Reprogramming GANs via Input Noise Design*

Kangwook Lee[1], Changho Suh[2] ✉, and Kannan Ramchandran[3]

[1] Department of Electrical and Computer Engineering
University of Wisconsin–Madison, Madison WI, USA
kangwook.lee@wisc.edu
[2] School of Electrical Engineering
Korea Advanced Institute of Science and Technology, Daejeon, South Korea
chsuh@kaist.ac.kr
[3] Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, Berkeley CA, USA
kannanr@eecs.berkeley.edu

Abstract. The goal of neural reprogramming is to alter the functionality of a fixed neural network just by preprocessing the input. In this work, we show that Generative Adversarial Networks (GANs) can be reprogrammed by shaping the input noise distribution. One application of our algorithm is to convert an unconditional GAN to a conditional GAN. We also empirically study the applicability, feasibility, and limitation of GAN reprogramming.

Keywords: GAN· neural reprogramming

1 Introduction

While deep neural networks have revolutionized a wide range of areas, their flexibility is limited as they cannot perform other tasks that are not taken into account during training. Various techniques have been proposed for general-purpose neural networks. Multi-task learning designs a versatile neural network that can perform multiple tasks at the same time when the target tasks are known at training time [25]. Transfer learning can help adapt to another task if the new task is similar enough to the original task [22]. For instance, given a new task, one can slightly modify the parameters of a pre-trained neural network so that it can perform well on the new task [31].

Recently, [5] proposes a new approach to design general-purpose neural networks, which they call *neural reprogramming*. The goal of neural reprogramming is to modify the functionality of a pre-trained neural network just by preprocessing the input. For instance, one can turn an ImageNet classifier into an MNIST classifier just by preprocessing input images. Compared to the standard approaches (multi-task learning and transfer learning), neural reprogramming does not assume any prior knowledge about target tasks and allows for modular design of a large neural network.

* This material is based upon work supported by the Air Force Office of Scientific Research under award number FA2386-19-1-4050.

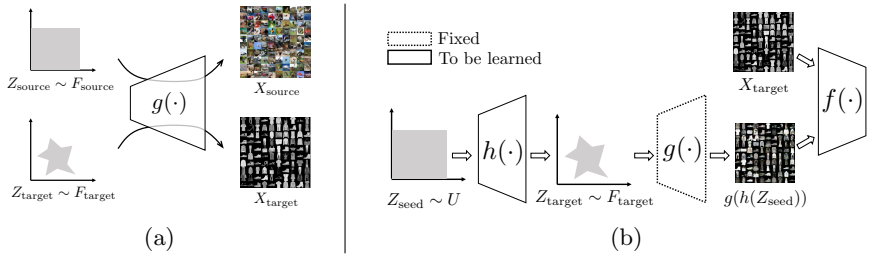


Fig. 1: GAN reprogramming. (a) Given a GAN $g(\cdot)$ that generates X_{source} (e.g., CIFAR10 images) with the latent random variable $Z_{\text{source}} \sim F_{\text{source}}$, reprogramming GAN aims to design another distribution F_{target} such that the generator fed by $Z_{\text{target}} \sim F_{\text{target}}$ yields X_{target} (e.g., FMNIST images). (b) GAN reprogramming algorithm.

This motivates us to investigate reprogramming ideas beyond the classification setting. Specifically we focus on unsupervised learning that has recently been revolutionized via Generative Adversarial Networks (GANs) [8]. A GAN is trained with a latent variable (input noise) with a certain distribution, e.g., uniform or Gaussian distributions. Denote the trained generator by $g(\cdot)$, the random latent variable by Z_{source}, and its distribution by F_{source}. For a well-trained GAN, $g(Z_{\text{source}})$ follows a similar distribution to that of the original training data, say X_{source}. That is, $g(Z_{\text{source}}) \approx X_{\text{source}}$ if $Z_{\text{source}} \sim F_{\text{source}}$, where the notion of '$\approx$' will be defined formally soon. The goal of GAN reprogramming is finding another latent variable distribution F_{target} such that $g(Z_{\text{target}}) \approx X_{\text{target}}$ when $Z_{\text{target}} \sim F_{\text{target}}$. See Fig. 1a for visual illustration. Consider a GAN trained with CIFAR10 images. Assume that the trained generator is able to produce diverse CIFAR10-like images when Z_{source} follows $F_{\text{source}} = \text{Uniform}$. Reprogramming GAN wishes to find a new latent variable distribution F_{target} so that the samples of $g(Z_{\text{target}})$ look like Fashion-MNIST images.

In this work, we propose a simple GAN reprogramming algorithm and show that it can convert unconditional GANs to conditional GANs without labeled datasets. We also study its applicability, feasibility, and limitation via a variety of controlled experiments.

2 Related Work

GANs Goodfellow et al. [8] propose the first GAN algorithm, in which two neural networks called generator and discriminator are alternatively trained. The goal of the generator is to produce realistic fake samples, while the goal of the discriminator is to discriminate fake samples from real ones. It has been shown that the original GAN algorithm minimizes the Jensen-Shannon (JS) divergence between the distribution of fake samples and that of real samples. The Wasserstein GAN (WGAN) is similar to the original GAN, but its goal is to

minimize the Wasserstein distance instead of the JS divergence [1]. It is shown that the WGAN algorithm is more stable than the original GAN algorithm. In the WGAN algorithm, two neural networks, called generator and critic, are trained. The role of the generator is the same as before while the role of critic is to approximate the Wasserstein distance. WGAN-GP (WGAN gradient penalty) uses a gradient-based regularization method to satisfy a certain constraint on the Lipshitz constant [10].

Multi-task learning Multi-task learning (MTL) has been successful in many applications of machine learning [25]. MTL is particularly useful when multiple tasks are similar to each other. The most basic algorithm is based on parameter sharing. That is, one may train a neural network with multiple last layers for different tasks using a combined dataset [2]. This approach is also called 'hard parameter sharing' since it shares exactly the same parameters (except for the last layer) between multiple tasks. On the other hand, in soft parameter sharing, each task maintains its own neural network while keeping multiple neural networks close to each other according to a certain metric [4, 30]. Differently from MTL, neural reprogramming can be applied to a new task *without* needing to know the new task at training time.

Transfer learning Transfer learning exploits the knowledge from previous learning experiences to better solve future tasks [22]. The most popular algorithm is based on sharing the weights of a neural network [12, 21]. That is, given a neural network trained on one task, one simply takes the lower part of the neural network (closer to the input end), treating it as a generic feature extractor. One then trains the remaining part of the neural network. If the original and new tasks are similar enough, such algorithms would perform well. On the other hand, neural reprogramming does not modify the parameters of the given neural network and only prepends the input processing module, allowing for modular design of a general-purpose neural network. To see this, consider N GANs that generate similar outputs. With reprogramming, one can just store the core GAN with N (small) input processors, significantly saving the memory/storage cost.

Neural reprogramming The concept of neural reprogramming was introduced by Elsayed et al. [5]. Specifically, they design an input preprocessor so that a neural network trained for classifying certain types of images can be used to classify other types of images. For instance, they show that it is possible to reprogram an ImageNet classifier as an MNIST classifier. A few recent studies are somewhat related to GAN reprogramming. Nguyen et al. propose an MCMC-based algorithm that can be used to maximize the activation of a carefully chosen neuron [20]. While this algorithm is shown to generate novel images using a fixed GAN, it only works with a specific type of GAN and cannot be applied to GANs trained with standard methodologies. Furthermore, it requires a classifier pre-trained on the target dataset and computationally expensive as it involves the joint training of three neural networks. Engel et al. propose an algorithm for

conditioning an unconditional Variational Autoencoder (VAE) [6]. On the other hand, our algorithm can reprogram a GAN even across different datasets.

Adversarial robustness Recently, it is shown that deep neural networks can be easily fooled when the input is perturbed even with imperceptibly small noise, bringing adversarial robustness of deep neural networks into question [26, 9]. Through the lens of robust optimization, Madry et al. [18] propose an adversarial training algorithm that can make machine learning models robust against adversarial attacks. While most of the studies focus on the robustness of classifiers, a notable exception is made in recent studies [13, 23], where attack scenarios for GANs have been discussed. They show that one can manipulate the latent variable so that a conditional GAN's generated output becomes inconsistent with the specified label. Moreover, they show that one can manipulate the latent code so that the generated sample is similar to a single target. Using our notation, this is equivalent to finding a value of z_{target} such that $g(z_{\text{target}})$ looks like a target sample. This is similar to our work in that it finds latent codes that make the generator produce different types of data. A key distinction, however, is that instead of finding a single value of z_{target}, we find a *distribution* of Z_{target} so that one can generate an arbitrary number of such samples by drawing latent codes from $Z_{\text{target}} \sim F_{\text{target}}$.

3 Reprogramming GANs

3.1 Preliminaries

In this section, we first recall the definition of elementary divergences between two probability measures P_r and P_g. The JS divergence is defined as $D_{\text{JS}}(P_r, P_g) := \frac{1}{2}D_{\text{KL}}(P_r\|P_m) + \frac{1}{2}D_{\text{KL}}(P_g\|P_m)$, where D_{KL} is the Kullback-Leibler (KL) divergence and $P_m = (P_r + P_g)/2$. The first-order Wasserstein distance is

$$W(P_r, P_g) := \inf_{\gamma \in \Pi(P_r, P_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|],$$

where $\Pi(P_r, P_g)$ is the set of all joint distributions whose marginals are respectively P_r and P_g. Slightly abusing notation, we write $W(X, Y)$ for $W(P_X, P_Y)$ if $X \sim P_X$ and $Y \sim P_Y$. By the Kantrovich-Rubinstein duality [27],

$$W(P_r, P_g) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x\sim P_r}[f(x)] - \mathbb{E}_{x\sim P_g}[f(x)], \tag{1}$$

where $\|f\|_L$ is the Lipschitz constant of f.

3.2 Problem Formulation and Algorithm

We formally define the problem of GAN reprogramming and describe our algorithm. Assume that a generator $g : \mathbb{R}^{d_i} \to \mathbb{R}^{d_o}$, which produces d_o-dimensional

data from d_i-dimensional latent codes is given. Consider a latent variable distribution F_{source} used for training. Given an unlabeled dataset X_{target}, we solve the following optimization problem

$$\min_{F_{\text{target}}} W(g(Z_{\text{target}}), X_{\text{target}}), \tag{2}$$

where $Z_{\text{target}} \sim F_{\text{target}}$ is a d_i-dimensional random vector. If we can find F_{target} such that $W(g(Z_{\text{target}}), X_{\text{target}}) \simeq 0$, it implies that the given GAN can be reprogrammed to produce X_{target} instead of X_{source} by drawing latent codes according to F_{target} instead of F_{source}.

We now describe our reprogramming algorithm. Due to the Kantrovich-Rubinstein duality, we have

$$\min_{F_{\text{target}}} \sup_{\|f\|_L \leq 1} \mathbb{E}_{X \sim X_{\text{target}}}[f(X)] - \mathbb{E}_{Z \sim Z_{\text{target}}}[f(g(Z))], \tag{3}$$

where $\|f\|_L$ is the Lipschitz constant of f. In order to parameterize the distribution of Z_{target} and to obtain samples from it, we apply a deep neural network to transform a uniform random variable into another random variable. That is, we first draw a uniform random variable $Z_{\text{seed}} \sim U[0,1]^d$ ($Z_{\text{seed}} \sim U$, for short), and then apply a *code generator* h_θ, $h_\theta : \mathbb{R}^d \to \mathbb{R}^{d_i}$, parameterized by θ. Here, d denotes the dimensionality of a random vector used to generate latent codes. Similarly, define a parameterized family of all functions $\{f_w\}_{w \in \mathcal{W}}$ that are 1-Lipschitz. Using these, (3) becomes

$$\min_{\theta} \sup_{f_w \in \mathcal{W}} \mathbb{E}_{X \sim X_{\text{target}}}[f_w(X)] - \mathbb{E}_{Z_{\text{seed}} \sim U}[f_w(g(h_\theta(Z_{\text{seed}})))]. \tag{4}$$

Since h_θ and f_w are parameterized via neural networks, one can solve the above optimization problem via gradient-based methods. Our algorithm is essentially a simple variation of WGAN-GP [10], where the key difference is that the outer optimization problem is optimized over h_θ instead of g, which is fixed in our setting.

We now describe our algorithm with further details for the sake of completeness. We first draw a minibatch of m pairs of a real data point x and a random number z for generating a latent code. Denote the generated output $g(h_\theta(x))$ by $\tilde{x}$. We now define the loss function for the i-th pair, denoted by $L^{(i)}$. The loss function contains the negative of the objective function since our goal is to maximize it. Moreover, in order to satisfy the constraint $f_w \in \mathcal{W}$, we use the gradient penalty. Specifically, we choose a random data point, say $\hat{x}$, lying in between the real data point x and the fake data point $\tilde{x}$. This can be done by taking a weighted sum of x and $\tilde{x}$, i.e., $\hat{x} = \epsilon x + (1-\epsilon)\tilde{x}$, where $\epsilon \sim U[0,1]$. We then compute the gradient of f measured at $x = \hat{x}$. Using λ as the coefficient for gradient penalty, the total loss function for the i-th pair is

$$L^{(i)} := f_w(g(h_\theta(z))) - f_w(x) + \lambda(\|\nabla_x f_w\| - 1)^2. \tag{5}$$

Using the minibatch of m pairs, we can compute the batch average loss $\frac{1}{m}\sum_{i=1}^{m} L^{(i)}$. We then compute the gradient of the batch loss with respect to

w and use the Adam optimizer to update w. We run this for n_{critic} iterations to closely approximate the Wasserstein distance. We then solve the outer optimization problem by treating that the current cost function is close to the true Wasserstein distance, i.e., the first-order Wasserstein distance is well approximated with the current choice of f_w. While f_w being fixed, we draw a minibatch of m generated data points, and then back-prop the cost function to find the gradient with respect to θ. We then apply the Adam optimizer to update θ. See Fig. 1b for the visual illustration of our algorithm. The pseudocode of GAN reprogramming algorithm is given in Algorithm 1.

Algorithm 1: GAN Reprogramming

Data: pretrained generator $g(\cdot)$ and target data X_{target}
Result: code generator $h_\theta(\cdot)$
initialize w and θ;
while θ *not converged* **do**
  for $t \leftarrow 1$ **to** n_{critic} **do**
    for $i \leftarrow 1$ **to** m **do**
      sample $x \sim X_{\text{target}}$, $z \sim U$, and $\epsilon \sim U[0,1]$;
      $\tilde{x} \leftarrow g(h_\theta(z))$, $\hat{x} \leftarrow \epsilon x + (1-\epsilon)\tilde{x}$;
      $L^{(i)} \leftarrow f_w(g(h_\theta(z))) - f_w(x) + \lambda(\|(\nabla_x f_w)_{\hat{x}}\| - 1)^2$;
    $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^m L^{(i)})$;
  sample a batch of noise $z^{(i)}{}_{i=1}^m$, $z^{(i)} \sim U$, $\forall i$;
  $\theta \leftarrow \text{Adam}(-\nabla_w \frac{1}{m} \sum_{i=1}^m f_w(g(h_\theta(z)))$;

3.3 Toy Example

Consider a fixed generator

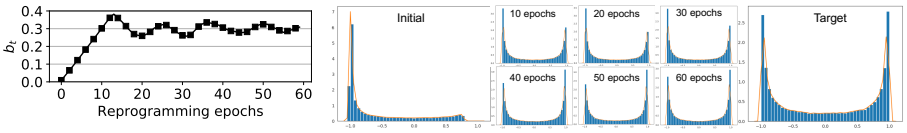
$$g(Z) = \sigma(10(Z - 0.8)) - (1 - \sigma(10(Z - 0.8))),$$

where $\sigma(\cdot)$ is the sigmoid function. Assume that our goal is to reprogram this generator given a target random variable

$$X_{\text{target}} = \sigma(10(X' - 0.5)) - (1 - \sigma(10(X' - 0.5))),$$

where $X' \sim U[0,1]$. See Fig. 2b for its density function, which has two modes at $X = \pm 1$. Consider $Z_{\text{seed}} \sim U[0,1]$ and $h_\theta(Z) = Z + b$, where b is a learnable bias. When $b = 0$, $g(h_\theta(Z_{\text{seed}})) \simeq -(1 - \sigma(10(Z - 0.8)))$. That is, the output of the reprogrammed does not properly capture the mode at $X = 1$. However, when $b = b^\star = 0.3$, we have $g(h_\theta(Z_{\text{seed}})) \overset{d}{=} X_{\text{target}}$, i.e., it perfectly recovers the target distribution.

Shown in Fig. 2 are the results of our GAN reprogramming algorithm applied to this setting. We can see that $b \to b^\star$ from Fig. 2a, and the distribution of the



(a) Evolution of b (b) Evolution of output distribution

Fig. 2: GAN reprogramming results for the toy example.

reprogrammed GAN's outputs converges to that of the target distribution as reprogramming algorithm proceeds from Fig. 2b.

4 Experiments and Analysis

4.1 Setting

For all of our experiments, we use Adam with $(\alpha, \beta_1, \beta_2) = (10^{-4}, 0, 0.9)$ and set $n_{\text{critic}} = 10$, $m = 32$, and $\lambda = 1$. The algorithm is run for 100 to 500 epochs.

We consider the following image datasets: MNIST [15], Fashion-MNIST (or FMNIST) [29], SVHN [19], CIFAR10 [14], CelebA [17], Cartoon Set (a collection of 2D cartoon avatar images) [24], and the Mel-Frequency Cepstrum (MFC) of drum sound clips [3]. When we reprogram a GAN between color images and grayscale images, we transform the grayscale images into RGB ones. For each labeled dataset D, we denote by D_i the subset of D containing of images whose label is i. Thus, we have MNIST_i, FMNIST_i, SVHN_i, and CIFAR10_i for $i \in \{0, 1, \ldots, 9\}$.

We denote the generator g trained with dataset D by g_D and call it 'D-GAN'. Each D-GAN is obtained via the standard WGAN-GP algorithm. We use standard neural network architectures for generator and discriminator. For the generator, we pass a 62-dimensional latent code ($d_i = 62$) through two fully connected layers followed by two transposed convolution layers: 1) fc1024, 2) fc$8WH$, 3) TrConv-4x4 (64 feature maps) 4) TrConv-4x4 (1 feature map), where (W, H) denotes the image dimensions. For the discriminator, we pass an image of size W by H through two convolution layers followed by two fully connected layers: 1) Conv-4x4 (64 feature maps), 2) Conv-4x4 (128 feature maps), 3) fc1024, 4) fc1. For CelebA and Cartoon Set, we use deeper architectures with four TrConv's and Conv's each. In order to generate a latent code from d-dimensional random seed, we use a six-layer neural network: 1) fc500, 2) fc500, 3) fc500, 4) fc500, 5) fc500, 6) fc62.

Case 1: GAN conditioning via reprogramming The first scenario we consider is GAN conditioning via reprogramming. That is, we reprogram a D-GAN as a D_i-GAN for $D \in \{\text{MNIST}, \text{FMNIST}, \text{SVHN}\}$ and $i \in \{0, 1, \ldots, 9\}$. For instance, when $D = \text{MNIST}$ and $i = 0$, the goal is to convert an MNIST-GAN into an MNIST_0-GAN, which only produces 0 images.

Case 2: GAN reprogramming across different datasets We also apply our reprogramming algorithm across different datasets: SVHN to MNIST$_i$, FMNIST to MNIST$_i$, CIFAR10 to MNIST$_i$, CIFAR10 to FMNIST$_i$, CIFAR10 to CelebA, CelebA to Cartoon Set, and CIFAR10 to Drum Sound Clips.

4.2 Qualitative Results

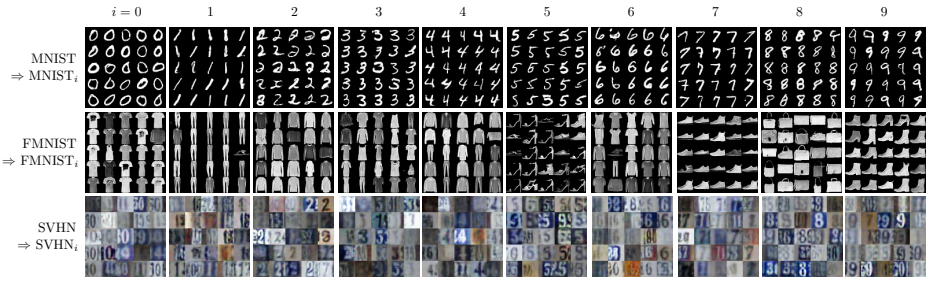


Fig. 3: GAN conditioning via reprogramming. Row corresponds to training data, and column corresponds to the target label.

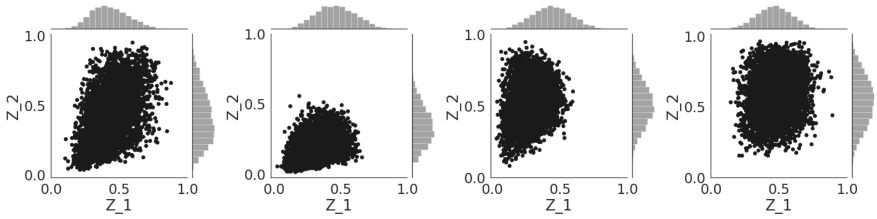


Fig. 4: The latent code distribution of reprogrammed GANs. We visualize the empirical distribution of the first two dimensions of latent code (Z_1 and Z_2). Column i is for FMNIST$_i$. Note that the original latent code is uniformly distributed in $[0, 1] \times [0, 1]$.

Shown in Fig. 3 are the experimental results for case 1. Each row represents source/target dataset D and each column represents different label i for $i \in \{0, 1, \ldots, 3\}$. Note that our algorithm successfully reprogram unconditional GANs as conditional GANs. To further confirm that each reprogrammed GAN successfully learned its corresponding input noise distribution, we sample 1000

random input noise according to F_{target} and visualize the distribution of the first two entries in Fig. 4. Column i is for F_{target} learned with FMNIST-i. Observe that F_{target} is not a uniform distribution while F_{source} is.

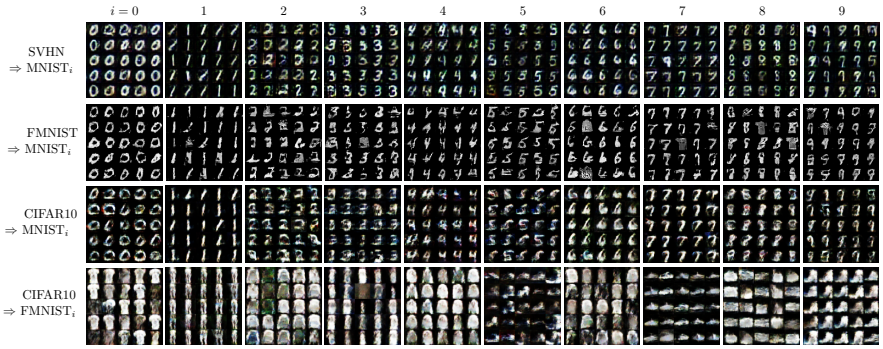


Fig. 5: GAN reprogramming across different datasets. Row corresponds to source/target data, and column corresponds to the label.

Shown in Fig. 5 are the experimental results for case 2. The reprogrammed GANs generate reasonable images but also show some inherent limitations. First, the texture remains the same as the original GAN. Also, if the given GAN learned a limited set of structures, the reprogrammed GAN attempts to generate target images by distorting the learned structures. For instance, consider the second row (FMNIST to MNIST) of Fig. 5. Since the FMNIST-GAN is able to draw fashion items only, the reprogrammed GAN draws digit images maintaining fashion item structures. For instance, it generates digit-2 images by distorting shoes.

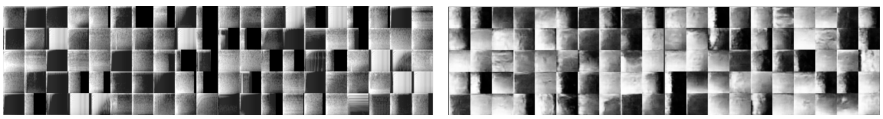


Fig. 6: MFC of real and synthetic audio clips.

We also apply reprogramming algorithm to a CIFAR10-GAN so that it can generate drum sound clips. The Mel-Frequency Cepstrum of the real drum sound clips (left) and those of the synthetic ones generated by a reprogrammed GAN (right) are shown in Fig. 6. Even though the MFC images look very differently from typical CIFAR10 images, our reprogramming algorithm successfully generates realistic MFC images. As an additional experiment, we reprogram a

Fig. 7: CIFAR10 to CelebA (left) and a CelebA to Cartoon (right).

CIFAR10-GAN as a CelebA-GAN and a CelebA-GAN to a Cartoon-GAN. From the left panel of Fig. 7, one can see that a reprogrammed GAN is able to generate human-face-like images even though it is originally trained with CIFAR10 images. On the right hand side, the reprogrammed GAN generates cartoon-like face images with white background, resembling the Cartoon dataset.

Comparison with an existing method We also evaluate the latent code sampling algorithm, called PPGN [20], to generate MNIST_0 images with a pre-trained FMNIST-GAN. Shown in Fig. 8 are the sample outputs of PPGN. Compared to the images generated by our reprogramming algorithm, the examples generated by PPGN barely resemble the digit 0. The poor performance of PPGN can be attributed to the fact that PPGN is designed specifically for GANs trained under a particular method called Joint PPGN. Furthermore, the sampling procedure of PPGN involves an iterative procedure, requiring many rounds of forward/backward-passes to generate a sample, while our sampling operation requires a single forward-pass.

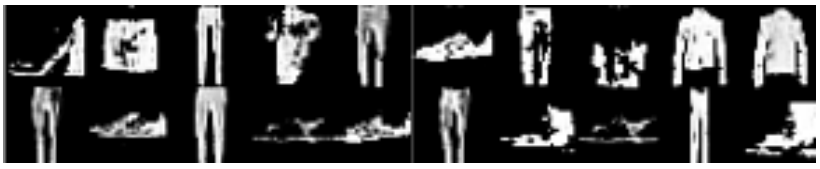
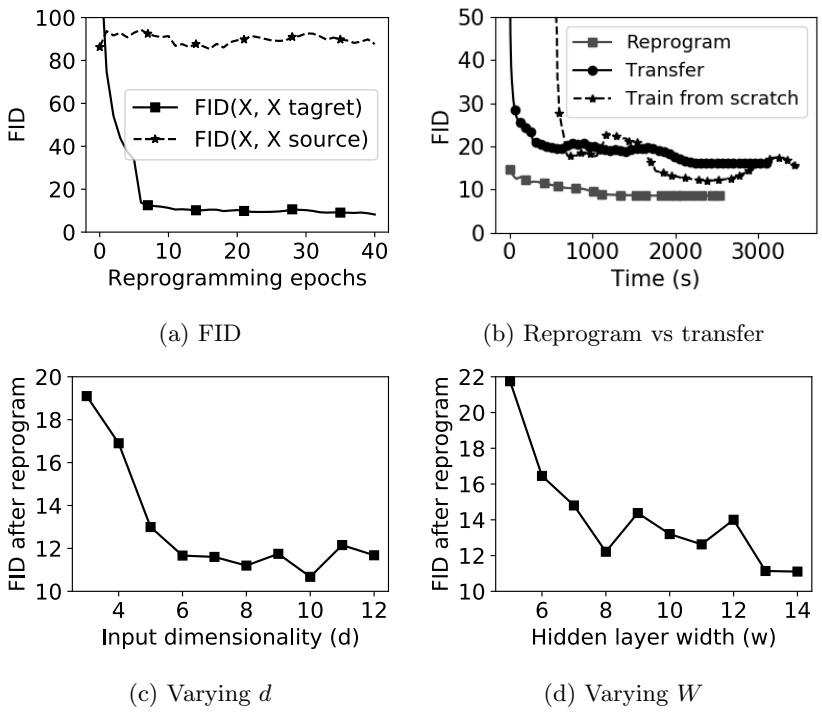


Fig. 8: MNIST_0 generated from FMNIST-GAN + PPGN [20].



(a) FID

(b) Reprogram vs transfer

(c) Varying d

(d) Varying W

Fig. 9: Practical analysis of GAN reprogramming. X_{source} is MNIST and X_{target} is MNIST_0. (a) The FID between $g(h_\theta(Z_{\text{seed}}))$ and X_{target} (blue) and the FID between $g(h_\theta(Z_{\text{seed}}))$ and X_{source} (orange). (b) Reprogramming vs transfer. (c) Reprogrammability as a function of d. (d) Reprogrammability as a function of the number of hidden neurons in each layer of h_θ.

4.3 Fréchet Inception Distance (FID)

In Fig. 10a, we plot the Fréchet Inception Distance (FID) [11] between the output generated by a reprogrammed GAN ($g(h_\theta(Z_{\text{seed}}))$) and the target distribution (X_{target}). One can observe that the FID decreases as the reprogramming algorithm proceeds, showing the validity of GAN reprogramming. We also observe a much larger value for the FID between the output generated by a reprogrammed GAN ($g(h_\theta(Z_{\text{seed}}))$) and the original distribution (X_{source}).

4.4 Reprogramming vs Transfer learning

We compare the convergence performance of GAN reprogramming algorithm and that of transfer learning. A standard approach to transfer knowledge between generative models is based on fine tuning [28]. That is, (i) one first trains a GAN (both G and D) with the original dataset; (ii) then fine-tunes G and D

with the target dataset. Shown in Fig. 10b are the comparison results. For the transfer learning curve, we report the performance of 'fine-tuning all layers' as we observed 'fine-tuning last layers' only performs worse. We first note that both 'transfer learning' and 'train from scratch' suffer from the notorious oscillation phenomena [16]. More importantly, the convergence of transfer learning is much slower than reprogramming.

4.5 Capacity of Code Generator

To evaluate the reprogrammability as a function of the capacity of code generator $h_\theta(\cdot)$, we vary d, the dimensionality of Z_{seed}. We also repeat the same experiment while varying the number of neurons in each layer, i.e., the capacity of $h_\theta(\cdot)$. Here, we use a three-layer FC network, whose hidden layers have w neurons each, where $w \in \{5, 6, \ldots, 14\}$. For each point, we then measure the FID five times and report the average. The experimental results are summarized in Fig. 9c and Fig. 9d, respectively. In both figures, we can observe that reprogramming is infeasible when the capacity of code generator is low and that the reprogrammability saturates as the capacity increases.

5 Theoretical Analysis

Clearly, reprogrammed generators cannot generate samples beyond the original range. Thus, the performance of GAN reprogramming is characterized by the range of the original generator. The following theorem formally states this phenomenon under a simplified linear setting [7].

Theorem 1. *Let X_{target} be a zero-mean non-degenerate d_o-dimensional Gaussian random vector. Assume a linear generator $g(Z) = AZ$, where $A \in \mathbb{R}^{d_o \times d_i}$, and denote the projection of X_{target} on the subspace spanned by matrix A by X^A_{target}. Then, the minimum second-order Wasserstein distance that can be achieved via* any *GAN reprogramming method is lower bounded by $\mathbb{E}[\|X_{target} - X^A_{target}\|^2]$. Moreover, if Z_{seed} is a zero-mean non-degenerate d_i-dimensional Gaussian random vector, the optimal reprogramming is achievable with a linear code generator $h(Z_{seed}) = BZ_{seed}$ for some $B \in \mathbb{R}^{d_i \times d_i}$.*

Proof. **(lower bound)** The second-order Wasserstein distance is $W_2^2(P_r, P_g) := \inf_\gamma \mathbb{E}_{(x,y)\sim\gamma}[\|x-y\|^2]$, where $\gamma \in \Pi(P_r, P_g)$, and $\Pi(P_r, P_g)$ is the set of all joint distributions whose marginals are respectively P_r and P_g. Under the second-order Wasserstein distance as an underlying metric, the GAN reprogramming problem can be written as $\min_{F_{\text{target}}} W_2^2(g(Z_{\text{target}}), X_{\text{target}})$, where $Z_{\text{target}} \sim F_{\text{target}}$ is a d_i-dimensional random vector. Assuming our code generator structure ($Z_{\text{target}} = h(Z_{\text{seed}})$) and the linear generator assumed in the theorem, we have $\inf_h W_2^2(Ah(Z_{\text{seed}}), X_{\text{target}})$, where X_{target} is a zero-mean non-degenerate d_o-dimensional Gaussian random vector, and $A \in \mathbb{R}^{d_o \times d_i}$ is a given generator matrix. By the definition of the second-order Wasserstein distance, it reduces to

$$\inf_h \inf_\gamma \mathbb{E}_{(Z_{\text{seed}}, X_{\text{target}})\sim\gamma} \|Ah(Z_{\text{seed}}) - X_{\text{target}}\|^2, \tag{6}$$

where $\gamma \sim \Pi(P_{Z_{\text{seed}}}, P_{X_{\text{target}}})$.

As $Ah(Z_{\text{seed}}) - X^A_{\text{target}} \perp X^A_{\text{target}} - X_{\text{target}}$, we have

$$\|Ah(Z_{\text{seed}}) - X_{\text{target}}\|^2 = \|Ah(Z_{\text{seed}}) - X^A_{\text{target}}\|^2 + \|X^A_{\text{target}} - X_{\text{target}}\|^2. \quad (7)$$

Since the second term in the RHS is independent of h and γ, the second-order Wasserstein distance is always lower bounded by $\mathbb{E}\|X^A_{\text{target}} - X_{\text{target}}\|^2$. This proves the first part of the theorem.

(achievability) We now show that the lower bound can be achieved with a linear code generator. Consider a linear code generator $h(Z_{\text{seed}}) = BZ_{\text{seed}}$ for some matrix $B \in \mathbb{R}^{d_i \times d_i}$ and a zero-mean non-degenerate d_i-dimensional Gaussian random vector Z_{seed}, i.e., $Z_{\text{seed}} \sim \mathcal{N}(0, K_s)$ for some positive definite matrix K_s.

Let the economic singular value decomposition (SVD) of A by $A = U_A \Sigma_A V_A^\intercal$. Then, the projection on the subspace spanned by A is a linear operator $P := U_A U_A^\intercal$. Since X_{target} is a zero-mean non-degenerate Gaussian random vector, $X_{\text{target}} \sim \mathcal{N}(0, K_t)$ for some positive definite matrix K_t. Thus, $X^A_{\text{target}} \sim \mathcal{N}(0, PK_tP^\intercal)$. Similarly, $Ah(Z_{\text{seed}}) = ABZ_{\text{seed}} \sim \mathcal{N}(0, ABK_sB^\intercal A^\intercal)$. Then,

$$\begin{aligned}
& PK_tP^\intercal = ABK_sB^\intercal A^\intercal \\
\Rightarrow\ & U_A U_A^\intercal K_t U_A U_A^\intercal = U_A \Sigma_A V_A^\intercal B K_s B^\intercal V_A \Sigma_A U_A^\intercal \\
\Rightarrow\ & U_A^\intercal K_t U_A = \Sigma_A V_A^\intercal B K_s B^\intercal V_A \Sigma_A \\
\Rightarrow\ & U_A^\intercal \sqrt{K_t} = \Sigma_A V_A^\intercal B \sqrt{K_s} \quad (8) \\
\Rightarrow\ & V_A \Sigma_A^{-1} U_A^\intercal \sqrt{K_t} \sqrt{K_s}^{-1} = B, \quad (9)
\end{aligned}$$

where (8) holds since K_t and K_s are positive definite, and (9) holds since $\sqrt{K_t}$ and $\sqrt{K_s}$ are also positive definite. Thus, by choosing

$$B = B^\star := V_A \Sigma_A^{-1} U_A^\intercal \sqrt{K_t} \sqrt{K_s}^{-1},$$

the marginal distribution of X^A_{target} and that of $AB^\star Z_{\text{seed}}$ are identical. Therefore, one can choose B and γ such that $X^A_{\text{target}} = AB^\star Z_{\text{seed}}$ w.p. 1. Together with (7), it implies

$$\mathbb{E}\|AB^\star Z_{\text{seed}} - X_{\text{target}}\|^2 = \mathbb{E}\|X^A_{\text{target}} - X_{\text{target}}\|^2, \quad (10)$$

proving the theorem. □

Experiments To further demonstrate that reprogrammed generators cannot generate samples beyond the range of the original generator, we conduct the following experiments with synthetic datasets. We first train a generator using images of three randomly positioned circles (Set 0) and then reprogram it respectively with images of one randomly positioned circle (Set 1), images of three co-located circles (Set 2), and images of two randomly positioned squares (Set 3). See Fig. 10a for sample images. In Fig. 10b, we plot normalized FIDs, i.e.,

FID divided by the initial FID measured before we apply our reprogramming algorithm. We can observe that reprogramming succeeds only for the first two datasets (Set 1 and Set 2), which consist of circles, and fails for the last dataset (Set 3), which consist of different shapes. This is expected as the square shapes are beyond the range of the original generator that can only draw curvy edges.

(a) Sample synthetic images (b) Normalized FID

Fig. 10: Reprogramming a generator with restricted range.

6 Conclusion

In this work, we studied the problem of GAN reprogramming. We showed that our algorithm can reprogram a GAN as another GAN that generates different datasets and studied the applicability, feasibility, and limitation. We concluded the paper with discussing related open questions.

Unexpected Behaviors of a GAN and Adversarial Reprogramming of GANs The fact that GANs can be reprogrammed implies that a GAN trained with a certain dataset is *not* guaranteed to generate valid samples only. Even though this observation is already made in a recent study [13], our work implies even a stronger message: There exist infinitely many latent codes which result in inconsistent samples with the original dataset. We believe that these inconsistent samples are generated with very low probability if latent codes are drawn at random according to F_{original}. However, for some critical applications, such an unexpected behavior might not be allowed. Thus, it still implies that the usage of GAN for highly critical applications should be avoided until we have a formal guarantee that GANs generate valid samples only. Moreover, it is an interesting open question whether or not one can design a GAN that provably generates valid samples for *every* latent code.

An Alternative for GAN Transfer Learning In Sec. 4.4, we observed that GAN reprogramming outperforms the standard fine-tuning learning algorithm in a transfer learning setting. This implies that reprogramming-based approach might be a useful alternative to GAN transfer learning, but this requires a more in-depth study.

References

1. Arjovsky, M., Chintala, S., Bottou, L.: Wasserstein generative adversarial networks. In: International Conference on Machine Learning (ICML) (2017)
2. Caruana, R.: Multitask learning. Machine learning **28**(1), 41–75 (1997)
3. Donahue, C., McAuley, J., Puckette, M.: Synthesizing audio with generative adversarial networks. arXiv preprint arXiv:1802.04208 (2018)
4. Duong, L., Cohn, T., Bird, S., Cook, P.: Low resource dependency parsing: Cross-lingual parameter sharing in a neural network parser. In: Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (2015)
5. Elsayed, G.F., Goodfellow, I., Sohl-Dickstein, J.: Adversarial reprogramming of neural networks. In: International Conference on Learning Representations (ICLR) (2019)
6. Engel, J., Hoffman, M., Roberts, A.: Latent constraints: Learning to generate conditionally from unconditional generative models. In: International Conference on Learning Representations (ICLR) (2018)
7. Feizi, S., Farnia, F., Ginart, T., Tse, D.: Understanding GANs: the LQG setting. arXiv preprint arXiv:1710.10793 (2017)
8. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial nets. In: Advances in Neural Information Processing Systems (NeurIPS) (2014)
9. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. In: International Conference on Learning Representations (ICLR) (2014)
10. Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., Courville, A.C.: Improved Training of Wasserstein GANs. In: Advances in Neural Information Processing Systems (NeurIPS) (2017)
11. Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., Hochreiter, S.: Gans trained by a two time-scale update rule converge to a local nash equilibrium. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems (NeurIPS) (2017)
12. Huang, J.T., Li, J., Yu, D., Deng, L., Gong, Y.: Cross-language knowledge transfer using multilingual deep neural network with shared hidden layers. In: IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (2013)
13. Kos, J., Fischer, I., Song, D.: Adversarial examples for generative models. In: 2018 IEEE Security and Privacy Workshops (SPW) (2018)
14. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images. MS thesis, Department of Computer Science, University of Toronto (2009)
15. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)
16. Li, J., Madry, A., Peebles, J., Schmidt, L.: On the limitations of first-order approximation in GAN dynamics. In: International Conference on Machine Learning (ICML) (2018)
17. Liu, Z., Luo, P., Wang, X., Tang, X.: Deep learning face attributes in the wild. In: International Conference on Computer Vision (ICCV) (2015)
18. Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks. In: International Conference on Learning Representations (ICLR) (2018)

19. Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., Ng, A.Y.: Reading digits in natural images with unsupervised feature learning. In: NeurIPS workshop on deep learning and unsupervised feature learning (2011)
20. Nguyen, A., Clune, J., Bengio, Y., Dosovitskiy, A., Yosinski, J.: Plug & play generative networks: Conditional iterative generation of images in latent space. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE (2017)
21. Oquab, M., Bottou, L., Laptev, I., Sivic, J.: Learning and transferring mid-level image representations using convolutional neural networks. In: IEEE conference on Computer Vision and Pattern Recognition (CVPR) (2014)
22. Pan, S.J., Yang, Q., et al.: A survey on transfer learning. IEEE Transactions on knowledge and data engineering **22**(10), 1345–1359 (2010)
23. Pasquini, D., Mingione, M., Bernaschi, M.: Out-domain examples for generative models. arXiv preprint arXiv:1903.02926 (2019)
24. Royer, A., Bousmalis, K., Gouws, S., Bertsch, F., Moressi, I., Cole, F., Murphy, K.: Xgan: Unsupervised image-to-image translation for many-to-many mappings. arXiv preprint arXiv:1711.05139 (2017)
25. Ruder, S.: An overview of multi-task learning in deep neural networks. arXiv preprint arXiv:1706.05098 (2017)
26. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199 (2013)
27. Villani, C.: Optimal transport: old and new, vol. 338. Springer Science & Business Media (2008)
28. Wang, Y., Wu, C., Herranz, L., van de Weijer, J., Gonzalez-Garcia, A., Raducanu, B.: Transferring gans: generating images from limited data. arXiv preprint arXiv:1805.01677 (2018)
29. Xiao, H., Rasul, K., Vollgraf, R.: Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. arXiv preprint arXiv:1708.07747 (2017)
30. Yang, Y., Hospedales, T.M.: Trace norm regularised deep multi-task learning. arXiv preprint arXiv:1606.04038 (2016)
31. Zhou, B., Lapedriza, A., Xiao, J., Torralba, A., Oliva, A.: Learning deep features for scene recognition using places database. In: Advances in Neural Information Processing Systems (NeurIPS) (2014)