OUT-DOMAIN EXAMPLES FOR GENERATIVE MODELS

A PREPRINT

Dario Pasquini  
Department of Computer Science  
Sapienza University  
Rome, Italy  
pasquini@di.uniroma1.it

Marco Mingione  
Department of Statistics  
Sapienza University  
Rome, Italy  
marco.mingione@uniroma1.it

Massimo Bernaschi  
Institute for Applied Computing (IAC)  
CNR  
Rome, Italy  
massimo.bernaschi@cnr.it

March 8, 2019

ABSTRACT

Deep generative models are being increasingly used in a wide variety of applications. However, the generative process is not fully predictable and at times, it produces an unexpected output. We will refer to those outputs as out-domain examples. In the present paper we show that an attacker can force a pre-trained generator to reproduce an arbitrary out-domain example if fed by a suitable adversarial input. The main assumption is that those outputs lie in an unexplored region of the generator’s codomain and hence they have a very low probability of being naturally generated. Moreover, we show that this adversarial input can be shaped so as to be statistically indistinguishable from the set of genuine inputs. The goal is to look for an efficient way of finding these inputs in the generator’s latent space.

1 Introduction

The existence of adversarial examples has been demonstrated for a quite large set of deep learning architectures [10][1][36]. An adversarial input is a carefully forged data instance that aims at driving the model into an incorrect or unexpected behaviour. Moreover, the adversarial setup requires that those instances must be as much as possible indistinguishable from genuine inputs.

In the present work, motivated by the extensive studies carried out on adversarial inputs for discriminative models, we extend the adversarial context into the increasingly popular generative models field. In particular, we focused on the most promising class of architectures, called Generative Adversarial Networks (GANs) [9]. GANs implicitly perform generative modeling of a target data distribution by training a deep neural network architecture. This is composed by two neural networks, a generator and a discriminator that are trained simultaneously in a zero-sum game. In the end, the generator learns a deterministic mapping between a latent representation and an approximation of the target data distribution. What we show with the present work is that a pre-trained generator can be forced to reproduce an arbitrary output if fed by a suitable adversarial input. In particular, our findings show that the data space, defined by the
generator, contains data instances having very low probability of lying in the space of the expected outputs (i.e., the target data distribution). We will refer to those outputs as out-domain examples and to the relative adversarial inputs as out-domain latent vectors or OLV in short. Figure 1 shows a set of out-domain examples for a generator trained by using a Progressive GAN [14]. In that example, we found a set of inputs capable to force the generative model to produce images completely different from those belonging to the generator training dataset, i.e., the CelebA dataset [22]. Moreover, we show that those OLV’s can be forged in order to be statistically indistinguishable from known-to-be trusted inputs. The existence of such adversarial inputs raises new practical questions about the use of GAN generators in an untrustworthy environment, as a web application. The main contributions of the present paper may be summarized as follows:

1. We show that a generator may be forced to produce out-domain data instances which are arbitrarily different from those for which the generator is trained. Our experiments refer to three common image datasets and for standard and conditional GAN architectures: Deep Convolutional GAN (DCGAN) [30] and Auxiliary Classifier GAN (ACGAN) [28].

2. We propose a first type of adversarial input for not encoder-based generative models.

3. We investigate the nature of out-domain examples showing that their quality strongly depends on the dimension of both the latent and the data space.

2 Background

The objective of a generative model is to learn a probability distribution \( \hat{p}_X \) that approximates a target data probability distribution \( p_X \). Actually, in general, \( p_X \) is unknown and it can only be inferred by a limited set of samples. One of the most powerful approaches to train a generative model is the recently proposed Generative Adversarial Networks (GANs) framework. GANs require the simultaneous training of two neural networks: a generator \( G \) and a discriminator \( D \). This unsupervised training process is modeled as a zero-sum game where \( D \)’s objective is to maximize the probability of discriminating between \( \hat{p}_X \) and \( p_X \), whereas \( G \)’s objective is to make \( \hat{p}_X \) and \( p_X \) indistinguishable in order to mislead \( D \). A trained generator \( G \) can be intended also as a deterministic mapping function between the latent space defined by \( p_Z \) and the data space defined by \( \hat{p}_X \), i.e., \( G: Z \rightarrow X \). In other words, during the training process, the neural network \( G \) receives as input a vector \( z = (z_1, \ldots, z_n) \) and produces \( x = G(z) \). Each element of \( z \) is a realization of a random vector \( Z_1, \ldots, Z_n \), with \( Z_i \overset{i.i.d.}{\sim} p_Z \), where \( p_Z \) is an arbitrary density function. Then, the optimization problem can be easily summarized by the following formulation:

\[
\min_{\theta_G} \max_{\theta_D} \mathbb{E}[\log(D(x))] + \mathbb{E}[\log(1 - G(z))]
\]

where \( \theta_G \) and \( \theta_D \) are, respectively, the parameters of the generator and the discriminator network and \( x \) is an instance from the training set \( X \).

The use of random latent instances as input of \( G \) makes possible to explore the data space by generating new data instances, not necessarily available in the training set.

Many extensions to the original GAN framework have been successfully developed [31][28][30]. One of the most influential work is [30], in which the DCGAN architecture was proposed. DCGAN is capable of exploiting the full potential of Convolutional Neural Network (CNN) in both the generator and discriminator perspective. The GANs framework can be easily extended to train conditional generators using the ACGAN architecture [25][28]. In this case, a supervised training approach is used to the purpose of learning a probability distribution conditioned to a set of classes \( Y \). During this training process, the class \( y \) is chosen randomly (e.g., with uniform probability) from the set \( Y \) including all the possible classes. Then, the generator is modeled as a bivariate function that receives as input an instance of the latent space \( z \) labelled with its related class \( y \in Y \). ACGAN architecture improves the performance of the training process by adding an auxiliary classification task to the discriminator. The latter outputs two probability distributions, the first over the source (i.e., the probability that the instance comes from \( p_X \)) and the second over the class labels (i.e., the probability that the instance belongs to class \( y \)). In this case, the optimization problem can be parametrized by extending [1] as follows:

\[
L_{source} = \mathbb{E}[\log(D_{source}(x))] + \mathbb{E}[\log(D_{source}(1 - G(z)))]
\]

1 A pre-trained Progressive GAN available at https://tfhub.dev/google/progan-128/1
2 Renamed Adversarial Training in this context.
3 which is a function of \( G \).
4 whereas \( D : X \rightarrow [0, 1] \).
where $L_D$ and $L_G$ are the loss functions of the discriminator and the generator, respectively.

### 3 Related works

Some examples of adversarial inputs for encoder-based generative models like 
Variational AutoEncoder (VAE) and VAE-GAN are analyzed in [19]. In the proposed scenario, given a data instance $x$, an attacker aims at producing an instance $\hat{x}$ that differs in a limited way from $x$ but which is capable of driving the VAE to reconstruct a $\tilde{x}$ far from the original $x$. The reconstructed $\tilde{x}$ can be an approximation of an arbitrary data instance chosen from the attacker. All the results described by the authors refer to $\tilde{x}$ as if it came from the same data distribution on which the VAE is trained. A similar attack scenario has been investigated also in [35]. At the best of our knowledge no other form of adversarial input targeting GAN generators has been proposed.

Many works investigate the possibility of finding or exploiting an inverse mapping from the data space to the latent space of a pre-trained generator [4, 20, 27]. In [27] a pre-trained generator $G$ is used to invert a discriminative model $C$ to the purpose of synthesizing novel images. It is noteworthy that the authors report a first case of partial out-domain example. In particular, a generator trained on ImageNet was able to reproduce images belonging to classes known to $C$ but unknown to $G$.

Additionally, a technique to map images into a latent representation with a pre-trained generator is proposed in [4] and in [20]. In those works, the authors mention the possibility of mapping images, which are not present in the training set, into the generator latent space. That is shown only for images coming from the same distribution on which the model is trained. The proposed inversion technique is essentially the same used in the present work and it is based on the direct optimization of the generator input by a gradient descent based approach. In [4] during the optimization process, the latent vectors are encouraged to be similar to those of the latent prior distribution by adding a penalty term in the loss function. This term is a weighted sum of the discrepancy between the mean and standard deviation of the latent vector and the latent prior distribution. We exploited the possibility of extending this penalty term beyond the second moment, given that just two moments might not be sufficient to correctly identify the latent vectors.

The same generator inversion technique is used in [32] as a defense against adversarial examples. In this work, the adversarial examples are mapped to unperturbed data instances by inverting a generator trained on the data distribution of clear data. The proposed model inversion aims at finding the closest generator codomain element for each input of a discriminative model $f$. At the end, these codomain elements are taken as input by $f$ instead of the original untrusted data.

A similar technique is also used in [21] to perform a data membership attack against generative models. That kind of attack aims at inferring the presence of a data instance $x$ in the training-set used during the training of a generative model $G$. In this case, the generator inversion is carried out by a neural network attacker called $A$. This attacker is trained as an encoder for $G$. Given a data instance $x$, the latent vector $z = A(x)$ is used to estimate the chances of $x$ of being in the $G$’s dataset by calculating the distance between $G(z)$ and $x$.

### 4 Out-domain examples

Let $X$ be the set of all possible data that can be generated by $G$ and let $Z$ be the set of all the possible latent vectors coming from $p_Z$. Then, an out-domain example for a generator $G$ is defined to be an element $\hat{x} \in X$ such that:

\[
G(\hat{z}) = \hat{x}
\]

\[
p_X(\hat{x} \in X) \leq \epsilon_x \quad \text{and} \quad p_Z(\hat{z} \notin Z) \leq \epsilon_z
\]

where $\hat{z}$ is the out-domain latent vector (OLV) used to generate an out-domain example $\hat{x}$ and both $\epsilon_x$ and $\epsilon_z$ are negligible probabilities. The underlying assumption of (3) is that the probability of $\hat{x}$ of belonging to the set of expected outputs is negligible. In the present work, we will refer to the set of expected outputs of a generator with the term **domain**. The domain of a generator can also be intended as the semantic contents defined by $p_X$. To the purpose of finding a suitable out-domain latent vector $\hat{z}$, we choose a target instance $\hat{x}$ and then, we look for the $\hat{z} \in Z$ such that there is the minimum distance between $\hat{x}$ and $\hat{x} = G(\hat{z})$. We refer to this process with the expression **latent search**. Coherently with (3), the target instance $\hat{x}$ is chosen ad hoc to be out of the generator’s domain. In our experiments, this

\[
\text{For instance, the domain of the MNIST dataset is the set of digits representation and the domain of CelebA dataset is a set of human faces.}
\]
is done by sampling \( \hat{x} \) from a dataset different and enough distant, from the semantic viewpoint\(^6\) from the one used in the training process of the generator \( G \). This scenario, is depicted in Figure 2 where a set of out-domain examples from a DCGAN generator trained on the CIFAR10 (right panel) is reported. In this case, 25 target instances have been randomly chosen from five datasets different from CIFAR10.

In addition, as required by \( \hat{z} \), an out-domain latent vector \( \hat{z} \) is considered a valid input for the generator \( G \) if it lies in a dense region of the latent space. This implies that \( \hat{z} \) must be statistically indistinguishable from a valid latent vector sampled from the latent probability distribution \( p_Z \). In the adversarial perspective, this means that a defender is unable to tell apart a valid latent vector from an out-domain latent vector before the generation of \( \hat{x} \).

4.1 Motivating adversarial scenario

The recent success of generative models in the scientific\(^5\) and in the entertainment field\(^12,37\), inspired the development of many GANs based software applications. These are often in the form of a web service with an interactive interface by which the user provides direct or indirect input to the model\(^7\). Assuming a white-box access to the generator model, an attacker can find out-domain latent vectors capable of driving the service to produce inadequate contents such as pornographic or offensive material. The attacker can use these out-domain examples in order to perform a very effective and straightforward defacing attack direct to the generator owner. Indeed, this type of web application and software allows to share\(^8\) and save internal copies of the images created by the users. This scenario resembles a reflected or stored Cross-site Scripting (XSS) attack where the attacker is able to arbitrary modify an image in the web page. The white-box assumption is supported by the observation that often these applications, in order to reduce the server load, run the generative model in the client-side\(^9\) and additionally pre-trained versions of open-source generator are frequently used.

We assume that the owner (referred as defender) performs a validation process on \( \hat{z} \) before the calculation of \( \hat{x} = G(\hat{z}) \). This validation can be intended as a function \( v : \mathcal{Z} \rightarrow \{0, 1\} \). Therefore, the defender accepts to calculate \( \hat{x} \) if and only if \( v(\hat{z}) \) is equal to 1. In our attack scenario, this function \( v \) is represented by a distributional hypothesis test. The null hypothesis \( (H_0) \) is that the vector \( \hat{z} \) is sampled from \( p_Z \). Thus, given a test statistic \( t \) and for fixed type I error \( \alpha \), the decision rule can be formalized as follows:

\[
Pr(T \geq t|H_0) \geq \alpha \Rightarrow v(\hat{z}) = 1
\]  

(4)

where \( T \) is the distribution of the test statistics under \( H_0 \) and \( Pr(T \geq t|H_0) \) corresponds to the classic \( p \)-value of confirmatory data analysis. The same scenario can be easily extended to conditional generators. In that case, we assume that the defender is able to arbitrary chose and fix a data class \( y \in Y \). The attacker aims at finding a suitable out-domain latent vector for the conditioned generator \( G(\cdot|Y = y) \).

---

\(^6\)For instance, we can consider CIFAR10 and CelebA, sufficient distant but MNIST and SVHN not.
\(^7\)One example can be found here: https://make.girls.moe.
\(^8\)On social networks.
\(^9\)using framework as https://js.tensorflow.org/
5 Methodology

As mentioned in Section 4, out-domain examples can be found by looking for the closest representation of an arbitrary chosen target instance in the data space defined by $G$. Actually, by leveraging the differentiable nature of the generator and the structure of a well formed latent-data mapping [30], we can transform this searching problem in an efficient optimization process as follows:

$$L(\hat{x}, z) = d(\hat{x}, G(z)) + \rho(z)$$

$$\hat{z} = \arg\min_{z \in \mathcal{Z}} L(\hat{x}, z)$$  \hspace{1cm} (5)

Where $\hat{x}$ is a given target instance, $d$ is a distance function and $\rho$ is a penalty term applied to $z$. The purpose of a penalty term is to force the solution $\hat{z}$ to be consistent with $p_Z$. More precisely, $\rho$ is defined as the weighted sum of the squared difference of the first $k$ sample moments of $z$ and the theoretical moments of a random variable $Z \sim p_Z$.

$$\rho(z) = \sum_{i=1}^{k} \omega_i ||\mu_Z(i) - \hat{\mu}_z(i)||^2_2$$  \hspace{1cm} (6)

Where $\mu_Z(i)$ is the $i^{th}$ moment of $Z$ and $\hat{\mu}_z(i)$ is the $i^{th}$ sample moment of the latent vector $z$. The parameter $\omega_i$ is the weight assigned to the $i^{th}$ moment difference. In the case of conditional generators, the searching process is performed by fixing a class $y$ as input of the generator function. This implies that the optimization process acts only on the latent vector $z$ and cannot modify the class representation $y$. More formally, in the conditional case, the problem can be reformulated as:

$$L(\hat{x}, z) = d(\hat{x}, G(z, y)) + \rho(z)$$

$$\hat{z} = \arg\min_{z \in \mathcal{Z}} L(\hat{x}, z)$$  \hspace{1cm} (7)

assuming that $y$ is randomly chosen from $Y$ by the defender.

Starting from a random initialization of $z$, say $z_0$ obtained by sampling from $p_Z$, we iteratively update the current latent vector according to the following rule:

$$z_{i+1} = z_i + \eta \nabla L(\hat{x}, z_i), \quad i = 1, \ldots, N$$  \hspace{1cm} (8)

and where $\eta$ is the learning rate. At each iteration of the optimization process, the distance function $d$ is computed between the target $\hat{x}$ and $G(z_i) = \hat{x}_i$. We tested and compared two distance functions: the mean squared error (MSE) and the cross entropy (XE). It is noteworthy that in the cross entropy case, the softmax function is used in order to ensure the unitary sum in both $\hat{x}$ and $G(z)$. However, given its non-bijectivity, we force the comparison between the target and generated image to be scale invariant. Nonethelss, although this approach diverts from the original objective of founding the closest codomain instant to $\hat{x}$, XE is able to provide a very good approximation (at least in the visual form) of $\hat{x}$ with fewer training iterations than MSE.

The penalty term is used to ensure the indistinguishability of the out-domain vector from a trusted input. The main objective is to find $\hat{z}$ such that the probability of $\nu(\hat{z}) = 1$ is maximized. This can be obtained by forcing the out-domain latent vector to have moments equal to those of a random variable distributed as $p_Z$. Indeed, in probability theory, Moment Generating Functions (MGFs) have great practical relevance not only because they can be used to easily derive moments, but also because they uniquely identify a probability distribution function, a feature that makes them a handy tool to solve several problems. The MGF (if it exists) can be seen as an alternative specification useful to characterize a random variable. On one hand, the MGF can be used to compute the $n^{th}$ moment of a distribution as the $n^{th}$ derivative of the MGF evaluated in [12]. On the other hand, a MGF is useful to compare two different random variables, studying their behavior under limit conditions. Given a random variable $X$, its MGF is defined as the expected value of $e^{tX}$:

$$M_X(t) = \mathbb{E}(e^{tX}), \quad t \in \mathbb{R}$$  \hspace{1cm} (9)

If $e^{tX}$ holds, then the $n^{th}$ moment of $X$, denoted by $\mu_X(n)$, exists and it is finite for any $n \in \mathbb{N}$:

$$\mu_X(n) = \mathbb{E}(X^n) = \frac{\partial^n M_X(t)}{\partial t^n} \bigg|_{t=0}$$  \hspace{1cm} (10)

Footnotes:

10 Only the brightness rate between pixels is compared.

11 In the Euclidean sense.

12 The idea is to write the series expansion of $e^{tX}$ and then apply the expected value both to the LHS and the RHS of the equation. For further details see [8].

5
6 Results

In our experiments, we tested and compared two common prior distributions, i.e., the standard normal and the continuous uniform distribution in $[-1, 1]$. Given the constraint imposed by the latter, we perform a hard clipping on $z$ values in order to force the latent vector to lie in the allowed hypercube. As proposed in \cite{20}, we tested the stochastic clipping method but results showed no substantial improvement.

We did not apply any clipping method for the normal prior distribution. Empirically, it has been observed that the penalty on the moments is sufficient to guarantee that $z$ assumes values in an acceptable range.

The quality of the out-domain examples is evaluated on different DCGAN generators and on conditional generators trained within an ACGAN framework. For the sake of exposition, we will refer to each trained generator with the following compact notation:

$\text{[Architecture]-[Training-dataset]-[Latent prior]-[Latent space dimension]}$

In particular, a generator is trained for any combination of the followings:

- **Architecture**: DCGAN, ACGAN
- **Training Dataset**: CIFAR10 \cite{17}, SVHN \cite{26} and a simple variation of MNIST \cite{19}, called ColorMNIST\footnote{Only for DCGAN generators},
- **Latent space dimension**: \{100, 256, 512\}
- **Latent prior distribution**: $N(0, 1)$ and $U[-1, 1]$

For instance, DCGAN-CIFAR10-Normal-100 defines a DCGAN generator trained on CIFAR10 with a normal latent prior distribution and latent space dimension equal to 100. The ColorMNIST dataset is obtained by applying a random background color to the original MNIST. The reason of that modification is to offer to the generator the chance of representing a larger set of output while keeping virtually unaltered the complexity of MNIST. All the generators and discriminators’ architectures as well as the hyper-parameters and the training process are the same proposed in \cite{30}. We tested three values of the latent space dimension that are commonly used in literature.

The weights used in the regularization function $\rho$ are listed in Table \ref{tab:weights}. The validation process of the out-domain vectors is performed by fixing $\alpha = 0.05$, $k = 4$ for the normal prior and $k = 6$ for the uniform prior. We performed three different distributional tests, i.e., Kolmogorov-Smirnov, Shapiro-Wilk\footnote{By adding the colors to MNIST images we let the generator learn a larger number of RGB triplets.} and Anderson-Darling\footnote{only for the Normal prior.}. Results showed that, given the penalty term $\rho$ and an OLV, all the distributional tests bring to the same decision. The following results refer to the Anderson-Darling test \cite{2}, which was finally chosen since its test statistics is based on the Cumulative Distribution Function CDF \cite{31} and, compared to other tests, it places more weight on the values in the tails of the distribution.

We defined a test-set of target instances for the purpose of evaluating the capability of different generators to reproduce out-domain examples. This test-set contains randomly chosen instances from four image datasets i.e., Omniglot \cite{23}, CelebA \cite{22}, UT-Zap50K \cite{38} and Tiny ImageNet \cite{24, 15}. A random sample of 32 images is selected for each dataset for a total of 128 target instances. In the case of Tiny ImageNet, the images are sampled from classes which are different from those of CIFAR10. All images are forced to share the same dimension of $32 \times 32$ pixels and to be normalized in the interval $[0, 1]$. To simplify the understanding of the results, the target distance function used for all the experiments is the Mean Squared Error (MSE). The average Mean Squared Error MSE\footnote{These datasets have null particular domain intersection with the training datasets.} and the percentage of successfully passed statistical tests are computed on the test-set and used as main evaluation scores. In the latent search process, the Adam \cite{15} optimizer is used with a learning rate equal to 0.01. All the experiments are performed using the TensorFlow framework\footnote{The MSE is calculated between the target distance and the generated out-domain example.}. The most relevant codes used for the present work along with an interactive proof of concept are available on: \url{https://github.com/pasquini-dario/OutDomainExamples}

6.1 DCGAN

Table \ref{tab:results} shows the results related to the DCGAN generators. All the forged OLV pass successfully the distributional test, regardless of the chosen prior distribution. Several checks on the biases of the OLV have been carried out providing pretty good results (see Section \ref{sec:bias} of the supplementary material for further details). This is due to the fact that the penalty term $\rho$ strongly constraints the values of the latent vectors in a well defined range. It is worth to notice that relaxing the moments penalty during the optimization process \footnote{The $MSE$ is calculated between the target distance and the generated out-domain example.} would reduce further the $MSE$. In contrast to
Figure 3: Out-domain examples for a set of DCGAN architectures trained with uniform latent prior. The central panel shows several different randomly chosen targets from the test-set. The upper panel shows the variability in the out-domain generation when the latent space dimension is fixed to 512 and the training dataset of the generator varies. The lower panel shows the variability in the out-domain generation when the training dataset of the generator is fixed to CIFAR10 and the latent space dimension varies.

Figure 4: Average MSE compared to the estimated entropy of each training set.

Figure 5: Two-dimensional representation of out-domain latent vectors for the DCGAN–CIFAR10–Uniform–100. Out-domain latent vectors with target instances coming from the same dataset are represented with the same marker.

In-domain inversion [4], we can state that out-domain examples do not take any significant advantage from latent vectors statistically close to those used during the training process. Figure 3 shows a set of target instances and related out-domain examples for a total of six generators. The upper panel reports the out-domain examples produced by three generators trained on different training sets but with same latent space dimension and prior. When the training set of the generator is ColorMNIST, the method fails in finding suitable OLV capable of reproducing the target images. For the other two, the generator is able to provide a valid reconstruction for all the targets. The failure of ColorMNIST may be connected to the fact that it is less heterogeneous with respect to SVHN and CIFAR10. By heterogeneity we intend the actual number of different pixels which are necessary in order to reproduce the same heterogeneity of the whole set (i.e., the entropy). It is reasonable to expect that the larger the variety of images in the training set, the larger will be the set of potential out-domain examples reproduced by the generator. As an estimator of that variety, we computed the Shannon entropy \[ H \] for a sample of $2^{10}$ images from each training set. Results are depicted in Figure 4 and show that there is a strong dependence between the average MSE and the entropy of the training set. Figure 5 depicts a two-dimensional projection of a set of out-domain latent vectors and latent vectors directly sampled by $p_z$. This representation is obtained by applying the dimension reduction algorithm called t-distributed Stochastic Neighborhood Embedding (t-SNE) [24] on vectors of size 100. It is possible to note how the out-domain latent vectors tend to be uniformly distributed in the space. In the case of the Omniglot dataset, the OLV tend to cluster in a specific vicinity of the latent vectors sampled at random.

\[ \text{on the normalized pixels’ distribution.} \]
### Table 1: Results concerning the out-domain generation process on the test-set for all the DCGAN generators. The column Test Succ. reports the percentage of out-domain latent vectors which successfully passed statistical tests. The column Avg MSE* reports the average MSE in the case of complete relaxation of the penalty term $\rho$.

| Dataset   | Z dim. | (a) Normal Latent distribution |         |         | (b) Uniform Latent distribution |         |         |
|-----------|--------|-------------------------------|---------|---------|-------------------------------|---------|---------|
|           |        | Avg MSE | Test Succ. | Avg MSE* | Avg MSE | Test Succ. | Avg MSE* |
| CIFAR10   | 100    | 0.010646 | 100%       | 0.008881 | 0.012131 | 100%       | 0.009354 |
| CIFAR10   | 256    | 0.005094 | 100%       | 0.003902 | 0.012131 | 100%       | 0.009354 |
| CIFAR10   | 512    | 0.003693 | 100%       | 0.002710 | 0.003603 | 100%       | 0.002826 |
| SVHN      | 100    | 0.018569 | 100%       | 0.011541 | 0.016730 | 100%       | 0.011359 |
| SVHN      | 256    | 0.011374 | 100%       | 0.006970 | 0.012121 | 100%       | 0.007213 |
| SVHN      | 512    | 0.009474 | 100%       | 0.005314 | 0.008323 | 100%       | 0.005594 |
| C.MNIST   | 100    | 0.063097 | 100%       | 0.040453 | 0.059342 | 100%       | 0.037457 |
| C.MNIST   | 256    | 0.052926 | 100%       | 0.029160 | 0.045851 | 100%       | 0.027178 |
| C.MNIST   | 512    | 0.043685 | 100%       | 0.025855 | 0.037946 | 99%        | 0.022415 |

Figure 6: Average MSE for all the DCGAN generators trained on all combinations of dataset, latent space dimension and latent prior distribution.

region and this may be due to its intrinsic homogeneity.

Even if the entropy is a sort of predictor of the success of our method, it is still possible, given a target image, a latent prior and a training set, to enhance the quality of the generated image by increasing the dimension of the latent space. As a matter of fact, we can observe, by looking at the lower panel of Figure 3 that an increase of the latent space dimension makes the generated image more similar to the target one. An additional motivation can be that the latent space acts as an information bottleneck for the target instance during the latent search process. All these possibilities are evaluated in terms of MSE. Figure 6 shows the average MSE for each latent space dimension, latent prior and training set confirming that the more is the entropy of the training set, the higher the probability of success in the generation and, at the same time, the larger the dimension of the latent space, the higher the quality of the reconstruction. Instead, there is no relevant difference in the quality of the out-domain examples when the latent prior distribution varies.

Figure 7 shows three examples of linear interpolation between latent vectors. The first row depicts a smooth and semantic meaningful transaction between two random vectors sampled from $p_Z$, referred as in-domain latent vectors. By semantic meaningful transaction, we mean that each image between the two interpolation points remains coherent with $p_X$. The second row depicts the interpolation between an in-domain vector and an out-domain vector. In contrast with the first case, the transaction is unbalanced and not particularly smooth. From the sequence, it can be noticed that the semantic valid attributes of the starting image, i.e. the black of the hair and the reflection on the forehead, are deformed to recreate the final MNIST digit. The last row shows the extreme case of interpolation between two out-domain latent vectors. In this case, all the intermediate data instances never cross the in-domain set.

### 6.2 ACGAN

Conditional generators are trained to the purpose of enforcing their outputs to be part of a meaningful, from the semantic viewpoint, data class. Typically, this implies a better global coherence and quality in the definition of the generator’s data space. This is especially true for models trained with the ACGAN framework in which the generators are encouraged to produce images that are correctly classified from the discriminator as genuine and belonging to its class.

A target image has a number of pixels equal to $32 \cdot 32 \cdot 3 = 3072$. The biggest latent size analyzed is 512.
The experiments described below aim at finding out if the conditional extension is sufficient to the purpose of preventing the generation of out-domain examples. We trained different ACGAN generators using the same set of parameters reported in Section 6.1. Also the architecture used for the generator and the discriminator is the same used for the previous DCGAN experiments. Training hyper-parameters are the same proposed in [28]. As aforementioned, in the conditional setup, the hypothesis is that the class $y$ is randomly chosen by the defender and the attacker can not modify its representation during the latent search process. Tests and validation are performed as in Section 6.1 but they are evaluated conditionally to each class $y$ in the generators’ classes set. All the tested training sets are composed by 10 classes. In this case, the MSE is calculated as the average over the classes.

We are able to find an out-domain example for each image in the test-set, conditionally to each class. We do not report the results when the training set is ColorMNIST, since it already failed in the less severe DCGAN experiment. As an example, Figure 8 shows the generated out-domain examples, conditionally to each class of CIFAR10, for four randomly chosen target instances in the test-set. It can be noticed that the class has no relevant impact on the quality of the out-domain examples: the attack succeeds regardless of the class. The same happens when attacking the generators trained on SVHN. Results in terms of MSE are summarized in Table 6.2. It is possible to observe that the average MSE is uniformly larger compared to the DCGAN experiments due to the conditional setup. In Figure 10 we also report the distribution of the MSE, conditionally to each class, for each training set. No specific patterns are registered: the MSE distribution is approximately the same for each class and training set. However, there is a slight variability for CIFAR10 given the larger heterogeneity among its classes. The validation of the out-domain latent vectors is the same described in Section 6.1. Also in this case, all the latent vectors pass successfully the Anderson-Darling test. Moments distributions for each dataset, latent prior and latent space dimension are also checked. No relevant difference has been observed with respect to the not-conditional generators experiments.

---

20The only difference is the number of neurons in the generator’s input layer and in the discriminator’s output layer due to the conditional setup
Table 2: Results for any ACGAN generator. Scores are obtained as the average over test-set results for each of the ten classes present in the generator training-set.

| Dataset | Z dim. | (a) Normal Latent distribution | (b) Uniform Latent distribution |
|---------|--------|-------------------------------|-------------------------------|
|         |        | Avg MSE | Test Succ. | Avg MSE | Test Succ. |
| CIFAR10 | 100    | 0.023457 | 100%       | 0.019615 | 100%       |
| CIFAR10 | 256    | 0.013144 | 100%       | 0.009547 | 100%       |
| CIFAR10 | 512    | 0.009075 | 100%       | 0.005944 | 99%        |
| SVHN    | 100    | 0.026686 | 100%       | 0.024732 | 100%       |
| SVHN    | 256    | 0.016879 | 100%       | 0.015268 | 100%       |
| SVHN    | 512    | 0.016291 | 100%       | 0.013707 | 99%        |

Figure 10: MSE distribution conditionally to each class for CIFAR10 (left) and SVHN (right) datasets. Black dashed lines represent the average MSE for CIFAR10 and SVHN datasets respectively, calculated as the average over all the generated out-domain examples for each class of the dataset.

7 Conclusion and further developments

We showed how to forge suitable adversarial inputs capable of driving a trained generator to produce arbitrary data instances in output. This is possible for both conditional and not-conditional generators. Additionally, we showed that an adversarial input can be shaped in order to be statistically indistinguishable from the set of trusted inputs. We also showed that the success of our method strongly depends on two main factors: the heterogeneity of the set on which the generator is trained and the latent space dimension.

In additional experiments we found a set of generators showing a greater resilience to the generation of out-domain examples. In particular, the Non-saturating GANs with ResNet architecture analyzed in [18] shows an inherent difficulty to produce out-domain examples even when the generator is trained on high entropic datasets such as CIFAR10. We conjecture that this property is strongly related to the generator architecture.

In the described adversarial scenario, we supposed that an aware defender can just test the validity of the model’s input in order to evaluate the genuineness of the latent vectors. However, it is possible to imagine a more powerful defender able to verify the generator’s output in order to spot unexpected generation.

As future directions of activity, we expect to i) investigate the generation of out-domain examples for other GAN architectures; ii) study the generation of out-domain examples in contexts other than those of images; iii) define a different way of finding these out-domain latent vectors, by relaxing the constraint imposed by the penalty term \( \rho \); iv) investigate the possibility of training an arbitrary complex generator which is resilient to the generation of out-domain examples; v) evaluate the possibility of extending this defacing attack to a black-box scenario, using a local copy of the generator by queering the remote model, as proposed in [29].

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and
X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] T. W. Anderson and D. A. Darling. A test of goodness of fit. *Journal of the American statistical association*, 49(268):765–769, 1954.

[3] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein GAN. *ArXiv e-prints*, Jan. 2017.

[4] A. Creswell and A. A. Bharath. Inverting the generator of a generative adversarial network (ii). *arXiv preprint arXiv:1802.05701*, 2018.

[5] N. De Cao and T. Kipf. MolGAN: An implicit generative model for small molecular graphs. *ArXiv e-prints*, May 2018.

[6] L. de Oliveira, M. Paganini, and B. Nachman. Learning Particle Physics by Example: Location-Aware Generative Adversarial Networks for Physics Synthesis. *ArXiv e-prints*, Jan. 2017.

[7] J. Donahue, P. Krähenbühl, and T. Darrell. Adversarial feature learning. *arXiv preprint arXiv:1605.09782*, 2016.

[8] W. Feller. *An introduction to probability theory and its applications*, volume 2. John Wiley & Sons, 2008.

[9] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[10] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. *ArXiv e-prints*, Dec. 2014.

[11] S. H. Huang, N. Papernot, I. J. Goodfellow, Y. Duan, and P. Abbeel. Adversarial attacks on neural network policies. *CoRR*, abs/1702.02284, 2017.

[12] Y. Jin, J. Zhang, M. Li, Y. Tian, H. Zhu, and Z. Fang. Towards the Automatic Anime Characters Creation with Generative Adversarial Networks. *ArXiv e-prints*, Aug. 2017.

[13] L. Jost. Entropy and diversity. *Oikos*, 113(2):363–375, 2006.

[14] T. Karras, T. Aila, S. Laine, and J. Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *CoRR*, abs/1710.10196, 2017.

[15] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[16] J. Kos, I. Fischer, and D. Song. Adversarial examples for generative models. *CoRR*, abs/1702.06832, 2017.

[17] A. Krizhevsky. Learning multiple layers of features from tiny images. 05 2012.

[18] K. Kurach, M. Lucic, X. Zhai, M. Michalski, and S. Gelly. The GAN landscape: Losses, architectures, regularization, and normalization. *CoRR*, abs/1807.04720, 2018.

[19] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010.

[20] Z. C. Lipton and S. Tripathi. Precise recovery of latent vectors from generative adversarial networks. *CoRR*, abs/1702.04782, 2017.

[21] K. S. Liu, B. Li, and J. Gao. Generative model: Membership attack, generalization and diversity. *CoRR*, abs/1805.09898, 2018.

[22] Z. Liu, P. Luo, X. Wang, and X. Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[23] B. M Lake, R. Salakhutdinov, and J. B Tenenbaum. Human-level concept learning through probabilistic program induction. 350:1332–1338, 12 2015.

[24] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[25] M. Mirza and S. Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[26] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011.

[27] A. Nguyen, J. Clune, Y. Bengio, A. Dosovitskiy, and J. Yosinski. Plug & play generative networks: Conditional iterative generation of images in latent space. In *CVPR*, page 7, 2017.

[28] A. Odena, C. Olah, and J. Shlens. Conditional Image Synthesis With Auxiliary Classifier GANs. *ArXiv e-prints*, Oct. 2016.

[29] N. Papernot, P. D. McDaniel, I. J. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against deep learning systems using adversarial examples. *CoRR*, abs/1602.02697, 2016.
[30] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015.

[31] S. Ross. *A first course in probability*. Pearson, 2014.

[32] P. Samangouei, M. Kabkab, and R. Chellappa. Defense-gan: Protecting classifiers against adversarial attacks using generative models. *CoRR*, abs/1805.06605, 2018.

[33] S. S. Shapiro. *How to test normality and other distributional assumptions*, volume 3. ASQC Milwaukee, WI, 1990.

[34] Stanford. Tiny imagenet visual recognition challenge.

[35] P. Tabacof, J. Tavares, and E. Valle. Adversarial images for variational autoencoders. *CoRR*, abs/1612.00155, 2016.

[36] C. Xie, J. Wang, Z. Zhang, Y. Zhou, L. Xie, and A. L. Yuille. Adversarial examples for semantic segmentation and object detection. *CoRR*, abs/1703.08603, 2017.

[37] L. Yang, S. Chou, and Y. Yang. Midinet: A convolutional generative adversarial network for symbolic-domain music generation using 1d and 2d conditions. *CoRR*, abs/1703.10847, 2017.

[38] A. Yu and K. Grauman. Fine-grained visual comparisons with local learning. In *Computer Vision and Pattern Recognition (CVPR)*, Jun 2014.
Supplementary materials

7.1 ColorMNIST dataset

Figure 11: ColorMNIST sample.

Figure 11 shows a sample from the ColorMNIST dataset. This dataset is obtained by summing to each of the original MNIST images scaled in [0, 1] a random positive number generated according to a uniform distribution, as described by the Algorithm 1.

Algorithm 1: ColorMNIST generation

Data: X MNIST dataset
Result: X_C ColorMNIST dataset

X_C = ∅;
for x ∈ X do
    x_c = gray_to_RGB(x);
    ∆r, ∆g, ∆b ∼ U(0, 1);
    ∆ = {∆r, ∆g, ∆b};
    x_c += ∆;
    x_c.clip([0, 1]);
    X_c = X_c ∪ {x_c}
end

7.2 Additional results

Weights of the regularization function

Statistical indistinguishability of the out-domain latent vectors

Figure 12 and 13 show the distribution of the biases, with respect to each single moment, for the normal and the uniform priors and for each combination of latent space dimension and dataset. In particular, Figure 12 highlights that the estimation of the odd moments for the normal is precise. The second and the fourth moment are instead slightly overestimated and underestimated, respectively. For the uniform prior, Figure 13 shows that the bias for the second and the sixth moment is slightly positive (overestimation) and for the remainings the estimation is precise, with a quite large variance in the estimation of the third moment. No evident patterns are worth to notice when looking at the training dataset or the latent space dimension changes, for both the prior distributions.

Remark that for the standard normal distribution, 1st and 4th moments are equal to 0, the 2nd (which coincides with the variance) is equal to 1 and the 4th is equal to 3. For the uniform in [−1, 1] instead, the odd moments are all equal to 0, whereas the 2nd is equal to 1/3, the 4th is equal to 1/5 and the 6th is equal to 1/7. Results are shown in Table 3 and 4.
Figure 12: Distribution of the empirical moments’ bias evaluated on the 128 out-domain latent vectors for the standard normal prior and the uniform prior for each combination of training dataset and latent space dimension.

Figure 13: Distribution of the empirical moments’ bias evaluated on the 128 out-domain latent vectors for the standard normal prior and the uniform prior for each combination of training dataset and latent space dimension.

| Dataset     | Dimension | Test success | 1st moment | 2nd moment | 3rd moment | 4th moment | 5th moment | 6th moment |
|-------------|-----------|--------------|------------|------------|------------|------------|------------|------------|
| CIFAR10     | 100       | 100%         | -0.00051   | 1.01152    | -0.00006   | 2.99186    |            |            |
| CIFAR10     | 256       | 100%         | -0.00006   | 1.00498    | -0.00003   | 2.99408    |            |            |
| CIFAR10     | 512       | 100%         | -0.00022   | 1.00293    | 0.00000    | 2.99440    |            |            |
| SVHN        | 100       | 100%         | 0.00047    | 1.01426    | 0.00004    | 2.97407    |            |            |
| SVHN        | 256       | 100%         | 0.00016    | 1.00598    | 0.00003    | 2.98810    |            |            |
| SVHN        | 512       | 100%         | 0.00021    | 1.00373    | -0.00000   | 2.98943    |            |            |
| ColorMNIST  | 100       | 100%         | 0.00066    | 1.01264    | 0.00003    | 2.98561    |            |            |
| ColorMNIST  | 256       | 100%         | -0.00107   | 1.00712    | 0.00007    | 2.98101    |            |            |
| ColorMNIST  | 512       | 100%         | -0.00002   | 1.00530    | -0.00002   | 2.97996    |            |            |

Table 3: Median 1st, 2nd, 3rd and 4th empirical moments of the out-domain latent vectors over 128 simulations for all the combinations of training dataset and latent space dimension with the Normal prior.
Table 4: Median 1st, 2nd, 3rd, 4th, 5th and 6th empirical moments of the out-domain latent vectors over 128 simulations for all the combinations of training dataset and latent space dimension with the Uniform prior.

Further examples of out-domain generation

Further out-domain examples for a DCGAN generator are depicted in Figure 7.2.

Figure 14: Out-domain examples (right column) and relative target instances (left column) for a DCGAN-CIFAR10-Normal-512.