They Might NOT Be Giants
Crafting Black-Box Adversarial Examples Using Particle Swarm Optimization

Rayan Mosli[1,2], Matthew Wright[1], Bo Yuan[1], and Yin Pan[1]

[1] Golisano College of Computing and Information Sciences, Rochester Institute of Technology, Rochester, New York 14623 {rhm6501, matthew.wright, bo.yuan, yin.pan}@rit.edu

[2] Faculty of Computing and Information Technology
King Abdul-Aziz University, Jeddah, Saudi Arabia

Abstract. As machine learning is deployed in more settings, including in security-sensitive applications such as malware detection, the risks posed by adversarial examples that fool machine-learning classifiers have become magnified. *Black-box* attacks are especially dangerous, as they only require the attacker to have the ability to query the target model and observe the labels it returns, without knowing anything else about the model. Current black-box attacks either have low success rates, require a high number of queries, produce adversarial images that are easily distinguishable from their sources, or are not flexible in controlling the outcome of the attack. In this paper, we present AdversarialPSO,[3] a black-box attack that uses few queries to create adversarial examples with high success rates. AdversarialPSO is based on Particle Swarm Optimization, a gradient-free evolutionary search algorithm, with special adaptations to make it effective for the black-box setting. It is flexible in balancing the number of queries submitted to the target against the quality of the adversarial examples. We evaluated AdversarialPSO on CIFAR-10, MNIST, and Imagenet, achieving success rates of 94.9%, 98.5%, and 96.9%, respectively, while submitting numbers of queries comparable to prior work. Our results show that black-box attacks can be adapted to favor fewer queries or higher quality adversarial images, while still maintaining high success rates.

1 Introduction

Deep learning (DL) is being used to solve a wide variety of problems in many different domains, such as image classification [**?**], malware detection [**?**], speech recognition [**?**], and medical imaging based diagnosis [**?**]. Despite state-of-the-art performance, DL models have been shown to suffer from a general flaw that makes them vulnerable to external attack. Adversaries can manipulate models to misclassify inputs by applying small perturbations to samples at test time [**?**]. These *adversarial examples* have also been successfully demonstrated against

[3] Code available: https://github.com/rhm6501/AdversarialPSOImages

real-world black-box targets, where adversaries would perform remote queries on a classifier to develop and verify their attack samples [**?**]. The possibility of such attacks poses a significant risk to any ML application, especially in security-critical settings or life-threatening environments.

Early adversarial attacks relied on model gradients to create examples [**?**,**?**,**?**], which requires internal knowledge of the target model. Since some adversarial examples transfer from one model to another [**?**], limited black-box attacks are possible using model gradients, but with low success rates [**?**]. More recent approaches either estimate the model's gradients [**?**,**?**,**?**,**?**] or iteratively apply perturbations to the input [**?**,**?**,**?**]. As demonstrated by Moon et al., however, the success rate of gradient-estimation approaches depends heavily on the choice of hyperparameters [**?**]. Consequently, attack methods with fewer hyperparameters to set or potentially tune would be less sensitive to hyperparameter values and thus more practical in a black-box setting where tuning might be impossible.

In practice, the feasibility of a black-box attack also depends greatly on the number of required queries submitted to the model. Against machine-learning-as-a-service (MLaaS) platforms like Google Vision, each query has a monetary cost. Too many queries make the attack costly. Perhaps more importantly, needing too many queries could trigger a monitor to detect an attack underway by observing many subtly modified versions of the same image submitted to the system in a short period. To evade such a monitor, one could conduct the attack very slowly or use a large number of accounts that all have different credit cards attached and different IP addresses. Either approach would significantly add to the real-world costs of conducting the attack.

In this paper, we examine how an adversary could generate adversarial examples with a *controllable trade-off* between the number of queries and the quality of the adversarial examples. In particular, we propose the use of Particle Swarm Optimization (PSO)—a gradient-free optimization technique—to craft adversarial examples. PSO maintains a population of candidate solutions called particles. Each particle moves in the search space to find better solutions based on a fitness function that we have designed for finding adversarial examples.

In our attack, called AdversarialPSO, we specify that particles move by making small perturbations to the input image that are virtually imperceptible to a human observer. PSO has been shown to quickly converge on good (though not globally optimal) solutions [**?**], making it very suitable for finding adversarial examples in a black-box setting, as it can identify sufficiently good examples with few queries. In AdversarialPSO, we also propose numerous adaptations to fit the black-box setting, including a novel method to minimize redundancy among the particles that greatly reduces the number of queries. We test the effectiveness of this approach on three image classification datasets—MNIST, CIFAR-10, and Imagenet—and find that AdversarialPSO attains high success rates with queries comparable to state-of-the-art attacks.

In a real attack, the adversary may be constrained to making fewer queries or, alternatively, be able to make more queries and want to improve the quality of the images further. AdversarialPSO allows the attacker to tune the number of

queries against the quality of images by simply changing the number of particles in the swarm. By using bigger swarms, more queries would be submitted to the model in exchange for higher-quality adversarial examples.

In addition to the flexibility offered by AdversarialPSO, due the gradient-free nature of the attack, no hyperparameters require tuning for the attack to be successful. As shown in Section 3.2, the attack only requires the number of particles and the initial block-size used in the attack. The two hyperparameters affect the quality vs number of queries trade-off and can be roughly estimated based on the dimensions of the input.

Contributions. In summary, we have made the following contributions:

- We present AdversarialPSO, a gradient-free black-box attack with controllable trade-offs between the number of queries and the quality of adversarial examples.
- We demonstrate the effectiveness of AdversarialPSO on both low-dimensional and high-dimensional datasets by empirically evaluating the attack on the MNIST, CIFAR-10, and Imagenet datasets. We show that AdversarialPSO produces adversarial examples comparable to the state-of-the-art.
- We show how AdversarialPSO can be adjusted to trade-off the number of queries against the quality of the images.

2 Related Work

In this section, we discuss related work in both the white-box and black-box settings.

2.1 White-box Attacks

Szegedy et al. were the first to discuss the properties of neural networks that make adversarial attacks possible [**?**]. They show that imperceptible non-random perturbations of an image can cause an otherwise accurate DL model to misclassify it. The authors also discuss the transferability of adversarial examples from one model to another, including scenarios where models may have different architectures or are trained using different subsets of training data.

Goodfellow et al. [**?**] presented an explanation as to why DL models are susceptible to adversarial examples. They argue that the linearity of neural networks is what leads to their sensitivity to small and directed changes in input. They also present the Fast Gradient Sign Method (FGSM), which calculates the perturbations needed to transform inputs to adversarial examples. FGSM determines the direction of perturbations according to model gradients with respect to input and adds minuscule values in that direction. Kurakin et al. [**?**] extend this approach by introducing IGSM, an iterative variant of FGSM that takes several smaller steps instead of one relatively large step. The authors printed the images of the adversarial examples and fed them to a model through a camera. The results demonstrate that adversarial examples can work in the physical world, and that these types of attacks are practical.

Papernot et al. take a different approach to find adversarial examples [**?**]. Instead of taking multiple small steps, they construct a saliency map that maintains relevant input features with a high impact on model outputs. They utilize the saliency map to perturb specific features and create adversarial examples. This approach allows an adversarial example constructed towards a target label specified by the attacker. In a later paper, Papernot et al. extended the techniques of both Goodfellow et al. and Papernot et al. to launch black-box attacks against remotely hosted targets [**?**]. As both attacks require knowledge of model internals—information that is not available in a black-box setting—the authors used a local white-box surrogate that approximates the black-box target. The surrogate is trained using the Jacobian-based Dataset Augmentation method, which expands the training set used to train the surrogate with data points that allow the surrogate to approximate the target's decision boundary closely.

Another approach was employed by Carlini and Wagner [**?**], who search for adversarial examples by iteratively performing $minimize\, \mathcal{D}(x, x+\delta)$, where $\mathcal{D}$ is either an L_0, L_2, or L_∞ distance metric. The attack finds the minimum distance required to generate an adversarial example according to the distance metric being minimized. To use it as a black-box attack, it can be launched on a surrogate model, where only examples with high confidence are likely to transfer to the target model.

2.2 Black-box Attacks

In a black-box attack, the attacker does not know the internals of a target model. Instead, the attacker can query the target with specially crafted inputs meant to help estimate the gradient or lead gradually to misclassified samples. Target models are typically assumed to return confidence scores along with each classification, and these are used in constructing the inputs for subsequent queries.

Gradient-Estimation Attacks. To launch black-box attacks, Chen et al. propose ZOO [**?**], a method to estimate model gradients using only the model inputs and the corresponding confidence scores provided by the model. The approach employs a finite difference method that evaluates image coordinates after adding a small perturbation to estimate the direction of the gradient for each coordinate. Since examining every coordinate requires a huge number of queries to the model, the authors applied the stochastic coordinate descent algorithm and attack-space dimension reduction to reduce the number of queries needed to approximate gradients. Moderate perturbations in the direction of the gradient are, as shown in the FGSM attack, sufficient to obtain an adversarial example from the input. Although ZOO can successfully create adversarial examples indistinguishable from the inputs, it requires up to a million queries for high-dimensional samples, such as from Imagenet. With so many queries, the attack could be easily detectable, and the cost could be prohibitive and impractical in a real-world setting for a single image.

To reduce the number of queries, Bhagoji et al. estimate the gradient of groups of features or coordinates instead of estimating one coordinate at a

time [?]. Although the attack was not evaluated on a high-dimensional dataset, it outperformed ZOO on low-dimensional datasets such as CIFAR-10 and MNIST. The proposed Gradient Estimation (GE) approach by the authors still requires up to 10,000 queries to generate an adversarial example. The authors considered PSO as a possible approach for searching adversarial examples but found it to be slow and not as useful as GE. As we show in Section 4.2, however, our modifications to the basic PSO algorithm enable it to outperform GE. Our version of PSO does not require a swarm of 100 particles to be effective, which would be slow as per Bhagoji et al.'s experience. Instead, it can search for adversarial examples with high success rates using swarms with as few as five particles.

Ilyas et al. propose Natural Evolutionary Strategies (NES) to estimate gradients of the model, and then use projected gradient descent on the estimated gradients to craft adversarial examples [?]. They also extend the approach in [?] to utilize the bandit optimization method to exploit prior information when estimating the gradients. Specifically, they incorporate a data-dependent prior, which exploits the similarity in gradient information exhibited by adjacent pixels. Furthermore, they also incorporate a time-dependent prior that utilizes the high correlation between gradients estimated in successive steps. Although the attack can generate high-quality adversarial examples with few queries, the approach has been shown to be very sensitive to changes in hyperparameter values. Moon et al. [?] have shown that having too many hyperparameters could lead to significant variability in attack performance, creating dependability on the values chosen for those hyperparameters. Gradient-estimation based approaches commonly have multiple hyperparameters that are necessary for the execution of attacks, such as the learning rate, search variance, decay rate, and update rules – in a real-world black-box setting, tuning these hyperparameters would either incur additional queries or might not be possible at all in many cases. In our approach, there are only two hyperparameters with predictable effects on the outcome of the attack.

Gradient-Free Attacks. Moon et al. formulate the problem of crafting adversarial examples as a set maximization problem that searches for the set of positive and negative perturbations that maximizes an objective function [?]. Similar to [?], the authors exploit the spatial regularity exhibited by adjacent pixels by searching for perturbations in blocks instead of individual pixels. They increase the granularity of the blocks as the search progresses. Our AdversarialPSO attack searches for perturbations in blocks as well and yields comparable results as Moon et al.'s approach. However, our approach is capable of adjusting hyperparameter values effectively for the trade-off between L2 and queries as we show in Section 4.6.

Guo et al. explore a simple attack that crafts adversarial examples by randomly sampling a set of orthonormal vectors and adding or subtracting them from the input [?]. The attack is shown to be successful in crafting adversarial examples despite its simplicity. However, the success of the attack diminishes as dimensionality increases, as shown when targeting InceptionV3, which expects

inputs (299x299) with higher dimensionality than that of ResNet and DenseNet (224x224). As the perturbations are applied randomly, many queries are wasted by the approach until a solution is found.

By utilizing Differential Evolution (DE), Su et al. show that some test samples can be misclassified by changing a single pixel [**?**]. Similar to the PSO algorithm used in this paper, DE is a population-based algorithm that maintains and manipulates a set of candidate solutions until an acceptable outcome is found. The objective of this one-pixel attack is to better understand the geometry of adversarial space and proximity of adversarial examples to their corresponding inputs. The attack does not achieve high success rates due to the tight constraints used in the study.

Another population-based black-box attack is GenAttack [**?**], which uses a *Genetic Algorithm (GA)* to find adversarial examples. This attack iteratively performs the three genetic functions–*selection*, *crossover*, and *mutation*–where selection extracts the fittest candidates in a population, crossover produces a child from two parents, and mutation encodes diversity to the population by applying small random perturbations. The authors propose two heuristics to reduce the number of queries used by GenAttack, namely dimensionality reduction and adaptive parameter scaling. Although the authors propose two heuristics to reduce the numbers of queries used by their approach, GenAttack uses a higher number of queries compared to our approach.

3 Particle Swarm Optimization

In this section, we provide an overview of the PSO algorithm and describe how we adapt it to generate adversarial examples against image classification models.

3.1 Conventional PSO

Kennedy and Eberhart first proposed PSO as a model to simulate how flocks of birds forage for food [**?**]. It has since been adapted to address a multitude of problems, such as text feature selection [**?**], grid job scheduling [**?**], and optimizing the generation of electricity [**?**]. The algorithm works by dispersing particles in a search space and moving them until a solution is found. The search space is assumed to be d-dimensional, where the position of each particle i is a d-dimensional vector $X_i = (x_{i,1}, x_{i,2}, x_{i,3}, \ldots, x_{i,d})$. The position of each particle is updated according to a velocity vector V_i where $V_i = (v_{i,1}, v_{i,2}, v_{i,3}, \ldots, v_{i,d})$. In each time-step or iteration, denoted as t, the velocity vector is used to update the particle's next position, calculated as:

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{1}$$

$$v_i(t+1) = wv_i(t) + c_1R_1(p_g - x_i(t)) + c_2R_2(p_i - x_i(t)) \tag{2}$$

Equation 2 contains three terms. The first term controls how much influence the current velocity has when calculating the next velocity and is constrained

with the *inertia* weight w. The second term, with weight c_1, is referred to as *exploration*, as it allows particles to explore further regions in the search space in the direction of the best position found by the swarm, denoted by p_g. The third term, with weight c_2, is referred to as *exploitation*, and it is based on the best position found by this particle, denoted by p_i. R_1 and R_2 are d-dimensional vectors containing uniformly distributed random numbers that are calculated for each iteration to encode randomness in the search process. Early implementations of PSO assigned a fixed value to w. Shi and Eberhart, however, found that linearly decreasing the inertia weight w improved PSO performance [**?**]. In each iteration, fixed values w_{start} and w_{end} together with a maximum number of iterations $t_{\max}$ were used to calculate the inertia as:

$$w(t) = w_{\text{end}} + (w_{\text{start}} - w_{\text{end}}) \left(\frac{t_{\max} - t}{t_{\max}} \right) \tag{3}$$

In the case of black-box adversarial attacks, however, the number of queries is a more appropriate measure for how much the attack has progressed. We thus modify Equation 3 to compute w with respect to the number of queries instead of number of iterations as:

$$w(t) = w_{\text{end}} + (w_{\text{start}} - w_{\text{end}}) \left(\frac{q_{\max} - q}{q_{\max}} \right), \tag{4}$$

where q_{max} is the query budget used in the attack and q is the number of queries submitted to the model. We set $w_{\text{start}} = 1$ and $w_{\text{end}} = 0$.

3.2 Adversarial PSO

Among the many applications of PSO, we show in this paper that it can also be used to craft adversarial examples for images. Shi and Eberhart [**?**] found that PSO is quick to converge on a solution and scales well to large dimensions, at the cost of slower convergence to global optima. This makes PSO an excellent fit for finding adversarial examples in the black-box setting, as it suggests that it can identify sufficiently good examples with few queries, even for high-dimensional image data.

In this section, we first describe the key adaptations we used to make PSO effective and query-efficient for black-box attacks, and especially highlight our technique for minimizing redundancy in the query process. We then lay out the overall algorithm.

PSO Adaptations. Our PSO includes several key adaptations for our problem:

Fitness Function. To adapt PSO to the problem of creating adversarial examples, we define a fitness function that measures the change in model output when perturbations are added to the input. In both targeted and untargeted attacks, the fitness function measures how much the model's confidence in the target label rises or drops, respectively. When performing untargeted attacks, the fitness

for each candidate solution is the confidence drop in the original class predicted by the model. Given the original image x, the perturbed image x', the model parameters θ, and the original label y we compute confidence $f(x, y, \theta)$. We then calculate the fitness using fitness $= f(x, y, \theta) - f(x', y, \theta)$. In targeted attacks, fitness is given by the *increase* in confidence in the desired class. For the target label y', we compute confidence $f(x, y', \theta)$ and fitness $= f(x', y', \theta) - f(x, y', \theta)$.

Constraints. To further control the perturbations added to the input image, we define an upper bound value B of maximum change to limit the L_∞ distance between the adversarial image and the original image. L_∞ measures the maximum change to any of the coordinates, where $L_\infty = max(|x_1 - x'_1|, |x_2 - x'_2|, \ldots, |x_d - x'_d|)$. To ensure the upper bound, we use the clip operator to get $x^{'} = clip(x_i + v_i, x_i - B, x_i + B)$. Additionally, we apply box constraints to maintain valid image values when adding perturbations. These constraints are applied to Equation 1 to yield:

$$x_i(t+1) = clip(clip(x_i(t) + v_i(t+1), x_i - B, x_i + B), 0, 1) \tag{5}$$

Block-Based Perturbation. Similar to related work [**?**,**?**,**?**], we exploit the spatial regularity of adjacent pixels by splitting the input into blocks and perturbing all the pixels in each block en masse. Perturbing pixels in blocks utilizes the gradient similarities that are shared between adjacent pixels. Essentially, as such pixels would have similar effects on the outcome of the prediction, perturbing them as a group would have a larger impact on the rise (or drop) on the model's confidence, which translates to requiring fewer queries to generate adversarial examples.

Reversals. Since we have a relatively small number of blocks, and thus a limited number of perturbations, we can examine the results of each modification separately. We take advantage of this by reversing all the perturbations that have caused a negative impact on the fitness with the goal of finding improvements. Note that instead of just undoing the perturbation, we actually move the particle in the opposite direction. In essence, this is similar to inferring the gradient, as we assume that the opposite of a bad direction will be a good direction. We note that this is different from the approach of Moon et al. of alternating between adding perturbations and removing perturbations [**?**], which just undoes some of the prior steps. In our tests, we find that our reversals do indeed lead to a better position in many cases.

Following the Edge of the L_∞ Ball. As observed by Moon et al. [**?**], the optimal solution when crafting adversarial examples often reside at the edges of the L_∞ ball. Based on this observation, when initializing and randomizing particles, we set their positions at the edge of the L_∞ ball to observe the highest (or lowest) fitness for each dimension. Particles are then moved inwards using Equations 2 and 5. Moving inwards from the edge ensures that particles get enough velocity to reach the other end quickly if the opposite position was found to have better fitness. Otherwise, particles would waste queries moving around the center of

the ball until they eventually build enough velocity towards the position with the highest fitness.

The Particle Explosion Problem. For long running attacks, the velocity would eventually become so large that it would overpower the exploration and exploitation terms in Equation 2. This would cause particles to get stuck at the edges of the L_∞ ball as the ever-increasing velocity would continuously push them to locations outside the search space. This is a well known problem in PSO, and although the inertia weight is meant to mitigate it, it does not completely solve the problem. Therefore, in addition to the inertia weight, we perform *velocity clamping* to limit the growth of the velocity vector, again leveraging the `clip` operator:

$$v_i(t) = clip(v_i(t), -B, B). \tag{6}$$

This is performed in every iteration for each particle before updating the particle positions.

Redundancy Minimization. Beyond these other adaptations to PSO, we found it very effective to minimize the redundancy across particles, which helps to minimize the number of particles and the number of queries to find good examples. The key insight of this approach is that relatively few of the possible changes to the image are going to be especially valuable to changing the classification result. If one of the particles includes one of these useful changes, then that benefit is likely to be seen in the query result. Having found that effective change in one particle, other particles can take advantage of this through the exploration attribute in the PSO algorithm, which moves particles towards the best position in the swarm. We thus aim to limit the possibility of redundant checks on already perturbed blocks. Essentially, if one of the particles has modified one of the blocks in a given way, e.g. it increased the red channel on all pixels in that block, then we prevent other particles from making the same modification. To do this, we first define a set β with all *available* blocks (which are still eligible to be modified), $\beta = (b_1, b_2, b_3, \ldots, b_n)$. Then, for each block in the set, we create a list of all possible directions containing the positive and negative directions for each channel in the block. For grayscale images, which contain only a single channel, the list of possible channel directions cd is given by $cd = \{(1), (-1)\}$. For RGB images, it is

$$\begin{aligned} cd = \{&(1,0,0), (-1,0,0),\\ &(0,1,0), (0,-1,0),\\ &(0,0,1), (0,0,-1)\}. \end{aligned}$$

In other words, any single channel could be increased or decreased.

When a direction in a block is assigned to a particle, that direction is then removed from the list to avoid multiple particles perturbing the same block in the same direction. When all the directions in a block are assigned to particles, we remove that block from the set β. When there are no more blocks in the set,

we increase the granularity of the blocks by dividing the block-size by half and recreate the block set to contain the smaller blocks.

For each particle, we maintain a list of all the blocks and directions assigned to it. This list is used to avoid assigning an opposite direction to the particle which would cancel out a direction that it was previously assigned. Section 3.2 discusses how this list is used.

PSO Algorithm. The threat model we assume for this attack consists of an attacker with exploratory capabilities that permits submissions to a remote black-box model, which returns confidence scores with each prediction. The attacker has no influence on the training process and has no access to internal model information. The attack is based solely on the confidence scores returned by the model.

The search for adversarial examples is performed in two stages: *initialization* and *optimization*. The *initialization* stage disperses the particles in the search space and tests the initial fitness for the starting point of each particle. The *optimization* stage moves the particles according to Equations 1 and 2, and tests the fitness for each new position until either an adversarial example is found or the query budget is exhausted, whichever comes first. The overall process of AdversarialPSO (Algorithm 1), and all other algorithms discussed in this section, can be seen in the appendix.

Initialization. For each image, the search process starts with initializing the particles by randomizing their positions in the search space (Algorithm 2). Particles are initialized by randomly assigning an equal number of blocks to each particle, without replacement to minimize redundancy (see Section 3.2). In large swarms, each particle is assigned relatively few blocks, resulting in a more fine-grained search for adversarial examples.

Two hyperparameters control how the swarm is initialized: the number of particles in the swarm P and the initial block-size b, which determines the number of initial blocks created and the number of blocks assigned to each particle. Each particle begins with the input image x and the set of blocks β with a single direction for each block. Particles are then dispersed in the search space by perturbing all the blocks assigned to them to the edge of the L_∞ ball according to the directions they were given. Once the particles are created and dispersed, their fitness is calculated and subsequently used in the optimization step.

Optimization. The optimization step of AdversarialPSO (Algorithm 3) is an iterative process that moves the particles in search of better fitness. Particle positions are updated using the velocity vector, which is calculated for each particle in every iteration. After moving the particles, their fitness is calculated and compared against the particle's best fitness to determine which particle position will be used to calculate future particle movements. The particle's fitness is also compared against the best fitness achieved in the swarm as a whole (i.e, best swarm fitness), and if the particle fitness was found to be better, the swarm is updated to account for the position with the highest fitness. The process is

repeated until an adversarial example is found or when the process exhausts the allowed number of queries.

In every iteration, in addition to particle movements, each particle is assigned the next set of blocks and directions as was done in the initialization stage. Again, the assignment is designed to minimize redundancy (see Section 3.2). This randomization is performed after the particles are moved according to their calculated velocity vectors to allow the exploration of additional regions of the search space (Algorithm 4).

After some number of iterations, all the directions in all the blocks will have been assigned to a particle. At that point, the granularity of the blocks is increased, and the particles are re-initialized with the swarm best position as a starting point. When re-initializing the particles, we also reset their best positions. We do this to prevent the particles from retracting to the previous granularity level.

After all the blocks are assigned to particles and before increasing the granularity of the blocks, we perform the reversal operation (see Section 3.2 on *Reversals*). The reversal is performed on the swarm best position by iterating through the past positions of each particle and applying an opposite step for any movement that caused a negative fitness for the particle (Algorithm 5).

4 Evaluation

4.1 Setup

To evaluate AdversarialPSO, we consider the the success rate (i.e., the ratio of successfully generated adversarial examples over the total number of samples) and the average number of queries needed to generate adversarial examples. We compare our results against the Parsimonious Black-Box Adversarial Attack [**?**], NES [**?**] and Bandits [**?**] using the benchmark dataset Imagenet. We use the results reported in the Parsimonious attack paper [**?**] for our comparison, and as L_2 distances were not reported by the authors for all three attacks, we omit this metric from our evaluation. Nonetheless, the same L_∞ bound was used in our experiments as the other three attacks. Furthermore, similar to the related work, we evaluate the attack using InceptionV3. The Imagenet results for both untargeted and targeted attacks are obtained from running AdversarialPSO on 1,000 correctly classified samples from the indices list provided in the Parsimonious attack. The same target labels used in [**?**] and [**?**] are used for the targeted experiment. Also, for related work that utilize block-based perturbations, we use the same initial block-sizes as related work. If not, we use block-size that are adequate to the dimensions of the input samples. The results for the untargeted and targeted Imagenet attacks are reported in Sections 4.3 and 4.4, respectively.

We also test the attack on an adversarially trained CIFAR-10 ResNet classier as was done in [**?**], by using the same pretrained network provided by MadryLab.[4] The results for this test are reported in Section 4.5.

[4] `https://github.com/MadryLab/cifar10_challenge`

	MNIST			CIFAR-10		
Attack	Succ. Rate	L2	Queries	Succ. Rate	L2	Queries
Finite Diff	92.9%	6.1	1568	86%	410.3	6144
GE	61.5%	6.0	196	66.8%	402.7	768
IFD	**100%**	2.1	62720	**100%**	65.7	61440
Iterative GE	98.4%	**1.9**	8000	99.0%	80.5	7680
Their PSO	84.1%	5.3	10000	89.2%	262.3	7700
SPSA	96.7%	3.9	8000	88.0%	**44.4**	7680
AdversarialPSO	98.52%	5.3	**183**	94.92%	338.2	**129**

Table 1. Results comparison: Untargeted attack on MNIST and CIFAR-10 against the PSO and GE attacks of Bhagoji et al. [**?**]. The results we list for the Bhagoji attacks are obtained from their paper

We compare the AdversatialPSO attack on MNIST and CIFAR-10 against the approach used by Bhagoji et al. [**?**] to show the improvements attained from our modifications to the PSO algorithm. Similar to the models used in [**?**], we use ResNet-32 and a two-layer convolutional neural network for CIFAR-10 and MNIST respectively. Furthermore, we use the same L_∞ limits of $L_\infty = 0.3$ for MNIST $L_\infty = {}^{8}/_{255}$ for CIFAR-10.Unlike Bhagoji et al., who used 100 particles, we only use 5 particles. Using fewer particles translates to fewer queries being submitted to the model and a less resource intensive attack. As we show in Section 4.2, we achieve higher success rates with much smaller swarms. For all MNIST evaluations, due to the low dimensionality of the inputs, we use an initial block-size of 2 without increasing the granularity. For CIFAR-10, we use an initial block-size of 8.

Finally, we explore the effect of using different-sized swarms on ImageNet. We report the average per-pixel L_2 distance between input images and their adversarial counterparts. As using more particles enables us to increase the granularity, we find that larger swarms produce better adversarial examples with a lower L_2 average. We show the results of this analysis in Section 4.6.

4.2 Untargeted MNIST and CIFAR-10

To demonstrate the effectiveness of AdversarialPSO, we compare our attack against the approach used by Bhagoji et al. [**?**]. As shown in Table 1, AdversarialPSO not only outperforms the standard PSO used by Bhagoji et al., it also outperforms the GE approach used by the authors. For MNIST, the only approach to have a higher success-rate is the Iterative Finite Difference (IFD) attack at 100%, however the average number of queries was above 60K. In our implementation, we set a maximum budget of 10K queries, which led to a handful of failures.

Regarding the average L_2, using a swarm with 5 particles produces adversarial examples with comparable distances. However, by increasing the number of particles in the swarm, better quality adversarial examples could be generated

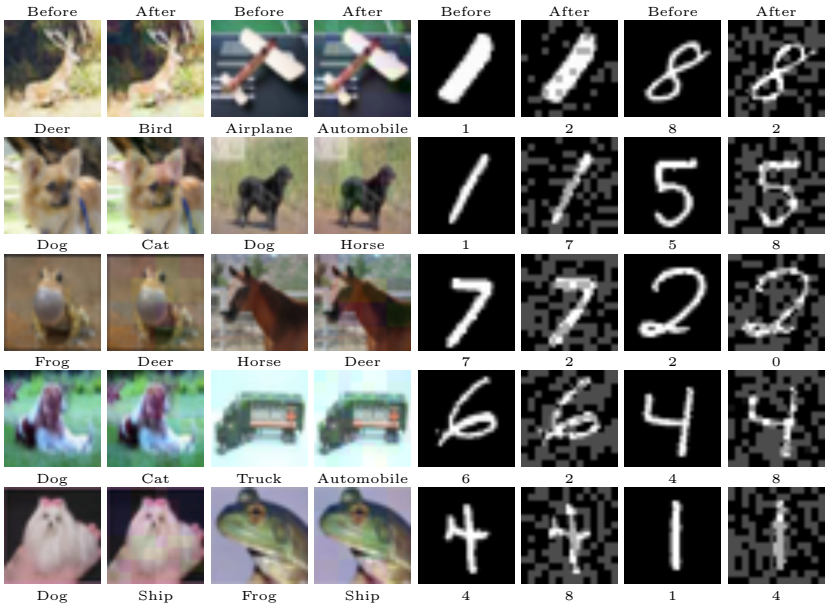

Fig. 1. Untargeted attack using AdversarialPSO on MNIST and CIFAR-10

	Untargeted		Targeted	
Attack	Success Rate	Avg. Queries	Success Rate	Avg. Queries
NES	80.3%	1660	99.7%	16284
Bandits	94.9%	1030	92.3%	26421
Parsimonious attack	**98.5%**	**722**	**99.9%**	**7485**
AdversarialPSO	96.9%	837	98.6%	14959

Table 2. Untargeted and targeted attacks on Imagenet

at the expense of more queries. Repeating the same experiment but with 10 particles produces an average L_2 of 4.9, but with an average of 296 queries.

Similarly for CIFAR-10, the only two approaches to have higher success rates are Iterative GE and IFD. Both of these, however, require many more queries on average (over 7500) than AdversarialPSO (under 200).

In examining the failed instances of the CIFAR-10 ResNet-32 model, we find that samples that failed were resistant to small perturbations. Particle movements had a low impact on the model's confidence scores and as such, executed for a large number of iterations until the the query budget was exhausted. For a majority of the samples, the adversarial examples were crafted rather quickly without using many queries. We speculate that the failed instances were far from the decision boundary, thus requiring large changes to be misclassified. Figure 1 shows randomly chosen examples of our attack on both CIFAR-10 and MNIST.

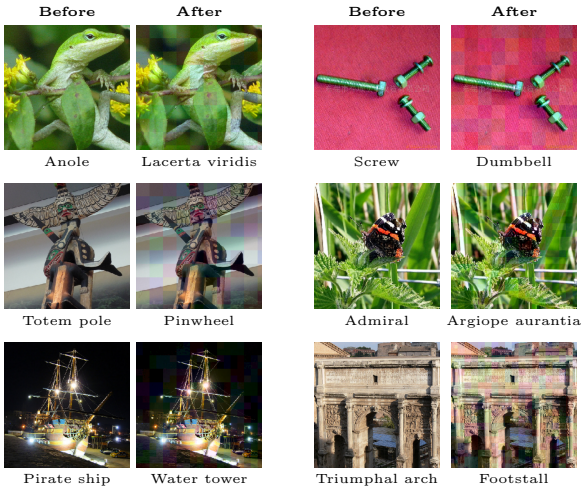

Fig. 2. Untargeted attacks on InceptionV3 (randomly selected samples)

4.3 Untargeted Imagenet

To evaluate the attack on the Imagenet dataset, we use the InceptionV3 model provided by Keras[5]. As per the Keras implementation, inputs are scaled to [-1,1], so we set the L_∞ bound to 0.1 (equivalent to the 0.05 L_∞ used by prior work). We choose the first 1000 samples from the indices list found in the Parsimonious Black-Box Attack GitHub page[6] and attack each sample with a query budget of 10,000 queries. We also use 32 for an initial block-size, similar to Moon et al. [**?**], and 5 particles in the swarm. Figure 2 shows randomly chosen examples of images generated from the attack, which we find have similar quality to those shown by Moon et al. [**?**]. As shown in Table 2, our attack achieves comparable success rates and number of queries as the related work, but with the advantage of providing controllable trade-offs between the number of queries and the quality of the adversarial examples.

4.4 Targeted Imagenet

To evaluate AdversarialPSO in a targeted attack, we use samples from the Parsimonious Black-box Attack's list of sample indices and we use the same labels as in [**?**]. Furthermore, similar to [**?**], we use an initial block-size of 32 and a query budget of 100,000 queries. Unlike the untargeted attack however, we use 10 particles to accommodate the more difficult attack setting. Table 2 summarizes our results and Figure 3 shows randomly chosen examples of the attack. Similar to the untargeted attack, we outperform both GE-based attacks. The Parsimonious attack however, generates adversarial examples with fewer queries.

[5] `https://keras.io/applications/#inceptionv3`
[6] `https://github.com/snu-mllab/parsimonious-blackbox-attack`

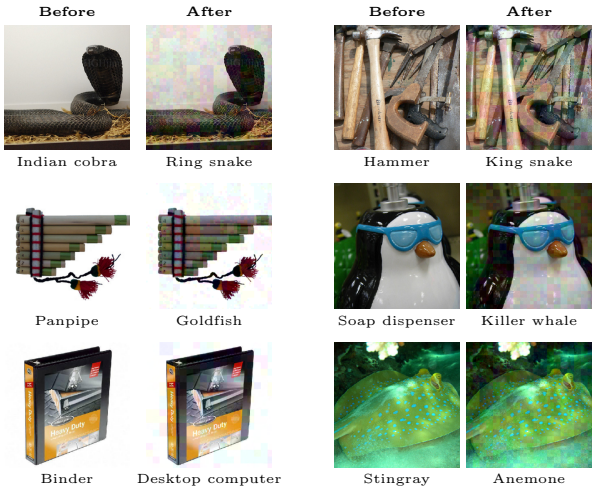

Fig. 3. Targeted attacks on InceptionV3 (randomly selected samples)

Attack	Success Rate	Avg. Queries
NES	29.5%	2872
Bandits	38.6%	1877
Parsimonious attack	**48%**	**1261**
AdversarialPSO	45.4%	2341

Table 3. Untargeted attack on adversarially trained CIFAR-10 ResNet classier

4.5 AdversarialPSO on Adversarially Trained Models

To test the attack against defended models, we evaluate AdversarialPSO against the adversarially trained CIFAR-10 model provided by MadryLabs. We use the same samples, L_∞ bound, and query budgets as used by Moon et al. [**?**]. As shown in Table 3, AdversarialPSO outperforms both Bandits and NES. Although the Parsimonious Black-box attack remains the highest in success rate, AdversarialPSO performs comparably with the added advantage of providing a trade-off between queries and L_2.

4.6 Swarm-size Analysis

By re-running the untargeted Imagenet attack using swarms with different sizes, we show that increasing the number of particles lowers the average L_2 at the expense of more queries. The results are based on samples that were successfully attacked by all swarm sizes. As shown in Figure 4, there is a 26% improvement in adversarial example quality when increasing the number of particles from 5 to 50. With this trade-off, an attacker that favors adversarial example quality over number of queries can use larger swarms. On the other hand, if fewer queries is more important to the attacker, then smaller swarms would be more beneficial.

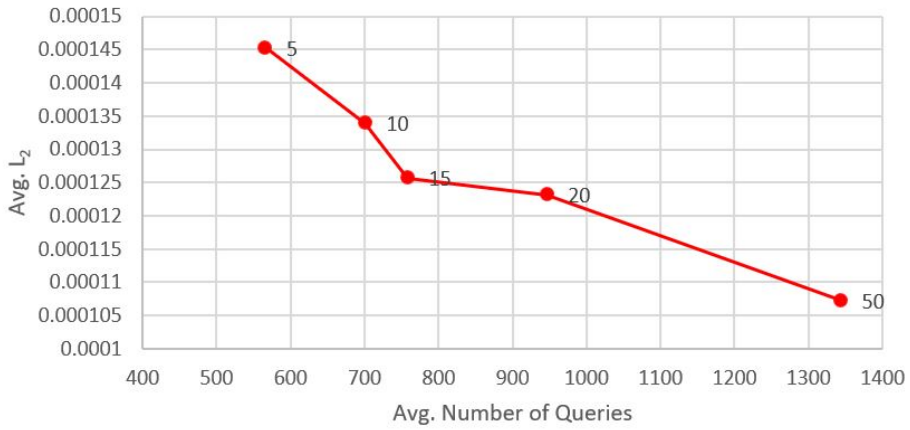

Fig. 4. The effect of swarm size on the average number of queries and per-pixel L_2 distance. In the figure, the x-axis represents the number of queries, the y-axis represents the per-pixel L_2, and the number of particles are shown by the markers

5 Conclusions

This paper presented a black-box attack based on the evolutionary search algorithm: Particle Swarm Optimization. The attack we call AdversarialPSO, adapts the traditional PSO algorithm to produce adversarial examples from images. Our experimental evaluations on the MNIST, CIFAR10, and Imagenet datasets suggest that AdversarialPSO can effectively generate adversarial examples in practical black-box settings with a limited number of queries to the target model. Furthermore, we demonstrate how the attack can be adjusted to control the trade-off between the number of queries submitted to the model and the L_2 distance between the original inputs and the generated adversarial examples. The purpose of the attack is to help evaluate security-critical models against black-box attacks and to promote the search for robust defenses.

Acknowledgment

We would like to thank the reviewers for their constructive comments that helped clarify and improve this paper. This material is based upon work supported by the National Science Foundation under Awards No. 1816851, 1433736, and 1922169.

A Appendix

As discussed in Section 3.2, the AdversarialPSO attack iteratively performs several operations to generate adversarial examples from images. Algorithm 1 provides a high-level view of the main AdversarialPSO loop that is responsible for

initializing the swarm, moving the particles, randomizing the particles, increasing the granularity of the search space, and reversing any movement with a negative fitness:

Algorithm 1 AdversarialPSO

Input: maximum queries q_{max},block-set B
Initialize swarm (See Algorithm 2)
while $q < q_{max}$ **do**
  if *Success* **then**
    return *bestPosition*
  end if
  Move Particles (See Algorithm 3)
  if *B is empty* **then**
    $performReversal$ (See Algorithm 5)
    $increaseGranularity$
    $initializeParticles(bestPosition)$
  else
    $randomizeParticles$ (See Algorithm 4)
  end if
end while
return *bestPosition*

In preparation for the attack, AdversarialPSO initializes the swarm by separating the image into blocks and assigning a different set of blocks to each particle. The attack then moves the particles according to the blocks they were assigned and evaluates the new position to calculate the fitness of each new position. Algorithm 2 provides the steps for the initialization process:

Algorithm 2 Initializing the swarm

Input: input image x, particle array par, block-set B, and maximum change m.
$n \leftarrow int(length(B)/P)$ # blocks per particle
for *p in par* **do**
  blocks $\leftarrow$ select random n elements from B
  $p.pos \leftarrow x$
  for *block in blocks* **do**
    direction $\leftarrow$ select random direction from $B[block]$
    pop direction from $B[block]$ and *push* direction to $p.blockList[block]$
    for *i in block* **do**
      $p.pos_i \leftarrow p.pos_i + m * direction$
    end for
  end for
  $fitness \leftarrow calculateFitness$ #includes update to q
  Compare new fitness against particle best and swarm best
end for
return *par*, *bestPosition*, *bestFitness*

In each iteration, particles are moved using traditional PSO operations, which consist of calculating the velocity of each particle and adding that velocity to the particle's current position. After each movement, the fitness for the new position is calculated and compared against the particle's best fitness and the swarn-wide best fitness. Future particle movements depend on the outcome of each fitness comparison. Algorithm 3 provides the steps for the velocity-based particle movements:

Algorithm 3 Move Particles

Input: particle array par, swarm-wide best fitness $bestFitness$, and swarm-wide best position $bestPosition$
for p *in* par **do**
  $v \leftarrow calculateVelocity$
  $p.pos \leftarrow updatePosition$
  $fitness \leftarrow calculateFitness$ #includes update to q
  Compare new fitness against particle best and swarm best
end for
return $bestPosition$

In addition to velocity-based movements, in every iteration, each particles is assigned new blocks with directions that are unique to that particle. Algorithm 4 shows the process of assigning blocks and directions to particles:

Algorithm 4 Randomize Particles

Input: particle list par, block-set B, change rate cr, and maximum change m.
for p *in* par **do**
  $blocks \leftarrow$ select random cr elements from B
  for $block$ in $blocks$ **do**
    if $block$ in $p.blockList$ **then**
      $d \leftarrow p.blockList[block]$
      $direction \leftarrow$select direction from $B[block]$ where $direction\,! = \{d, -d\}$
    else
      push block to $p.blockList$
      $direction \leftarrow$ select random direction from $B[block]$
    end if
    pop direction from $B[block]$ and *push* direction to $p.blockList[block]$
    for i *in* $block$ **do**
      $p.pos_i \leftarrow p.pos_i + m * direction$
    end for
  end for
  $fitness \leftarrow calculateFitness$ #includes update to q
  Compare new fitness against particle best and swarm best
end for

If a given particle movement produced a negative fitness, we observed that moving in the opposite direction would most likely produce a positive fitness. Algorithm 5 provides the steps for these reversal operations:

Algorithm 5 Reverse movements with negative fitness

```
1: Input: best position bestPosition and Particle list par
2: for p in par do
3:     for pastPos in p.pastPos do
4:         if pastPos.fitness < 0 then
5:             bestPosition ← bestPosition − pastPos.pos
6:             if Fitness did not improve then
7:                 Undo last changes
8:             end if
9:             pop pastPos from p.pastPos
10:        end if
11:    end for
12: end for
13: return bestPosition
```