Abstract

One major factor impeding more widespread adoption of deep neural networks (DNNs) is their issues with robustness, which is essential for safety critical applications such as autonomous driving. This has motivated much recent work on adversarial attacks for DNNs, which mostly focus on pixel-level perturbations void of semantic meaning. In contrast, we present a general framework for adversarial black-box attacks on trained agents, which covers semantic perturbations to the environment of the agent performing the task as well as pixel-level attacks. To do this, we re-frame the adversarial attack problem as learning a distribution of parameters that always fool the agent. In the semantic case, our proposed adversary (denoted as BBGAN) is trained to sample parameters that describe the environment with which the black-box agent interacts, such that the agent performs its dedicated task poorly in this environment. We apply BBGAN on three different tasks (primarily targeting aspects of autonomous navigation): object detection, self-driving, and autonomous UAV racing. On these tasks, BBGAN can generate failure cases that consistently fool an agent. We also demonstrate the usefulness of our framework as an analysis tool by visualizing systemic failure cases and uncovering semantic insights about the agents themselves.

1. Introduction

As a result of recent advances in machine learning and computer vision, deep neural networks (DNNs) are now interleaved with many aspects of our daily lives. DNNs are used to suggest news articles to read and movies to watch, automatically edit our photos and videos, and translate between hundreds of languages. They are also bound to disrupt transportation with autonomous driving slowly becoming a reality. While there are already impressive demos and some successful deployments, safety concerns for boundary conditions persist. While current models work very well on average, they struggle with robustness in certain cases. Recent work in the adversarial attack literature shows how sensitive DNNs are to input noise. These attacks usually utilize the information about the network structure to perform gradient updates in order to derive targeted perturbations (coined white-box attacks). These perturbations are injected into the input image at the pixel-level, so as to either confuse the network or enforce a specific behavior [45, 18, 9, 24]. There are also attacks that assume the DNN to be a black-box (i.e. undisclosed network architecture), but they still operate at the pixel-level [11, 33]. Targeted pixel attacks can lead to severe performance degradation not only in classification, but also on tasks like object detection [12], face recognition [43], and segmentation [50].

In practice, such pixel attacks are much less likely to naturally occur than semantic attacks which include changes in camera viewpoint, lighting conditions, street layouts, etc. The literature on semantic attacks is much sparser, since they are much more subtle and difficult to analyze [53]. Yet, this type of attack is critical to understand/diagnose failure cases that might occur in the real-world. While it is very difficult to investigate semantic attacks on real data, we can leverage simulation as a proxy that can unearth use-
ful insights transferable to the real-world. Figure 1 shows an example of an object misclassified by the YOLOV3 detector [38] applied to an image generated by Blender [7], an autonomous UAV racing [31] failure case in a recently developed general purpose simulator (Sim4CV [30]), and an autonomous driving failure case in a popular driving simulator (CARLA [14]). These failures arise from adversarial attacks on the semantic parameters of the environment.

In this work, we consider environments that are adequately photo-realistic and parameterized by a compact set of variables that have direct semantic meaning (e.g. camera viewpoint, lighting/weather conditions, road layout, etc.). Since the generation process of these environments from their parameters is quite complicated and in general non-differentiable, we treat it as a black-box function that can be queried but not back-propagated through. We seek to learn an adversary that can produce fooling parameters to construct an environment, in which the agent (which is also a black-box) fails in its task. Unlike most adversarial attacks that generate sparse instances of failure, our proposed adversary provides a more comprehensive view on how an agent can fail; we learn the distribution of fooling parameters for a particular agent and task and sample from it. Since Generative Adversarial Networks (GANs [17, 4]) have emerged as a promising family of unsupervised learning techniques that can model high-dimensional distributions [52, 21, 20, 25], we model our adversary as a GAN, appropriately denoted as black-box GAN (BBGAN).

Contributions. (1) We formalize adversarial attacks in a more general setup to include both semantic and conventional pixel attacks. (2) We propose BBGAN in order to learn the underlying distribution of semantic adversarial attacks. (3) We show promising results on three different safety-critical applications used in autonomous navigation, thus, illustrating the usefulness of our approach in analyzing agent failure, along with some analysis and insights about transferability of these attacks.

2. Related Work

Pixel Attacks on Classifiers. The work of Szegedy [45] was the first to introduce a formulation of attacking neural networks as an optimization. The method produces a minimal perturbation of the image pixels that fools a trained classifier (incorrect predictions). Several works followed the same approach but with different formulations, such as Fast Gradient Sign Method (FGSM) [18] and Projected Gradient Descent [24, 28]. The notable work of [9] gives a comprehensive study of different ways to fool the network with minimal pixel perturbation. However, all these methods are limited to pixel perturbations and only fool classifiers, while we consider more general cases of attacks, e.g. changes in camera viewpoint to fool a detector or change in weather conditions to fool a self-driving car.

Most of these attacks are white-box attacks, in which the algorithm has access to network gradients. However, some methods generate black-box attacks, which only have access to the classifier output. Most of these methods approximate the gradient to employ techniques similar to white-box attacks, e.g. through finite differences [11] or PCA and randomized dimension selection. In this work, our core attack is black-box in nature, but we are interested in the distribution of the semantic parameters that fool the agent, more so than individual fooling examples. This limits the usefulness of approximating the gradient to reach a single solution.

Attacks beyond Classifiers and Pixels. Image classification is not the only application for adversarial pixel attacks. Other tasks include object detection [39] and semantic segmentation [50], where the image pixels are perturbed through additive components or spatial transformations. Beyond pixel perturbations, Alcorn et al. attack famous classifiers and detectors by changing object pose [3], while Zeng et al. [53] attack deep classifiers and VQA (Visual Question and Answering) agents by perturbing lighting and surface normals. They show that most common image space adversarial attacks are not natural and cannot be realized in real 3D scenes. Inspired by this insight, we tackle the topic of semantic attacks by using readily available virtual environments with plausible 3D setups to systematically test the agents. In fact, our formulation includes attacks not only on static agents like object detectors, but also agents that interact with dynamic environments, such as self-driving agents. To the best of our knowledge, this is the first work to introduce adversarial attacks in CARLA [14], a standard autonomous navigation benchmark.
Adversarial Attacks and Reinforcement Learning. We formulate a black-box adversarial attack in a general setup that includes an environment that rewards an agent for some task. An adversary outside the environment is tasked to fool the agent by modifying the environment. Our formulation is naturally inspired by Reinforcement Learning (RL), in which agents can have multiple actions and receive partial rewards as they proceed in their task [44]. However, in RL, the agent is the focus of learning and the goal is to achieve a task in the environment. In adversarial attacks, the agent is usually fixed and learning is performed by the adversary in order to fool the agent or diagnose it. Recent work in [51] introduces an RL environment update as a way to improve the performance of agents inside this environment. Other works suggest the opposite, i.e., the environment is updated to fool the agent for training robust RL agents [29, 36]. In contrast, our virtual environment is parameterized by semantic attributes with the goal of fooling and diagnosing the failures of an already trained agent.

3. Methodology

Typical adversarial pixel attacks involve a neural network agent C (e.g., classifier or detector) that takes an image \( x \in [0, 1]^n \) as input and outputs a multinoulli distribution over \( K \) class labels with softmax values \( [l_1, l_2, \ldots, l_K] \), where \( l_j \) is the softmax value for class \( j \). The adversary (attacker) tries to produce a perturbed image \( x' \in [0, 1]^n \) that is as close as possible to \( x \), such that \( C \) changes its class prediction from \( x \) to \( x' \). The objective to be optimized is:

\[
\min_{x' \in [0, 1]^n} d(x, x') \text{ s.t. } \arg \max C(x) \neq \arg \max C(x') \tag{1}
\]

where \( d(x, x') \) is the distance, e.g., \( \|x - x'\|_2 \) or \( \|x - x'\|_\infty \).

3.1. Generalizing Adversarial Attacks

Extending attacks to general agents. In this work, we generalize the adversarial attack setup beyond pixel perturbations. Our more general setup (refer to Figure 2) includes semantic attacks, e.g., perturbing the camera pose or lighting conditions of the environment that generates observations (e.g., pixels in 2D images). An environment \( E_{\mu} \) is parametrized by \( \mu \in [\mu_{\min}, \mu_{\max}]^d \). It has an internal state \( s_t \) and produces observations \( o_t \in \mathbb{R}^n \) at each time step \( t \in \{1, \ldots, T\} \). The environment interacts with a trained agent \( A \), which takes \( o_t \) from \( E_{\mu} \) to produce actions \( a_t \). At each time step \( t \) and after the agent performs \( a_t \), the internal state of the environment is updated: \( s_{t+1} = E_{\mu}(s_t, a_t) \). The environment rewards the agent with \( r_t = R(s_t, a_t) \), for some reward function \( R \). We define the episode score

\[
Q(A, E_{\mu}) = \sum_{t=1}^{T} r_t \text{ of all intermediate rewards. The goal of } A \text{ is to complete a task by maximizing } Q. \text{ Outside the black-box (environment and agent), the adversary } G \text{ attacks } A \text{ by modifying } E_{\mu} \text{ through its parameters } \mu.
\]

Distribution of Adversarial Attacks. We define \( P_{\mu'} \) to be the fooling distribution of semantic parameters \( \mu' \) representing environment \( E_{\mu'} \), which fools the agent \( A \).

\[
\mu' \sim P_{\mu'} \leftrightarrow Q(A, E_{\mu'}) \leq \epsilon; \quad \mu' \in [\mu_{\min}, \mu_{\max}]^d \tag{2}
\]

Here, \( \epsilon \) is a small threshold that is defined for each adversarial attack. Distribution \( P_{\mu'} \) ensures both exploration and exploitation, since it covers all failure cases of \( A \) and still results in successful attacks for all its samples. Its PDF is unstructured and depends on the complexity of the agent. We seek an adversary \( G \) that tries to learn \( P_{\mu'} \), so it can be used to comprehensively analyze the weaknesses of \( A \).

Unlike the common practice of finding individual adversarial examples (most often images), we address the attacks distribution-wise in a compact semantic parameter space. We denote our analysis technique as Semantic Adversarial Diagnostic Attack (SADA). It is semantic because of the nature of the environment parameters and diagnostic because a fooling distribution is sought. Section 5.4 highlights what these semantic attacks can reveal about the nature of the agents used in safety-critical tasks (e.g., object detectors and autonomous navigation agents).

For the adversary \( G \) to achieve this challenging goal, we propose to optimize the following objective:

\[
\arg \min_{G} \mathbb{E}_{\mu \sim G} [Q(A, E_{\mu})] \tag{3}
\]

s.t. \( \{\mu : \mu \sim G\} = \{\mu' : \mu' \sim P_{\mu'}\} \)

Algorithm 1 describes a general setup for \( G \) to learn to generate fooling parameters. It also includes a mechanism for evaluating \( G \) in the black-box environment \( E_{\mu} \) for \( L \) iterations after training it to attack agent \( A \). An attack is considered a fooling attack, if parameter \( \mu \) sampled from \( G \) achieves an episode score \( Q(A, E_{\mu}) \leq \epsilon \). Consequently, the Attack Fooling Rate (AFR) is defined as the rate at which samples from \( G \) are fooling attacks. In addition to AFR, the algorithm returns the set \( S'_{\mu'} \) of adversarial examples that can be used to diagnose the agent. The equality constraint in Eq (3) is very strict to include all the fooling parameters \( \mu' \) of the fooling distribution. It acts as a perceptuality metric in our novel generalized attack to prevent unrealistic attacks. In the following, we relax this equality constraint and leverage recent advances in GANs to learn an estimate of the distribution \( P_{\mu'} \).

3.2. Black Box Generative Adversarial Network

Generative Adversarial Networks (GANs) are a promising family of unsupervised techniques that can model complex domains, e.g., natural images [17, 4, 19]. GANs consist of a Discriminator \( D_X \) and a Generator \( G_X \) that are adversarially trained to optimize the loss \( L_{GAN}(G_X, D_X, P_X) \) in Eq (4), where \( P_X \) is the distribution of images in domain \( X \) and \( z \in \mathbb{R}^c \) is a latent random Gaussian vector.
Figure 3: BBGAN: Learning Fooling Distribution of Semantic Environment Parameters. We learn an adversary $G$, which samples semantic parameters $\mu$ that parameterize the environment $E_{\mu}$, such that an agent $A$ fails in a given task in $E_{\mu}$. The inducer produces the induced set $S_{\mu'}$ from a uniformly sampled set $\Omega$ by filtering the lowest scoring $\mu$ (according to $Q$ value), and passing $S_{\mu'}$ for BBGAN training. Note that $Q_1 \leq Q_2 \ldots \leq Q_N$, where $s = |S_{\mu'}|$, $N = |\Omega|$. The inducer and the discriminator are only used during training (dashed lines), after which the adversary learns the fooling distribution $P_{\mu'}$. Three safety-critical applications are used to demonstrate this in three virtual environments: object detection (in Blender [7]), self-driving cars (in CARLA [14]), and autonomous racing UAVs (in Sim4CV [30]).

Algorithm 1: Generic Adversarial Attacks on Agents

| Returns: | Attack fooling Rate (AFR) |
|---|---|
| Requires: | Agent $A$, Adversary $G$, Environment $E_{\mu}$, number of episodes $T$, training iterations $L$, test size $M$, fooling threshold $\epsilon$ |
| Training $G$: | for $i \leftarrow 1$ to $L$ do |
| | Sample $\mu_i \sim G$ and initialize $E_{\mu_i}$ with initial state $s_1$ |
| | for $t \leftarrow 1$ to $T$ do |
| | $E_{\mu_i}$ produces observation $\alpha_t$ from $s_t$ |
| | $A$ performs $a_t(\alpha_t)$ and receives $r_t \leftarrow R(s_t, a_t)$ |
| | State updates: $s_{t+1} \leftarrow E_{\mu_i}(s_t, a_t)$ |
| | $G$ receives the episode score $Q_i(A, E_{\mu_i}) \leftarrow \sum_{t=1}^{T} r_t$ |
| | Update $G$ to solve for Eq (3) |
| Testing $G$: | Initialize fooling counter $f \leftarrow 0$ |
| for $j \leftarrow 1$ to $M$ do |
| | Sample $\mu_j \sim G$ and initialize $E_{\mu_j}$ with initial state $s_1$ |
| for $t \leftarrow 1$ to $T$ do |
| | $a_t(\alpha_t) : r_t \leftarrow R(s_t, a_t) : s_{t+1} \leftarrow E_{\mu_j}(s_t, a_t)$ |
| | State updates: $s_{t+1} \leftarrow E_{\mu_j}(s_t, a_t)$ |
| if $Q_j(A, E_{\mu_j}) \leq \epsilon$ then |
| | $f \leftarrow f + 1$ |
| end |
| end |

Returns: $\text{AFR} = f / M$

$D_X$ tries to determine if a given sample (e.g. image $x$) is real (from the training dataset) or fake (generated by $G_X$). On the other hand, $G_X$ tries to randomly generate samples that fool $D_X$ (e.g. misclassification). Both networks are proven to converge when $G_X$ can reliably produce the underlying distribution of the real samples [17].

In this paper, we propose to learn the fooling distribution $P_{\mu'}$ using a GAN setup, which we denote as black-box GAN (BBGAN). We follow a similar GAN objective but replace the image domain $x$ by the semantic environment parameter $\mu$. However, since we do not have direct access to $P_{\mu'}$, we propose a module called the inducer, which is tasked to produce a parameter set $S_{\mu'}$ (induced set) that belongs to $P_{\mu'}$. Thus, this setup relaxes Eq (3) to:

$$\text{arg min}_G \mathbb{E}_{\mu \sim G}[Q(A, E_{\mu})]$$

s.t. $\{\mu : \mu \sim G \subset \{\mu' : \mu' \sim P_{\mu'}\}$

So, the final BBGAN loss becomes:

$$\min_{G_{\mu}} \max_{D_{\mu}} L_{\text{BBGAN}}(G_{\mu}, D_{\mu}, S_{\mu'}) =$$

$$\mathbb{E}_{x \sim p_x(z)}[\text{log} D_{\mu}(x)] + \mathbb{E}_{z \sim p_z(x)}[\text{log}(1 - D_{\mu}(G_{\mu}(z)))]

(6)$$

Here, $G_{\mu}$ is the generator acting as the adversary, and $z \in \mathbb{R}^m$ is a random variable sampled from a normal distribution. A simple inducer can be just a filter that takes an uniformly sampled set $\Omega = \{\mu_i \sim \text{Uni}(\{\mu_{\text{min}}, \mu_{\text{max}}\}) : i = 1 \ldots N\}$ and suggests the lowest $Q$-scoring $\mu$, that satisfies the condition $Q(\mu_i) \leq \epsilon$. The selected samples constitute the induced set $S_{\mu'}$. The BBGAN treats the induced set as a training set,
so the samples in $S_{\mu'}$ act as virtual samples from the fooling distribution $P_{\mu'}$ that we want to learn. As the induced set size $S_{\mu'}$ increases, the BBGAN learns more of $P_{\mu'}$. As $|S_{\mu'}| \to \infty$, any sample from $S_{\mu'}$ becomes a sample of $P_{\mu'}$ and the BBGAN in Eq (6) satisfies the strict problem in Eq (3). Consequently, sampling from $G_{\mu'}$ would consistently fool agent $A$. We show an empirical proof for this in the appendix and in Section 5 we show how we consistently fool three different agents by $G_{\mu}$ samples. The number of samples needed for $S_{\mu'}$ to be representative of $P_{\mu'}$ depends on the dimensionality $d$ of $\mu$. Because of the black-box and stochastic nature of $E_{\mu}$ and $A$ (similar to other RL environments), we follow the random sampling scheme common in RL [26] instead of deterministic gradient estimation. In Section 5.3, we compare our method against baselines that use different approaches to solve Eq (3).

### 3.3. Special Cases of Adversarial Attacks

One can show that the generic adversarial attack framework detailed above includes well-known types of attacks as special cases, summarized in Table 1. In fact, the general setup allows for static agents (e.g. classifiers and detectors) as well as dynamic agents (e.g. an autonomous agent acting in a dynamic environment). It also covers pixel-wise image perturbations, as well as, semantic attacks that try to fool the agent in a more realistic scenario. The generic attack also allows for a more flexible way to define the attack success based on an application-specific threshold $\epsilon$ and the agent score $Q$. In the appendix, we provide more details on the inclusiveness of our generic setup.

### 4. Applications

#### 4.1. Object Detection

Object detection is one of the core perception tasks commonly used in autonomous navigation. In short, its goal is to determine the bounding box and class label of objects in an image. You Only Look Once (YOLO) object detectors popularized a single-stage approach, in which the detector observes the entire image and regresses the boundaries of the bounding boxes and the classes directly [37]. This trades off the accuracy of the detector for speed, making real-time object detection possible (passing the 30-FPS threshold).

**Agent.** Based on its suitability for autonomous applications, we choose the very fast, state-of-the-art YOLOv3 object detector as the agent in our SADA framework [38].

**Environment.** We use Blender [7] to construct a scene based on freely available 3D scenes and CAD models from free3D [2] and 3Dwarehouse [1]. The scene was picked to be an urban scene with an open area to allow for different rendering setups. The scene includes one object of interest, one camera, and one main light source all directed toward the center of the object. The light is a fixed strength spotlight located at a fixed distance from the object. The material of each object is semi-metallic, which is common for the classes under consideration. The 12 object classes (10 from Pascal-3D [49] and 2 from ShapeNet [10]) are: aeroplane, bench, bicycle, boat, bottle, bus, car, chair, dining table, motorbike, train, and truck. The total number of shapes across all 12 classes is 100 as shown in Figure 2. We augment Pascal3D shapes with others from Shapenet [10] and Modelnet40 [48] from the same class. At each iteration, one shape from the intended class is randomly picked and placed in the middle of the scene. Then, the Blender rendered image is passed to YOLOV3 for detection. Please
refer to the appendix for more details on the dataset.

Environment parameters. We take \( \mathbf{\mu} \in \mathbb{R}^n \), defining parameters that have shown to affect detection performance and frequently occur in real setups (refer to Figure 4). The object is centered in the virtual scene, and the camera circles around the object keeping the object in the center of the rendered image.

4.2. Self-Driving

There is a lot of recent work in autonomous driving especially in the fields of robotics and computer vision [8, 15, 34, 13]. In general, complete driving systems are very complex and difficult to analyze or simulate. By learning the underlying distribution of failure cases, our work provides a safe way to analyze the robustness of such a complete system. While our analysis is done in simulation only, we would like to highlight that sim-to-real transfer is a very active research field nowadays [35, 40, 41, 42, 46].

Agent. We use an autonomous driving agent (based on CIL [13]), which was trained on the environment \( \mathbf{E}_\mathbf{\mu} \) with default parameters. The driving-policy was trained end-to-end to predict car controls given an input image and is conditioned on high-level commands (e.g. turn right at the next intersection) in order to facilitate autonomous navigation.

Environment. We use the recent CARLA driving simulator [14], the most realistic open-source urban driving simulator currently available. We consider the three common tasks of driving in a straight line, completing one turn, and navigating between two random points [14]. The score is measured as the average success of five pairs of start and end positions.

Environment parameters. Since experiments are time-consuming, we restrict ourselves to three parameters, two of which pertain to the mounted camera viewpoint and the third controls the appearance of the environment by changing the weather setting (e.g. ‘clear noon’, ‘clear sunset’, ‘cloudy after rain’, etc.). As such, we construct an environment by randomly perturbing the position and rotation of the default camera along the z-axis and around the pitch axis respectively, and by picking one of the weather conditions. Intuitively, this helps measure the robustness of the driving policy to the camera position (e.g. deploying the same policy in a different vehicle) and to environmental conditions.

4.3. UA V Racing

In recent years, UA V (unmanned aerial vehicle) racing has emerged as a new sport where pilots compete in navigating small UA Vs through race courses at high speeds. Since this is a very interesting research problem, it has also been picked up by the robotics and vision communities [23, 32].

Agent. We use a fixed agent to autonomously fly through each course and measure its success as percentage of gates passed [32]. If the next gate was not reached within 10 seconds, we reset the agent at the last gate. We also record the time needed to complete the course. The agent uses a perception network that produces waypoints from image input and a PID controller to produce low-level controls.

Environment. We use the general-purpose simulator for computer vision applications, Sim4CV [30]. Sim4CV is not only versatile but also photo-realistic and provides a suitable environment for UA V racing.

Environment parameters. Here, we change the geometry of the race course environment. We define three different race track templates with 3-5 2D anchor points, respectively. These points describe a second order B-spline and are perturbed to generate various race tracks delineated by colored cones and populated by uniformly spaced gates. Please refer to the appendix for more details and visualizations of the generated tracks.

5. Experiments

5.1. BBGAN

In this section, we show how our proposed BBGAN can be used to estimate the distribution of failure cases with respect to some environment parameters \( \mathbf{\mu} \) that affect the agent observations of the environment.

Training. To learn the fooling distribution \( P_{\mathbf{\mu}'} \), we train the BBGAN as described in Section 3.2. For this, we use a vanilla GAN model [17], but any GAN architecture can be used. We use a simple FCN with 2 layers for the Generator \( G \) and Discriminator \( D \). We train the GAN following convention, but since we do not have access to the true distribution that we want to learn (i.e. real samples), we induce the set by randomly sampling \( N \) parameter vector samples \( \mathbf{\mu} \), and then picking the \( K \) worst among them (according to \( Q \) score). For object detection, we use \( N = 20000 \) image renderings for each class (a total of 240K images), as described in Section 4.1. Due to the computational cost, our
Table 2: Attack Fooling Rate (AFR) Comparison: AFR of adversarial attacks on three safety-critical applications: YOLOV3 object detection, self-driving, and UAV racing. For each application, an estimate of the fooling distribution can be learned using our BBGAN models or any of the baselines. Fooling samples are generated from each method and evaluated by computing the AFR. For detection, we report the average AFR performance across all 12 classes and we highlight 3 of them since they frequently occur in autonomous driving scenarios. For autonomous driving, we compute the AFR for the three common tasks in CARLA. For UAV racing, we compute AFR for race tracks of varying complexity (3, 4, or 5 anchor points describe the track). We see that our BBGAN outperforms all the baselines, and with larger margins for higher dimensional tasks (e.g., detection). Adding the boosting strategy to BBGAN helps improve the AFR further. Due to the expensive computations and sequential nature of the Bayesian baseline, we omit it for the two autonomous navigation applications. Best results are in bold. More details are provided in Section 4 and Section 5.

dataset for the autonomous navigation tasks comprises only $N = 1000$ samples. For instance, to compute one data point in autonomous driving, we need to run a complete episode that requires 15 minutes. The samples for autonomous driving and UAV racing are collected as described in Section 4.2 and Section 4.3 respectively.

**Boosting.** Inspired by the classical Adaboost algorithm [16], we use a boosting strategy to improve the performance of our BBGAN. Our boosting strategy simply utilizes the samples generated by the previous stage adversary $G_{k-1}$ in inducing the training set for the current stage adversary $G_k$. This is done by including the generated samples to $\Omega$ before training $G_k$. This strategy can be applied iteratively in a sequential manner more than once. The intuition here is that the main computational burden in training the BBGAN is not the GAN training itself, but computing the agent episodes, each of which can take multiple hours for the case of self-driving. For more details, including the algorithm, a mathematical justification and more experimental results please refer to the appendix.

5.2. Testing, Evaluation, and Baselines

To highlight the merits of BBGAN, we seek to compare it against baseline methods (see below), which also aim to estimate the fooling distribution $P_{\mu'}$. In this comparative study, each method produces $M$ fooling/adversarial samples (250 for object detection and 100 for self-driving and UAV racing) based on its estimate of $P_{\mu'}$. Then, the attack fooling rate (AFR) for each method is computed as the percentage of the $M$ adversarial samples that fooled the agent. To determine whether the agent is fooled, we use a fooling rate threshold $\epsilon = 0.3$ [12], $\epsilon = 0.6$, and $\epsilon = 0.7$ for object detection, self-driving, and UAV racing, respectively. In the following, we briefly explain the baselines.

**Random.** We uniformly sample random parameters $\mu$ within an admissible range that is application dependent.

**Gaussian Mixture Model (GMM).** We fit a full covariance GMM of varying Gaussian components to estimate the distribution of the samples in the induced set $S_{\mu'}$. The variants are denoted as Gaussian (one component), GMM10% and GMM50% (number of components as percentage of the samples in the induced set).

**Bayesian Expected Improvement.** We use the hyper-opt [6] implementation of the Expected Improvement (EI) Bayesian Optimization algorithm [22] (similarly adopted by [47]) to minimize the score $Q$ for the agent. The optimizer runs for $10^4$ steps and it tends to gradually sample more around the global minimum of the function. So, we use the last $N = 1000$ samples to generate the induced set $S_{\mu'}$ and then learn a GMM on that with different Gaussian components. Finally, we sample $M$ parameter vectors from the GMMs and report results for the best model. The advantage of this baseline over GMM is that the samples are more structured and usually clustered around the global minimum, but this happens after a long expensive trace which prevents using it for the autonomous applications.

**Multi-Class SVM.** We bin the output score $Q$ into 5 equally sized bins and train a multi-class SVM classifier on the complete set $\Omega$ to predict the correct bin. We then randomly sample parameter vectors $\mu$, classify them, and sort them by the predicted score. We pick the $M$ samples with the lowest $Q$ score.

**Gaussian Process Regression.** Similar to the SVM case,
we train a Gaussian Process Regressor [27] with an exponential kernel to regress \( Q \) scores from the corresponding \( \mu \) parameters that generated the environment on the dataset \( \Omega \).

5.3. Results

Table 2 summarizes the AFR results for the aforementioned baselines and our BBGAN approach across all three applications. For object detection, we show 3 out of 12 classes and report the average across all classes. For autonomous driving, we report the results on all three driving tasks. For UAV racing, we report the results for three different track types, parameterized by an increasing number of 2D anchor points (3, 4 and 5) representing \( \mu \) (refer to Section 4.2 and Section 4.3).

Our results show that we consistently outperform the baselines, even the ones that were trained on the complete \( \Omega \) rather than the smaller induced set \( S_{\mu} \), such as the multi-class SVM and the GP regressor. While some baselines perform well on the autonomous driving application where \( \mu \) consists of only 3 parameters, our approach outperforms them by a large margin on the tasks with higher dimensional \( \mu \) (e.g., object detection and UAV 5-anchors). To ensure that mode collapse (a GAN phenomenon where the generator collapses to generate a single point because of imbalance loss with the discriminator) does not occur, we do the following. (1) We visualize the individual \( \mu \) distributions of the generated samples (as in Figure 5) and ensure the variety of the samples. (2) We measure the average standard deviation per parameter dimension to make sure it is not zero for BBGAN (would be less than some of other baselines). (3) We visualize the images/tracks created by these parameters as in Figure 6.

5.4. Analysis

The usefulness of SADA lies in that it is not only an attacking scheme using BBGAN, but it can also serve as a diagnosis tool to assess the systematic failures of agents.

Diagnosis. To identify weaknesses and cases that result in systematic failure for the YOLOv3 detector, we fix some semantic parameters (distance to camera and object color) and attack the others (the four angles in Figure 4). We focus on two object classes that are relevant to autonomous driving: cars and motorbikes. We use rough models with homogeneous texturing (obtained from the training in Section 4.1) as well as detailed textured models. Figure 5 visualizes the learned fooling distribution for a detailed car model.

Transferability. To demonstrate the transferability of the fooling parameter distribution to the real-world, we photograph a toy car using a standard mobile phone camera and use office desk lights to simulate the light source of the virtual environment. We orient the camera and the lighting source according to the fooling distribution learned from the virtual world. In Figure 5, we present a fooled real-world photo of the toy car and the rendered virtual-world image, both of which are similarly fooled with the same erroneous class label. Please refer to the appendix for a similar analysis of the motorbike class, the transfer of attacks between different CAD models and the effect of occlusion (as yet another environment parameter) on detection.

Figure 5: Visualization of the Fooling Distribution. Using BBGAN, we learn how to sample from a distribution of camera positions and light source angles, which fool the YOLOv3 detector for a car model. (Top): On the right we plot the camera positions and light source directions of 250 sampled parameters in a 3D sphere around the object. On the left, we show how real photos of a toy car, captured from the same angles as rendered images, confuse the YOLOv3 detector in the same way. (Bottom): We visualize the distribution of individual parameters sampled from the learned BBGAN (range normalized to [-1,1]).

Figure 6: Qualitative Examples: (Top): BBGAN generated samples that fool YOLOv3 detector on the Truck class. (Bottom): BBGAN generated tracks that fool the UAV navigation agent.
6. Insights and Future Work

Object Detection with YOLOV3. In this case, we consistently found that for most objects, top-rear or top-front views of the object tend to fool the YOLOV3 detector. The color of the object does not play a significant role in fooling the detector, but usually colors that are closer to the background color tend to be preferred by the BBGAN samples (as shown in the qualitative examples).

Self-Driving. In this case, we found that weather is the least important parameter for fooling the driving policy, which indicates that the policy was trained to be insensitive to this factor. Interestingly, we observe that the learned policy is very sensitive to slight perturbations in the camera pose (height and pitch), indicating a systemic weakness that should be ratified with more robust training.

UAV Autonomous Navigation. We observe that the UAV fails if the track has very sharp turns. This makes intuitive sense and the results that were produced by our BBGAN consistently produce such tracks. While the tracks that are only parameterized by three control points can not achieve sharp turns, our BBGAN is still able to make the UAV agent fail by placing the racing gates very close to each other, thereby increasing the probability of hitting them.

Future Work. In the future, we plan to investigate defense mechanisms against semantic attacks described in this work. Our method can be used to identify difficult samples that lie in the fooling distribution. These samples can then be used to adversarially train more robust agents (e.g. classifiers, detectors, and autonomous navigation agents).

Acknowledgments. This work was supported by the King Abdullah University of Science and Technology (KAUST) Office of Sponsored Research through the Visual Computing Center (VCC) funding.

References

[1] 3d warehouse. https://3dwarehouse.sketchup.com/?hl=en, 2018.
[2] Free3d. https://free3d.com/, 2018.
[3] M. A. Alcorn, Q. Li, Z. Gong, C. Wang, L. Mai, W. Ku, and A. Nguyen. Strike (with) a pose: Neural networks are easily fooled by strange poses of familiar objects. CoRR, abs/1811.11553, 2018.
[4] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein gan, 2017. arXiv:1701.07875.
[5] C. Audet and W. Hare. Derivative-Free and Blackbox Optimization, chapter:3. pages 33–54. Springer International Publishing, Cham, 2017.
[6] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, and D. Cox. Hyperopt: A python library for model selection and hyperparameter optimization. Computational Science & Discovery, 8:014008, 07 2015.
[7] Blender Online Community. Blender - a 3D modelling and rendering package. Blender Foundation, Blender Institute, Amsterdam, 2018.
[8] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, et al. End to end learning for self-driving cars. arXiv:1604.07316, 2016.
[9] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In IEEE Symposium on Security and Privacy (SP), 2017.
[10] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, J. Xiao, L. Yi, and F. Yu. ShapeNet: An Information-Rich 3D Model Repository. Technical Report arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015.
[11] P.-Y. Chen, H. Zhang, Y. Sharma, J. Yi, and C.-J. Hsieh. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security, AISec ’17, pages 15–26, New York, NY, USA, 2017. ACM.
[12] S. Chen, C. Cornelius, J. Martin, and D. H. Chau. Robust physical adversarial attack on faster R-CNN object detector. CoRR, abs/1804.05810, 2018.
[13] F. Codevilla, M. Müller, A. Dosovitskiy, A. López, and V. Koltun. End-to-end driving via conditional imitation learning. In ICRA, 2018.
[14] A. Dosovitskiy, G. Ros, F. Codevilla, A. López, and V. Koltun. CARLA: An open urban driving simulator. In CoRL, 2017.
[15] U. Franke. Autonomous driving. In Computer Vision in Vehicle Technology, 2017.
[16] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. J. Comput. Syst. Sci., 55(1):119–139, Aug. 1997.
[17] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In Z. Ghahramani, N. D. Lawrence, and K. Q. Weinberger, editors, Advances in Neural Information Processing Systems 27, pages 2672–2680. Curran Associates, Inc., 2014.
[18] I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In International Conference on Learning Representations, 2015.
[19] S. Gurumurthy, R. Kiran Sarvadevabhatla, and R. Venkatesh Babu. Deligan: Generative adversarial networks for diverse and limited data. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), July 2017.
[20] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros. Image-to-image translation with conditional adversarial networks. arxiv, 2016.
[21] J. Johnson, A. Alahi, and F. Fei Li. Perceptual losses for real-time style transfer and super-resolution. In ECCV, 2016.
[22] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black-box functions. J. of Global Optimization, 13(4):455–492, Dec. 1998.
A. Empirical Justification for BBGAN

We want to show that as the size of the induced set $|S_{\mu'}| \to \infty$, learning an adversary according to the BBGAN objective in Eq (7) converges to the fooling distribution of semantic parameters $P_{\mu'}$ defined in Eq (8).

\[
\min_{G_{\mu}} \max_{D_{\mu}} L_{\text{BBGAN}}(G_{\mu}, D_{\mu}, S_{\mu'}) =
E_{\mu \sim S_{\mu'}}[\log D_{\mu}(\mu)] + E_{z \sim p_z(z)}[\log(1 - D(G(z)))]
\] (7)

\[
\mu' \sim P_{\mu'} \leftrightarrow Q(A, E_{\mu'}) \leq \epsilon; \ \mu' \in [\mu_{\text{min}}, \mu_{\text{max}}]^d
\] (8)

A.1. Empirical Proof

We use the definition of Exhaustive Search (Algorithm 3.1) from the Audet and Hare book on derivative-free and black box optimization [5]. In this algorithm, we try to optimize an objective $f : \mathbb{R}^d \to \mathbb{R}$ defined on a closed continuous global set $\Omega$ by densely sampling a countable subset $S = \{\mu_1, \mu_2, ..., \mu_N\} \subset \Omega$. Theorem 3.1 [5] states that as long as the exhaustive search continues infinitely from the set $S$, the global solutions of $f$ can be reached. Let’s assume the global solutions $\mu^*$ of $f$ exists and defined as follows.

\[
\mu^* = \arg \min_{\mu} f(\mu) \ \text{s.t.} \ \mu \in \Omega
\] (9)

Let’s denote the best solution reached up to the sample $\mu_N$ to be $\mu_{N}^{\text{best}}$. If the set $S_{\mu^*}$ is the set of all global solutions $\mu^*$, then Theorem 3.1 [5] states that

\[
\mu_{N}^{\text{best}} \in S_{\mu^*} = \{\mu^*\}, \ \text{as} \ N \to \infty
\] (10)

Now let $f(\mu) = \max(0, Q(E_{\mu}, A) - \epsilon)$ and let $\Omega = [\mu_{\text{min}}, \mu_{\text{max}}]^d$, then the global solutions of the optimization:

\[
\mu^* = \arg \min_{\mu} \max(0, Q(E_{\mu}, A) - \epsilon) \ \text{s.t.} \ \mu \in [\mu_{\text{min}}, \mu_{\text{max}}]
\] (11)

satisfy the two conditions in Eq (8) as follows.

\[
Q(A, E_{\mu^*}) \leq \epsilon; \ \mu^* \in [\mu_{\text{min}}, \mu_{\text{max}}]^d
\] (12)

Hence, the set of all global solutions includes all the points in the fooling distribution:

\[
S_{\mu^*} = \{\mu^*\} = \{\mu' : \mu' \sim P_{\mu'}\}
\] (13)

Therefore, as the sampling set size $|S| \to \infty$, all the points $\mu$ that lead to $Q(E_{\mu}, A) \leq \epsilon$ achieve the minimum objective in Eq (11) of zero and the set of best observed values $|\{\mu_{\text{best}}\}| \to \infty$. This set is what we refer to as the induced set $S_{\mu'}$. From Eq (10) and Eq (13), we can infer that the induced set will include all fooling parameters as follows.

as $N \to \infty, \ S_{\mu'} \to \{\mu' : \mu' \sim P_{\mu'}\}$ (14)

Finally if the set $S_{\mu'}$ has converged to the distribution $P_{\mu'}$ and we use $S_{\mu'}$ to train the BBGAN in Eq (7), then according to proposition 2 from the original GAN paper by Goodfellow et al. [17], the adversary $G_{\mu}$ has learnt the distribution $P_{\mu'}$ and hence satisfies the following equation:

\[
\arg \min_{G_{\mu}} \mathbb{E}_{\mu \sim G}[Q(A, E_{\mu})] \ \text{s.t.} \ \{\mu : \mu \sim G_{\mu}\} = \{\mu' : \mu' \sim P_{\mu'}\}
\] (15)

This concludes our empirical proof for our BBGAN.
B. Special Cases of Generic Adversarial attacks:

In Table 3, we summarize the substitutions in the generic adversarial attack to get different special cases of adversarial attacks. In summary, the generic adversarial attack allows for static agents (like classifiers and detectors) and dynamic agents (like an autonomous agent acting in a dynamic environment). It also covers the direct attack case of pixel perturbation on images to attack classifiers, as well as semantic attacks that try to fool the agent in a more realistic scenario (e.g., camera direction to fool a detector). The generic attack also allows for a more flexible way to define the attack success based on an application-specific threshold $\epsilon$ and the agent score $Q$. In the following we provide detailed explanation and mathematical meaning of the substitutions.

B.1. Pixel Adversarial Attack on Image Classifiers

Probably the most popular adversarial attack in the literature is a pixel-level perturbation to fool an image classifier. This attack can be thought of as a special case of our general formulation. In this case, the agent $A$ is a classifier $C : [0, 1]^n \rightarrow [l_1, l_2, ..., l_K]$ and the environment $E_\mu$ is a dataset containing the set $\Phi$ of all images in the classification dataset along with their respective ground truth labels, i.e., $\{(x_i, y_i)\}_{i=1}^{\Phi}$ and $y_i \in \{1, \ldots, K\}$. The softmax value of the true class is given by $l_{y_i} = \max C(x_i)$. Parameter $\mu$ defines the fooling noise to be added to the images ($i.e.$ $d = n$). The observation is simply an image from $\Phi$ with noise added: $a_i = x_i + \mu$ for some $i \in \{1, 2, \ldots, \Phi\}$. In classification, the environment is static with $T = 1$. To ensure the resulting image is in the admissible range, the noise added $\mu$ should fall in the range $[-x_i, \min, 1 - x_i, \max]$, where $x_i, \min, x_i, \max$ are the min and max pixel value for the image $x_i$. The sole action $a_i$ is simply the softmax score of the highest scoring class label predicted by $C$ that is not the true class $y_i$. Formally, $a_i = \max_{j \neq y_i} C(x_i + \mu) = l_j$. The reward function is $R(s_i, a_i) = Q(C, \Phi') = \max(l_{y_i} - l_j, 0)$. Here, $\epsilon = 0$ for the classifier fooling to occur, which means fooling occurs if $l_{y_i} - l_j \leq 0$. Using these substitutions in the hard constraint in Eq (8) translates to the following constraints on the perturbed image.

$$l_{y_i} \leq l_j, x_i + \mu \in [0, 1]^n \tag{16}$$

For a single image attack, we can rewrite Eq (16) as follows:

$$\max C(x) \leq \max_{j \neq y} C(x'), x' \in [0, 1]^n \tag{17}$$

We observe that the constraints in Eq (17) become the following constraints of the original adversarial pixel attack formulation on a classifier $C$. 

$$\min_{x' \in [0, 1]^n} d(x, x') \; s.t. \; \arg \max C(x) \neq \arg \max C(x') \tag{18}$$

B.2. Semantic Adversarial Attack on Object Detectors

Extending adversarial attacks from classifiers to object detectors is straightforward. We follow previous work [12] in defining the object detector as a function $F : [0, 1]^n \rightarrow (\mathbb{R}^{N \times K}, \mathbb{R}^{N \times 4})$, which takes an $n$-dimensional image as input and outputs $N$ detected objects. Each detected object has a probability distribution over $K$ class labels and a 4-dimensional bounding box for the detected object. We take the top $J$ proposals according to their confidence and discard the others. Analyzing the detector in our general setup is similar to the classifier case. The environment $E_\mu$ is static ($i.e.$ $T = 1$), and it contains all images with ground truth detections. For simplicity, we consider one object of interest per image (indexed by $i$). The observation in this case is a rendered image of an instance of object class $i$, where the environment parameter $\mu$ determines the 3D scene and how the image is rendered ($e.g.$ camera position/viewpoint, lighting directions, textures, etc.). Here, the observation is defined as the rendering function $o_i : [\mu_{\min}, \mu_{\max}]^d \rightarrow \mathbb{R}^n$. We use Blender [7] to render the 3D scene containing the object and to determine its ground truth bounding box location in the rendered image. The action $a_i$ by the agent/detector is simply the highest confidence score $l_i$, corresponding to class $i$ from the top $J$ detected boxes in $o_i$. The final score of $F$ is $Q(F, E_\mu) = l_i$. The attack on $F$ is considered successful, if $l_i \leq \epsilon$.

B.3. Semantic Adversarial Attack on Autonomous Agents

The semantic adversarial attack of an autonomous agent can also be represented in the general formulation of Algorithm 1 in the paper. Here, $A$ corresponds to the navigation policy, which interacts with a parametrized environment $E_\mu$. The environment parameter $\mu \in \mathbb{R}^d$ comprises $d$ variables describing the weather/road conditions, camera pose, environment layout etc. In this case, an observation $o_i$ is an image as seen from the camera view of the agent at time $t$. The action $a_i$ produced by the navigation policy is the set of control commands ($e.g.$ gas and steering for a car or throttle, pitch, roll and yaw for a UAV). The reward function $R(s_t, a_t)$ measures if the agent successfully completes its task ($e.g.$ if it safely reaches the target position at time $t$ and 0 otherwise). The episode ends when either the agent completes its task or the maximum number of iterations $T$ is exceeded. Since the reward is binary, the $Q$ score is the average reward over a certain number of runs (five in our case).
| Generic Attack Variables | Pixel Adversarial Attack on Image Classifiers | Semantic Adversarial Attack on Object Detectors | Semantic Adversarial Attack on Autonomous Agents. |
|--------------------------|---------------------------------------------|---------------------------------------------|-------------------------------------------------|
| **Agent A**              |                                             |                                             | self-driving policy agent A                      |
|                          | K-class classifier                          | K-class object detector                     | e.g. network to regress controls                |
|                          | \( C : [0, 1]^n \to [l_1, l_2, \ldots, l_K] \) | \( F : [0, 1]^n \to (\mathbb{R}^{N \times K}, \mathbb{R}^{N \times 4}) \) | e.g. road shape, weather, camera                |
|                          | \( l_j : \) the softmax value for class \( j \) | \( N : \) number of detected objects       |                                                 |
| **Parameters \( \mu \)** | some semantic parameters describing the scene | some semantic parameters involved in the simulation |                                     |
|                          | e.g. camera pose, object, light              | e.g. road shape, weather, camera             |                                                 |
| **Parameters Size \( d \)** | \( n: \) the image dimension \( n = h \times w \times c \) | number of semantic parameters parameterizing \( \mu \) | number of semantic parameters parameterizing \( \mu \) |
| **Environment \( \Phi_{\mu} \)** | dataset \( \Phi \) containing all images and their true class label | dataset \( \Phi \) containing all images and their true class label | simulation environment partially described by \( \mu \) that \( A \) navigates in to reach |
| **Parameters Range \( [\mu_{\min}, \mu_{\max}] \)** | \( x_{\min}, x_{\max} \) are the min and max of each pixel value in the image \( x \) | \([-1, 1]^d \) | \([-1, 1]^d \) |
| **Environment States \( s_t \)** | static environment \( s_t = \Phi \) | static environment \( s_t = \Phi \) | the state describing the simulation at time step \( t \) |
|                          |                                             |                                             | e.g. the new position of car                    |
| **Observation \( o_t \)** | the attacked image after adding fooling noise | the rendered image using the scene parameters \( \mu \) | the sequence of rendered images the \( A \) observes during the simulation episode |
| **Episode Size \( T \)** | 1                                           | 1                                           | \( t_{\text{stop}} \in \{1, 2, 3, \ldots, T_{\text{max}}\} \) |
| **Time step \( t \)**    | 1                                           | 1                                           | \( t_{\text{stop}} : \) step when success reached |
| **Fooling Threshold \( \epsilon \)** | 0 | fixed small fraction of 1 \( e.g. = 0.3 \) | fixed small fraction of 1 \( e.g. = 0.6 \) |
| **Observation \( o_t \)** | attacked image after added noise \( = x_t + \mu \), where \( x, \mu \in \mathbb{R}^n \) | the rendered image using the scene parameters \( \mu \) | sequence of rendered images \( A \) observes during the simulation episode |
| **Agent Actions \( a_t(\alpha_t) \)** | predicted softmax vector of attacked image | predicted confidence of the true class label | steering command to move the car/UA V in the next step |
| **Reward Functions \( R(s_t, a_t) \)** | the difference between true and predicted softmax | predicted confidence of the true class label | 1: if success state reached | 0: if success not reached |
| **Score \( Q(A, \Phi_{\mu}) \)** | the difference between true and predicted softmax | predicted confidence of the true class label | the average sum of rewards over five different episodes |

Table 3: Special Cases of Generic Adversarial attacks: summarizing all the variable substitutions to get common adversarial attacks.
C. Boosting Strategy for BBGAN

C.1. Intuition for Boosting

Inspired by the classical Adaboost meta-algorithm [16], we use a boosting strategy to improve the performance of our BBGAN trained in Section 5 of our paper with results reported in Table 1. The boosting strategy of BBGAN is simply utilizing the information learned from one BBGAN by another BBGAN in a sequential manner. The intuition is that the main computational burden in training the BBGAN is not the GAN training but computing the agent A episodes (which can take multiple hours per episode in the case of the self-driving experiments).

C.2. Description of Boosting for BBGANs

We propose to utilize the generator to generate samples that can be used by the next BBGAN. We start by creating the set $\Omega_0$ of the first stage adversary $G_0$. We then simply add the generated parameters for each stage and hence the average score of the added boosting samples from the previous stage $\Omega_{k-1}$ is trained according to Eq (7). Here $\beta$ is the boosting rate of our boosting strategy which dictates how much emphasis is put on the previous stage (exploitation ratio) when training the next stage. The whole boosting strategy can be repeated more than once. The global set $\Omega$ of all $N$ sampled points and the update rule from one stage to another is described by the following two equations:

$$\Omega_0 = \{\mu_j \sim \text{Uniform}(\mu_{\min}, \mu_{\max})\}_{j=1}^N$$

$$\Omega_k = \Omega_{k-1} \cup \{\mu_j \sim G_{k-1}\}_{j=1}^{\beta N}$$

These global sets $\Omega_k$ constitute the basis from which the inducer produces the induced sets $S_{\mu}^k$. The adversary $G_k$ of boosting stage $k$ uses this induced set when training according to the BBGAN objective in Eq (7). Algorithm 2 summarizes the boosting meta-algorithm for BBGAN.

C.3. Empirical proof for BBGAN Boosting

Here we want to show the effectiveness of boosting (Algorithm 2) on improving the performance of BBGAN from one stage to another. Explicitly, we want to show that the following statement holds, under some conditions.

Algorithm 2: Boosting Strategy for BBGAN

Requires: environment $E_\mu$, Agent A number of boosting stages $K$, boosting rate $\beta$, initial training size $N$

Sample $N$ points to form $\Omega_0$ like in Eq (19)

induce $S_{\mu}^0$ from $\Omega_0$

learn adversary $G_0$ according to Eq (7)

for $i \leftarrow 1$ to $K$

update boosted training set $\Omega_i$ from $\Omega_{i-1}$ as in Eq (20)

obtain $S_{\mu}^i$ from $\Omega_i$

train adversary $G_i$ as in Eq (7)

end

Returns: last adversary $G_K$

$$E_{\mu_k \sim G_k}[Q(A, E_{\mu_k})] \leq E_{\mu_k \sim G_{k-1}}[Q(A, E_{\mu_{k-1}})]$$

This statement says that the expected score $Q$ of the sampled parameters $\mu_k$ from the adversary $G_k$ of stage ($k$) BBGAN is bounded above by the score of the previous stage $G_{k-1}$, which indicates iterative improvement of the fooling adversary $G_k$ by lowering the score of the agent $A$ and hence achieving a better objective at the following realization of Eq (13).

$$\arg \min_G E_{\mu \sim G}[Q(A, E_{\mu})]$$

s.t. $\{\mu : \mu \sim G\} \subset \{\mu' : \mu' \sim P_{\mu'}\}$

Proof of Eq (21).

Let’s start by sampling random $N$ points as our initial $\Omega_0$ set as in Eq (19) and then learn BBGAN of the first stage and continue boosting as in Algorithm 2. Assume the following assumption holds,

$$|S_{\mu}^k| = \lfloor \beta N \rfloor, \forall k \in \{1, 2, 3, \ldots\}$$

then by comparing the average score $Q$ of the entire global set $\Omega_k$ at stage $k$ (denoted simply as $Q(\Omega_k)$) with the average score of the added boosting samples from the previous stage $\{\mu_j \sim G_{k-1}\}_{j=1}^{\beta N}$ as in Eq (20) (denoted simply as $Q(G_{k-1})$), two possibilities emerge:

1. Exploration possibility: $Q(\Omega_k) \leq Q(G_{k-1})$.

This possibility indicates that there is at least one new sample in the global set $\Omega_k$ that are not inherited from the previous stage adversary $G_{k-1}$, which is strictly better than $G_{k-1}$ samples with strictly lower $Q$ score. If the assumption in Eq (23) holds, then the induced set $S_{\mu}^k$ will include at least one new parameter that is not inherited from previous stage and hence the average score of the induced set will be less than that of the generated by previous stage.

$$Q(S_{\mu}^k) < Q(G_{k-1})$$

However since the BBGAN of stage $k$ uses the induced set $S_{\mu}^k$ for training, we expect the samples to be correlated: $G_k \sim S_{\mu}^k$, and the scores to be similar as follows:
E_{\mu_k \sim G_k}[Q(A, E_{\mu_k})] = Q(S^k_{\mu}) \quad (25)

Substituting Eq (25) in Eq (24) results in the inequality which makes the less strict inequality Eq (21) holds.

2. Exploitation possibility: $Q(\Omega_k)) > Q(G_{k - 1})$.

In this scenario, we don’t know for sure whether there is a new sample in $\Omega_k$ that is better than the inherited samples, but in the worst case scenario we will get no new sample with lower score. In either case, the assumption in Eq (23) ensures that the new induced set $S^k_{\mu}$ is exactly the inherited samples from $G_{k - 1}$ and the following holds.

$Q(S^k_{\mu}) \leq Q(G_{k - 1}) \quad (26)$

Using the same argument as in Eq (25), we deduce that in this exploitation scenario Eq (21) is still satisfied. Hence, we prove that Eq (21) holds given the assumption in Eq (23).

C.4. Experimental Details for Boosting

We note that low $\beta$ values do not affect the training of our BBGAN since the induced set will generally be the same. Hence, we use $\beta = 0.5$, a high boosting rate. For practical reasons (computing 50% of the training data per boosting stage is expensive) we just compute 10% of the generated data and repeat it 5 times. This helps to stabilize the BBGAN training and forces it to focus more on samples that have low scores without having to evaluate the score function on 50% of the training data.

D. Detailed Results

Tables 4 and 5 show the detailed results for all three applications.
Table 4: Fooling rate of adversarial attacks on different classes of the augmented Pascal3D dataset. We sample 250 parameters after the training phase of each model and sample a shape from the intended class. We then render an image according to these parameters and run the YOLOV3 detector to obtain a confidence score of the intended class. If this score $Q \leq \epsilon = 0.3$, then we consider the attack successful. The fooling rate is then recorded for that model, while $\mu_{std}$ (the mean of standard deviations of each parameter dimensions) is recorded for each model. We report the average over all classes. This metric represents how varied the samples from the attacking distribution are.

| Model               | aeroplane | bench | bicycle | boat | bottle | bus | car | chair | diningtable | motorbike | train | truck | avg | $\mu_{std}$ |
|---------------------|-----------|-------|---------|------|--------|-----|-----|-------|-------------|-----------|-------|-------|-----|-------------|
| Full Set            | 8.64%     | 35.2% | 14.6%   | 33.4%| 22.5%  | 53.1%| 39.8%| 44.1% | 46.1%       | 32.5%     | 58.1% | 56.8% | 37.1%| 0.577      |
| Random:             | 11.3%     | 42.7% | 18.6%   | 41.8%| 28.4%  | 65.7%| 49.9%| 55.3% | 56.4%       | 40.3%     | 72.8% | 70.8% | 46.2%| 0.584      |
| Multi-Class SVM     | 12.0%     | 45.6% | 20.0%   | 39.6%| 26.0%  | 64.4%| 49.6%| 50.4% | 53.6%       | 45.6%     | 72.0% | 70.8% | 45.8%| 0.576      |
| GP Regression       | 13.6%     | 15.6% | 17.6%   | 41.2%| 31.6%  | 71.6%| 51.6%| 48.0% | 56.0%       | 43.6%     | 69.2% | 83.6% | 45.2%| 0.492      |
| Gaussian            | 11.2%     | 45.6% | 19.6%   | 41.6%| 31.2%  | 70.4%| 48.0%| 56.8% | 55.6%       | 40.4%     | 71.2% | 72.4% | 47.0%| 0.548      |
| GMM10%              | 14.8%     | 45.2% | 26.0%   | 42.8%| 34.0%  | 67.2%| 53.2%| 56.4% | 54.8%       | 48.4%     | 70.4% | 75.2% | 49.0%| 0.567      |
| GMM50%              | 12.0%     | 44.0% | 16.4%   | 46.4%| 33.2%  | 66.4%| 51.6%| 53.2% | 58.4%       | 46.8%     | 73.6% | 72%   | 47.8%| 0.573      |
| Bayesian [47]       | 9.2%      | 42.0% | 48.0%   | 68.8%| 32.4%  | 91.6%| 42.0%| 75.6% | 58.4%       | 52.0%     | 77.2% | 75.6% | 56.1%| 0.540      |
| BBGAN (ours)        | 13.2%     | 91.6% | 44.0%   | 90.0%| 54.4%  | 91.6%| 81.6%| 93.2% | 99.2%       | 45.2%     | 99.2% | 90.8% | 74.5%| 0.119      |
| BBGAN (boost)       | 33%       | 82.4% | 65.8%   | 78.8%| 67.4%  | 100% | 67.4%| 100%  | 90.2%       | 82.0%     | 98.4% | 100% | 80.5%| 0.100      |

Table 5: Autonomous Driving (CARLA) and UAV Racing Track Generation (Sim4CV). Each method produces 50 samples and we show the fooling rate (FR) and the mean of the standard deviation per parameter. We set the fooling threshold to 0.6 and 0.7 for autonomous driving and racing track generation respectively.

| Method               | Straight | One Curve | Navigation | 3 control points | 4 control points | 5 control points |
|----------------------|----------|-----------|------------|------------------|------------------|------------------|
|                      | FR  | $\mu_{std}$ | FR  | $\mu_{std}$ | FR  | $\mu_{std}$ | FR  | $\mu_{std}$ | FR  | $\mu_{std}$ | FR  | $\mu_{std}$ |
| Full Set             | 10.6% | 0.198      | 19.5% | 0.596      | 46.3% | 0.604      | 17.0% | 0.607      | 23.5% | 0.544      | 15.8% | 0.578      |
| Random               | 8.0%  | 0.194      | 18.0% | 0.623      | 48.0% | 0.572      | 22.0% | 0.602      | 30.0% | 0.550      | 16.0% | 0.552      |
| Multi-Class SVM      | 96.0% | 0.089      | 100%  | 0.311      | 100%  | 0.517      | 24.0% | 0.595      | 30.0% | 0.510      | 14.0% | 0.980      |
| GP Regression        | 100%  | 0.014      | 100%  | 0.268      | 100%  | 0.700      | 74.0% | 0.486      | 94.0% | 0.492      | 44.0% | 0.486      |
| Gaussian             | 54.0% | 0.087      | 30.0% | 0.528      | 64.0% | 0.439      | 49.3% | 0.573      | 56.0% | 0.448      | 28.7% | 0.568      |
| GMM10%               | 90.0% | 0.131      | 72.0% | 0.541      | 98.0% | 0.571      | 57.0% | 0.589      | 63.0% | 0.460      | 33.0% | 0.558      |
| GMM50%               | 92.0% | 0.122      | 68.0% | 0.556      | 100%  | 0.559      | 54.0% | 0.571      | 60.0% | 0.478      | 40.0% | 0.543      |
| BBGAN (ours)         | 100%  | 0.048      | 98.0% | 0.104      | 98.0% | 0.137      | 42.0% | 0.161      | 94.0% | 0.134      | 86.0% | 0.202      |
| BBGAN (boost)        | 100%  | 0.014      | 100%  | 0.001      | 100%  | 0.058      | 86.0% | 0.084      | 98.0% | 0.030      | 92.0% | 0.003      |
E. Qualitative Examples

Figure 7 shows some qualitative examples for each of the 12 object classes. These images were rendered according to parameters generated by BBGAN which fooled the detector.

F. Qualitative Comparison

Figure 10 shows a qualitative comparison between samples of the BBGAN distribution and different baselines distributions in the YOLOV3 attacks experiments.

G. Training data

In the following we show some examples of the training data for each of the applications. Please refer to Figure 11 for some sample images of the object detection dataset. Figure 12 shows some images of the CARLA [14] simulation environment used for the self-driving car experiments. Figure 13 visualizes the distribution of the training data. Figure 14 shows some images of the Sim4CV simulator used for the UAV racing application. Figures 15, 16 and 17 show some samples from the UAV datasets.
Figure 7: **BBGAN Qualitative Examples in Object Detection** - Some sample images for each class that were rendered according to parameters generated by BBGAN which fooled the object detector.
| object class | parameters distribution (input) | scores histogram(output) |
|--------------|---------------------------------|-------------------------|
| aeroplane    | ![Input Parameters Distribution](image1) | ![Histogram Output](image2) |
| bench        | ![Input Parameters Distribution](image3) | ![Histogram Output](image4) |
| bicycle      | ![Input Parameters Distribution](image5) | ![Histogram Output](image6) |
| boat         | ![Input Parameters Distribution](image7) | ![Histogram Output](image8) |
| bottle       | ![Input Parameters Distribution](image9) | ![Histogram Output](image10) |
| bus          | ![Input Parameters Distribution](image11) | ![Histogram Output](image12) |

Figure 8: **BBGAN Distribution Visualization 1**: visualizing the input parameters marginal distributions (the range is normalized from -1 to 1). Also, the Agent scores histogram for these parameters vs random parameters scores histogram are shown in the right column.
Figure 9: BBGAN Distribution Visualization 2: visualizing the input parameters marginal distributions (the range is normalized from -1 to 1). Also, a histogram of agent scores for generated parameters and a histogram of scores for random parameters are shown in the right column.
| GP Regression | GMM 50% | Multi-SVM | BBGAN(ours) |
|---------------|--------|-----------|-------------|

Figure 10: **Qualitative Comparison for YOLOV3 Experiments**: Comparing the distribution of the best baselines with the distribution learned by our BBGAN. The samples shown are from the truck class experiment.
Figure 11: **Training Data for YOLOV3 Experiment** - Some sample images from the dataset used for object detection with YOLOV3. Note that in the actual dataset each object has a random color regardless of its class. For clarity we uniformly color each class in this figure.
Figure 12: **Environment for Self-driving** - Some samples of the CARLA [14] simulator environment.
Figure 13: **Training Data for Self-driving** - Visualization of training data distribution for 2 parameters (camera height, camera pitch angles).
Figure 14: **Environment for UAV Racing** - Some samples of the Sim4CV [30] simulator environment.
Figure 15: **Training Data for 3-Anchors UAV Racing** - Some sample tracks from the dataset with 3 control points.

Figure 16: **Training Data for 4-Anchors UAV Racing** - Some sample tracks from the dataset with 4 control points.
Figure 17: **Training Data for 5-Anchors UAV Racing** - Some sample tracks from the dataset with 5 control points.
H. Analysis

H.1. Diagnosis

To identify weaknesses and cases that result in systematic failure for the YOLOv3 detector, we fix some semantic parameters and attack the others. We conduct two different case studies. The first one involves the camera viewpoint and light direction; these have the largest impact in determining the pixel values of the final image. The second case study examines occlusion, which is one of the most common causes of detection failure. In both cases, we focus on two classes that are relevant in the autonomous driving application: cars and motorbikes. Since we have several 3D models for each class, we include the effect of model shapes in the analysis. We use roughly sampled models with homogeneous texturing (obtained from the training in Section 4.1) as well as detailed fully-textured models. This also has the nice side-effect to show how well our insights transfer to more realistic renderings and ultimately the real world. In total, we consider five different scenarios per case. Scenario 1: simple car, Scenario 2: detailed car, Scenario 3: simple motorbike, Scenario 4: detailed motorbike 1, Scenario 5: detailed motorbike 2. We use these scenarios for both cases.

Case 1: Viewpoint.

We restrict the number of parameters to 4 (θ_{cam}, φ_{cam}, θ_{light}, φ_{light}), and fix the object class and RGB colors (pure blue). Figures 21, 22, 23, 24 and 25 show qualitative results for samples generated by learning a BBGAN on each scenario in the viewpoint case. Figures 19 and 20 visualize the learned distribution in scenario 3 and scenario 5 and some examples of transferability to real-world.

Case 2: Occlusion.

Since occlusion plays an important role in object misdetection, we introduce an occlusion experiment. Here, we investigate how occlusion (e.g. by a pole) can result in failure of a detector (e.g. from which viewpoint). Therefore, we include the camera viewpoint angles (θ_{cam}, φ_{cam}) and introduce a third parameter to control horizontal shift of a pole that covers 15% of the rendered image and moves from one end to another. The pole keeps a fixed distance to the camera and is placed between the camera and the object. Figures 26, 27, 28, 29 and 30 show qualitative results for samples generated by learning a BBGAN on each scenario in the occlusion case.

H.2. Transferability

Across Shape Variations.

We use the same scenarios as above to construct transferability experiments. The goal is to validate generalization capabilities of the learned fooling distribution from one scenario to another. Also, it shows what role the model shape plays with regard to the strength of the learned attacks. Tables 6 and 7 show the transferability of the adversarial attacks for Case 1 and Case 2. We see that most attacks transfer to new scenarios that are similar, indicating the generalization of the learned fooling distribution. However, the attacks that were learned on more detailed CAD models transfer better to generic less detailed models (e.g. PASCAL3D[49] and ShapeNet[10] models).

To validate generalization capabilities of the learned fooling distribution, we learn this distribution from samples taken from one setup and then test it on another. Table 6 shows that adversarial distributions learned from detailed textured models transfer better (i.e. maintain similar AFR after transferring to the new setup) than those learned from rough ones from Pascall3D [49] and ModelNet [48]. This observation is consistent with that of [3].

Table 6: Case 1: viewpoint attack transferability: Attack Fooling Rate for sampled attacks on each scenario and transferred attacks from one scenario to another. Random attacks for each scenario are provided for reference.

Virtual to Real World.

To demonstrate the transferability of the fooling parameter distribution to the real world, we photograph a toy motorbike, similar to the 3D model we are attacking. We use a mobile phone camera and an office spotlight to replace the light source in the virtual environment. The photos are taken under different camera views and lighting directions (uniform sampling). We also take photos based on samples from the distribution learned by the BBGAN. We apply the
Table 7: **Case 2: occlusion attack transferability.** Attack Fooling Rate for sampled attacks on each scenarios and also for transferred attacks from one scenario to another. Random attacks for each scenario are put for reference.

| Scenario # | BBGAN | Random | Transferred | Scenario # | BBGAN | Random |
|------------|-------|--------|-------------|------------|-------|--------|
| 1          | 96.8% | 58.0%  | 2           | 36.0%      | 32.4% |
| 2          | 94.8% | 32.4%  | 1           | 77.2%      | 58.0% |
| 3          | 95.6% | 52.0%  | 4           | 90.8%      | 37.2% |
| 3          | 95.6% | 52.0%  | 5           | 94.0%      | 39.6% |
| 4          | 99.2% | 37.2%  | 3           | 100.0%     | 52.0% |
| 5          | 100.0%| 39.6%  | 3           | 100.0%     | 52.0% |

YOLOv3 detector on these images and observe the confidence score for the ‘motorbike’ class of interest. On the samples generated from the BBGAN distribution, the attack fooling rate is 21% compared to only 4.3% when picking a random viewpoint. In Figures 19 and 20, we visualize the fooling distribution generated by our BBGAN and provide some corresponding real-world images.

**Visualization.** In Figures 19 and 20, we visualize of the fooling distribution generated by our BBGAN in the two previous experiments (Section H.1). We also include some real-world images captured according to the parameters generated by the BBGAN.

Figure 19: **Analysis: Visualization of the Fooling Distribution.** We fix the object to be a car and fix the distance to the camera and train a BBGAN to learn the fooling camera and light source angles to fool the YOLOV3 detector. **Top:** on the right we plot the camera positions and light source directions of 250 sampled parameters in a 3D sphere around the object. On the left we show how taking real photos from the same rendered angles of some toy car confuses the YOLOV3 detector as the rendered image. **Bottom:** on the right we visualize the distribution of parameters normalized from (-1,1), while on the left we visualize the histogram of scores (0 to 1) of the learned parameters distribution vs random distribution.
Figure 20: Analysis: Visualization of the Fooling Distribution. We fix object to be a motorbike and fix the distance to the camera and train a BBGAN to learn the fooling camera and light source angles to fool the YOLOV3 detector. Top: on the right we plot the camera positions and light source directions of 250 sampled parameters in a 3D sphere around the object. On the left we show how taking real photos from the same rendered angles of some toy motorbike confuses the YOLOV3 detector as the rendered image. Bottom: on the right we visualize the distribution of parameters normalized from (-1,1), while on the left we visualize the histogram of scores (0 to 1) of the learned parameters distribution vs random distribution.

Figure 21: Case 1, Scenario 1, Qualitative Examples: generated by BBGAN

Figure 22: Case 1, Scenario 2, Qualitative Examples: generated by BBGAN

Figure 23: Case 1, Scenario 3, Qualitative Examples: generated by BBGAN
Figure 24: Case 1, Scenario 4, Qualitative Examples: generated by BBGAN

Figure 25: Case 1, Scenario 5, Qualitative Examples: generated by BBGAN

Figure 26: Case 2, Scenario 1, Qualitative Examples: generated by BBGAN

Figure 27: Case 2, Scenario 2, Qualitative Examples: generated by BBGAN
Figure 28: Case 2, Scenario 3, Qualitative Examples: generated by BBGAN

Figure 29: Case 2, Scenario 4, Qualitative Examples: generated by BBGAN

Figure 30: Case 2, Scenario 5, Qualitative Examples: generated by BBGAN
I. Insights Gained by BBGAN Experiments

I.1. Object Detection with YOLOV3:

In our YOLOV3 experiments, we consistently found that for most objects top rear or top front views of the object are fooling the YOLOV3 detector. Furthermore, the light angle which will result in highest reflection off the surface of the object also results in higher fooling rates for the detector. The color of the object does not play a big role in fooling the detector, but usually colors that are closer to the background color tend to be preferred by the BBGAN samples (as shown in the qualitative examples). From the analysis in Section H of transferability of these attacks, we note that attacks on more detailed CAD shapes and models transfer better to less detailed shapes, but the opposite is not true.

I.2. Self-driving cars:

In our experiments we found that weather is the least important parameter for determining success. This is probably due to the fact that the driving policy was trained on multiple weather conditions. This allows for some generalization and robustness to changing weather conditions. However, the driving policy was trained with a fixed camera. We observe, that the driving policy is very sensitive to slight perturbations of the camera pose (height and pitch).

I.3. UAV Autonomous Navigation:

We observe that the UAV fails if the track has very sharp turns. This makes intuitive sense and the results that were produced by our BBGAN consistently produce such tracks. For the tracks that are only parameterized by three control points it is difficult to achieve sharp turns. However, our BBGAN is still able to make the UAV agent fail by placing the racing gates very close to each other, thereby increasing the probability of hitting them.