flow of our sketch is as follows. It begins by initializing a priority queue with all possible location-perturbation pairs. Then, while the queue is not empty, it pops the first pair, perturbs the image accordingly, and submits to the network. If the perturbed image is a successful adversarial example, it returns this pair. Otherwise, it reorders the closest pairs to this (failed) pair. Some pairs are pushed to the back of the queue, and some pairs are conceptually pushed to the front of the queue by eagerly checking them. We next provide details.

Location-perturbation pairs. The complete space of perturbations is infinite, because every pixel's RGB channel can be perturbed to any value in $[0, 1]$. Even if we assume a finite computer representation, the number of possible perturbations is still too high for our sketch to explicitly maintain all possible location-perturbation pairs. Instead, we adopt the insight of Croce and Hein [2019] and Croce et al. [2022], who show that the vast majority of successful one and few pixel adversarial examples can be defined by a perturbation corresponding to one of the eight corners of the RGB color cube. Thus, we focus on perturbations in which every RGB channel is 0 or 1. In total, the number of possible location-perturbation pairs is $8 \cdot d_1 \cdot d_2$, where $d_1 \cdot d_2$ is the number of pixels. In Section 5, we extend our sketch to support finer granularity perturbations.

Distances over pairs and closest pairs. As described, the sketch iteratively pops a pair from the location-perturbation queue, and if its corresponding perturbed image is not a successful adversarial example, the sketch reorders its closest pairs. We define closest pairs with respect to two distance metrics over the pairs: one for the location and another for the pixel value. The distance between two locations $l_1 = (i_1, j_1), l_2 = (i_2, j_2) \in [d_1] \times [d_2]$ is their L_∞ distance: $||l_1 - l_2|| \triangleq \max\{|i_1 - i_2|, |j_1 - j_2|\}$. The distance between two pixels $p_1 = (r_1, g_1, b_1), p_2 = (r_2, g_2, b_2) \in [0, 1]^3$ is their L_1 distance: $||p_1 - p_2|| \triangleq |r_1 - r_2| + |g_1 - g_2| + |b_1 - b_2|$. Given a pixel p and the set $S = \{0, 1\}^3$ consisting of all eight corners of the RGB cube, the farthest pixel in S is the pixel p_1 maximizing the pixel distance from p. Similarly, the second farthest pixel in S is the next pixel maximizing the pixel distance from p, and so on. We next define the closest pairs. Given the queue of location-perturbation pairs L and the last pair (l, p) that has been popped (i.e., it is not in L): (1) the closest pairs with respect to the location are all pairs in L whose location distance from (l, p) is 1 and their perturbation is p, and (2) the closest pair with respect to the perturbation is the next pair in L whose location is l.

Conditions for reordering. Since there are two kinds of closest pairs and a pair can be pushed back or front, our sketch has four conditions. The goal of the conditions is to identify similarities between the failed pair and its closest pairs. Each condition can express a different kind of similarity, and thus some closest pairs are pushed back and others are pushed front. The conditions are defined over pixel locations, pixel values and the network's output for the failed pair (as described in Section 4.2).

One pixel attack sketch. We now present our sketch, depicted in Algorithm 1. The sketch takes as input a classifier N, an image x, and its true class c_x. It outputs a location-perturbation pair that corresponds to a successful adversarial example or $\perp$, if there is no such pair. The sketch begins by initializing a priority queue L with all the possible location-perturbation pairs by the following order. The primary order of the pairs is by the pixel distance from the corresponding pixel in

Algorithm 1: OnePixelSketch(N, x, c_x)

Input : A neural network classifier N, a colored image $x \in [0,1]^{d_1 \times d_2 \times 3}$ and its true class c_x.
Output: A location and perturbation of a successful adversarial example or $\perp$.

```
1  L = initialize()
2  while L ≠ ∅ do
3      (l, p) = pop(L)
4      if argmax(N(x[l ← p])) ≠ c_x then return (l, p)
5      if [B1] then pushBack(L, closest_loc(l, p))
6      if [B2] then pushBack(L, closest_pert(L, l))
7      locQ = pertQ = [(l, p)]
8      while locQ ≠ ∅ or pertQ ≠ ∅ do
9          while locQ ≠ ∅ do
10             (l', p') = pop(locQ)
11             if [B3] then
12                 for (l'', p'') ∈ closest_loc(l', p') do
13                     remove(L, (l'', p''))
14                     if argmax(N(x[l'' ← p''])) ≠ c_x then return (l'', p'')
15                     locQ = locQ :: [(l'', p'')]
16                     pertQ = pertQ :: [(l'', p'')]
17         while pertQ ≠ ∅ do
18             (l', p') = pop(pertQ)
19             if [B4] then
20                 for (l'', p'') ∈ closest_pert(L, l') do
21                     remove(L, (l'', p''))
22                     if argmax(N(x[l'' ← p''])) ≠ c_x then return (l'', p'')
23                     locQ = locQ :: [(l'', p'')]
24                     pertQ = pertQ :: [(l'', p'')]
25     return ⊥
```

x, from the farthest to the closest. That is, the first $d_1 \cdot d_2$ pairs have the farthest perturbation, the next $d_1 \cdot d_2$ pairs have the second farthest perturbation, and so on. A secondary order sorts the pairs by the pixel location, from the center of the image to the boundaries. For example, the first $d_1 \cdot d_2$ pairs are sorted by the location. We remind that we assume a black-box access to the classifier and thus we consider a prioritization relying only on the pairs' location and perturbation. However, if one considers a white-box access or additional networks, our prioritization can leverage images' semantic features (e.g., the features considered by Bhattad et al. [2020] and Hosseini and Poovendran [2018]) or the classifier's gradients (e.g., for computing adversarial saliency maps as proposed by Papernot et al. [2016] and Wiyatno and Xu [2018]). After the initialization, the sketch enters its main loop, which continues until L is empty. At each iteration, a location-perturbation pair (l, p) is popped from L. Then, its corresponding adversarial example is generated. The adversarial example, denoted $x[l \leftarrow p]$, is identical to x except that at location l it has value p. This example is submitted to N. If its classification is not c_x, the pair (l, p) is returned. Otherwise, its closest pairs

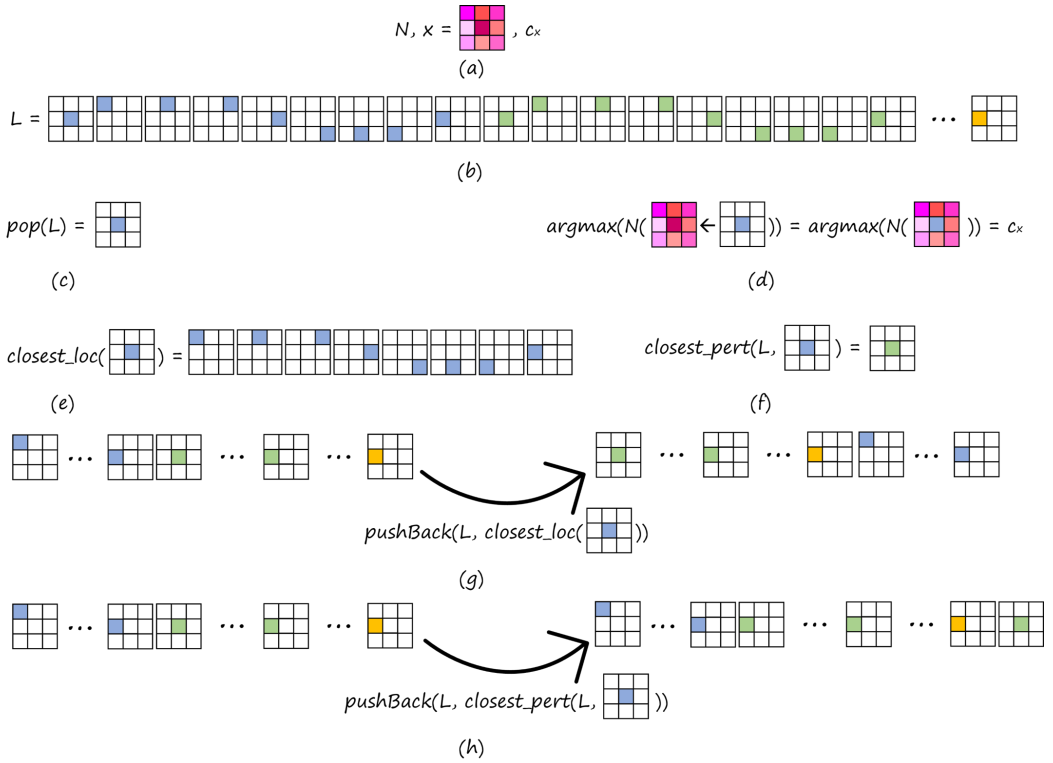

Fig. 1. Illustration of our sketch.

are reordered based on four conditions (to be synthesized). The goal of the first two conditions is to identify pairs whose corresponding adversarial examples are likely to be unsuccessful, and thus these pairs are pushed to the back of the queue. Similarly, the last two conditions identify pairs whose corresponding adversarial examples are likely to be successful and are thus conceptually pushed to the front. If the first condition is true, all closest pairs with respect to the location are pushed back (Line 5). If the second condition is true, the closest pair with respect to the perturbation is pushed back (Line 6). The sketch continues by identifying pairs to conceptually push to the front of L. This part is different from the previous reordering: it is defined iteratively and it eagerly checks pairs. It has two conditions, one for reordering the closest pairs with respect to the location and another one for reordering the closest pair with respect to the perturbation. If a condition is satisfied, then for every closest pair, a candidate adversarial example is generated and submitted to N. If a candidate is successful, its pair is returned. Otherwise, this process repeats for the closest pairs of this pair. Technically, this part begins by initializing a queue `locQ`, for the closest pairs with respect to the location, and `pertQ`, for the closest pairs with respect to the perturbation. Both queues are initialized with the (failed) pair (l, p). While the queues are not empty, a (failed) pair is popped from one of the queues (first from `locQ` and, after it becomes empty, from `pertQ`). Then, if a condition is satisfied, its closest pairs are eagerly checked. That is, every closest pair is removed from L and its corresponding example is submitted to N. If it is a successful adversarial example, the pair is returned. Otherwise, this pair is added to `locQ` and `pertQ`, to check its closest pairs. We note that the inner loops, Line 9–Line 16 and Line 17–Line 24, are identical, except that the first loop is for the closest pairs with respect to the location and the second loop is for the closest pair with respect to the perturbation.

Illustration. Figure 1 illustrates the operation of our sketch. The inputs are a network N, a 3×3 colored image x, and its true class c_x (Figure 1(a)). The priority queue L is initialized with all location-perturbation pairs, a pair for each location and a corner of the RGB cube (Figure 1(b)). The first nine pairs in L have the farthest perturbation from their corresponding pixel in x, and they are ordered by their distance from the center of the image. The next nine pairs in L have the second farthest perturbation from their corresponding pixel, and they are ordered by their distance from the center of the image. An iteration of the main loop pops a pair from L (Figure 1(c)) and submits the corresponding adversarial example to N (Figure 1(d)). If the example is not successful, it pushes to the back or front of the queue L the closest pixels with respect to the location (Figure 1(e)) or perturbation (Figure 1(f)). This is determined by a condition for each case. Figure 1(g) and Figure 1(h) show a push back to the closest pairs with respect to the location and perturbation.

Advantages. Our sketch has several advantages. First, it is guaranteed to find a successful adversarial example, among the possible location-perturbation pairs, if it exists. This is because every pair in the queue L is removed just before its corresponding example is submitted to the network. Second, the sketch has the flexibility of identifying whether a pair should be checked immediately or after other pairs. Third, an instantiated sketch provides insights in a high-level language about the robust and weak characteristics of the network.

4.2 The Condition Language

In this section, we describe the language of the sketch's conditions. We begin with a motivation for our language, then present its grammar and finally demonstrate its expressiveness.

Motivation. Our language consists of conditions over pixel locations, pixel values, and the network's output, which is all the information our adversarial programs have because of the black-box setting. The language leverages insights of prior works, providing a post-hoc analysis of the network's weaknesses for one pixel attacks [Alatalo et al. 2022; Vargas and Su 2020]. Alatalo et al. [2022] focus on medical imaging classifiers and analyze chromatic and spatial distributions of one pixel adversarial examples, successful and unsuccessful. They discretize the possible RGB values, and check whether each possibility leads to a successful one pixel adversarial example, over a given classifier and a set of images. They summarize the results and reach several conclusions. First, perturbing pixels close to the center of the image is more likely to lead to successful adversarial examples. Thus, our language includes a condition over the distance of a pixel location to the center of the image. Alatalo et al. [2022] also observe that changing a dark pixel, within a dark spot, sometimes leads to successful adversarial examples. Inspired by this observation, our language supports conditions requiring high or low values of the minimum, maximum and average of the RGB values. For example, to express a condition satisfied by very dark pixels, we can require that the maximum value is lower than a very small number. To express a condition satisfied by very bright pixels, we can require that the minimum value is greater than a very high number. We note that we experimented with conjunctions and disjunctions over these conditions, but observed that due to overfitting they performed worse. We also note that Alatalo et al. [2022] show that successful adversarial examples are typically obtained by pixels whose value is very far from nearby pixels. However, we do not introduce a condition for this case, because our initial ordering generalizes this property: it first considers all farthest perturbations, then the second farthest and so on. Vargas and Su [2020] focus on CIFAR-10 classifiers and provide a locality analysis. They show that nearby pixels have a similar level of vulnerability. Our language thus supports a condition that compares the difference in the network's confidence in the true class before and after the perturbation to a given number. For example, to express a condition satisfied by highly vulnerable pixels, we can require that this difference is greater than a high number. To express a condition satisfied by robust

$$
\begin{array}{rrcl}
\textit{(Condition)} & B & ::= & F > r \mid F < r \\
\textit{(Function)} & F & ::= & \max(p) \mid \min(p) \mid \text{avg}(p) \mid \text{score\_diff}(N(x_1), N(x_2), c') \mid \text{center}(l) \\
\textit{(Pixel)} & p & \in & [0,1]^3 \\
\textit{(Location)} & l & \in & [d_1] \times [d_2] \\
\textit{(Constant)} & r & \in & \mathbb{R} \\
\textit{(Network)} & N & \in & (\mathbb{R}^c)^{[0,1]^{d_1 \times d_2 \times 3}} \\
\textit{(Image)} & x & \in & [0,1]^{d_1 \times d_2 \times 3} \\
\textit{(Class)} & c' & \in & \{1, \ldots, c\}
\end{array}
$$

Fig. 2. Syntax of the condition language.

pixels, we can require that this difference is lower than a small number. That is, if $[B_3]$ or $[B_4]$ are instantiated with a condition requiring a high difference, satisfied by vulnerable pixels, the closest pairs are immediately checked. Similarly, if $[B_1]$ or $[B_2]$ are instantiated with a condition requiring a low difference, satisfied by robust pixels, the closest pairs are pushed to the back of the queue.

Grammar. Figure 2 shows the grammar of our condition language. A condition B is an inequality constraint over a function F and a real number r. A function F is: (1) `max`, `min`, or `avg` over a pixel $p \in [0,1]^3$, (2) `score_diff` that given two output vectors $N(x_1), N(x_2)$ and a class c' returns the difference between $N(x_1)_{c'}$ and $N(x_2)_{c'}$, and (3) `center` that given a location $l \in [d_1] \times [d_2]$ returns the location distance from the center of the image.

Example. As an example, consider the conditions synthesized for one of our adversarial programs:

- $[B_1]$: `score_diff(`$N(x), N(x[l \leftarrow p]), c_x$`) < 0.21`,
- $[B_2]$: `max(`x_l`) > 0.19`,
- $[B_3]$: `score_diff(`$N(x), N(x[l \leftarrow p]), c_x$`) > 0.25`,
- $[B_4]$: `center(`l`) < 8`.

The first condition leads to pushing to the back of the queue nearby pixels if the confidence difference is below 0.21, while the third condition leads to pushing to the front nearby pixels if the confidence difference is above 0.25. The second condition leads to pushing to the back of the queue the next pair in the queue with the same location if the pixel at x_l has an RGB channel that is above 0.19 (i.e., it is not very dark). The last condition leads to pushing to the front the next pair with the same location if the (failed) pair is close to the center of the image (the L_∞ distance is at most 7).

5 OPPSLA: A SYNTHESIZER FOR ONE PIXEL PROGRAM SKETCH FOR ATTACKS

In this section, we present OPPSLA, our synthesizer for instantiating the sketch presented in Section 4.1 with conditions over the language presented in Section 4.2. OPPSLA relies on a stochastic search, inspired by the Metropolis-Hastings algorithm [Chib and Greenberg 1995], that supports our typed conditions and unique scoring. Our scoring does not count successes, but rather the number of queries until reaching the successes. We begin with a background on Metropolis-Hastings and then introduce our adaptations. Afterwards, we introduce our extensions to support few pixel attacks and finer granularity perturbations.

The Metropolis-Hastings (MH) algorithm. The MH algorithm is a sampling algorithm based on Markov Chain Monte Carlo (MCMC) [Andrieu et al. 2003]. Similarly to other stochastic searches, e.g., random search [Karnopp 1963] and genetic search [Koza 1994], MH samples candidates based on a learned distribution over the search space. The learned distribution, updated based on the chosen candidates, provides an efficient guidance to find optimal solutions. The goal of MH is to effectively obtain random samples from a probability distribution that is difficult to directly sample.

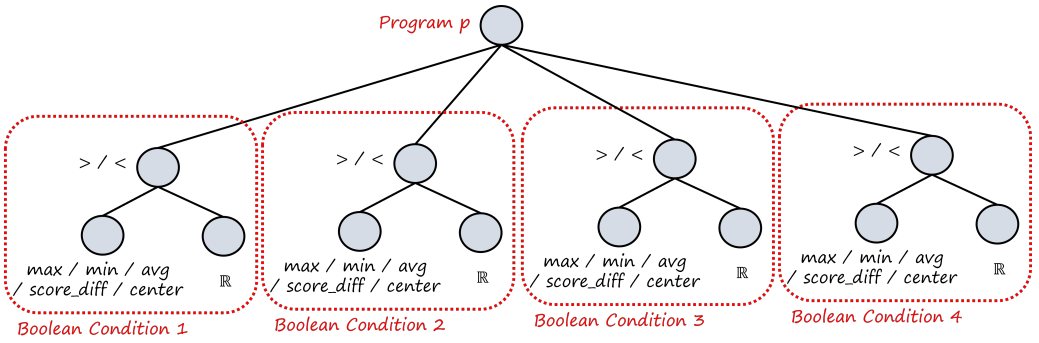

Fig. 3. Illustration of the abstract syntax tree representing a program in our search space.

Instead, MH relies on a score function $S : P \to \mathbb{R}$, mapping a candidate solution to a real-valued score. The advantage of MH is that it only requires the score function to be proportional to the probability density of the true probability distribution. MH operates iteratively. It begins from a random candidate p. At every iteration, it defines the next candidate p' by mutating the current candidate p. It keeps (accepts) p' with a probability determined by comparing the scores of p and p'. If it keeps p', it sets $p = p'$. Generally, MH is unaware of structures posed on the possible solutions (e.g., typed expressions). However, prior works leverage MH even for structured search spaces by enforcing the allowed structure by other means. Schkufza et al. [2013] leverage MH for loop-free binary superoptimization and enforce well-typed expressions by mutating a binary as follows: an operand is replaced by an operand and an operator is replaced by an operator. Their score function is based on the candidate's correctness (i.e., whether it is equivalent to the original binary) and performance. Thereby, their scoring enables MH to converge to a correct and more efficient binary. Goodman et al. [2008] leverage MH for analysis of human concept learning, where the search space is defined by first-order logical formulas. To enforce the structure of a formula when mutating a candidate, they represent formulas as parse trees and mutate node by node, based on their grammar. Their score function is the likelihood that the formula is evaluated to true for the observed examples, while considering the probability that some observed examples are outliers.

Our stochastic search. We design a stochastic search, inspired by MH, to synthesize conditions for our sketch. Our search space is the space of all programs that instantiate our sketch. We represent a program in this space as an abstract syntax tree, similarly to Gulwani et al. [2017], but only over the four conditions. The root is the program p and it has four children, one for each condition. Each condition node has two children, one for the function F and one for the number r. Figure 3 illustrates this tree. Given a program p, a mutation is computed by mutating its tree. A tree mutation begins by randomly picking a node (the root, a condition, a function or a real number), and mutating all nodes in its subtree. Each node (except the root) is replaced based on its respective grammar rule to guarantee that the overall condition is well-typed. Consequently, the mutated tree corresponds to a well-typed program in our search space. Figure 4 shows an example of a mutation.

The score function. We next define our score function. Recall that by construction, every instantiation of our sketch is guaranteed to find a successful adversarial example, if exists. The goal is to find such example with a minimal number of queries. Thus, we define the score function based on the average number of queries over a given training set. Formally, given a program p, its score is computed by first executing p on the classifier N and every pair of an image and its true class in the given training set. Let $\overline{Q}_p$ be the average number of queries submitted to N by p for inputs for which

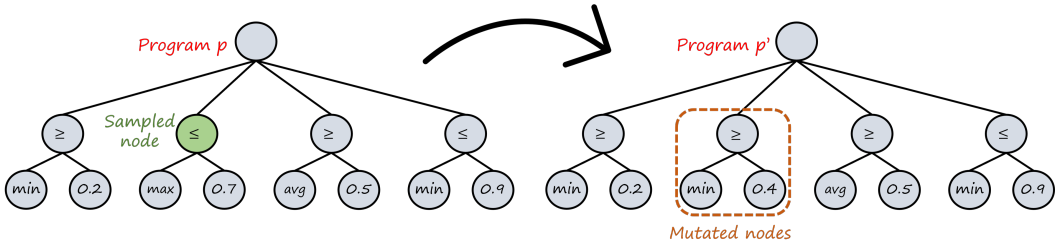

Fig. 4. An example of a tree and a mutated tree.

p finds a successful adversarial example (we ignore inputs for which p does not find a successful example, because their number of queries is fixed). Then, our score is $S(p) = exp(-\beta \cdot \overline{Q}_p)$, for $\beta \in \mathbb{R}^+$. Since $\overline{Q}_p \geq 0$, this is a positive, monotonically decreasing function. The maximal score is 1 and it is obtained for $\overline{Q}_p = 0$ (which is infeasible, since any program submits at least one query).

The algorithm of OPPSLA. Algorithm 2 presents OPPSLA. It takes as input a classifier N and a training set $\mathcal{D}_{Tr}$, consisting of pairs of images and their true classes. It returns an adversarial program. OPPSLA begins with a random program p, computed by instantiating our sketch with random well-typed conditions. It then computes the average number of queries of p by executing p on N and every pair in $\mathcal{D}_{Tr}$ and counting the number of queries posed for inputs for which p returns a successful adversarial example. Accordingly, the score S_p is computed. Then, OPPSLA begins a loop for `MAX_ITER` iterations (a hyper-parameter). An iteration begins by mutating p to define the next program candidate p', as described before. Then, the average number of queries of p' is computed (as described before), and accordingly the score $S_{p'}$ is computed. Then, p' is kept for the next iteration based on the learned distribution. Technically, a random number in $[0, 1]$ is sampled, and if it is smaller than the ratio of the new score and old score, p is set to p'. Namely, the smaller the average number of queries of p' compared to p, the higher the ratio, and thus the higher the probability of updating p with p'. We note that OPPSLA does not require a large training set: few dozens of pairs suffice to synthesize a program posing a low number of queries on average (Section 6). Also, if the training set of the classifier is not available, it is possible to synthesize a training set, for example as suggested by Li et al. [2020]; Papernot et al. [2017]; Zhang et al. [2022].

Few pixel attacks. We next explain how to extend OPPSLA to few pixel attacks. As defined in Definition 2.1, the goal is to compute an attack with a minimal number of queries, a minimal number of perturbed pixels, and with up to k perturbed pixels, where $k > 1$ is a small number. Naturally, the larger the k, the higher the success rate of the attack, but the perturbation is more perceptible. Our extension builds on CornerSearch [Croce and Hein 2019], which computes attacks with a minimal number of perturbed pixels, but does not aim to minimize the number of queries. CornerSearch begins by enumerating over all location-perturbation pairs and submits their corresponding examples to the network. It then sorts the pairs by the difference in the network's confidence in the true class and in the other classes. Next, it iteratively increases the number of perturbed pixels $k' \in \{2, \ldots, k\}$. For each k', it samples k' pairs based on a distribution determined by the confidence difference of the pairs. It repeats the sampling for a fixed number of iterations or until finding a successful adversarial example. OPPSLA is similar to CornerSearch but relies on the prioritization computed by the adversarial program and does not compute all one pixel adversarial examples. Technically, given an adversarial program and inputs N, x, c_x, OPPSLA executes the adversarial program until finding a successful candidate or until all pairs in the priority queue have been popped, removed or pushed back. Then, it computes the top-K pairs (K is a hyper-parameter) that have not been

Algorithm 2: OPPSLA (N, $\mathcal{D}_{Tr}$)

Input : A classifier N and a training set $\mathcal{D}_{Tr}$ consisting of pairs of images and their classes.
Output : An adversarial program.

```
p = random_program()                          // randomly instantiate the sketch
Q̄_p = 0                                       // the average number of queries of p
cnt_p = 0     // the number of successful adversarial examples generated by p
for (x, c_x) ∈ D_Tr do
    if p(N, x, c_x) ≠ ⊥ then
        Q̄_p += Q(p, N, x, 1)
        cnt_p += 1
Q̄_p /= cnt_p
S_p = e^{-β·Q̄_p}
for i = 1 to MAX_ITER do
    p' = mutate(p)                  // pick a node in the tree and mutate its subtree
    Q̄_p' = 0                                  // the average number of queries of p'
    cnt_p' = 0 // the number of successful adversarial examples generated by p'
    for (x, c_x) ∈ D_Tr do
        if p'(N, x, c_x) ≠ ⊥ then
            Q̄_p' += Q(p', N, x, 1)
            cnt_p' += 1
    Q̄_p' /= cnt_p'
    S_p' = e^{-β·Q̄_p'}
    if random([0, 1]) < S_p'/S_p then
        p = p'
        S_p = S_p'
return p
```

pushed back and maximize the confidence difference. It proceeds as CornerSearch. In Section 6, we show that our extension improves CornerSearch, indicating the effectiveness of our prioritization, determined by the synthesis.

Finer granularity perturbations. Our sketch considers as perturbations the eight corners of the RGB color cube. While it is very effective in practice, it sometimes misses successful adversarial examples obtained by other RGB values. We next explain how to extend OPPSLA to support finer granularity perturbations. Given a level g of granularity, for $g \in \{0, 1, 2, \ldots\}$, OPPSLA considers for every color channel the values $0, \frac{1}{2^g}, \frac{2}{2^g} \ldots, \frac{2^g}{2^g}$. For example, for $g = 0$, it considers the values $0, 1$, for $g = 1$, it considers $0, \frac{1}{2}, 1$, and for $g = 2$, it considers $0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1$. That is, for granularity g, the number of RGB values is $(2^g + 1)^3$, i.e., every increase of g increases the number of perturbations exponentially. While one can extend the possible perturbations in the priority queue L, this would result in a very high number of queries to the network, thus defeating the goal of our sketch. Instead, OPPSLA trades off success rate and number of queries and limits the number of pixel locations for which it considers finer granularity perturbations to `MAX_G` (a hyper-parameter). Thereby, the

number of additional queries posed to the network is bounded by $[(2^g+1)^3-8]\cdot$MAX_G. Technically, OPPSLA plants in the priority queue L a flag and runs the adversarial program, while recording the maximal confidence difference in the true class before and after the perturbation, for each pixel location. When it pops the flag from the queue, it attempts all finer granularity values (excluding the 0, 1 values) for the MAX_G locations with the highest confidence difference. The role of the flag is to check finer granularity perturbations only for pairs that have not been pushed to the back of L.

6 EVALUATION

In this section, we evaluate OPPSLA. We first describe the implementation and the evaluation setup. We then present our experiments that show that (1) OPPSLA obtains a state-of-the-art success rate with significantly fewer queries, (2) our conditions enable to reduce the number of queries, (3) our adversarial programs are transferable, (4) the number of synthesis queries posed to the classifier by OPPSLA is relatively low, (5) integrating OPPSLA in an adversarial training can reduce our attack's success rate with a minor impact on the classifier's accuracy, (6) our few pixel attacks require fewer queries, and (7) our finer granularity perturbations slightly improve the success rate.

Implementation and setup. We implemented OPPSLA in Python using Pytorch[5]. Our implementation supports GPU parallelization. Experiments ran on an Ubuntu 20.04 OS on a dual AMD EPYC 7742 server with 1TB RAM and eight NVIDIA GeForce RTX 2080 Ti GPUs. The hyper-parameters of Algorithm 2 are $\beta = 0.4$ and MAX_ITER = 210. For few pixel attacks, the top-K is top-100. We evaluate OPPSLA on two image datasets. First, CIFAR-10 [Krizhevsky et al. 2009], consisting of $32 \times 32 \times 3$ colored images, each is classified as one of ten possible classes. Second, ImageNet [Deng et al. 2009], consisting of $224 \times 224 \times 3$ colored images, each is classified as one of 1000 possible classes. For CIFAR-10, we use three pre-trained convolutional neural networks: VGG-16-BN [Simonyan and Zisserman 2015] (33.6M parameters, test accuracy 94%), ResNet18 [He et al. 2016] (11.2M parameters, test accuracy 93.07%), and GoogLeNet [Szegedy et al. 2015] (5.5M parameters, test accuracy 92.85%). For ImageNet, we use two pre-trained convolutional neural networks: DenseNet121 [Huang et al. 2017] (8M parameters, test accuracy 74.43%) and ResNet50 [He et al. 2016] (25.6M parameters, test accuracy 76.13%). For every CIFAR-10 network, we run OPPSLA with ten training sets, one for each class, each consisting of 50 images. We evaluate the adversarial programs over CIFAR-10's test set, consisting of 1000 images for each class (misclassified images are discarded). For every ImageNet network, we run OPPSLA with 11 training sets, each with a different class and consisting of only ten images. We evaluate the adversarial programs over a test set, consisting of 50 images for each class (misclassified images are discarded).

Baselines. We compare OPPSLA to several black-box one pixel and few pixel attacks. The first baseline is One Pixel Attack [Su et al. 2017], which we denote by SuOPA, relying on differential evolution to compute one pixel attacks. Unlike OPPSLA, SuOPA considers every perturbation in $[0, 1]^3$ (and not only the eight corners of the RGB color cube), and it does not aim to minimize the number of queries. We use the code from Torchattacks[6], with the default hyper-parameters. The second baseline is Sparse-RS [Croce et al. 2022], the state-of-the-art for one pixel and few pixel attacks. It relies on random search. As mentioned, its perturbations are limited to the eight corners of the RGB color cube. We use the authors' code, with the default hyper-parameters. The third baseline is CornerSearch [Croce and Hein 2019], a few pixel attack minimizing the number of perturbed pixels. It relies on local search and limits its perturbations to the eight corners of the RGB color cube. We use the authors' code, with the default hyper-parameters.

[5]The code is available at https://github.com/TomYuviler/OPPSLA
[6]https://github.com/Harry24k/adversarial-attacks-pytorch

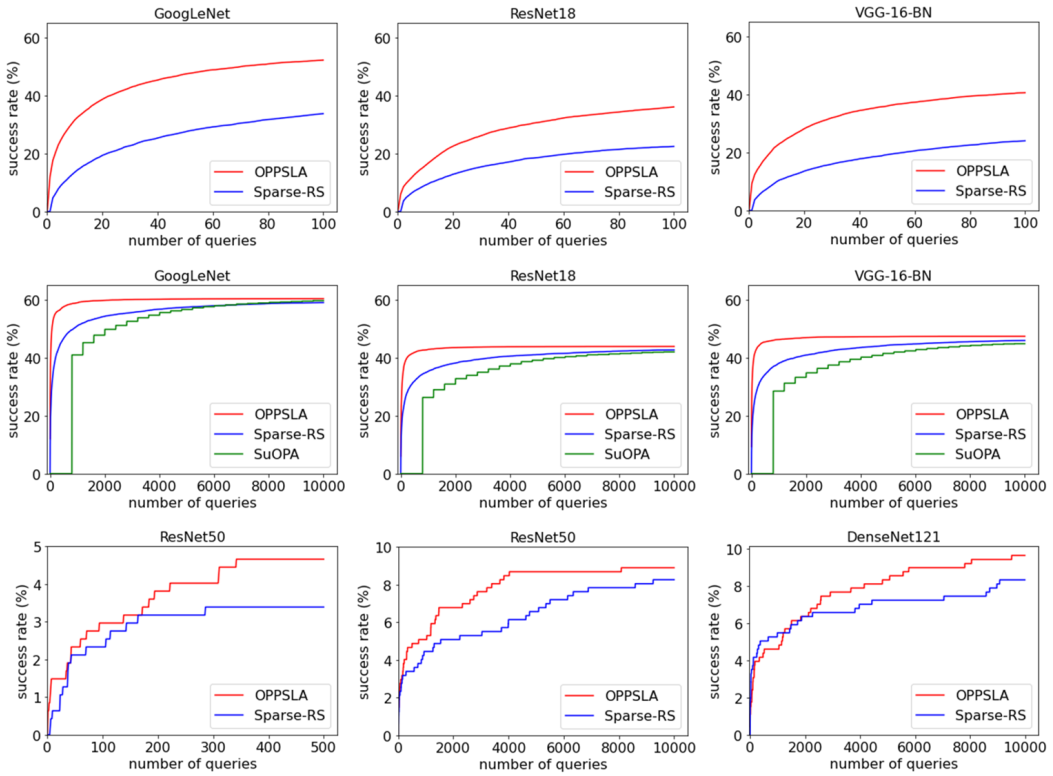

Fig. 5. OPPSLA vs. two baselines over one pixel attacks for CIFAR-10 classifiers and ImageNet classifiers.

One pixel attacks: comparison to baselines. We begin by comparing the one pixel attacks of OPPSLA and Sparse-RS on our CIFAR-10 and ImageNet classifiers. For our CIFAR-10 classifiers, we also compare to SuOPA. We let each approach run on all images in the test set. We record the success rate for every number of queries $q \in \{1, 2, \ldots, 10000\}$. We note that the minimal number of queries submitted by SuOPA is 400 (determined by `population_size`). We also note that for ImageNet classifiers the number of one pixel adversarial examples is over 400000, so our query limit ($q \leq 10000$) is very challenging. Figure 5 shows, for a given maximal number of queries (up to 100, 500, or 10000) and a classifier, the success rate over the test set. For CIFAR-10 classifiers, the results show that OPPSLA significantly outperforms both baselines in terms of computing minimal-query one pixel attacks. In particular, if the number of queries is smaller or equal than 100, OPPSLA success rate is 59% higher than Sparse-RS's success rate. When the number of queries is a few thousand, the success rate of both baselines comes close to OPPSLA's success rate. However, OPPSLA's success rate is still higher. Interestingly, even though SuOPA does not limit the possible perturbations to the eight corners of the RGB cube, its success rate is still lower than OPPSLA's. For ImageNet classifiers, OPPSLA generally outperforms Sparse-RS. Because the test set is relatively small, the curves are fluctuated, however, the results show that (1) for 10000 queries, OPPSLA's success rate is 12% higher than Sparse-RS's success rate, and (2) for a few hundred queries, OPPSLA often has a higher success rate.

Importance of the synthesized conditions. We continue by evaluating the importance of the sketch's conditions in reducing the number of queries. We compare OPPSLA's adversarial programs to the

Table 1. Impact of the synthesized conditions on the number of queries to the classifier.

Classifier	Approach	Average #Queries	Median #Queries
GoogLeNet	Synthesis	**104.07**	**9.0**
	`False`	393.20	18.0
ResNet18	Synthesis	**115.23**	**19.0**
	`False`	370.92	40.0
VGG-16-BN	Synthesis	**105.54**	**13.0**
	`False`	232.78	19.0

Table 2. Transferability: the number of queries when running programs synthesized for another classifier.

Synthesized for:	GoogLeNet	ResNet18	VGG-16-BN
Target	Average #Queries	Average #Queries	Average #Queries
GoogLeNet	**104.07**	135.32	140.92
ResNet18	215.16	**115.23**	139.00
VGG-16-BN	202.45	115.10	**105.54**

constant program that instantiates the sketch's conditions with `False`. That is, the prioritization of the location-perturbation pairs is fixed and determined by the initial ordering. We remind that the success rate of all instantiations is equal. Table 1 presents the results for our CIFAR-10 classifiers, where for each, we consider ten programs, one for each class. The results show that the synthesis significantly lowers the number of queries: the average is lowered by 3x and the median by 2x.

Transferability. Next, we study the transferability of our adversarial programs. Transferability is the ability of an adversarial attack, computed for a particular classifier, to be effective for other classifiers [Demontis et al. 2019; Hashemi et al. 2020; Wei et al. 2022; Zhou et al. 2018]. Transferable attacks are very important as often, in practice, attackers do not have an unlimited access to the attacked network. Instead, they prepare the attack based on a network that they trained by themselves for the same task. To evaluate the transferability of OPPSLA, we let it synthesize programs for one CIFAR-10 classifier and then use these programs to attack another CIFAR-10 classifier. We measure the average number of queries and check whether it remains relatively low (recall that the success rate is independent of the synthesis process). Table 2 reports the results. The diagonal shows the average number of queries when running the programs on the classifiers they are synthesized for (provided as a baseline). The results show that the programs synthesized for VGG-16-BN and ResNet18 are transferable with only a small increase to the number of queries. The program synthesized for GoogLeNet requires more queries for the other classifiers, however, the number of queries remains very small. We demonstrate this in Figure 6, showing for each target classifier, five approaches: the two baselines and the adversarial programs of each classifier. The results show that while, naturally, the best approach is to use the adversarial programs computed for the target classifier, the other adversarial programs obtain very close success rates given the same number of queries. In contrast, SuOPA and Sparse-RS obtain significantly lower success rates. This shows that OPPSLA is transferable, because its programs remain very query-efficient for unknown classifiers. This means that an attacker does not need to run OPPSLA for every classifier, but rather once for every set of classifiers trained for the same task.

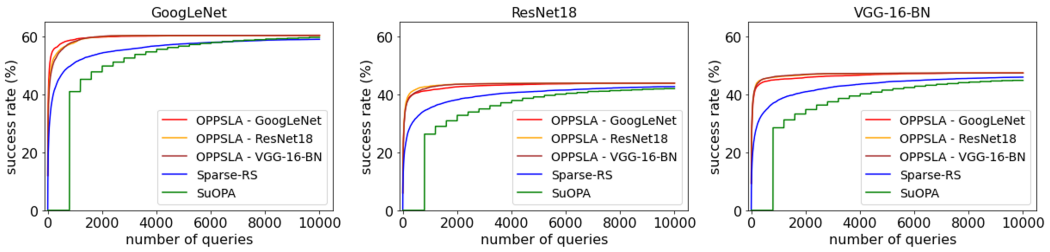

Fig. 6. Success rate of the transferred programs, compared to the original program and the baselines.

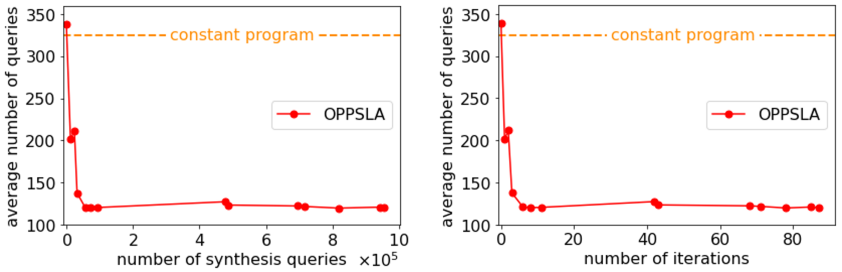

Fig. 7. Number of queries as a function of the number of synthesis queries (left) and iterations (right).

Number of queries posed by OPPSLA's synthesis. We next study how many queries OPPSLA requires to synthesize adversarial programs posing a low number of queries. Naturally, the synthesis process (Algorithm 2) introduces a high number of queries to the classifier. However, it runs once for a given classifier and a training set, and afterwards the user may invoke the adversarial program on as many images or classifiers (trained for the same task) as she wishes. In this experiment, we let OPPSLA synthesize an adversarial program for the VGG-16-BN classifier and a training set consisting of 50 images of the class *Airplane*. We limit OPPSLA to pose up to 1000000 synthesis queries to the classifier. During OPPSLA's synthesis process, we record the intermediate (accepted) programs p and the number of iteration in which they are generated. Then, we run each program on a test set consisting of 1000 *Airplane* images and measure the average number of queries. We compare these programs to the constant program whose conditions are `False` and its prioritization is thus fixed. That is, we compare to a baseline which does not require any synthesis queries. Figure 7 shows the average number of queries (on the test set) as a function of the number of synthesis queries posed until generating the program p (left) and as a function of the number of iterations (right). Results show that already with ~50000 synthesis queries, posed during six iterations, OPPSLA reduces the average number of queries by 2.7x compared to the constant program. Afterwards, more synthesis queries and iterations generally lower the average number of queries, but by up to 0.8%. We note that allowing more synthesis queries shows the same trend, which is why we focus, in this experiment, on up to 1000000 synthesis queries. We further note that the meaning of 50000 synthesis queries, in six iterations, on a training set of 50 images implies an average of 1000 queries per image over all six iterations run by Algorithm 2. This is about 2% of the maximum number of queries which can be posed in six iterations, since the maximal number of queries for each image in every iteration is 8192.

Adversarial training. We continue by studying OPPSLA's performance on classifiers trained to be robust. In this experiment, we consider two approaches to increase the robustness of the VGG-16-BN classifier. First, we employ randomized ablation [Levine and Feizi 2020], which increases

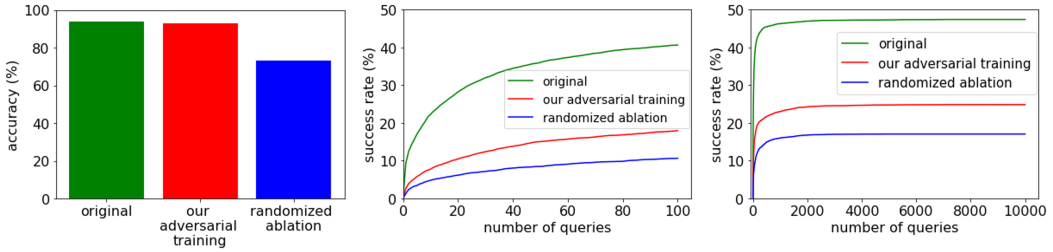

Fig. 8. The accuracy of classifiers trained to be robust and the success rate of OPPSLA on these classifiers.

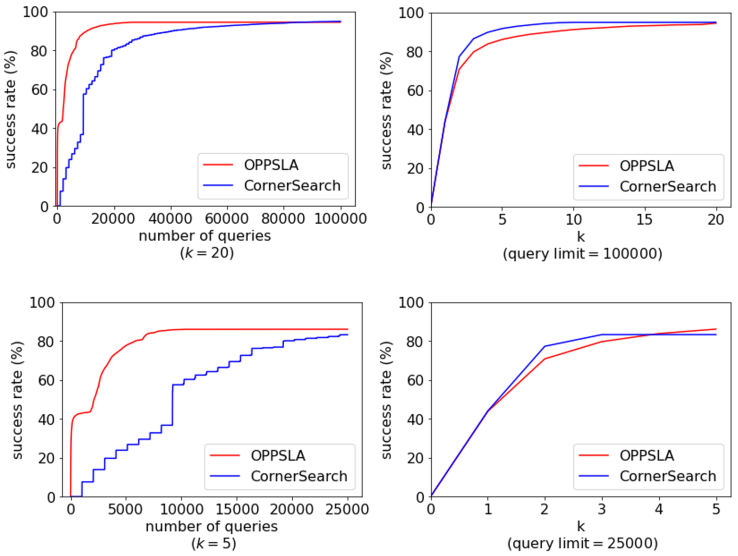

Fig. 9. OPPSLA vs. CornerSearch for few pixel attacks on ResNet18.

robustness to few pixel adversarial attacks. Randomized ablation employs smoothing during training and inference as follows. Given an input image, several variants are generated, each darkens a different set of pixels. All variants are submitted to the classifier, and the input's classification is the majority vote. For the training and inference, we rely on the authors' code. For OPPSLA's adversarial programs, we generate the variants upon each query to the classifier and determine the classification by the majority vote. Second, we consider an adversarial training based on our approach, defined as follows: (1) it runs OPPSLA on the original classifier, (2) it executes the resulting adversarial program on 2000 training inputs (200 of each class), (3) it adds the resulting adversarial examples to the training set, and (4) it retrains the classifier for 30 more epochs. Figure 8 (left) shows the accuracy of the three classifiers. It shows that our adversarial training leads to a minor decrease in the classifier's accuracy, in contrast to the other robust classifier. Next, we run OPPSLA on the original classifier and on both robust classifiers. Figure 8 (middle / right) shows, for a given maximal number of queries, up to 100 / 10000, the success rate over the test set. Results show that our adversarial training reduces our attack's success rate by 56% / 48% compared to our attack's success rate on the original classifier. Random ablation reduces our attack's success rate by 74% / 64%, compared to the original classifier. However, it comes with a cost: it reduces the classifier's accuracy by 22%, while our approach for adversarial training only reduces it by 1%.

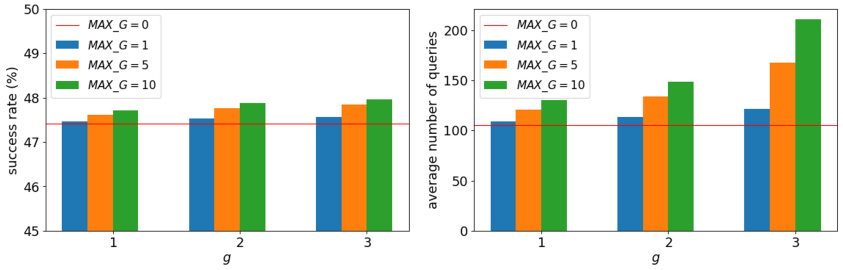

Fig. 10. Effect of allowing OPPSLA finer granularity perturbations on VGG-16-BN.

Few pixel attacks. Next, we compare the few pixel attacks of OPPSLA and CornerSearch [Croce and Hein 2019], on which OPPSLA builds. We run both approaches on the ResNet18 classifier, for different bounds on the number of perturbed pixels $k \in \{2, \ldots, 20\}$. Figure 9 shows the success rate as a function of the number of queries (left) and as a function of the bound k (right). The results show that, given a large number of queries, the success rates of both approaches are similar, regardless of the bound on the number of perturbed pixels. However, OPPSLA computes few pixel attacks with 4.7x fewer queries. This demonstrates the effectiveness of our dynamic prioritization.

Finer granularity perturbations. Lastly, we study OPPSLA's performance given perturbations of finer granularity than the eight corners of the RGB cube. We run OPPSLA with $g \in \{1, 2, 3\}$ and $\texttt{MAX\_G} \in \{1, 5, 10\}$ for the VGG-16-BN classifier. Figure 10 shows the success rate and the average number of queries as a function of g and MAX_G. Results show that the success rate slightly improves, by less than 1%. As expected, the number of queries increases when increasing g and MAX_G.

7 RELATED WORK

In this section, we discuss the closely related work.

L_0 attacks. Several works present one and few pixel attacks as well as general L_0 attacks, in a black-box setting. Su et al. [2017] design a one pixel attack based on differential evolution. In parallel, Narodytska and Kasiviswanathan [2017] propose to rely on random search to compute one and few pixel attacks. CornerSearch [Croce and Hein 2019] is a few pixel attack minimizing the number of perturbed pixels using local search. Sparse-RS [Croce et al. 2022] relies on random search to compute few pixel attacks, for a given number of perturbed pixels, which minimizes the number of queries posed to the network. Other kinds of L_0 attacks aim to minimize the number of perturbed pixels. Pointwise [Schott et al. 2019] minimizes the L_0 norm by finding a large noise causing a misclassification and then iteratively unperturbs pixels as long as the misclassification remains. Pixle [Pomponi et al. 2022] generates adversarial examples by rearranging a small number of pixels using random search. Other kinds of attacks minimize the L_0 norm and another metric (e.g., L_∞) to make the perturbations less visible [Croce and Hein 2019; Nguyen-Son et al. 2021]. Several works propose white-box L_0 attacks. GreedyFool [Dong et al. 2020] selects the most effective candidate positions for perturbation and iteratively removes the less effective perturbations. JSMA [Papernot et al. 2016] relies on saliency maps to select the most effective pixels for perturbation. Croce and Hein [2019] propose a white-box attack that projects the adversarial noise generated by the PGD attack [Madry et al. 2018] onto the L_0-ball. Carlini and Wagner [2017] propose an iterative algorithm to identify pixels with a larger effect on the classifier's output based on an L_2 attack.

Universal adversarial attacks. A universal adversarial attack, introduced by Moosavi-Dezfooli et al. [2017], computes a general perturbation whose goal is to cause a misclassification when

added to any input. The first universal attack computes the general perturbation by iteratively accumulating perturbations defined by every input in a given dataset. Other attacks rely on singular vectors [Khrulkov and Oseledets 2018] or Generative Adversarial Networks (GANs) [Hashemi et al. 2020; Mopuri et al. 2018]. Other works propose universal attacks that can be physically executed by stickers or patches [Brown et al. 2017; Casper et al. 2021; Liu et al. 2020; Zolfi et al. 2021]. Zhang et al. [2021] provides a survey on universal attacks. Our adversarial programs can be viewed as a customized universal attack. Like universal attacks, we rely on a precomputation (synthesis) phase to compute the attack (the program), and afterwards there is no costly optimization process for every new input. Unlike existing universal attacks, our adversarial programs do not rely on a constant perturbation, but rather they customize it for every given input.

Query-efficient attacks. Several black-box attacks minimize the number of queries to the network. Square Attack [Andriushchenko et al. 2020] proposes L_2 and L_∞ attacks generated by localized squared perturbations randomly selected. NP-Attack [Bai et al. 2020] proposes an L_∞ attack utilizing high-level image structure information to model the distribution of adversarial examples efficiently. Ilyas et al. [2018] propose an L_∞ attack based on Natural Evolutionary Strategies (introduced by Wierstra et al. [2008]). Li et al. [2018] leverage active learning to minimize the number of queries.

Program synthesis. OPPSLA builds on the sketch technique [Solar-Lezama 2009], which restricts the program space by a predefined template, and the synthesizer's goal is to complete the missing parts. Sketch has been shown to be successful in a variety of applications including code completions [Raychev et al. 2014], arithmetic and sorting programs [Srivastava et al. 2013], entity matching rules [Singh et al. 2017], lists manipulation [Feser et al. 2015] and reinforcement learning policies [Verma et al. 2018]. OPPSLA relies on the Metropolis-Hastings algorithm to synthesize boolean conditions for the sketch. Prior works for synthesizing conditional expressions rely on divide-and-conquer [Alur et al. 2017; Ferdowsifard et al. 2021], general decidable refinement types [Polikarpova et al. 2016], unification constraints [Alur et al. 2015], and goal graphs [Albarghouthi et al. 2013].

8 CONCLUSION

We presented OPPSLA, a synthesizer for generating adversarial programs for one and few pixel attacks. Our key insight is to identify a program sketch for one pixel attacks, checking the possible candidate adversarial examples according to a dynamic prioritization. The dynamic prioritization is determined by conditions, synthesized for a given network and a training set. OPPSLA completes the sketch such that the resulting program minimizes the number of queries on the training set. To this end, it employs a stochastic search, inspired by the Metropolis-Hastings algorithm, that synthesizes well-typed conditions and considers a score function evaluating candidate programs by their average number of queries. We show how to extend OPPSLA to support few pixel attacks and finer granularity perturbations. We evaluate OPPSLA on several CIFAR-10 and ImageNet classifiers and compare it to existing attacks. Results show that OPPSLA obtains a state-of-the-art success rate with significantly fewer queries. For example, given a 100 query limit, its success rate on CIFAR-10 classifiers is 59% higher than the success rate of the current state-of-the-art. We demonstrate that our adversarial programs are transferable to other classifiers, with a small increase to the number of queries. We also show that the dynamic prioritization of the adversarial programs, determined by the synthesized conditions, improves a prior minimal-perturbation attack. Lastly, we show that finer granularity perturbations enable to slightly improve the success rate, but require more queries.

ACKNOWLEDGMENTS

We thank our shepherd Satish Chandra and the anonymous reviewers for their insightful feedback. This research was supported by the Israel Science Foundation (grant No. 2605/20).

REFERENCES

Janne Alatalo, Joni Korpihalkola, Tuomo Sipola, and Tero Kokkonen. 2022. Chromatic and Spatial Analysis of One-Pixel Attacks Against an Image Classifier. *In NETYS* (2022). https://doi.org/10.1007/978-3-031-17436-0_20

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. *In CAV* (2013). https://doi.org/10.1007/978-3-642-39799-8_67

Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. 2015. Synthesis Through Unification. *In CAV* (2015). https://doi.org/10.1007/978-3-319-21668-3_10

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. *In TACAS* (2017). https://doi.org/10.1007/978-3-662-54577-5_18

Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. 2003. An Introduction to MCMC for Machine Learning. *Mach. Learn.* 50, 1-2 (2003), 5–43. https://doi.org/10.1023/A:1020281327116

Maksym Andriushchenko, Francesco Croce, Nicolas Flammarion, and Matthias Hein. 2020. Square Attack: A Query-Efficient Black-Box Adversarial Attack via Random Search. *In ECCV* (2020). https://doi.org/10.1007/978-3-030-58592-1_29

Yang Bai, Yuyuan Zeng, Yong Jiang, Yisen Wang, Shu-Tao Xia, and Weiwei Guo. 2020. Improving Query Efficiency of Black-Box Adversarial Attack. *In ECCV* (2020). https://doi.org/10.1007/978-3-030-58595-2_7

Anand Bhattad, Min Jin Chong, Kaizhao Liang, Bo Li, and David A. Forsyth. 2020. Unrestricted Adversarial Examples via Semantic Manipulation. *In ICLR* (2020).

Tom B. Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. 2017. Adversarial Patch. *CoRR* abs/1712.09665 (2017). arXiv:1712.09665 http://arxiv.org/abs/1712.09665

Nicholas Carlini and David A. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. *In IEEE Symposium on Security and Privacy, SP* (2017). https://doi.org/10.1109/SP.2017.49

Stephen Casper, Max Nadeau, and Gabriel Kreiman. 2021. One Thing to Fool them All: Generating Interpretable, Universal, and Physically-Realizable Adversarial Features. *CoRR* abs/2110.03605 (2021). arXiv:2110.03605 https://arxiv.org/abs/2110.03605

Siddhartha Chib and Edward Greenberg. 1995. Understanding the metropolis-hastings algorithm. *The american statistician* 49, 4 (1995), 327–335.

Francesco Croce, Maksym Andriushchenko, Naman D. Singh, Nicolas Flammarion, and Matthias Hein. 2022. Sparse-RS: A Versatile Framework for Query-Efficient Sparse Black-Box Adversarial Attacks. *In AAAI* (2022). https://ojs.aaai.org/index.php/AAAI/article/view/20595

Francesco Croce and Matthias Hein. 2019. Sparse and Imperceivable Adversarial Attacks. *In ICCV* (2019). https://doi.org/10.1109/ICCV.2019.00482

Swagatam Das and Ponnuthurai Nagaratnam Suganthan. 2011. Differential Evolution: A Survey of the State-of-the-Art. *IEEE Trans. Evol. Comput.* 15, 1 (2011), 4–31. https://doi.org/10.1109/TEVC.2010.2059031

Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. 2019. Why Do Adversarial Attacks Transfer? Explaining Transferability of Evasion and Poisoning Attacks. *In USENIX Security Symposium* (2019). https://www.usenix.org/conference/usenixsecurity19/presentation/demontis

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. *In CVPR* (2009). https://doi.org/10.1109/CVPR.2009.5206848

Xiaoyi Dong, Dongdong Chen, Jianmin Bao, Chuan Qin, Lu Yuan, Weiming Zhang, Nenghai Yu, and Dong Chen. 2020. GreedyFool: Distortion-Aware Sparse Adversarial Attack. *In NeurIPS* (2020). https://proceedings.neurips.cc/paper/2020/hash/8169e05e2a0debcb15458f2cc1eff0ea-Abstract.html

Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. LooPy: interactive program synthesis with control structures. *In OOPSLA* (2021). https://doi.org/10.1145/3485530

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *In PLDI* (2015). https://doi.org/10.1145/2737924.2737977

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. *In ICLR* (2015). http://arxiv.org/abs/1412.6572

Noah D. Goodman, Joshua B. Tenenbaum, Jacob Feldman, and Thomas L. Griffiths. 2008. A Rational Analysis of Rule-Based Concept Learning. *Cogn. Sci.* 32, 1 (2008), 108–154. https://doi.org/10.1080/03640210701802071

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Found. Trends Program. Lang.* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

Atiye Sadat Hashemi, Andreas Bär, Saeed Mozaffari, and Tim Fingscheidt. 2020. Transferable Universal Adversarial Perturbations Using Generative Models. *CoRR* abs/2010.14919 (2020). arXiv:2010.14919 https://arxiv.org/abs/2010.14919

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. *In CVPR* (2016). https://doi.org/10.1109/CVPR.2016.90

Daniel Hein, Steffen Udluft, and Thomas A. Runkler. 2018. Interpretable policies for reinforcement learning by genetic programming. *In Eng. Appl. Artif. Intell.* (2018). https://doi.org/10.1016/j.engappai.2018.09.007

Hossein Hosseini and Radha Poovendran. 2018. Semantic Adversarial Examples. *In CVPR Workshops* (2018). http://openaccess.thecvf.com/content_cvpr_2018_workshops/w32/html/Hosseini_Semantic_Adversarial_Examples_CVPR_2018_paper.html

Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. *In CVPR* (2017). https://doi.org/10.1109/CVPR.2017.243

Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. 2018. Black-box Adversarial Attacks with Limited Queries and Information. *In ICML* (2018). http://proceedings.mlr.press/v80/ilyas18a.html

Dean C. Karnopp. 1963. Random search techniques for optimization problems. *Autom.* 1, 2-3 (1963), 111–121. https://doi.org/10.1016/0005-1098(63)90018-9

Valentin Khrulkov and Ivan V. Oseledets. 2018. Art of Singular Vectors and Universal Adversarial Perturbations. *In CVPR* (2018). http://openaccess.thecvf.com/content_cvpr_2018/html/Khrulkov_Art_of_Singular_CVPR_2018_paper.html

John R Koza. 1994. Genetic programming as a means for programming computers by natural selection. *Statistics and computing* 4, 2 (1994), 87–112.

Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. 2017. Adversarial examples in the physical world. *In ICLR* (2017). https://openreview.net/forum?id=HJGU3Rodl

Alexander Levine and Soheil Feizi. 2020. Robustness Certificates for Sparse Adversarial Attacks by Randomized Ablation. *In AAAI* (2020). https://ojs.aaai.org/index.php/AAAI/article/view/5888

Pengcheng Li, Jinfeng Yi, and Lijun Zhang. 2018. Query-Efficient Black-Box Attack by Active Learning. *In ICDM* (2018). https://doi.org/10.1109/ICDM.2018.00159

Qizhang Li, Yiwen Guo, and Hao Chen. 2020. Practical No-box Adversarial Attacks against DNNs. *In NeurIPS* (2020). https://proceedings.neurips.cc/paper/2020/hash/96e07156db854ca7b00b5df21716b0c6-Abstract.html

Aishan Liu, Jiakai Wang, Xianglong Liu, Bowen Cao, Chongzhi Zhang, and Hang Yu. 2020. Bias-Based Universal Adversarial Patch Attack for Automatic Check-Out. *In ECCV* (2020). https://doi.org/10.1007/978-3-030-58601-0_24

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. *In ICLR* (2018). https://openreview.net/forum?id=rJzIBfZAb

Laurent Meunier, Jamal Atif, and Olivier Teytaud. 2019. Yet another but more efficient black-box adversarial attack: tiling and evolution strategies. *CoRR* abs/1910.02244 (2019). arXiv:1910.02244 http://arxiv.org/abs/1910.02244

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. 2017. Universal Adversarial Perturbations. *In CVPR* (2017). https://doi.org/10.1109/CVPR.2017.17

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks. *In CVPR* (2016). https://doi.org/10.1109/CVPR.2016.282

Konda Reddy Mopuri, Utkarsh Ojha, Utsav Garg, and R. Venkatesh Babu. 2018. NAG: Network for Adversary Generation. *In CVPR* (2018). http://openaccess.thecvf.com/content_cvpr_2018/html/Mopuri_NAG_Network_for_CVPR_2018_paper.html

Nina Narodytska and Shiva Prasad Kasiviswanathan. 2017. Simple Black-Box Adversarial Attacks on Deep Neural Networks. *In CVPR workshop* (2017). https://doi.org/10.1109/CVPRW.2017.172

Hoang-Quoc Nguyen-Son, Tran Phuong Thao, Seira Hidano, Vanessa Bracamonte, Shinsaku Kiyomoto, and Rie Shigetomi Yamaguchi. 2021. OPA2D: One-Pixel Attack, Detection, and Defense in Deep Neural Networks. *In IJCNN* (2021). https://doi.org/10.1109/IJCNN52387.2021.9534332

Nicolas Papernot, Patrick D. McDaniel, Ian J. Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. 2017. Practical Black-Box Attacks against Machine Learning. *In AsiaCCS* (2017). https://doi.org/10.1145/3052973.3053009

Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. 2016. The Limitations of Deep Learning in Adversarial Settings. *In IEEE European Symposium on Security and Privacy, EuroS&P* (2016). https://doi.org/10.1109/EuroSP.2016.36

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *In PLDI* (2016). https://doi.org/10.1145/2908080.2908093

Jary Pomponi, Simone Scardapane, and Aurelio Uncini. 2022. Pixle: a fast and effective black-box attack based on rearranging pixels. *In IJCNN* (2022). https://doi.org/10.1109/IJCNN55064.2022.9892966

Hao Qiu, Leonardo Lucio Custode, and Giovanni Iacca. 2021. Black-box adversarial attacks using evolution strategies. *In GECCO* (2021). https://doi.org/10.1145/3449726.3463137

Weize Quan, Deeraj Nagothu, Nihal A. Poredi, and Yu Chen. 2021. CriPI: an efficient critical pixels identification algorithm for fast one-pixel attacks. *In Defense + Commercial Sensing* (2021). https://doi.org/10.1117/12.2581377

Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. *In PLDI* (2014). https://doi.org/10.1145/2594291.2594321

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *In ASPLOS* (2013). https://doi.org/10.1145/2451116.2451150

Lukas Schott, Jonas Rauber, Matthias Bethge, and Wieland Brendel. 2019. Towards the first adversarially robust neural network model on MNIST. *In ICLR* (2019). https://openreview.net/forum?id=S1EHOsC9tX

Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. *In ICLR* (2015). http://arxiv.org/abs/1409.1556

Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed K. Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. 2017. Synthesizing Entity Matching Rules by Examples. *In Proc. VLDB Endow.* (2017). http://www.vldb.org/pvldb/vol11/p189-singh.pdf

Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. *In APLAS* (2009). https://doi.org/10.1007/978-3-642-10672-9_3

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2013. Template-based program verification and program synthesis. *In Int. J. Softw. Tools Technol. Transf.* (2013). https://doi.org/10.1007/s10009-012-0223-4

Rainer Storn and Kenneth V. Price. 1997. Differential Evolution - A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *J. Glob. Optim.* 11, 4 (1997), 341–359. https://doi.org/10.1023/A:1008202821328

Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. 2017. One pixel attack for fooling deep neural networks. *CoRR* abs/1710.08864 (2017). arXiv:1710.08864 http://arxiv.org/abs/1710.08864

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. *In CVPR* (2015). https://doi.org/10.1109/CVPR.2015.7298594

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. *In ICLR* (2014). http://arxiv.org/abs/1312.6199

Pedro Tabacof and Eduardo Valle. 2016. Exploring the space of adversarial images. *In IJCNN* (2016). https://doi.org/10.1109/IJCNN.2016.7727230

Danilo Vasconcellos Vargas and Jiawei Su. 2020. Understanding the One Pixel Attack: Propagation Maps and Locality Analysis. *In IJCAI-PRICAI workshop* (2020). http://ceur-ws.org/Vol-2640/paper_4.pdf

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. 2018. Programmatically Interpretable Reinforcement Learning. *In ICML* (2018). https://proceedings.mlr.press/v80/verma18a.html

Yixiang Wang, Jiqiang Liu, Xiaolin Chang, Ricardo J. Rodríguez, and Jianhua Wang. 2022. DI-AA: An interpretable white-box attack for fooling deep neural networks. *In Inf. Sci.* (2022). https://doi.org/10.1016/j.ins.2022.07.157

Zhipeng Wei, Jingjing Chen, Micah Goldblum, Zuxuan Wu, Tom Goldstein, and Yu-Gang Jiang. 2022. Towards Transferable Adversarial Attacks on Vision Transformers. *in AAAI* (2022). https://ojs.aaai.org/index.php/AAAI/article/view/20169

Daan Wierstra, Tom Schaul, Jan Peters, and Jürgen Schmidhuber. 2008. Natural Evolution Strategies. *In CEC* (2008). https://doi.org/10.1109/CEC.2008.4631255

Rey Wiyatno and Anqi Xu. 2018. Maximal Jacobian-based Saliency Map Attack. *CoRR* abs/1808.07945 (2018). http://arxiv.org/abs/1808.07945

Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. 2019. Adversarial Examples: Attacks and Defenses for Deep Learning. *In IEEE Trans. Neural Networks Learn. Syst.* (2019). https://doi.org/10.1109/TNNLS.2018.2886017

Chaoning Zhang, Philipp Benz, Chenguo Lin, Adil Karjauv, Jing Wu, and In So Kweon. 2021. A Survey on Universal Adversarial Attack. *In IJCAI* (2021). https://doi.org/10.24963/ijcai.2021/635

Qilong Zhang, Chaoning Zhang, Chaoqun Li, Jingkuan Song, Lianli Gao, and Heng Tao Shen. 2022. Practical No-box Adversarial Attacks with Training-free Hybrid Image Transformation. *CoRR* abs/2203.04607 (2022). https://doi.org/10.48550/arXiv.2203.04607

Wen Zhou, Xin Hou, Yongjun Chen, Mengyun Tang, Xiangqi Huang, Xiang Gan, and Yong Yang. 2018. Transferable Adversarial Perturbations. *In ECCV* (2018). https://doi.org/10.1007/978-3-030-01264-9_28

Alon Zolfi, Moshe Kravchik, Yuval Elovici, and Asaf Shabtai. 2021. The Translucent Patch: A Physical and Universal Attack on Object Detectors. *In CVPR* (2021). https://openaccess.thecvf.com/content/CVPR2021/html/Zolfi_The_Translucent_Patch_A_Physical_and_Universal_Attack_on_Object_CVPR_2021_paper.html

Received 2022-11-10; accepted 2023-03-31