One Pixel Adversarial Attacks via Sketched Programs

Tom Yuviler [1] **Dana Drachsler-Cohen** [1]

Abstract

Neural networks are susceptible to adversarial examples, including one pixel attacks. Existing one pixel attacks iteratively generate candidate adversarial examples and submit them to the network until finding a successful candidate. However, current attacks require a very large number of queries, which is infeasible in many practical settings. In this work, we leverage program synthesis and identify an expressive program sketch that enables the computation of adversarial examples using significantly fewer queries. We introduce OPPSLA, a synthesizer that instantiates the sketch with customized conditions. Experimental results show that OPPSLA achieves a state-of-the-art success rate while requiring an order of magnitude fewer queries than existing attacks.

1. Introduction

Over the past decade, many works have demonstrated that deep neural networks (DNNs) are susceptible to adversarial example attacks (Goodfellow et al., 2015; Kurakin et al., 2017; Szegedy et al., 2014; Yuan et al., 2019; Madry et al., 2018). Su et al. (2017) consider an extremely limited setting for attacking an image classifier 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; Nguyen-Son et al., 2021; Quan et al., 2021; Croce & Hein, 2019; Croce et al., 2022). 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 generate candidate adversarial examples and submit them to the network, until finding a successful candidate. However, existing approaches necessitate thousands of queries or more, 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, Microsoft Azure Computer Vision API, Clarifai moderation-recognition API and Google Cloud Vision API allow at most 5000 queries for free. To make few pixel attacks more practical, the Sparse-RS attack (Croce et al., 2022) aims at minimizing the number of queries posed to the network. Sparse-RS obtains a state-of-the-art success rate 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 5). 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. Our programs are based on a program sketch (Solar-Lezama, 2009). Like prior one pixel attacks, an adversarial program generates candidate adversarial examples and submits them to the network. However, a synthesized program dynamically prioritizes the adversarial example candidates, in order to reduce the number of queries. This is obtained by the synthesized conditions. Unlike prior one pixel attacks, the large number of queries is only required for the synthesis process. We introduce OPPSLA, a synthesizer for *One Pixel Program Sketch for adversariaL Attacks*. OPPSLA employs a stochastic search inspired by Metropolis-Hastings (Chib & Greenberg, 1995). Our search produces grammatically correct programs, minimizing the number of queries.

We evaluate OPPSLA on CIFAR-10 and ImageNet networks and show that it obtains a state-of-the-art success rate. Further, it requires significantly fewer queries than Sparse-RS (Croce et al., 2022), the current state-of-the-art attack that minimizes the number of queries. We further show that the number of queries posed to the classifier during the synthesis is relatively low. Lastly, we show that our adversarial programs transfer to other networks with

[1]Technion, Israel. Correspondence to: Tom Yuviler <tom.yuviler@campus.technion.ac.il>, Dana Drachsler-Cohen <ddana@ee.technion.ac.il>.

Presented at the 2nd Workshop on Formal Verification of Machine Learning, co-located with the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA., 2023. Copyright 2023 by the author(s).

a relatively small increase to the number of queries. This makes our adversarial programs practical, because attackers need not synthesize a program for every network.

2. Problem Definition

In this section, we define our problem. An image classifier is a function mapping a colored image to a score vector over the possible classes $C = \{1, \dots, c\}$, that is: $N : [0, 1]^{d_1 \times d_2 \times 3} \to \mathbb{R}^c$. We focus on classifiers implemented by a neural network, whose output layer returns a vector $N(x) \in \mathbb{R}^c$ assigning a score for each class. Given an input x, the classification of the network is the class with the highest score, $c' = \text{argmax}(N(x))$.

In one pixel attacks, an attacker perturbs a single pixel with the goal of causing the network to misclassify. Formally, given a classifier N and an image $x \in [0, 1]^{d_1 \times d_2 \times 3}$ correctly-classified as class c_x, the goal 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, they can perturb every RGB channel. We address the problem of computing one pixel attacks in a black-box setting (the attacker can only submit inputs to the classifier to obtain their output), with a minimal number of queries.

3. Key Idea: Adversarial Programs

To compute minimal-query one pixel attacks, 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. Note that a prioritizing program is guaranteed to find a successful adversarial example, if exists one in our space of perturbations, and its goal is to minimize the number of queries. We next present our sketch and then the language of conditions.

3.1. A Sketch for One Pixel Attacks

At a high-level, the flow of our sketch is as follows (the full sketch is in Appendix A). It first initializes 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.

Our Perturbation Space 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 Sparse-RS (Croce et al., 2022), showing that the vast majority of successful one 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.

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.

3.2. The Condition Language

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 weak-

$$\begin{array}{rlll}
\textit{(Program)} & P & ::= & (B_1, B_2, B_3, B_4) \\
\textit{(Condition)} & B & ::= & F > r \mid F < r \\
\textit{(Function)} & F & ::= & \max(p) \mid \min(p) \mid \text{avg}(p) \mid \\
 & & & \text{score_diff}(N(x_1), N(x_2), c') \mid \\
 & & & \text{center}(l) \\
\textit{(Pixel)} & p & \in & [0,1]^3 \\
\textit{(Location)} & l & \in & [d_1] \times [d_2] \\
\textit{(Constant)} & r & \in & \mathbb{R} \\
\textit{(Network)} & N & \in & (\mathbb{R}^c)^{[0,1]^{d_1 \times d_2 \times 3}} \\
\textit{(Image)} & x & \in & [0,1]^{d_1 \times d_2 \times 3} \\
\textit{(Class)} & c' & \in & \{1, \ldots, c\}
\end{array}$$

Figure 1. Syntax of the condition language.

nesses for one pixel attacks (Alatalo et al., 2022; Vargas & Su, 2020). Alatalo et al. (2022) show that 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. Vargas & 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.

Grammar Figure 1 shows the grammar of our condition language. A program P consists of four conditions. 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, these are the four 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).

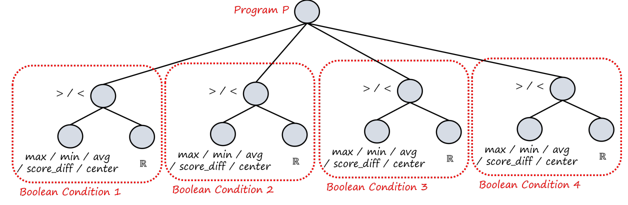

Figure 2. Illustration of the abstract syntax tree representing a program in our search space.

4. OPPSLA: A Synthesizer for our Sketch

We next present OPPSLA, a synthesizer for instantiating the sketch with conditions (its algorithm is in Appendix B). OPPSLA relies on a stochastic search, inspired by the Metropolis-Hastings algorithm (Chib & Greenberg, 1995), that supports our typed conditions and unique scoring. Our scoring does not count successes, but rather the number of queries until reaching them. We begin with a background on Metropolis-Hastings and then introduce our adaptations.

The Metropolis-Hastings (MH) Algorithm The MH algorithm is a sampling algorithm based on Markov Chain Monte Carlo (MCMC) (Andrieu et al., 2003). The goal of MH is to effectively obtain random samples from a probability distribution that is difficult to directly sample. Instead, MH relies on a score function $S : \mathcal{P} \to \mathbb{R}$, mapping a candidate solution to a real-valued score. 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; Goodman et al., 2008).

Our Stochastic Search Our stochastic search is inspired by MH. Its search space is 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 2 illustrates this tree. Given a program P, a mutation is computed by mutating its tree. A tree mutation

involves uniformly randomly selecting a node (the root, a condition, a function or a real number), and mutating all nodes in its subtree. Each non-root node 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.

The Score Function By construction, every instantiation of our sketch finds a successful adversarial example, if exists in our perturbation space. The goal is to find such an 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 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 $\overline{Q}_P = 0$.

5. Evaluation

We evaluate OPPSLA[1] on CIFAR-10 (Krizhevsky et al., 2009) and ImageNet (Deng et al., 2009). For CIFAR-10, we use three pre-trained convolutional neural networks: VGG-16-BN (Simonyan & Zisserman, 2015), ResNet18 (He et al., 2016), and GoogLeNet (Szegedy et al., 2015). For ImageNet, we use two pre-trained convolutional neural networks: DenseNet121 (Huang et al., 2017) and ResNet50 (He et al., 2016). 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 (great white shark, tiger shark, hammerhead, electric ray, stingray, cock, hen, house finch, junco, bulbul, jay) and consisting of ten images. We evaluate the adversarial programs over a test set, consisting of 50 images for each class (misclassified images are discarded).

Comparison to Baselines We compare OPPSLA to Sparse-RS (Croce et al., 2022), the state-of-the-art for one pixel and few pixel attacks, and to One Pixel Attack (Su et al., 2017), denoted by SuOPA. Unlike OPPSLA and Sparse-RS, 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 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 3 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. For ImageNet classifiers, 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.

Table 1. Transferability: the number of queries when running programs synthesized for another classifier.

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

Transferability Transferability is the ability of an adversarial attack, computed for a particular classifier, to be effective for other classifiers (Hashemi et al., 2020; Zhou et al., 2018; Demontis et al., 2019; Wei et al., 2022). Transferable attacks are practical since the attacker does not need to have an unlimited access to the attacked network. Instead, they prepare the attack on a network that they trained by themselves. To evaluate OPPSLA's transferability, 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). Table 1 shows the results. The diagonal is the baseline, showing the average number of queries when running the programs on the classifiers they are synthesized for. 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 programs synthesized for GoogLeNet require more queries, however, the number of queries remains very small.

Synthesis Queries Lastly, we study how many queries OPPSLA requires during the synthesis. We remind that OPPSLA runs once for a given classifier and a training set,

[1] `https://github.com/TomYuviler/OPPSLA`

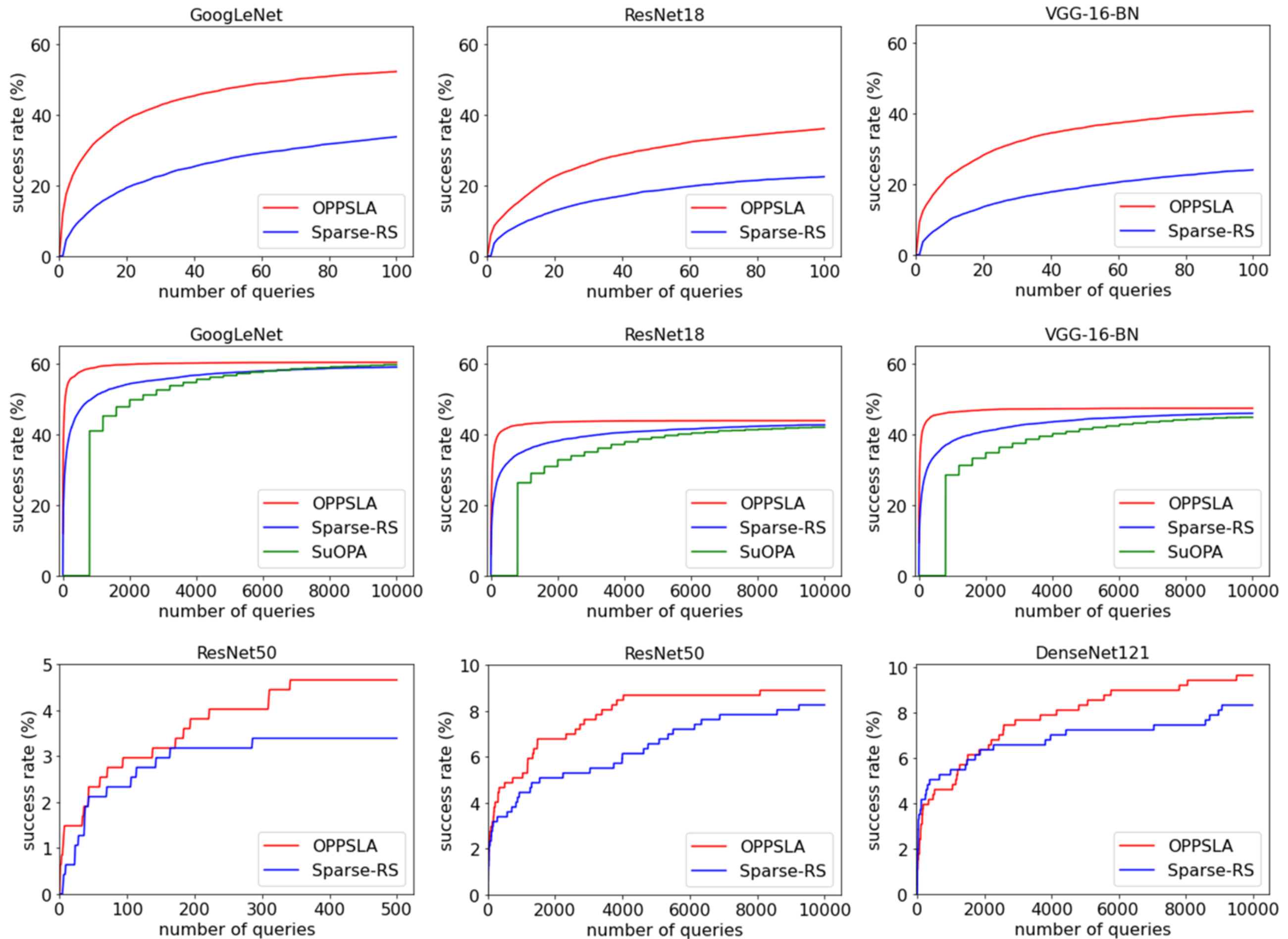

Figure 3. OPPSLA vs. two baselines over one pixel attacks for CIFAR-10 classifiers and ImageNet classifiers.

and afterwards users may invoke the adversarial program on as many images or classifiers (trained for the same task) as they wish. In this experiment, we let OPPSLA synthesize an adversarial program for the VGG-16-BN classifier and a training set consisting of 50 *Airplane* images. We limit it to 10^6 queries. We record the intermediate (accepted) programs P. Then, we run each program on a test set consisting of 1000 *Airplane* images and measure the average number of queries. We compare to a fixed-prioritization program (all conditions are `False`), which does not pose any synthesis queries. Figure 4 shows the average number of queries 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. Namely, the average is 1000 queries per image over all six iterations. 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. Afterwards, more synthesis queries and iterations lower the average number of queries, but by up to 0.8%.

Importance of the Conditions and Stochastic Search Appendix C provides an additional experiment showing that our synthesized conditions and stochastic search enable OPPSLA to reduce the number of queries.

6. Related Work

One Pixel Attacks Several works present one and few pixel attacks. Su et al. (2017) design a one pixel attack based on differential evolution. Narodytska & Kasiviswanathan (2017) rely on random search to compute one and few pixel attacks. CornerSearch (Croce & 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.

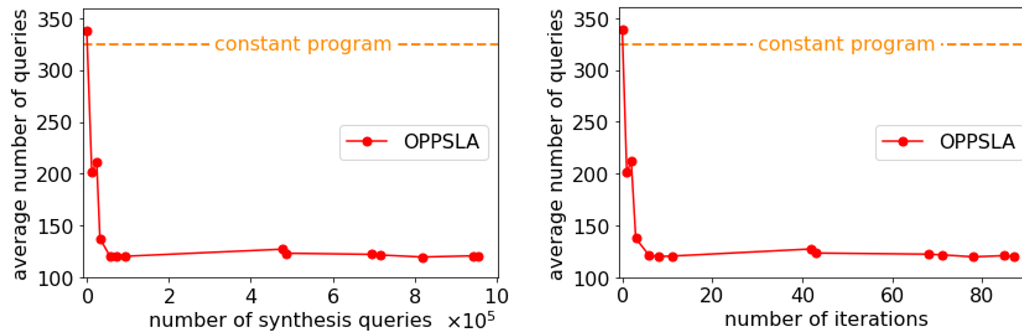

Figure 4. Number of queries as a function of the number of synthesis queries (left) and iterations (right).

Query-Efficient Attacks Several 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 (Wierstra et al., 2008). Li et al. (2018) leverage active learning to minimize the number of queries.

Program Synthesis The sketch technique (Solar-Lezama, 2009) has been shown to be successful in many applications, e.g., 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). 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).

7. Conclusion

We presented OPPSLA, a synthesizer for generating adversarial programs for one 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 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. We also demonstrate that our adversarial programs are transferable to other classifiers, with a small increase to the number of queries.

References

Alatalo, J., Korpihalkola, J., Sipola, T., and Kokkonen, T. Chromatic and spatial analysis of one-pixel attacks against an image classifier. *In NETYS*, 2022. URL `https://doi.org/10.1007/978-3-031-17436-0_20`.

Albarghouthi, A., Gulwani, S., and Kincaid, Z. Recursive program synthesis. *In CAV*, 2013. URL `https://doi.org/10.1007/978-3-642-39799-8_67`.

Alur, R., Cerný, P., and Radhakrishna, A. Synthesis through unification. *In CAV*, 2015. URL `https://doi.org/10.1007/978-3-319-21668-3_10`.

Alur, R., Radhakrishna, A., and Udupa, A. Scaling enumerative program synthesis via divide and conquer. *In TACAS*, 2017. URL `https://doi.org/10.1007/978-3-662-54577-5_18`.

Andrieu, C., de Freitas, N., Doucet, A., and Jordan, M. I. An introduction to MCMC for machine learning. *Mach. Learn.*, 50(1-2):5–43, 2003. URL `https://doi.org/10.1023/A:1020281327116`.

Andriushchenko, M., Croce, F., Flammarion, N., and Hein, M. Square attack: A query-efficient black-box adversarial attack via random search. *In ECCV*, 2020. URL `https://doi.org/10.1007/978-3-030-58592-1_29`.

Bai, Y., Zeng, Y., Jiang, Y., Wang, Y., Xia, S., and Guo, W. Improving query efficiency of black-box adversarial attack. *In ECCV*, 2020. URL `https://doi.org/10.1007/978-3-030-58595-2_7`.

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

Croce, F. and Hein, M. Sparse and imperceivable adversarial attacks. *In ICCV*, 2019. URL https://doi.org/10.1109/ICCV.2019.00482.

Croce, F., Andriushchenko, M., Singh, N. D., Flammarion, N., and Hein, M. Sparse-rs: A versatile framework for query-efficient sparse black-box adversarial attacks. *In AAAI*, 2022. URL https://ojs.aaai.org/index.php/AAAI/article/view/20595.

Demontis, A., Melis, M., Pintor, M., Jagielski, M., Biggio, B., Oprea, A., Nita-Rotaru, C., and Roli, F. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. *In USENIX Security Symposium*, 2019. URL https://www.usenix.org/conference/usenixsecurity19/presentation/demontis.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. *In CVPR*, 2009. URL https://doi.org/10.1109/CVPR.2009.5206848.

Ferdowsifard, K., Barke, S., Peleg, H., Lerner, S., and Polikarpova, N. Loopy: interactive program synthesis with control structures. *In OOPSLA*, 2021. URL https://doi.org/10.1145/3485530.

Feser, J. K., Chaudhuri, S., and Dillig, I. Synthesizing data structure transformations from input-output examples. *In PLDI*, 2015. URL https://doi.org/10.1145/2737924.2737977.

Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples. *In ICLR*, 2015. URL http://arxiv.org/abs/1412.6572.

Goodman, N. D., Tenenbaum, J. B., Feldman, J., and Griffiths, T. L. A rational analysis of rule-based concept learning. *Cogn. Sci.*, 32(1):108–154, 2008. URL https://doi.org/10.1080/03640210701802071.

Gulwani, S., Polozov, O., and Singh, R. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017. URL https://doi.org/10.1561/2500000010.

Hashemi, A. S., Bär, A., Mozaffari, S., and Fingscheidt, T. Transferable universal adversarial perturbations using generative models. *CoRR*, abs/2010.14919, 2020. URL https://arxiv.org/abs/2010.14919.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *In CVPR*, 2016. URL https://doi.org/10.1109/CVPR.2016.90.

Huang, G., Liu, Z., van der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. *In CVPR*, 2017. URL https://doi.org/10.1109/CVPR.2017.243.

Ilyas, A., Engstrom, L., Athalye, A., and Lin, J. Black-box adversarial attacks with limited queries and information. *In ICML*, 2018. URL http://proceedings.mlr.press/v80/ilyas18a.html.

Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. 2009.

Kurakin, A., Goodfellow, I. J., and Bengio, S. Adversarial examples in the physical world. *In ICLR*, 2017. URL https://openreview.net/forum?id=HJGU3Rodl.

Li, P., Yi, J., and Zhang, L. Query-efficient black-box attack by active learning. *In ICDM*, 2018. URL https://doi.org/10.1109/ICDM.2018.00159.

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., and Vladu, A. Towards deep learning models resistant to adversarial attacks. *In ICLR*, 2018. URL https://openreview.net/forum?id=rJzIBfZAb.

Narodytska, N. and Kasiviswanathan, S. P. Simple black-box adversarial attacks on deep neural networks. *In CVPR workshop*, 2017. URL https://doi.org/10.1109/CVPRW.2017.172.

Nguyen-Son, H., Thao, T. P., Hidano, S., Bracamonte, V., Kiyomoto, S., and Yamaguchi, R. S. OPA2D: one-pixel attack, detection, and defense in deep neural networks. *In IJCNN*, 2021. URL https://doi.org/10.1109/IJCNN52387.2021.9534332.

Polikarpova, N., Kuraj, I., and Solar-Lezama, A. Program synthesis from polymorphic refinement types. *In PLDI*, 2016. URL https://doi.org/10.1145/2908080.2908093.

Quan, W., Nagothu, D., Poredi, N. A., and Chen, Y. Cripi: an efficient critical pixels identification algorithm for fast one-pixel attacks. *In Defense + Commercial Sensing*, 2021. URL https://doi.org/10.1117/12.2581377.

Raychev, V., Vechev, M. T., and Yahav, E. Code completion with statistical language models. *In PLDI*, 2014. URL https://doi.org/10.1145/2594291.2594321.

Schkufza, E., Sharma, R., and Aiken, A. Stochastic superoptimization. *In ASPLOS*, 2013. URL https://doi.org/10.1145/2451116.2451150.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *In ICLR*, 2015. URL http://arxiv.org/abs/1409.1556.

Singh, R., Meduri, V. V., Elmagarmid, A. K., Madden, S., Papotti, P., Quiané-Ruiz, J., Solar-Lezama, A., and Tang, N. Synthesizing entity matching rules by examples. *In Proc. VLDB Endow.*, 2017. URL http://www.vldb.org/pvldb/vol11/p189-singh.pdf.

Solar-Lezama, A. The sketching approach to program synthesis. *In APLAS*, 2009. URL https://doi.org/10.1007/978-3-642-10672-9_3.

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

Su, J., Vargas, D. V., and Sakurai, K. One pixel attack for fooling deep neural networks. *CoRR*, abs/1710.08864, 2017. URL http://arxiv.org/abs/1710.08864.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. J., and Fergus, R. Intriguing properties of neural networks. *In ICLR*, 2014. URL http://arxiv.org/abs/1312.6199.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. *In CVPR*, 2015. URL https://doi.org/10.1109/CVPR.2015.7298594.

Vargas, D. V. and Su, J. Understanding the one pixel attack: Propagation maps and locality analysis. *In IJCAI-PRICAI workshop*, 2020. URL http://ceur-ws.org/Vol-2640/paper_4.pdf.

Verma, A., Murali, V., Singh, R., Kohli, P., and Chaudhuri, S. Programmatically interpretable reinforcement learning. *In ICML*, 2018. URL https://proceedings.mlr.press/v80/verma18a.html.

Wei, Z., Chen, J., Goldblum, M., Wu, Z., Goldstein, T., and Jiang, Y. Towards transferable adversarial attacks on vision transformers. *in AAAI*, 2022. URL https://ojs.aaai.org/index.php/AAAI/article/view/20169.

Wierstra, D., Schaul, T., Peters, J., and Schmidhuber, J. Natural evolution strategies. *In CEC*, 2008. URL https://doi.org/10.1109/CEC.2008.4631255.

Yuan, X., He, P., Zhu, Q., and Li, X. Adversarial examples: Attacks and defenses for deep learning. *In IEEE Trans. Neural Networks Learn. Syst.*, 2019. doi: 10.1109/TNNLS.2018.2886017. URL https://doi.org/10.1109/TNNLS.2018.2886017.

Zhou, W., Hou, X., Chen, Y., Tang, M., Huang, X., Gan, X., and Yang, Y. Transferable adversarial perturbations. *In ECCV*, 2018. URL https://doi.org/10.1007/978-3-030-01264-9_28.

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 [B_1] then pushBack(L, closest_loc(l, p))
6      if [B_2] 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 [B_3] 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 [B_4] 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 ⊥
```

A. One Pixel Attack Sketch

Our sketch (Algorithm 1) 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 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. 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 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

Figure 5. Illustration of our sketch.

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 5 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 5(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 5(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 5(c)) and submits the corresponding adversarial example to N (Figure 5(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 5(e)) or perturbation (Figure 5(f)). This is determined by a condition for each case. Figure 5(g) and Figure 5(h) show a push back to the closest pairs with respect to the location and perturbation.

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

```
Input  : A classifier N and a training set 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
```

B. The Algorithm of OPPSLA

OPPSLA (Algorithm 2) 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'.

C. Importance of the Synthesized Conditions and Stochastic Search

In this section, we present an additional experiment showing the importance of our conditions and stochastic search in reducing the number of queries. We compare OPPSLA's adversarial programs to several baselines. First, to understand the importance of the synthesized conditions, we compare to a 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. Second, to understand the importance of our stochastic search, we compare it to a random baseline that randomly samples 210 program instantiations (since OPPSLA runs the stochastic search for 210 iterations) and returns the one with the minimal number of queries posed to the classifier on the training set. Note that OPPSLA and these two baselines have the same success rate, since the success rate of all instantiations of our sketch is equal. The last baseline is Sparse-RS, the current

Table 2. Impact of the synthesized conditions and the stochastic search on the number of queries to the classifier.

Classifier	Approach	Average #Queries	Median #Queries
GoogLeNet	OPPSLA	**104.07**	**9.0**
	Sketch+`False`	393.20	18.0
	Sketch+Random	114.16	10.0
	Sparse-RS	557.20	62.0
ResNet18	OPPSLA	**115.23**	**19.0**
	Sketch+`False`	370.92	40.0
	Sketch+Random	186.03	26.0
	Sparse-RS	690.62	78.0
VGG-16-BN	OPPSLA	**105.54**	**13.0**
	Sketch+`False`	232.78	19.0
	Sketch+Random	152.12	**13.0**
	Sparse-RS	706.10	84.0

state-of-the-art. We let each approach run on every CIFAR-10 classifier and every CIFAR-10 class' training set. Table 2 presents the results. Compared to the constant program (Sketch+`False`), OPPSLA significantly lowers the number of queries: the average is lowered by 3x and the median by 2x. Compared to the random baseline (Sketch+Random), OPPSLA lowers the number of queries by 1.4x and the median by 1.2x. Sparse-RS incurs a significantly higher number of queries compared to OPPSLA and the other baselines.