Deep Generative Models and Their Inversions For Bidirectional Transformation Between Data and Latent Distributions

Jeongik Cho

A Thesis
In the Department of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy

at Concordia University
Montréal, Québec, Canada

April 2025

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Jeongik Cho

Entitled: Deep Generative Models and Their Inversions For Bidirectional Transformation Between Data and Latent Distributions

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Abdessamad Ben Hamza

_____ External Examiner
Dr. Mirek Pawlak

_____ Examiner
Dr. Charalambos Poullis

_____ Examiner
Dr. Eugene Belilovsky

_____ Examiner
Dr. Nizar Bouguila

_____ Thesis Supervisor
Dr. Adam Krzyżak

Approved by

_____
Dr. Sabine Bergler, Graduate Program Director

May 2025

_____
Dr. Mourad Debbabi, Dean of Faculty

Abstract

Deep Generative Models and Their Inversions For Bidirectional Transformation Between Data and Latent Distributions

Jeongik Cho, Ph.D.

Concordia University, 2025

Generative models aim to transform a simple latent distribution into a complex data distribution, enabling the synthesis of high-dimensional, realistic data. In contrast, generative model inversion addresses the reverse process, mapping a complex data distribution back into a simple latent representation. In this thesis, we introduce several novel contributions to architecture-agnostic algorithms of generative models and their inversions, as well as applications utilizing these methods.

First, we show that using multiple adversarial losses improves the performance and requires fewer hyperparameters than using an auxiliary classifier. Then, we introduce a novel encoder-based GAN inversion method for better convergence than a simple mean squared error by dynamically adjusting the scale of each element of the latent random variable. Furthermore, we propose an out-of-distribution detection method that leverages the log probability of the latent vector predicted by the encoder-based GAN inversion framework. Next, we introduce a novel method that combines the perceptual VAE and the GAN inversion technique from the second contribution to improve the GAN inversion performance. Finally, we introduce a novel GAN that allows the model to perform self-supervised class-conditional data generation and clustering using a classifier gradient penalty loss.

Acknowledgments

I would like to express my sincere gratitude to several individuals who have supported me throughout the completion of this thesis. Firstly, I extend my deepest appreciation to my supervisor, Dr. Krzyżak, for continuous guidance and encouragement. His expertise and mentorship have been invaluable throughout the writing process, providing me with invaluable advice and inspiration.

In addition, I am deeply grateful for the unwavering support and love of my family. My parents have always believed in me and have provided unwavering support. In addition, the constant encouragement of my younger sister has fueled my efforts and passion. I am immensely grateful for their endless love and support.

I extend my heartfelt thanks to Dr. Yoon and all those who have contributed to the completion of this thesis.

Contents

List of Figures

List of Tables

Table 1: Summary of Abbreviations

Abbreviation	**Full Term**
GANs	Generative Adversarial Networks
VAEs	Variational Autoencoders
CGAN	Conditional GAN
ACGAN	Auxiliary Classifier GAN
CAGAN	Conditional Activation GAN
i.i.d.	Independent and identically distributed
DLSGAN	Dynamic Latent Scale GAN
PVDGAN	Perceptual VAE DLSGAN
OOD	Out-of-distribution
ID	In-distribution
CGPGAN	Classifier Gradient Penalty GAN
VQ	Vector Quantization
MSE	Mean Squared Error
NLL	Negative Log-Likelihood
Rec	Reconstruction

Table 2: Summary of Notation

Symbol	Description
X	d_x-dimensional data random variable
x	d_x-dimensional data point
c_r	d_c-dimensional categorical vector corresponding to x
Z	d_z-dimensional continuous latent random variable
z	d_z-dimensional continuous latent vector
z'	d_z-dimensional continuous latent vector predicted by encoder
C	d_c-dimensional categorical latent random variable
c_f	d_c-dimensional categorical latent vector
G	Generator
D	Discriminator
Q	Classifier
E_l	Encoder predicting the continuous latent vector
E_{lv}	Encoder predicting the log variance vector
D^*	Discriminator integrated with E_l
E_i	Trainable i-th sub-encoder, i.e., $E_i(x) = [E(x)]_i$
A_g	Adversarial loss function for the generator
A_d	Adversarial loss function for the discriminator
L	Loss
λ	Loss weight
$\mathbb{E}_{x\sim p(x)}[\cdot]$	Expectation over distribution $p(x)$
$\|\cdot\|_2$	ℓ_2 norm (Euclidean distance)
$avg(x)$	Average over dimensions of a vector $x \in \mathbb{R}^{d_x}$: $avg(x) = \frac{1}{d_x}\sum_{j=1}^{d_x} x_j$
∇_x	Gradient with respect to x
$\cdot$	Dot product between vectors
$\circ$	Hadamard (element-wise) product
$argmax\ onehot(\cdot)$	One-hot encoding of the input vector's index with the largest value
$sample(\mathcal{X}, n)$	Function that returns n samples from random variable $\mathcal{X}$

1 Introduction

1.1 Motivation

Recently, deep generative models such as Diffusion Models [70], GANs (Generative Adversarial Networks) [1], and VAEs (Variational Autoencoders) [2] have demonstrated remarkable performance in data generation. These generative models are trained to transform simple latent distributions into complex data distributions. A simple latent distribution refers to an easy-to-sample distribution like a standard Gaussian, while a complex data distribution refers to the intricate, real-world distribution of observed data. In general, latent distribution is simple, easy to sample, and continuous. In particular, when the latent distribution is a continuous distribution, the generated data distribution is also a continuous distribution. This means that even when the data distribution is an empirical distribution, it can be approximated continuously by the generative model. These features of latent distribution allow generative models to be used not only for data generation, but also for multiple tasks such as data understanding, data generation, data augmentation, and data post-processing.

Inverse transform sampling and Gaussian Mixture Models (GMMs) are simple examples of transformations between a simple latent distribution and a complex data distribution. Inverse transform sampling uses the inverse cumulative distribution function (CDF) to convert uniform random variables into samples from arbitrary target distributions. It is widely used in univariate settings where the inverse CDF is tractable. Gaussian Mixture Models approximate complex distributions by linearly combining multiple simple Gaussian components. Since each component follows a Gaussian distribution, GMMs can represent a complex distribution as a weighted sum of Gaussian components.

Another topic in generative models is generative model inversion. Generative model inversion addresses the reverse process, mapping a complex data distribution back into a simple latent representation. These features of generative model inversion allow generative model inversion to be used not only for tasks where generative models are used but also for

various tasks such as domain transfer, data preprocessing, data manipulation, and anomaly detection.

Thus, this bidirectional transformation with generative model and generative model inversion allows complex data distributions to be replaced by simple latent distributions. This feature allows bidirectional transformation with generative models and generative model inversion to be used in a very wide range of tasks, including data processing and end-to-end models.

1.2 Summary

In this thesis, we introduce several contributions about architecture-agnostic algorithms of generative models and their inversions, as well as applications utilizing these methods. The main objective of this work is to investigate and improve the invertibility of generative models for effective representation (feature) learning.

Representation learning, including generative model inversion, aims to extract meaningful and compact features from high-dimensional data. It can be used for various downstream tasks, such as classification, regression, clustering, or anomaly detection. For example, latent vectors obtained through inversion can be clustered to discover inherent data structures, or used as input to train lightweight classifiers for few-shot learning. Additionally, the latent space enables interpretable manipulation of attributes, transfer learning across domains, and multimodal alignment when extended to other data types.

We propose novel inversion techniques that allow accurate recovery of latent vectors from generated data, enabling better interpretability, reconstruction, and downstream tasks such as anomaly detection and attribute editing. In most contributions, we discuss generative adversarial networks (GAN).

• **In the first contribution**, we introduce two advanced techniques for conditional GAN (CGAN) [38]. Auxiliary classifier GAN (ACGAN) [3] is a class-conditional GAN that uses an auxiliary classifier for class-conditional data generation. The classifier of ACGAN is

trained to classify the label of the input data, and the generator is trained so that the generated class-conditional data is correctly classified by the classifier. We analyze the problem ACGAN and introduce conditional activation GAN (CAGAN) [4] that can replace ACGAN. ACGAN's generator is trained to minimize classification loss for class-conditional data generation, which results in a trade-off between classification loss and adversarial loss and decreases the model's generative performance. On the other hand, the proposed CAGAN is a composite of multiple GANs, where each GAN is trained to generate each class. Since each GAN shares all hidden layers, it can be considered a single model. Compared to ACGAN, CAGAN does not require an additional hyperparameter to adjust the loss ratio between adversarial loss and classification loss and has better generative performance.

When training conditional GAN, real conditional distribution and fake conditional distribution can be different. If the discriminator has batch-wise operations, it can use the conditional distribution of the input batch for real/fake discrimination, which lowers the generative performance of the model. To prevent the discriminator from using conditional distribution of input batch, we propose mixed batch training. Mixed batch training is configuring each batch with always equal proportions of real data and fake data. If the ratio of real data and fake data in every batch is the same, the conditional distribution of every batch is also the same. For stable training, mixed batch training gradually changes the ratio of real data and fake data in each batch.

• **In the second contribution**, we introduce a novel GAN inversion method and a continuous attribute edit method that utilizes it. When each dimension of a latent random variable is independent and identically distributed (i.i.d.), the encoder trained with mean squared error loss to invert the generator does not converge. This is because the entropy of the latent random variable is too high, so the generator loses information of the latent random variable. We introduce a dynamic latent scale GAN (DLSGAN) that allows the encoder to invert the generator by dynamically adjusting the element-wise scale of a latent random variable so that the generator does not lose information. The scale of the latent

random variable is approximated by tracing the element-wise variance of the predicted latent random variable of the encoder from previous training steps. The encoder can be integrated with the discriminator, and the loss for the encoder is added to the generator loss for fast training.

InterFaceGAN [41] showed that the attributes of generated data are linearly separable in the latent space of GAN, utilizing it to perform continuous attribute editing of generated data. InterFaceGAN trains a linear classifier that predicts the attributes of generated data from latent vectors. By gradually changing the latent vector to change the output of the linear classifier, the attributes of the generated data can be continuously changed. Based on this idea, we introduce a method to continuously edit attributes of input data using DLSGAN. The proposed method trains a class-conditional DLSGAN to fit a fixed linear classifier. By gradually changing the predicted latent vector to change the output of the fixed linear classifiers, the attributes of the input data can be continuously changed.

• **In the third contribution**, we introduce an advanced self-supervised (unsupervised) out-of-distribution (OOD) detection method using DLSGAN. The entropy of DLSGAN's latent random variable decreases gradually as training progresses, converging to an appropriate value. When the DLSGAN converges, in-distribution (ID) data are densely mapped to latent vectors that are likely sampled from the latent distribution. Therefore, the log probability of the predicted latent vector represents the relative log probability of the data, and it can be used for OOD detection. We propose AnoDLSGAN which uses the log probability of the predicted latent vector of DLSGAN for OOD detection. Each dimension of DLSGAN's encoder output is independent of each other and follows simple latent distributions with different variances. Therefore, it is easy to calculate the log probability of the predicted latent vector. AnoDLSGAN does not require metric function, pre-trained model, ID data, or additional hyperparameters for prediction.

• **In the fourth contribution**, we introduce perceptual VAE DLSGAN (PVDGAN), a method of adding perceptual variational autoencoder (VAE) loss to DLSGAN efficiently.

Generally, GAN is suffering from low diversity of the generated data. Some works [68, 69, 34, 33] tried to combine VAE or autoencoder loss with GAN to improve the generative performance, especially data diversity, of GAN. We propose PVDGAN, which efficiently integrates perceptual VAE loss into DLSGAN. When each dimension of DLSGAN's latent random variable is i.i.d. normal distribution, each dimension of the predicted latent random variable of real data is independent of each other and follows normal distribution. Therefore, the sum of element-wise scaled normal noise and predicted latent random variable becomes a random variable with each dimension is i.i.d. normal distribution again. Since this random variable is paired with real data and follows the latent random variable, it can be used for both VAE and GAN training. Furthermore, by considering the intermediate layer output of the discriminator as the feature encoder output, the VAE can be trained with perceptual reconstruction loss, instead of simple reconstruction loss. The forward propagation & backpropagation for minimizing this perceptual reconstruction loss can be integrated with those of GAN training. Therefore, PVDGAN does not require additional computations compared to typical GAN or DLSGAN.

• **In the fifth contribution**, we introduce classifier gradient penalty GAN (CGPGAN) for self-supervised class-conditional data generation and clustering. Class-conditional GAN generates class-conditional data from continuous latent distribution and categorical distribution. Typically, a class-conditional GAN can be trained only when the label of the target data is given. We propose CGPGAN that allows the model to perform self-supervised class-conditional data generation and clustering without knowing labels, optimal prior categorical probability, or metric function. CGPGAN uses a discriminator, a classifier, and a generator. The classifier is trained with cross-entropy loss to predict the conditional vector of the fake data. Also, the conditional vector of real data predicted by the classifier is used to train the class-conditional GAN. When training class-conditional GAN with this classifier, the decision boundary of the classifier falls to the local optima where the density of the data is minimized. CGPGAN adds a classifier gradient penalty loss to the classifier loss to prevent the classifier's

decision boundary from falling into a narrow range of local optima. It regulates the gradient of the classifier's output to prevent the gradient near the decision boundary from becoming too large. As the classifier gradient penalty loss weight increases, the decision boundary falls into a wider range of local optima. Therefore, the sensitivity of each class can be adjusted by the weight of the gradient penalty loss. Additionally, CGPGAN updates the prior categorical probability with the categorical probability of real data predicted by the classifier. As training progresses, the entropy of the prior categorical probability decreases and converges according to the classifier gradient penalty loss weight. Furthermore, we propose codebook architecture for CGPGAN to strengthen the discrete representation and make it easier to interpret the discrete representation. Instead of directly inputting a one-hot categorical latent vector into the generator, the codebook architecture inputs the trainable page vector of the corresponding index of the categorical latent vector. Through memorization of codebook architecture, CGPGAN can enrich the discrete representation and generate high-quality data.

1.3 Publications and Contributions of the Co-authors

"Conditional Activation GAN: Improved Auxiliary Classifier GAN," In IEEE Access, vol. 8, pp. 216729-216740, 2020. [4]

Mainly discussed in section 3.1.

Jeongik Cho	Conception and design of the study, contributions to theoretical analysis, experimental work, data analysis, writing the original draft, editing and proofing
Kyoungro Yoon	Research supervision, funding, editing and proofing

"Dynamic Latent Scale for GAN Inversion," in Proceedings of the 11th International Conference on Pattern Recognition Applications and Methods (ICPRAM), pp. 221-228, 2022. [51]

Mainly discussed in section 3.2.

Jeongik Cho	Conception and design of the study, contributions to theoretical analysis, experimental work, data analysis, writing the original draft, editing and proofing
Adam Krzyżak	Research supervision, funding, editing and proofing

"Self-supervised Out-of-distribution Detection with Dynamic Latent Scale GAN," in Structural, Syntactic, and Statistical Pattern Recognition (S+SSPR), pp. 113-121, 2022. [52]

Mainly discussed in section 3.3.

Jeongik Cho	Conception and design of the study, contributions to theoretical analysis, experimental work, data analysis, writing the original draft, editing and proofing
Adam Krzyżak	Research supervision, funding, editing and proofing

"Efficient integration of perceptual variational autoencoder into dynamic latent scale generative adversarial network," in Expert Systems, e13618m 2024. [53]

Mainly discussed in section 3.4.

Jeongik Cho	Conception and design of the study, contributions to theoretical analysis, experimental work, data analysis, writing the original draft, editing and proofing
Adam Krzyżak	Research supervision, funding, editing and proofing

"Training Self-supervised Class-conditional GAN with Classifier Gradient Penalty and Dynamic Prior," under review in Expert Systems Journal. [86, 87]

Mainly discussed in section 3.5.

Jeongik Cho	Conception and design of the study, contributions to theoretical analysis, experimental work, data analysis, writing the original draft, editing and proofing
Adam Krzyżak	Research supervision, funding, editing and proofing

"Analysis of the rate of convergence of two regression estimates defined by neural features which are easy to implement," in Electronic Journal of Statistics 18.1 (2024): 553-598. [85]

Alina Braun	Conception and design of the study, contributions to theoretical analysis, experimental work, data analysis, writing the original draft, editing and proofing
Michael Kohler	Research supervision, major contributions to theoretical analysis, editing and proofing
Jeongik Cho	Experiments and proofing
Adam Krzyżak	Research supervision, contributions to theoretical analysis, editing and proofing

"On the rate of convergence of an over-parametrized deep neural network regression estimate with ReLU activation function learned by gradient descent," under revision in Electronic Journal of Statistics.

Michael Kohler	Conception and design of the study, contributions to theoretical analysis, writing the original draft, editing and proofing
Jeongik Cho	Experiments and proofing
Adam Krzyżak	Research supervision, contributions to theoretical analysis, editing and proofing

1.4 Background

1.4.1 Deep Generative Model

Recently, generative models have shown impressive performance in deep learning models. The objective of a generative model is to transform an easy-to-sample distribution, such as a multi-dimensional Gaussian distribution, into a complex target data distribution. The distribution that serves as the input for the generative model is called a latent random variable. In general, the latent distribution is a distribution where each dimension is i.i.d. and follows a simple distribution, and the target distribution is a complex, high-dimensional empirical distribution. Some dimensions of the latent random variable should be continuous, which means a well-trained generative model can sample data continuously from the target data distribution. Among various generative models, variational autoencoder (VAE) [2], generative adversarial network (GAN) [1], and diffusion models [70] are demonstrating state-of-the-art generative performance [12].

A VAE consists of an encoder and a decoder. The encoder of VAE takes input data and predicts means and variances of a Gaussian distribution. Each dimension of this Gaussian distribution is independent of the others. Then, the decoder generates data from a latent vector sampled from the predicted Gaussian distribution. The encoder and the decoder are trained to minimize the distance between input data and generated data with reconstruction

	VAE [2]	GAN [1]	Diffusion [70]
Data Quality	Low	High	High
Data Diversity	High	Low	High
Sampling Speed	Fast	Fast	Slow
Latent Dimension	Low	Low	High

Table 3: Characteristics of deep generative models by type [12].

loss. Additionally, the encoder is trained with KL (Kullback-Leibler) loss so that the predicted Gaussian distribution follows the latent distribution. VeryDeepVAE [72] exhibits the state-of-the-art performance among VAEs by stacking multiple VAEs.

A GAN consists of a discriminator and a generator. In GANs, the discriminator is trained to distinguish between the samples generated by the generator and the real samples from the training data. The generator is trained to deceive the discriminator by generating data samples that are indistinguishable from real samples. Through this adversarial training between the generator and discriminator, the generator learns to transform the latent distribution into the data distribution. StyleGAN [19, 20] used architecture consists of a mapping network and a synthesis network for the generator, and it has shown state-of-the-art performance among GANs.

A diffusion model consists of an autoencoder. The autoencoder of the diffusion model is trained to remove Gaussian noise from noisy data. The strength of the noise added to the data depends on the time step and scheduling function. The diffusion model's data generation process starts with the noisiest data and gradually removes the noise. Therefore, the noise removal process is repeated iteratively by multiple inferences of the autoencoder. In addition, each denoising step involves adding high-dimensional Gaussian noise, which makes the latent dimension very high. Latent diffusion model [71] showed state-of-the-art performance among diffusion models by using a pre-trained autoencoder and performing an inverse diffusion process only in the latent space. To address the slow sampling speed of diffusion models, DDGAN [12], which combines GAN and diffusion model, has also shown state-of-the-art performance.

Table 3 displays the general performance of each generative model. In Table 3, data quality refers to how much each generated data resembles real data. The data diversity represents how diverse the generated data is. The generative performance of the model can be considered high when both generated data quality and diversity are high. In general, the diffusion models exhibit the best generative performance.

Sampling speed represents the computations required to generate each data. VAEs and GANs require only one forward propagation of the generative model for each data. On the other hand, diffusion models gradually replace noise with data, requiring multiple forward propagation. Generally, diffusion models require more than 100 forward propagation for image generation.

The latent dimension refers to the dimension of the latent random variable required to generate data. The high latent dimension makes it difficult to represent data meaningfully. Since VAEs and GANs perform dimensionality reduction, the latent dimension is typically lower than the data dimension. On the other hand, diffusion models do not perform dimensionality reduction, and require a latent dimension of the data dimension for each denoising step. When generating 256×256 resolution images, the latent dimension in VAEs or GANs is typically between 256 to 1024, while in diffusion models is $256 \times 256 \times 3 = 196,608$ (deterministic) or $256 \times 256 \times 3 \times 100 = 19,660,800$ (non-deterministic, reference algorithm).

1.4.2 Class-conditional GAN

Among variations of GANs, a conditional GAN [38] (CGAN) is a GAN that can generate conditional data distribution. CGAN's generator takes an unconditional latent vector and a conditional vector as input and generates conditional data corresponding to the conditional vector. In general, the training dataset for CGAN should consist of a target (real) data vector and the corresponding conditional vector, which can be continuous or discrete. Pix2Pix [14] is an example of a conditional GAN using a continuous conditional vector. Pix2Pix takes an image, which is a continuous conditional vector, and generates a corresponding conditional

image. For example, Pix2Pix can be trained to take a grayscale image as input and output a corresponding color image. The work by [15] also employed a conditional GAN to augment the data for scene graph prediction. In particular, the conditional GAN generates visual features corresponding to rare scene graphs by conditioning on both the graph structure and continuous visual features. The generated visual features improve the performance of the scene graph prediction model through data augmentation.

Class-conditional GAN is a conditional GAN where the conditional vector is a discrete categorical vector. Auxiliary Classifier GAN (ACGAN) [3] is an example of class-conditional GAN. ACGAN takes one or multiple discrete categorical vectors as input and generates data corresponding to the categorical vectors. In ACGAN, a classifier is trained to predict the label of real data, and a generator is trained so that the fake data generated with the discrete categorical vector is correctly classified by the classifier. However, these class-conditional GANs can only be trained given the labels (class-conditional vector) of the data. Therefore, these methods cannot be used with unlabeled datasets.

Unlike ACGAN, class-conditional InfoGAN [39] can generate class-conditional data even if the data is not labeled. In class-conditional InfoGAN, the classifier and generator are trained so that the conditional vector of the generated data is correctly classified by the classifier. InfoGAN has shown that it is possible to generate class-conditional data without knowing the conditional vector of the real data if the generator and the classifier are trained with classification loss. This is because the generator tries to generate class-conditional data that is easy to be classified by the classifier. For example, the MNIST handwritten digits dataset [61] consists of handwritten images of 10 different digits, each with a proportion of 0.1. If a class-conditional InfoGAN is trained on the MNIST dataset using a 10-dimensional categorical conditional vector, with each category assigned a probability of 0.1, then each conditional vector uniquely represents one of the ten digits. Although InfoGAN does not require a conditional vector of the real data, it can only be trained given the optimal categorical probability.

Elastic InfoGAN [11] proposed a method for class-conditional data generation even when the optimal prior categorical probability is not known. In elastic InfoGAN, the categorical latent probability is updated to minimize generator loss through gradient descent. Elastic InfoGAN used Gumbel softmax [16, 17] to perform gradient descent on the categorical latent probability. It also restricted each class to have the same identity by using contrastive loss [13] with identity-preserving transformations.

1.4.3 Generative Model Inversion

Another topic in deep learning generative models is generative model inversion. Generative models transform a simple latent distribution into a complex data distribution. Therefore, the inverse mapping of the generative model transforms a complex data distribution into an easy-to-sample continuous latent distribution. Generative model inversion can be considered as unsupervised feature learning (or representation learning), and it can be used for various tasks such as dimensionality reduction, clustering, independent component analysis, transfer learning, or multimodal application. For example, some works [45, 46, 52] used generative model inversion for out-of-distribution detection. InterFaceGAN [41] showed that attributes of data are linearly separable in latent space, and used it for continuous attribute edit of generated data. Work [47] improved classifier performance by augmenting the input sample using generative model inversion. Some works [48, 49] used generative model inversion for semantic segmentation.

VAEs have an encoder-decoder architecture, where during training, the decoder serves as a generative model, and the encoder automatically serves for the inversion of the generative model. Therefore, no additional methods are required to invert VAEs.

GAN does not have a separate encoder to invert the generator during training. Therefore, an additional method is needed to invert the generator in GAN. Optimization-based methods [23, 24, 25] perform gradient descent iteratively on the latent vector to minimize data reconstruction loss between input data and generated data. Those methods require

a lot of computation to predict the latent vector of input data since each gradient descent iteration requires forward propagation & backpropagation in the generator. For example, StyleGAN2 [20] used an optimization-based method with 1k iterations. It means 1k forward propagation & backpropagation of the generator are required to predict the latent vector of a single image. Also, there is no guarantee that the predicted latent vector follows the latent distribution.

Learning-based methods train an encoder to predict the latent vector of input data points. Therefore, learning-based methods require additional training of the encoder compared to optimization-based methods. Instead, learning-based methods require only one inference of the encoder to predict the latent vector of a single data. This means that learning-based methods can predict very quickly compared to optimization-based methods. Among the learning-based methods, [28, 29, 30] used CGAN [38] to train an encoder that inverts the generator. These methods consider the encoder as a conditional generative model, and train the encoder to generate a predicted latent vector from generated data. However, since those methods use conditional GAN, they are difficult to train and the performance is not good.

Recently, StyleGAN [19, 20] has shown remarkable performance in image generation. StyleGAN converts latent vectors to style vectors through a mapping network consisting of nonlinear fully connected layers, and then uses the style vectors to generate fake images through a synthesis network. Thus, the style vector is the intermediate layer output of the generative model. Some works [35, 36, 37] proposed the StyleGAN-specific inversion method to find an inverse mapping of the synthesis network of StyleGAN. However, since these methods are based on StyleGAN architecture, they are not applicable to other model architectures and cannot be extended to any data domain other than the image domain. Also, these StyleGAN inversion methods do not map images to latent vectors, but to style vectors. Since style vectors are intermediate layer outputs of the generator, each dimension of the style vector follows an unknown complex distribution rather than simple distributions such as normal or uniform distribution, and each dimension of the style vector is not independent.

Therefore, these StyleGAN inversion methods cannot be utilized in generative model inversion applications using latent space, such as InterFaceGAN [41] or AnoDLSGAN [52].

Works [31, 39] used squared error to train an encoder that predicts the latent vector. Assuming that the encoder is a Gaussian model, training the encoder with mean squared error (MSE) loss is a maximum likelihood estimation of the encoder (minimizing negative log-likelihood). Works [32, 33, 34] added data reconstruction loss to MSE loss for better performance.

Hybrid methods [26, 27] use both learning-based methods and optimization-based methods. Hybrid methods use an encoder to predict the latent vector, and then perform gradient descent on the latent vector to find a more precise latent vector.

Unlike VAEs or GANs, diffusion models are difficult to invert, and inversion may not be meaningful. The first reason is that diffusion models inherently lose information about the latent distribution. This is because they are trained such that different latent vectors (i.e., different noise samples) can generate the same output data. More specifically, during the reverse diffusion process, the model is explicitly trained to generate the same data point with a different latent vector. This many-to-one mapping makes it fundamentally difficult to recover the original latent vector, thus hindering invertibility. The second reason is that the latent dimension of diffusion models is typically high, as explained in Table 3. The latent dimension in diffusion models is equal to or n times larger than the data dimension. This high-dimensionality not only increases the difficulty of inversion but also makes the latent variables harder to interpret or manipulate in a meaningful way.

Many methods and applications of GAN inversion are introduced in the GAN inversion survey paper [18].

While most existing works on generative model inversion are empirical, there are some papers that cover theoretical aspects of generative model inversion. For instance, work [90] showed that when the generator is L-Lipschitz function, latent vectors can be accurately recovered using optimization-based methods. Specifically, they proved that $O(d_z \log L)$ random

Gaussian measurements are sufficient to guarantee an ℓ_2/ℓ_2 recovery under certain conditions.

Work [91] showed that when the generator's Jacobian is ill-conditioned, training becomes unstable and the model's performance degrades. To address this, they proposed Jacobian Clamping, which constrains the range of the singular values of the generator's Jacobian during training. This result suggests that generative models with well-conditioned Jacobians are not only more stable to train but also potentially more invertible and effective. This shows that when training a generative model to be invertible, the model can be trained more stably.

1.4.4 Out-of-distribution Detection

One of the fields in deep learning is out-of-distribution (OOD) detection. Most deep learning models assume that their input data and train data are sampled from the same distribution. However, the input data may not always be sampled from the same distribution as the train data. Those models do not work properly when the probability that input data is sampled from train data distribution is low (e.g., outliers, out-of-distribution data). For example, if a cat image is an input to a dog species classifier, the classifier may output the wrong results.

Adversarial attack (e.g., FGSM [54]) is also an example where the model does not properly work if input data is not sampled from the train data distribution. An adversarial attack is manipulating the model's predictions by adding artificial noise to the input data. When the model architecture and weights are given, very small noise generated by an adversarial attack can significantly change the model's predictions.

OOD detection is detecting inputs that are sampled from a distribution other than the in-distribution (ID), which is the train data distribution. Therefore, OOD detection is a binary classification problem of distinguishing between ID data and OOD data.

The main difference between simple classification and OOD detection is that in OOD detection, no information is given about the OOD data. In other words, the model is trained only on ID data and needs to detect OOD data that it has never seen before. In general, it is

almost impossible to define an OOD dataset and use it for ID-OOD classification in complex high-dimensional datasets, since the range of OOD data is very wide in high-dimensional data space.

Some methods [55, 56, 57, 58] used classifiers trained with ID data for OOD detection. These methods consider the intermediate layer output of the pre-trained classifier as an encoder, and use the feature vector (intermediate layer output) of input data to calculate the OOD score of input data. If the OOD score of input data is greater than the threshold, the input data is classified as OOD data. Otherwise, input data is classified as ID data.

Works [46, 45, 65] used reconstruction error for OOD detection. These methods reconstruct input data through an encoder and decoder, then use reconstruction error between the input data and reconstructed data as OOD score.

Works [59, 60] presented image domain-specific OOD detection methods. These methods cannot be applied to domains other than images.

1.4.5 Deep Generative Model with Codebook

Recently, some generative models, such as vector Quantization VAE (VQVAE) [84] and Vector Quantization GAN (VQGAN) [83], have used vector quantization to improve the performance of generative models. These models use a codebook architecture to ensure that the generators represent discrete features and improve the quality of the generated data. These models use an encoder to encode real data into low-dimensional quantized latent vectors and store them in a codebook. Then, the generative model generates data using the quantized vectors stored in the codebook. Thus, the codebook acts as a kind of memory for the generative model. This memory architecture allows the generator to generate high-quality data.

In general, the encoder used in vector quantization methods is trained with autoencoder loss (reconstruction loss). That is, the encoder encodes the input train data into a complex low-dimensional quantized vector, and the decoder is trained to reconstruct the input data

from the quantized vector. These quantized vectors are stored in the codebook and then loaded when the generator generates data. The quantized vectors loaded to generate data are determined by the prior categorical probability of the model.

2 Previous Works and Analysis

2.1 Class-Conditional GAN

2.1.1 Auxiliary Classifier GAN

ACGAN [3] is a class-conditional GAN that can generate class-conditional data distribution, given a labeled dataset. ACGAN consists of a discriminator, generator, and (auxiliary) classifier. The discriminator of ACGAN is trained with the adversarial loss of a typical GAN to distinguish between real and fake data, and the classifier is trained to predict the label (categorical conditional vector) of the input data. The generator takes latent random variables and categorical latent random variables as input and generates class-conditional data. To generate class-conditional data, the generator is trained so that the generated data is correctly classified by the classifier. Additionally, the classifier of ACGAN is integrated with a discriminator (shares hidden layers).

The following equations show losses for training ACGAN.

$$L_d = L^d_{adv} + L^r_{cls} + L^f_{cls} \tag{1}$$

$$L_g = L^g_{adv} + L^f_{cls} \tag{2}$$

$$L^r_{cls} = \mathbb{E}_{x,c_r}\left[-c_r \cdot \log Q(x)\right] \tag{3}$$

$$L^f_{cls} = \mathbb{E}_{z,c_f}\left[-c_f \cdot \log Q(G(z,c_f))\right] \tag{4}$$

$$L_{adv}^{d} = \mathbb{E}_{x,z,c'}\left[A_d(D(x), D(G(z, c_f)))\right] \tag{5}$$

$$L_{adv}^{g} = \mathbb{E}_{z,c_f}\left[A_g(D(G(z, c_f)))\right] \tag{6}$$

In Eqs. 1 and 2, L_d and L_g represent discriminator loss and generator loss for ACGAN training, respectively. L_{adv}^{d} and L_{adv}^{g} represent adversarial loss for discriminator and generator, respectively. L_{cls}^{r} and L_{cls}^{f} represent classification loss on real data and generated (fake) data, respectively. In ACGAN, the discriminator is integrated with the classifier. Therefore, classification losses L_{cls}^{r} and L_{cls}^{f} are included in the discriminator loss L_d.

In Eq. 3, x and c_r represent a real data point and its categorical conditional vector, respectively. Q represents d_c-dimensioanl classifier of the discriminator, where d_c represents class size (dimension of categorical vector). Operation "$\cdot$" represents the inner product. One can see that L_{cls}^{r} is a categorical cross-entropy loss to predict the labels of real data. In Eq. 4, z and c_f represent a latent vector and a categorical latent vector, respectively. G represents the generator. Therefore, $G(z, c_f)$ is a fake data point. Same as L_{cls}^{r}, L_{cls}^{f} is a categorical cross-entropy loss to predict the labels of fake data.

In Eqs. 5 and 6, A_d and A_g represent adversarial loss functions for the discriminator and generator, respectively. There are several adversarial loss functions [40] and regularizations [7].

The first problem of ACGAN is that fake data classification loss L_{cls}^{f} in discriminator loss L_d (Eq. 1) lowers model performance. If real data distribution and fake data distribution are the same, fake data classification loss L_{cls}^{f} is the same as real data classification loss L_{cls}^{r}. However, if fake data distribution and real data distribution are different, fake data classification loss L_{cls}^{f} is meaningless and can generate bad gradients.

The second problem is that there should be an additional hyperparameter to adjust the ratio of adversarial loss and classification loss. There are several types of adversarial losses [40] and regularization [7], and the optimal scale for each adversarial loss can be different. It

means that the optimal ratio between adversarial loss and classification loss depends on the type of adversarial loss and regularizations.

Therefore, one can think of a revised ACGAN loss that added a weighting hyperparameter for classification loss and removed fake data classification loss L_{cls}^{f} in discriminator loss L_d. Several works (e.g., [43, 44]) used revised ACGAN loss for class conditional GAN, but those papers did not analyze it. The revised ACGAN loss is as follows.

$$L_d = L_{adv}^{d} + \lambda_{cls} L_{cls}^{r} \tag{7}$$

$$L_g = L_{adv}^{g} + \lambda_{cls} L_{cls}^{f} \tag{8}$$

Eqs. 7 and 8 show losses for revised ACGAN. In Eqs. 7 and 8, there is no fake data classification loss L_{cls}^{f}, and the ratio between adversarial loss and classification loss can be adjusted by classification loss weight λ_{cls}. However, there are still some problems with the revised ACGAN loss.

The first problem is that there is an important hyperparameter, classification loss weight λ_{cls}. The optimal classification loss weight λ_{cls} can vary depending on the adversarial loss function and the type of regularization. Therefore, it may take a lot of costs to find optimal classification loss weight λ_{cls} for the model.

The second problem is that there is a conflict between adversarial loss and classification loss, resulting in a decrease in the generative performance of the model. To minimize classification loss, the generated data points should be moved away from the classifier's decision boundary. This produces a conflict between classification loss and adversarial loss.

The third problem is that at the beginning of training, the fake data classification loss L_{cls}^{f} of the generator loss produces a meaningless gradient. If the real data distribution is different from the generated data distribution, fake data classification loss L_{cls}^{f} will produce a meaningless gradient for the generator since the classifier is not trained with the generated data. Therefore, when the real and generated data distributions are different at the beginning

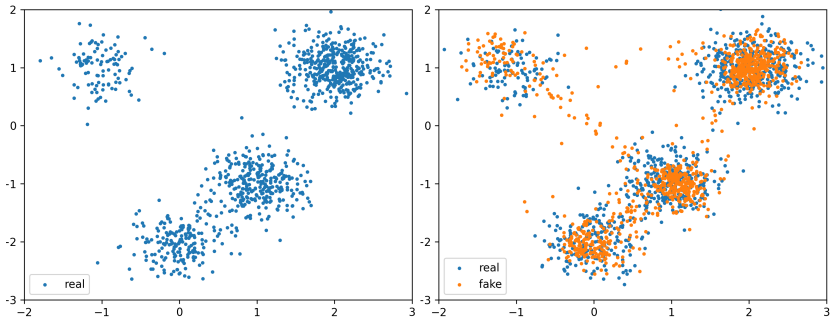

Figure 1: Left plot: Two-dimensional dataset consisting of four Gaussian clusters. The centers and probabilities of each cluster are $(-1, 1), (0, -2), (1, -1), (2, 1)$ and $[0.1, 0.2, 0.3, 0.4]$, respectively. The standard deviation for all clusters is 0.3. Right plot: Samples generated by GAN trained only with a continuous latent distribution.

of training, the fake data classification loss L_{cls}^{f} of the generator loss can lower the training speed of the model.

2.1.2 Unsupervised Class-Conditional GAN

Typically, when training a GAN, everything is assumed to be continuous. It means that the data distribution and latent distribution are assumed to be continuous, and the generator and discriminator of GAN are assumed to be continuous functions (in general, deep learning models are differentiable continuous functions for error backpropagation). However, some data distributions may be better represented as both discrete and continuous latent distributions. Specifically, target data distributions with concavities in the probability density function in the feature space can be better represented using both discrete and continuous latent random variables. Still, approximating the concave part with a low probability density of the target data distribution is still not easy for most deep generative models, which is a continuous function.

The left plot of Fig. 1 shows a data distribution example consisting of four Gaussian clusters. The probability density function of the data distribution is continuous, and there are no points where $P(x) = 0$. However, it is easier and better to represent this data distribution with a continuous latent distribution and a 4-dimensional discrete categorical

latent distribution.

The right plot of Fig. 1 shows generated data with GAN trained only with a continuous latent distribution. One can see that the generator generates samples between each cluster, which appear to be connecting each cluster. This is because the latent distribution is continuous, and the generator is a continuous function, making it difficult to represent concave parts with low probability density. As training progresses, the probability density of the concave part decreases, but it requires a long training period, and it is hard to say that the continuous latent distribution correctly represents data distribution (i.e., entangled data representation).

For data distributions whose probability density function has concave parts in the feature space, using a discrete latent distribution is more appropriate for model training and disentangled data representation. Class-conditional generative models, such as ACGAN [3] or CAGAN [4], take both continuous latent distribution and discrete categorical latent distribution as inputs and generate class-conditional data distribution. Training model with discrete categorical latent random variable allows the model to represent concave parts of the dataset appropriately. However, ACGAN or CAGAN can only be trained when class-conditional vectors (labels) of real data are given.

Class-conditional InfoGAN [39] can perform class-conditional data generation and inversion (clustering) by maximizing mutual information of generator input categorical latent distribution and classifier output distribution, even if the data is not labeled. The following equations show losses for (class-conditional) InfoGAN.

$$L_q = \lambda_{cls} L_{cls} \tag{9}$$

$$L_d = L^d_{adv} \tag{10}$$

$$L_g = L^g_{adv} + \lambda_{cls} L_{cls} \tag{11}$$

$$L_{cls} = \mathbb{E}_{z,c_f}\left[-c_f \cdot \log(Q(G(z, c_f)))\right] \tag{12}$$

$$L^d_{adv} = \mathbb{E}_{x,z,c_f}\left[A_d(D(x), D(G(z, c_f)))\right] \tag{13}$$

$$L^g_{adv} = \mathbb{E}_{z,c_f}\left[A_g(D(G(z, c_f)))\right] \tag{14}$$

In Eqs. 9, 10, and 11, L_q, L_d, and L_g represent classifier loss, discriminator loss, and generator loss of InfoGAN, respectively. L_{cls} and λ_{cls} represent classification loss and classification loss weight, respectively. In Eqs. 12, 13 and 14, Q, D, and G represent classifier, discriminator, and generator, respectively. In Eq. 12, L_{cls} is categorical cross entropy between categorical latent vector c and predicted probability of generated data $Q(G(z, c_f))$. c_f and z represent the categorical latent vector and continuous latent vector sampled from the categorical latent distribution C and continuous latent distribution Z, respectively. Operation "$\cdot$" represents the inner product. In Eqs. 13 and 14, A_d and A_g represent adversarial loss function [6] for GAN training. In InfoGAN, a classifier Q can share hidden layers with a discriminator D for efficiency.

From the above equations, one can see that a classifier Q and a generator G are trained to minimize classification loss L_{cls}. InfoGAN has shown that, given an appropriate categorical latent distribution C, it can perform class-conditional data generation and clustering (inversion) even when the data is unlabeled. However, InfoGAN still needs prior probability of categorical latent distribution C. Without knowing the appropriate categorical latent distribution C, InfoGAN cannot perform class-conditional data generation and clustering appropriately. Additionally, InfoGAN's generator is trained with both adversarial loss and classification loss like ACGAN. It means that InfoGAN shares the same problems as ACGAN: adversarial loss and classification loss conflict with each other in generator loss, resulting in a decrease in the generative performance of the model. Specifically, the density of fake data is always lower than the density of real data near the decision boundary of classifier Q. This is because the generator is trained to move the generated data away from the decision boundary

due to the classification loss.

Elastic InfoGAN [11] proposed a method for class-conditional data generation when the prior probability of a categorical latent distribution is not known. In Elastic InfoGAN, the categorical latent distribution probability is updated to minimize generator loss through gradient descent. To allow the gradient to flow up to a categorical latent distribution probability, Elastic InfoGAN uses Gumbel softmax [16, 17]. Elastic InfoGAN also used contrastive loss [13] to ensure that each class has the same identity. Contrastive loss allows augmented data with identity-preserving transformations to be classified in the same class. By training the classifier with contrastive loss, generators are constrained to generate data with the same identity if they are of the same class. For example, Elastic InfoGAN has used rotation, zoom, flip, crop, and gamma change as identity-preserving transformations for image data.

However, there are still several problems with Elastic InfoGAN. First, contrastive loss can only be used if a good transformation that preserves the identity of the data is known. Therefore, it cannot be used in data domains where good identity-preserving transformations are not known. Second, clustering only can be performed based on identity-preserving transformations. For example, on the MNIST handwritten digits dataset, Elastic InfoGAN will consider digits 6 and 9 as the same class if 180-degree rotation is used for identity-preserving transformation. Also, like InfoGAN, Elastic InfoGAN uses classification loss for the generator like ACGAN, which causes conflict between adversarial loss and classification loss and decreases the generative performance of the model.

2.2 GAN Inversion

Assume that generator G maps the latent random variable Z to the data random variable X (i.e., $X = G(Z)$). The purpose of the learning-based GAN inversion method is to train an encoder E that inverts the generator G (i.e., $Z = E(G(Z))$).

When the latent random variable Z is a d_z-dimensional random variable where each dimension is i.i.d., the encoder E can be considered as an integration of d_z encoders, where

each encoder is trained to recover each element of a latent random variable Z (i.e., $Z_1 = E_1(G(Z)), Z_2 = E_2(G(Z)), \ldots, Z_{d_z} = E_{d_z}(G(Z))$). Assuming each encoder is a Gaussian model, training each encoder with an MSE loss minimizes the negative log-likelihood of each encoder.

However, the integrated encoder E cannot fully recover the latent random variable Z since there is no guarantee that the generator G uses all the information of the latent random variable Z. For example, when the latent random variable Z has too many dimensions, generator G can be trained to ignore some dimensions of the latent random variable Z. Or, generator G can be trained so that some elements of the latent random variable Z are less important than others.

It means that different latent vectors z_a and z_b sampled from the latent random variable Z can be mapped to the same or similar generated data points $G(z_a)$ and $G(z_b)$. In this case, some encoders of the integrated encoder E cannot converge or converge very slowly to predict some element of the latent random variable Z. In other words, the generator loses the information of the latent random variable Z, and the encoder E cannot perfectly recover the latent random variable Z from the generated data random variable $G(Z)$.

The upper part of Fig. 2 shows an example of the problem that occurs when training an encoder without a dynamic latent scale. In the upper part of Fig. 2, the generator ignores the third element of the latent vector. Therefore, different latent vectors ($[0.9, -0.7, 0.5]$ and $[0.9, -0.7, 0.3]$), are mapped to the same generated data point A. In this case, the encoder cannot recover the third element of the latent vector and vibrates to minimize the non-converging loss, which degrades the performance of the encoder.

Nevertheless, several works [32, 33, 34] used MSE loss to train the encoder to predict the latent vector of the generated data.

$$L_{enc} = avg\left((Z - E_l(G(Z)))^2\right) \tag{15}$$

Eq. 15 shows MSE loss for training the encoder that inverts the generator. In Eq. 15,

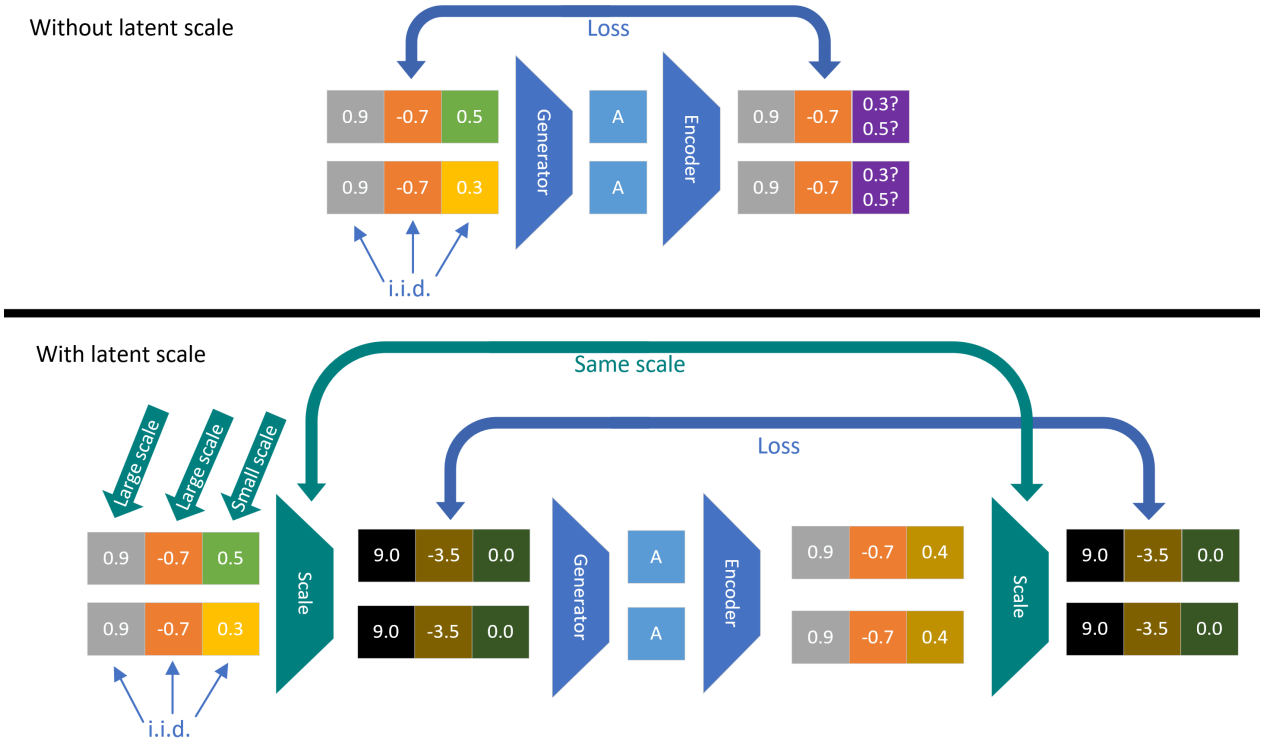

Figure 2: Training encoder with and without latent scale

L_{enc} represents the encoder loss for the encoder. E_l and G represent the (latent) encoder and the generator, respectively. Z represents the latent random variable, and $G(Z)$ is generated data random variable. avg is the vector average function. When using simple MSE loss, encoder E_l can share hidden layers with discriminator D.

InfoGAN [39] proposed a method to train an encoder to predict some dimension of the latent random variable Z. In InfoGAN, both generators and encoders are trained to minimize encoder loss. In addition, InfoGAN's encoder predicts the mean and variance of the latent vector and is trained to maximize the log-likelihood of the predicted latent vector. It was simply assumed that the additional noise of the predicted latent vector follows the normal distribution in InfoGAN. InfoGAN can be extended to train the encoder to predict all dimensions of the latent random variable Z. The following equation shows the loss for InfoGAN predicting all dimensions of the latent random variable.

$$L_{enc} = avg\left(E_{lv}(G(Z)) + \frac{(Z - E_l(G(Z)))^2}{e^{E_{lv}(G(Z))}}\right) \tag{16}$$

In Eq. 16, the log variance encoder E_{lv} predicts the log variance of the additional normal noise. In InfoGAN, encoders E_l, E_{lv}, and discriminator D can share hidden layers.

When encoders are integrated with the discriminator, the losses for the above GAN inversion methods are as follows:

$$L_d = L^d_{adv} + \lambda_{enc} L_{enc} \tag{17}$$

$$L_g = L^g_{adv} + \lambda_{enc} L_{enc} \tag{18}$$

In Eqs. 17 and 18, L_d and L_g represent discriminator loss and generator loss, respectively. L^d_{adv} and L^g_{adv} represent adversarial loss for GAN training. Several adversarial loss functions can be found in the work [6]. λ_{enc} represents encoder loss weight.

VAEGAN [68] integrated perceptual VAE and GAN for better generative and inversion performance. In VAEGAN, the generator (decoder) is trained with both adversarial loss of GAN and perceptual VAE loss. The VAE latent random variable predicted from the encoder is used to train both GAN and VAE. The following equations show losses for VAEGAN training when $Z \sim N(0, I_{d_z})$, where d_z represents the dimension of latent random variable Z.

$$Z_x = E_l(X) + N(0, I_{d_z}) \circ e^{E_{lv}(X)/2} \tag{19}$$

$$L_{rec} = avg\left((E_f(G(Z_x)) - E_f(X))^2\right) \tag{20}$$

$$L_{prr} = avg\left(E_l(X)^2 - E_{lv}(X) + e^{E_{lv}(X)}\right) \tag{21}$$

$$L_d = L^d_{adv} \tag{22}$$

$$L_g = L^g_{adv} + \lambda_{rec} L_{rec} \tag{23}$$

$$L_e = \lambda_{rec} L_{rec} + \lambda_{prr} L_{prr} \tag{24}$$

Eqs. 19-24 show losses for training VAEGAN. Z_x in Eq. 19 shows the VAE latent random variable that is used for both GAN and VAE training. Eq. 20 shows perceptual reconstruction loss for perceptual VAE training. E_f represents the feature encoder. VAEGAN uses feature encoder E_f for perceptual reconstruction loss, instead of using simple MSE reconstruction loss. The intermediate layer output of the discriminator D is used as the feature encoder E_f. Encoders E_l, E_{lv}, and generator G are trained to minimize perceptual reconstruction loss L_{rec}. Eq. 21 shows a prior loss for perceptual VAE training. To use Z_x for GAN training, Z_x should follow $Z \sim N(0, I_{d_z})$. VAEGAN uses prior loss L_{prr} to make Z_x follow Z. Encoders E_l and E_{lv} are trained to minimize prior loss L_{prr}. Eqs. 22, 23, and 24 show losses for the discriminator, generator, and encoders, respectively. λ_{rec} and λ_{prr} represent reconstruction loss weight and prior loss weight, respectively.

In VAEGAN, if the prior loss weight λ_{prr} is too low, Z_x may not follow $Z \sim N(0, I_{d_z})$. If $\lambda_{prr} = 0$, VAEGAN is equivalent to training the model with perceptual autoencoder and GAN, and latent encoder E_l becomes equivalent to the encoder of the perceptual autoencoder. It means that Z_x does not have the useful properties of Z (e.g., each dimension is independent, and each dimension follows a normal distribution). Furthermore, since Z_x is used for GAN training, the generative performance of VAEGAN can decrease. This is because the model is trained to generate data using Z_x and Z, but only Z should be used to evaluate the generation performance of the model.

2.3 Out-Of-Distribution Detection

OOD detection is a binary classification problem. Therefore, input data should be mapped to a one-dimensional scalar to classify input data. Most previous works set this one-dimensional scalar as an OOD score, and if the OOD score of input data is greater than a threshold, input data is classified as OOD data, otherwise as ID data.

Works [55, 56, 57, 58] used a classifier for OOD detection. Given labels of ID data, these

methods train a classifier to predict the label of input data. Considering the intermediate layer output of the trained classifier as a feature vector of the input data, these methods calculate the OOD score of input data using the feature vector of it.

The theoretical problem of these methods is that the encoder (input layer $\sim$ intermediate layer of classifier) may lose the information of input data. For example, consider an ID dataset of MNIST handwritten digit images [61] (examples in Fig. 29) with additional noise. In this dataset, the top left pixel (or dimension) a has a random noise value independent of the other pixels (or dimensions). The pixel a follows $U(0.5, 1.0)$. Assuming that pixel a does not affect the label of data, the ideal classifier will ignore pixel a so that pixel a does not affect the prediction. Then, consider a modified ID image where only pixel $a = 0.0$ and other pixels are the same as the ID image. This image is definitely an OOD image, but OOD detection methods using the ideal classifier cannot detect it since the ideal classifier will ignore pixel a.

Furthermore, other OOD detection methods using pre-trained models potentially have the same problem (encoder losing information problem). Also, these methods require a labeled ID dataset.

Works [46, 45, 65] used reconstruction error for OOD detection. These methods first train autoencoder or GAN with ID data. Then, calculate the OOD score with the difference between input data and reconstructed data.

The problem with these methods is that it is hard to define a good reconstruction error function (i.e., metric function). For example, consider the ID dataset is MNIST handwritten digit images, and the pixel-wise mean squared error function is used as the reconstruction error function. One can simply think of OOD images that have low pixel-wise mean squared error but large perceptual error between ID data (e.g., OOD images in Fig. 29). These methods using pixel-wise mean squared error function may not detect these OOD images. The opposite is also true. It is hard to define a good perceptual error function, but just assume that there is a good perceptual error function. If these OOD detection methods use a good

perceptual error function, one can also simply think of OOD images that have low perceptual error but large pixel-wise mean squared error (e.g., slightly increase the brightness). These OOD detection methods may not detect these OOD images.

Therefore, reconstruction error-based OOD detection methods are greatly affected by the type of OOD data and the distance metric for the reconstruction error function.

2.4 Training Generative Model with Discrete Latent Random Variable

Some generative models use latent random variables that are discrete in some dimensions. For example, class-conditional generative models, such as class-conditional GAN discussed in section 2.1, use discrete conditional vectors to generate class-conditional data. Also, generative models using vector quantization, discussed in section 1.4.5, use quantized vector selected from the codebook with prior probability as the input of the generative model. It can be said that generative models with vector quantization use discrete latent random variable, as they select quantized vectors from discrete categorical distributions.

However, models with latent random variables that are discrete in some dimensions should be careful not to overweight the discrete dimensions relative to the continuous dimensions. Because, if the latent random variable is completely discrete, the models may not have some useful properties compared to generative models that use continuous latent random variables.

The following algorithm shows an example of a discrete generative model.

Algorithm 1 Sampling process of perfect discrete generative model

Require: $x_1, x_2, \ldots, x_n$

1: $book \leftarrow [x_1, x_2, \ldots, x_n]$
2: $z \leftarrow sample(U(0, 1))$
3: $i \leftarrow round(z \times n + 0.5)$
4: $return\ book\,[i]$

Algorithm 1 shows the sampling process of the perfect discrete generative model. In algorithm 1, training data $x_1, x_2, \ldots, x_n$ were given to train the model. In line 1, the generative

model simply stores all data in the book. Then, it returns data sampled uniformly from this book. This simple generative model generates a data distribution that perfectly matches the train data distribution. Also, latent distribution is continuous ($Z \sim U(0, 1)$). However, it is impossible to interpret data from the latent distribution, and the model cannot perform latent interpolation. This is because the model is perfectly discrete. Thus, in addition to the generative performance of the model, the generalization performance of the model is also important for a generative model. Methods such as Perceptual Path Length (PPL) [20] can be used to evaluate the generalization performance of the generative models.

3 Deep Generative Models and Their Inversions

3.1 Conditional Activation GAN: Improved Auxiliary Classifier GAN

In section 2.1.1, we discussed the problem of ACGAN and revised ACGAN. In this section, we propose Conditional Activation GAN (CAGAN) to reduce a hyperparameter of ACGAN and improve the model performance.

A labeled dataset can be thought of as a combination of datasets for each class. The proposed CAGAN trains GAN for each class to generate class-conditional data, instead of using a classifier. Among multiple GANs, class-conditional data can be generated by activating only the GAN corresponding to the input conditional vector to generate data. Furthermore, since the datasets of each GAN are similar to each other, they can be considered as a single model by making them share hidden layers.

The following equations represent losses for CAGAN.

$$L_d = \Sigma_{i=1}^{d_c} L_d^i \tag{25}$$

$$L_g = \Sigma_{i=1}^{d_c} L_g^i \tag{26}$$

$$L_d^i = \mathbb{E}_{x,c_r,z,c_f}\left[A_d(D^i(x)\cdot c_r^i, D^i(G(z,c_f))\cdot c_f^i)\right] \tag{27}$$

$$L_g^i = \mathbb{E}_{z,c_f}\left[A_g(D^i(G(z,c_f))\cdot c_f^i)\right] \tag{28}$$

Eqs. 25 and 26 show losses for CAGAN. In Eqs. 25 and 26, L_d^i and L_g^i represent i-th discriminator loss and i-th generator loss, respectively. d_c represents the dimension of the category. "$\cdot$" represents the inner product. Eqs. 25 and 26 show that CAGAN losses (L_d and L_g) are the sum of GAN losses for each class (L_d^i and L_g^i).

In Eqs. 27 and 28, D^i represent i-th discriminator. c_r and c_f represent labels of real data and generated (fake) data, respectively. A_d and A_g represent adversarial loss functions for training GAN. Work [6] summarized several adversarial loss functions for training GAN. c_r^i and c_f^i represent i-th value of c_r and c_f, respectively. Since c_r and c_f are one-hot vectors, they have only binary (activation) values (0 or 1). In Eq. 27, one can see that i-th discriminator D^i is trained only if activation value c_r^i or c_f^i is 1. Likewise, in Eq. 28, one can see that generator G is trained with D^i only if activation value c_f^i is 1. Each GAN will not be trained if its activation value is 0. Since each GAN shares all hidden layers in CAGAN, CAGAN loss can be combined as the following losses.

$$L_d = \mathbb{E}_{x,c_r,z,c_f}\left[A_d(D(x)\cdot c_r, D(G(z,c_f))\cdot c_f)\right] \tag{29}$$

$$L_g = \mathbb{E}_{z,c_f}\left[A_g(D(G(z,c_f))\cdot c_f)\right] \tag{30}$$

Eqs. 29 and 30 show combined adversarial losses for training CAGAN. Unlike ACGAN, the output of D is d_c-dimensional adversarial value since D of CAGAN is an integration of D^is.

Note that the binary condition value should be encoded as a 2-dimensional one-hot vector in CAGAN. For example, consider the 3-label human face image generation task. The first label has 3 classes (black hair, blond hair, and white hair). The second and third labels are

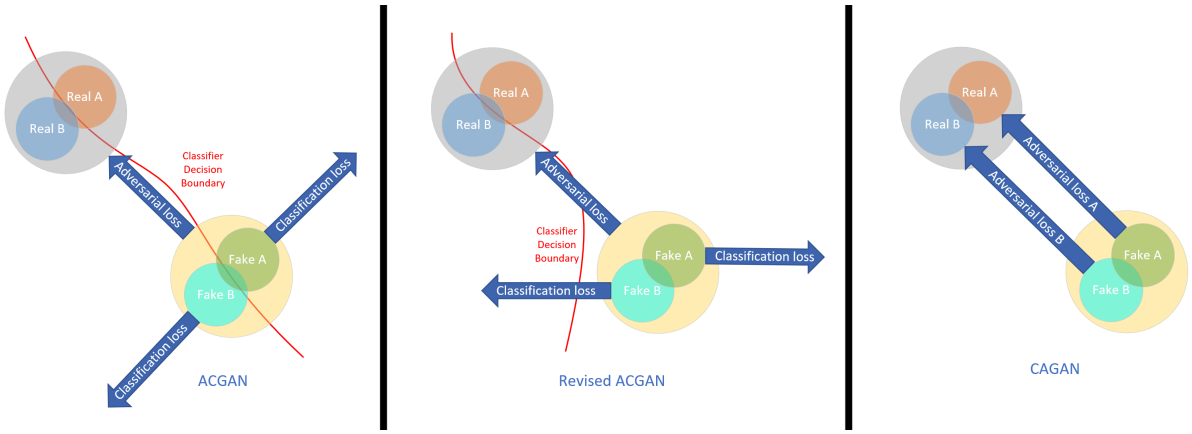

Figure 3: Gradient direction (blue arrow) for the fake data in ACGAN, revised ACGAN, and CAGAN at the beginning of the training. The red line represents the classifier decision boundary of ACGAN or revised ACGAN.

binary conditions (smiling/not smiling and bangs/no bangs). In this case, condition vector dimension d_c of CAGAN should be $3 + 2 + 2 = 7$. Therefore, the discriminator output dimension of CAGAN is 7. On the other hand, condition vector dimension d_c of ACGAN can be $3 + 2 + 2 = 7$, but $3 + 1 + 1 = 5$ or $3 + 1 + 2 = 6$ is also available. Therefore, the discriminator output dimension of ACGAN can be $6 \sim 8$.

Fig. 3 shows the gradient direction for the fake data in ACGAN, revised ACGAN, and CAGAN at the beginning of the training. One can see that the gradient direction for the fake data is in the wrong direction due to classification loss. This is because the generator should generate data away from the decision boundary of the classifier to minimize classification loss. This classification loss lowers the performance of the model because it continues to generate bad gradients, even as training progresses and the real and fake data distributions become nearly identical or identical. On the other hand, the gradient direction of CAGAN is good because it uses only adversarial losses, not classification losses.

Some works [7, 5] used multiple adversarial losses for class-conditional data generation, but did not analyze why using multiple adversarial losses is better than using classification loss.

3.1.1 Mixed Batch Training

Batch-wise operations such as batch normalization [67] or minibatch standard deviation layer [8] have been widely used in GAN architecture [8, 19, 20].

However, when training conditional GAN, adversarial loss and conditional generation can conflict with each other if the discriminator has batch-wise operations, and real conditional distribution and fake conditional distribution are different. This is because when the conditional distribution of real data and fake data are different, the discriminator uses the conditional distribution of the input batch for real/fake discrimination.

For example, consider biased MNIST handwritten digit dataset [61], where the percentage of zero class is far greater than any other number classes. If each class of categorical latent distribution has the same probability, the discriminator with batch-wise operations will use a class-conditional distribution of input batch for real/fake discrimination. If there are many zeros in the input batch, the probability that the input batch was real images becomes high, and otherwise, it becomes low in the discriminator. Therefore, the adversarial loss to generate real data and the conditional loss to generate conditional data conflict with each other in the generator. This lowers the generative performance of conditional GAN. If the conditional distribution is categorical, the categorical latent distribution can be easy to adjust. However, if the conditional distribution is complex and continuous (e.g., Pix2Pix [14]), it can be difficult to control the conditional latent distribution.

To address the problems described above, we introduce mixed batch training to prevent conflict between adversarial loss and conditional generation. Mixed batch training is to configure each batch for the discriminator to always have the same ratio of real data and fake data. If all batches consist of an equal ratio of real data and fake data, each batch always has the same conditional distribution. Therefore, the discriminator will not discriminate between real/fake by conditional distribution, and there will be no conflict between adversarial loss and conditional loss in the generator.

However, mixed batch training makes it easy for the discriminator to discriminate input

data. This causes an imbalance between the generator and the discriminator, especially at the beginning of training when the distance between the fake data distribution and the real data distribution is far.

The proposed method gradually changes the ratio of real data to fake data to alleviate this imbalance. For example, one can configure $real\ data : fake\ data$ in each 2 batch equals $100 : 0$ and $0 : 100$ in epoch 1, $10 : 90$ and $90 : 10$ in epoch 11, $50 : 50$ and $50 : 50$ in epoch 51. After that, one can continue to configure each batch as $50 : 50$.

3.2 Dynamic Latent Scale GAN for GAN Inversion

In section 2.2, we discussed problems of previous GAN inversion methods. In this section, we introduce a DLSGAN that dynamically adjusts the scale of each element of the latent random variable Z to prevent the generator G from losing information of the latent random variable Z.

Assume that the latent random variable Z is d_z-dimensional random variable where each dimension is i.i.d., with the variance σ^2. When the encoder E is trained enough to predict the latent random variable Z from the generated data random variable $G(Z)$ with MSE loss, the variance of each element of the predicted latent random variable $Z' = E(G(Z))$ represents information of the latent random variable Z that can be recovered from the generated data random variable $G(Z)$. If the variance of n-th predicted latent random variable Z'_n is zero, it means that the encoder E cannot recover any information of n-th latent random variable Z_n from the generated data random variable $G(Z)$.

On the other hand, if the variance of the n-th predicted latent random variable Z'_n is σ^2, then the encoder E can recover all information of n-th latent random variable Z_n from the generated data random variable $G(Z)$. If the variance of the n-th predicted latent random variable Z'_n is greater than 0 but lower than σ^2, it means that the encoder E can recover some information of n-th latent random variable Z_n from the generated data random variable $G(Z)$. Therefore, if the element-wise variance of the predicted latent random variable Z' and

the element-wise variance of the latent random variable Z are the same, it means that the generator G does not lose the information of the latent random variable Z, and the encoder can converge to predict the latent random variable Z from the generated data random variable $G(Z)$.

DLSGAN dynamically adjusts the scale of each element of the latent random variable Z according to the variance of each element of the predicted latent random variable Z' so that the element-wise variance of scaled latent random variable and scaled predicted latent random variable are equal. The following equations show the latent scale vector and encoder loss of DLSGAN.

$$v \approx E(G(Z \circ s))^2 \tag{31}$$

$$s = \frac{\sqrt{d_z v}}{\|\sqrt{v}\|_2} \tag{32}$$

$$L_{enc} = avg\left(((Z - E(G(Z \circ s))) \circ s)^2\right) \tag{33}$$

$$L_d = L_{adv}^d + \lambda_{enc} L_{enc} \tag{34}$$

$$L_g = L_{adv}^g + \lambda_{enc} L_{enc} \tag{35}$$

In Eqs. 31, 32 and 33, it was assumed that the latent random variable Z is an random variable with mean 0 and variance 1 where each dimension is i.i.d., and multiplication-invariant distribution.

Eqs. 31 and 32 show latent variance vector v and latent scale vector s, respectively. Operation "$\circ$" represents the element-wise multiplication, and $\|vec\|_2$ represents the L2 norm of vector vec. DLSGAN uses scaled latent random variable $Z \circ s$ to generate data $G(Z \circ s)$. The latent variance vector v is the approximated element-wise variance of the predicted latent random variable. More specifically, DLSGAN uses the moving average of squared predicted fake latent random variable $E_l(G(Z \circ s))^{\circ 2}$ to approximate the latent variance vector v during the training. It is ideal to approximate the predicted latent variance vector

v for every training step with many samples, but for efficiency, the predicted latent variance vector v is approximated through predicted latent vectors from the past training steps.

The latent scale vector s is the normalized latent variance vector. When all elements of v are the same (i.e. when the variance of all elements of a predicted latent random variable Z' are the same), all elements of the latent scale vector s are 1, and the scaled latent random variable $Z \circ s$ has the largest differential entropy. On the other hand, when the variance of only one element of the latent variance vector is not 0, and the other elements are 0, only that dimension has the scale of $\sqrt{d_z}$, and other scales are 0, and the scaled latent random variable $Z \circ s$ has the least differential entropy. At the beginning of training, the variance of all dimensions of the encoder output is similar, so the scaled latent random variable $Z \circ s$ has large differential entropy. As training progresses, the variance of each dimension of the encoder output changes, and the scaled latent random variable $Z \circ s$ decreases.

Eq. 33 shows encoder loss for DLSGAN. The encoder loss L_{enc} in DLSGAN is the squared error loss between the scaled latent random variable $Z \circ s$ and the predicted scaled latent random variable $E(G(Z \circ s)) \circ s$. In DLSGAN, both encoder E and generator G are trained to minimize encoder loss L_{enc} like InfoGAN. Also, encoder E can be integrated into the discriminator D. Therefore, losses for DLSGAN can be summarized in Eqs. 34 and 35, which is the same as Eqs. 17 and 18.

The lower part of Fig. 2 shows an example of the dynamic latent scale preventing the problem caused by the generator losing information. In Fig. 2, the generator ignores the third element of the latent vector, so the scale of the third element becomes 0. Therefore, the generator maps the same scaled latent vector to the same generated data point A, and the encoder can converge because there is no loss for the third element.

DLSGAN is still the maximum likelihood estimation of the encoder (minimize negative log-likelihood), but the generator G does not lose the information of the latent random variable Z, which allows the encoder E to converge when training the encoder E with squared error loss.

Lemma.

When Z_i is i-th element of Z and s_i is i-th element of s, entropy of scaled latent random variable $h(s \circ Z)$ can be calculated as follows:

$$h(s \circ Z) = h(Z) + \sum_{i=1}^{d_z} \log s_i$$

Proof.

Let $Y_i = s_i Z_i$. Then the probability density function of Y_i is

$$f_{Y_i}(y) = \frac{1}{s_i} f_{Z_i}\left(\frac{y}{s_i}\right)$$

Hence,

$$\begin{aligned} h(s_i Z_i) &= -\int f_{Y_i}(y)\ \log f_{Y_i}(y)\, dy \\ &= -\int \frac{1}{s_i} f_{Z_i}\left(\frac{y}{s_i}\right) \log\left[\frac{1}{s_i} f_{Z_i}\left(\frac{y}{s_i}\right)\right] dy \\ &= -\int f_{Z_i}(z)\ \log f_{Z_i}(z)\, dz\ +\ \log s_i \\ &= h(Z_i) + \log s_i \end{aligned}$$

Therefore,

$$h(s \circ Z) = h(Z) + \sum_{i=1}^{d_z} \log s_i$$

Furthermore, $\sum_{i=1}^{d_z} s_i^2 = d_z$.

And let $f(s) = \sum_{i=1}^{d_z} \log s_i$. Then, from the Lagrangian,

$$\mathcal{L}(\mathbf{s}, \lambda) = \sum_{i=1}^{d_z} \log s_i - \lambda \left(\sum_{i=1}^{d_z} s_i^2 - d_z \right)$$

Setting gradient to zero,

$$\frac{\partial \mathcal{L}}{\partial s_i} = \frac{1}{s_i} - 2\lambda s_i = 0$$

Therefore,

$$s_i^2 = \frac{1}{2\lambda}$$

Since all s_i are the same, the entropy of scaled latent random variable $h(s \circ Z)$ is maximized when $s = \mathbf{1_{d_z}}$

This proof follows from Theorem 8.6.4 in book [89].

Algorithm 2 shows the algorithm to train DLSGAN.

In algorithm 2, D^*, G, Z, and X represent integrated discriminator, generator, latent random variable, and data random variable, respectively. Because the encoder E is integrated with the discriminator D, the integrated discriminator D^* outputs two values: 1-dimensional adversarial value and d_z-dimensional predicted latent vector.

In lines 1 and 2, Z is a d_z-dimensional latent random variable where each dimension is i.i.d., and X is a data random variable. In algorithm 2, it was assumed that latent random variable Z follows a distribution with a mean of 0 and a variance of 1 for convenience (e.g., $N(0, 1^2)$ or $U(-\sqrt{3}, \sqrt{3})$). *sample* is a function that samples a single sample from a random variable. z represents a latent vector sampled from the latent random variable Z. x represents a data point sampled from the data random variable X.

In line 4, integrated discriminator D^* predicts fake adversarial value a_f and predicted latent vector z' from generated data point $G(z \circ s)$. In line 5, a_r represents the adversarial value of a real data point x. "_" represents not using value. Because the latent vector of the real data point x is unknown, the predicted latent vector for the real data point x is discarded in DLSGAN training. In line 6, L_{enc} is encoder loss. One can see that the encoder loss L_{enc} is equal to the MSE loss between the scaled latent vector $z \circ s$ and the scaled predicted latent

Algorithm 2 Algorithm to train DLSGAN

Require: D^*, G, Z, X, v

▷ update D^*

$z \leftarrow sample(Z)$
$x \leftarrow sample(X)$
$s \leftarrow \frac{\sqrt{d_z}v}{\|\sqrt{v}\|_2}$

$a_f, z' \leftarrow D^*(G(z \circ s))$
$a_r, \_ \leftarrow D^*(x)$
$L_{enc} \leftarrow \frac{1}{d_z}\|(z - z') \circ s\|_2^2$
$L_{adv}^d \leftarrow A_d(a_r, a_f)$

$L_d \leftarrow L_{adv}^d + \lambda_{enc} L_{enc}$
$D^* \leftarrow minimize(D^*, L_d)$

$v_1 \leftarrow z'^2$

▷ update G

$z \leftarrow sample(Z)$

$a_f, z' \leftarrow D^*(G(z \circ s))$
$L_{enc} \leftarrow \frac{1}{d_z}\|(z - z') \circ s\|_2^2$
$L_{adv}^g \leftarrow A_g(a_f)$

$L_g \leftarrow L_{adv}^g + \lambda_{enc} L_{enc}$
$G \leftarrow minimize(G, L_g)$

▷ update v

$v_2 \leftarrow z'^2$
$v \leftarrow update(v, concat(v_1, v_2))$

vector $z' \circ s$. In line 7, A_d is the adversarial loss function for the discriminator D^*. One can find many adversarial losses in GAN adversarial losses compare paper [6].

In line 8, λ_{enc} is encoder loss weight. In line 9, integrated discriminator D^* is updated to minimize discriminator loss L_d.

In line 14, A_g represents the adversarial loss function for the generator.

In lines 8 and 15, one can see encoder loss L_{enc} is added to both generator loss L_g and discriminator loss L_d. This means that the generator G and discriminator D are trained cooperatively to reduce the encoder loss L_{enc}. Training the encoder E during the GAN training enables adding encoder loss L_{enc} to generator loss L_g.

In line 16, generator G is updated to minimize the generator loss L_g.

In line 18, latent variance vector v is updated with squared predicted latent vector v_1 and v_2. Because the mean of the predicted latent random variable Z' becomes automatically zero, simply squared predicted latent vector z'^2 can be considered as the sample variance of the predicted latent random variable Z'. A moving average, exponential moving average, or other scheduling functions can be used for the *update* function.

In DLSGAN, we assume that each dimension of the latent distribution is independent and follows a simple distribution. This assumption allows the transformed data distribution to remain interpretable and easy to understand. If the latent dimensions were not independent and followed a more complex distribution, such as the latent distribution of autoencoders, the resulting representation might be more difficult to interpret and analyze. In this sense, a simple latent distribution, where each dimension is independent of the other and follows a simple distribution, is more suitable for representation learning through generative model inversion. Nevertheless, it is worth considering alternative latent distributions such as multivariate Gaussian distributions or Gaussian mixture models, where the dimensions are dependent but the overall distribution remains relatively simple and interpretable. Such distributions may require additional techniques to appropriately control or adjust the entropy of the latent distribution.

3.2.1 Continuous Attribute Edit with Fixed Linear Classifier

InterFaceGAN [41] showed that the latent random variable of GAN learns disentangled representation after linear transformation. InterFaceGAN edits data attributes continuously through the decision boundary of a linear classifier that predicts attributes of the latent vector.

However, InterFaceGAN can edit attributes only when the latent vector of the data point is known, and training a linear classifier requires a highly complex method. First, train the GAN with the dataset. Second, trains a classifier that classifies each attribute of the dataset. Third, label the attributes of the latent vector through the trained generator and classifier. Finally, train the linear classifier to predict attributes from the latent vectors.

In this section, we introduce AEDLSGAN [88] for integrating InterFaceGAN into the DLSGAN training stage without training a separate classifier or linear classifier. AEDLSGAN assumes that randomly initialized weights of the linear classifier are already ideal. Therefore, the only need to do is to train DLSGAN to fit the linear classifier.

$$c_f = argmax\ onehot((z \circ s)w + b) \tag{36}$$

Eq. 36 shows a fake categorical conditional vector c_f. AEDLSGAN predicts fake categorical conditional vector c_f from latent vector z with linear classifier weight w and bias b. Linear classifier weight w and bias b are $d_z \times d_c$ size and d_c size matrix, respectively, where d_c is the dimension of the attribute vector. AEDLSGAN assumes that linear classifier weights w and biases b are already ideal, so those values never change after initialization. $argmax\ onehot$ is a function that converts the continuous output value of the linear classifier into a discrete categorical vector. For example, $argmax\ onehot([0.2, 0.5, 0.3])$ is $[0.0, 1.0, 0.0]$. And, this predicted fake categorical conditional vector c_f is used to train class-conditional GAN with DLSGAN.

Algorithm 3 shows the algorithm to train AEDLSGAN.

Algorithm 3 Algorithm to train AEDLSGAN

Require: D^*, G, Z, X, v, w, b

▷ update D^*

$z \leftarrow sample(Z)$
$c_f \leftarrow argmax\ onehot((z \circ s)w + b)$
$x, c_r \leftarrow sample(X)$
$s \leftarrow \frac{\sqrt{d_z v}}{\|\sqrt{v}\|_2}$

$a_r, \_ \leftarrow D^*(x)$
$a_f, z' \leftarrow D^*(G(z \circ s))$
$L^d_{adv} \leftarrow A_d(a_r \cdot c_r, a_f \cdot c_f)$
$L_{enc} \leftarrow \frac{1}{d_z}\|(z - z') \circ s\|_2^2$

$L_d \leftarrow L^d_{adv} + \lambda_{enc} L_{enc}$
$D^* \leftarrow minimize(D^*, L_d)$

$v_1 \leftarrow z'^2$

▷ update G

$z \leftarrow sample(Z)$
$c_f \leftarrow argmax\ onehot((z \circ s)w + b)$

$a_f, z' \leftarrow D^*(G(z \circ s))$
$L^g_{adv} \leftarrow A_g(a_f \cdot c_f)$
$L_{enc} \leftarrow \frac{1}{d_z}\|(z - z') \circ s\|_2^2$

$L_g \Leftarrow L^g_{adv} + \lambda_{enc} L_{enc}$
$G \leftarrow minimize(G, L_g)$

▷ update v

$v_2 \leftarrow z'^2$
$v \leftarrow update(v, concat(v_1, v_2))$

In line 3 of algorithm 3, c_r represents a real categorical latent vector (label) of real data point x.

In lines 5 and 6, a_f and a_r represent an adversarial vector of generated data and real data, respectively. In algorithm 3, CAGAN [4] loss is used for class-conditional GAN loss. Therefore, a_f and a_r are vectors, not one-dimensional values. In lines 7 and 15, one can see that CAGAN loss was used for class-conditional GAN loss.

Unlike simply adding class conditional GAN to DLSGAN, AEDLSGAN can edit attributes continuously like InterFaceGAN.

3.3 Self-supervised Out-of-distribution Detection with Dynamic Latent Scale GAN

In section 2.3, we discussed the problems of previous OOD detection methods. DLSGAN [51] is a learning-based GAN inversion method [18] with maximum likelihood estimation of the encoder. It solves the problem that the generator loses information when training the InfoGAN [39] that predicts all dimensions of the latent vector through the dynamic latent scale. The encoder of DLSGAN maps input data to predicted latent vectors.

When the DLSGAN's encoder maps the input data to the predicted latent vector, we found that the probability that the predicted latent vector is sampled from latent distribution can be used for OOD detection. There are two features that allow the predicted latent vector to be used for OOD detection.

First is the latent entropy optimality. As DLSGAN training progresses, the entropy of the scaled latent random variable decreases, and the entropy of the scaled encoder output increases. When DLSGAN converges, the generator generates ID data with a scaled latent random variable, and the entropy of the scaled latent random variable and scaled encoder output becomes optimal entropy for expressing ID data with the generator and encoder. It means that ID data generated by the generator is densely mapped to latent vectors that are likely sampled from the latent distribution. Therefore, by the pigeonhole principle, OOD data

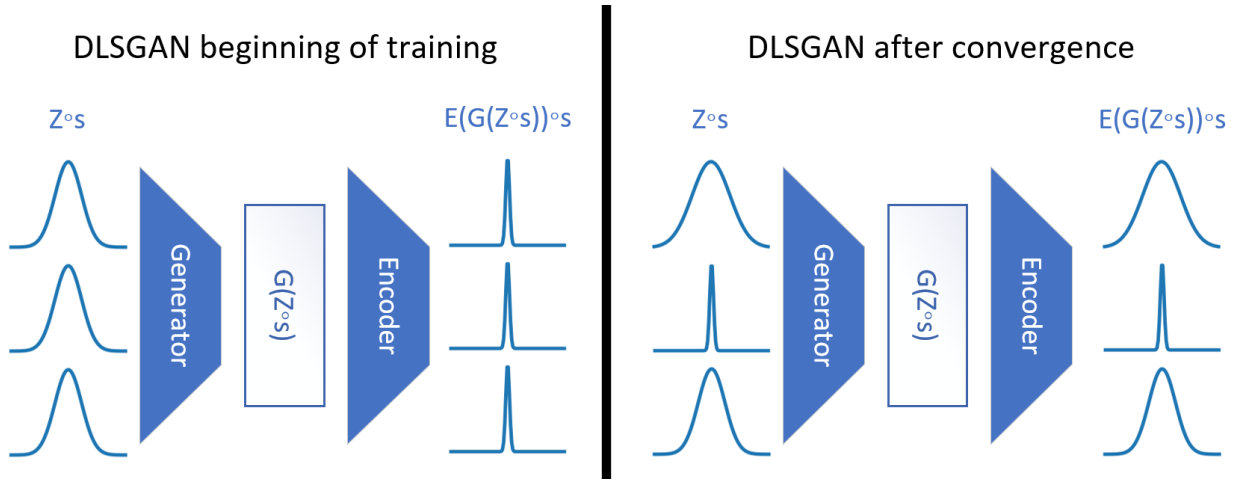

Figure 4: DLSGAN beginning of the training and after convergence.

can only be mapped to latent vectors that are unlikely sampled from the latent distribution.

Fig. 4 shows the training process of the DLSGAN. In Fig. 4, Z, G, s, and E represent a latent random variable, generator, latent scale vector, and encoder, respectively. "$\circ$" represents element-wise multiplication. One can see that the scaled latent random variable $Z \circ s$ has high entropy, and the scaled encoder output $E(G(Z \circ s)) \circ s$ has low entropy at the beginning of the training. As the training progresses, the entropy of the scaled latent random variable $Z \circ s$ decreases, and the entropy of scaled encoder output $E(G(Z \circ s)) \circ s$ increases. After DLSGAN converges, the scaled latent random variable $Z \circ s$ and scaled encoder output $E(G(Z \circ s)) \circ s$ have the same optimal entropy to represent ID data with the generator and the encoder.

Secondly, elements of the DLSGAN encoder output are independent of each other and follow a simple distribution (the same as the latent distribution). Therefore, it is very easy to calculate the log probability of the predicted latent vector.

The following equation shows the negative log probability of the predicted latent vector of input data.

$$OOD\ score = -\Sigma_{i=1}^{d_z} \log f(E_i(x)|\mu_i, v_i) \tag{37}$$

In Eq. 37, x and E represent the input data point and DLSGAN's encoder, respectively.

$E(x)$ represents the d_z-dimensional predicted latent vector of input data point x. f represents the probability density function of the latent random variable Z where each dimension is i.i.d.. μ and v represent the latent mean vector and traced latent variance vector for the probability density function f. μ is the mean vector of latent random variable Z. v is traced latent variance vector of DLSGAN (i.e., v is an approximation of element-wise variance of $E(G(Z))$). $E_i(x)$, μ_i, and v_i represent an i-th element of $E(x)$, μ, and v, respectively. For example, assume $\mu = [0, 0, 0]$ and $v = [1.0, 0.5, 0.1]$, and $f(x)$ is probability density function of normal distribution. When $E(x_a) = [0.1, -0.2, 0.3]$ and $E(x_b) = [-0.3, 0.2, -0.1]$, the OOD score of x_a is higher than x_b's.

One can see that the OOD score of input data represents the relative probability that the latent vector is sampled from the approximated encoder output distribution. Since each element of the encoder output $E(x)$ is independent of each other, the negative log probability of the predicted latent vector can be simply calculated by adding the negative log probability of each element. If the OOD score is greater than the threshold, the input data is classified as OOD data. Otherwise, it is classified as ID data.

AnoDLSGAN is a self-supervised OOD detection method because it directly estimates the probability that input data is sampled from ID. Therefore, AnoDLSGAN does not require any conditional distribution of ID data or a pre-trained model, and only one inference of encoder E is required for prediction.

3.4 Efficient Integration of Perceptual VAE into Dynamic Latent Scale GAN

In section 3.2, we introduced DLSGAN, a novel architecture-agnostic GAN inversion method. In this section, we introduce an architecture-agnostic generative model inversion method that efficiently integrates perceptual VAE [2] loss into DLSGAN to achieve better performance. When training DLSGAN, the latent encoder E_l of DLSGAN is trained to predict latent random variable Z from fake data random variable $G(Z \circ s)$. It is clear that $E_l(X) \circ s = Z \circ s$ if

generator G perfectly generates real data random variable X, and the latent encoder E_l perfectly inverts generator G. During DLSGAN training, if latent encoder E_l and discriminator D are integrated, it will be difficult to distinguish between real data random variable X and fake data random variable $G(Z \circ s)$ for latent encoder E_l. This is because latent encoder E_l shares hidden layers with discriminator D in adversarial training with generator G. Based on this intuition, we assumed that latent encoder E_l tries to map real data random variable X to the latent random variable Z during DLSGAN training, even without explicit loss. Under this assumption, VAE latent random variable Z_x that follows GAN latent random variable Z can be generated by adding scaled normal noise to the predicted real latent random variable $E_l(X)$.

When the latent random variable Z follows $N(0, I_{d_z})$, we assumed that predicted real latent random variable $E_l(X)$ follows $N(0, I_{d_z}) \circ \sqrt{v}$, where v is approximated latent variance vector in DLSGAN (Eq. 31). Therefore, the sum of scaled normal noise and real latent random variable $E_l(X)$ follows the normal random variable where each dimension is i.i.d..

$$Z_x = E_l(X) + N(0, I_{d_z}) \circ \sqrt{1 - v} \tag{38}$$

Eq. 38 shows VAE latent random variable Z_x. VAE latent random variable Z_x can be used for GAN training because it follows latent random variable Z. It is clear that $Z_x \sim Z \sim N(0, I_{d_z})$ because $E_l(X) \sim N(0, I_{d_z}) \circ \sqrt{v}$. If VAE latent random variable Z_x is different from GAN latent random variable Z, GAN training with VAE latent random variable Z_x is not only meaningless but rather makes GAN training more difficult. This is because generator G and discriminator D are trained with two different random variables Z_x and Z, but only Z should be used for generative performance evaluation.

VAE latent random variable Z_x is also used for VAE training since there is a corresponding real data random variable X. The following equation shows perceptual reconstruction loss for perceptual VAE training in PVDGAN.

$$L_{rec} = avg\left((E_f(G(Z_x \circ s)) - E_f(X))^2\right) \tag{39}$$

In Eq. 39, $G(Z_x \circ s)$ represents the reconstructed data random variable of real data random variable X. Like VAEGAN, the generator of PVDGAN is trained to minimize the perceptual reconstruction loss L_{rec} using the feature encoder E_f (See Eq. 20).

Finding a good distance metric for reconstruction loss is not an easy problem. For example, if image VAE is trained with pixel-wise mean squared error, the fake images will be very blurry. In general, perceptual reconstruction loss tends to generate better gradients than simple MSE reconstruction loss because humans recognize data perceptually. For example, if the brightness of the image changes slightly, the MSE distance will change significantly, but the perceptual distance for humans may change very little. Conversely, if the color of the pupils of the face picture changes, the perceptual distance for humans can change significantly, even though the MSE distance changes very little.

Using a pre-trained model (e.g., pre-trained inception model) is one of the most basic solutions for estimating perceptual reconstruction loss. It is known that the intermediate layer output of a deep neural network contains the perceptual features of the input [22]. However, using a separate pre-trained model requires additional computations for forward propagation & backpropagation of the pre-trained model to minimize the distance. Also, there might be no good pre-trained models for some data domains. Furthermore, it can be hard to customize a pre-trained model (e.g., the input resolution of the pre-trained model is fixed, or the model is too large or small).

Instead of using a pre-trained model, PVDGAN uses discriminator intermediate layer output as feature encoder L_f output like VAEGAN. Also, PVDGAN uses VAE latent random variable Z_x for GAN training too because it follows GAN latent random variable Z. During GAN training with VAE latent vector z_x, there are forward propagation & backpropagation on generator G and discriminator D with real data x and reconstructed data $G(z_x \circ s)$. Therefore, forward propagation & backpropagation for minimizing perceptual reconstruction

loss L_{rec} can be integrated into the forward propagation & backpropagation for the GAN training. It means that no additional forward propagation & backpropagation for minimizing reconstruction loss L_{rec} is required.

In short, when training DLSGAN with GAN latent random variable $Z \sim N(0, I_{d_z})$, and latent encoder E_l is integrated into discriminator D, VAE latent random variable $Z_x = E_l(X) + N(0, I_{d_z}) \circ \sqrt{1 - v}$ follows GAN latent random variable Z. Therefore, VAE latent random variable Z_x can be used for both GAN and VAE training. Also, there is already forward propagation & backpropagation with real data x and reconstructed data $G(z_x \circ s)$ in GAN training with VAE latent random variable Z_x. Therefore, VAE training (minimizing perceptual reconstruction loss L_{rec}) does not require additional forward propagation & backpropagation.

Algorithm 4 shows the algorithm to train PVDGAN. In algorithm 4, D^*, G, Z, and X represent the integrated discriminator, generator, latent random variable, and real data random variable, respectively. In algorithm 4, it was assumed that the latent random variable Z follows $N(0, I_{d_z})$. D^* is the integrated discriminator in which discriminator D, latent encoder E_l, and feature encoder E_f are integrated. Therefore, the integrated discriminator D^* has 3 outputs: 1-dimensional adversarial value output, d_z-dimensional latent encoder output, and d_f-dimensional feature encoder output. b and v represent the batch size and latent variance vector, respectively.

Lines 1-9 show the update process of integrated discriminator D^*. In lines 1 and 2, $sample(R, n)$ is a function that returns n samples from random variable R. x represents sampled real data points. In line 3, s is the d_z-dimensional latent scale vector of DLSGAN (Eq. 32).

In lines 4 and 5, the integrated discriminator D^* outputs 3 values. The first outputs a_r and a_f are $b \times 1$ shape adversarial values of real data and fake data, respectively. The second outputs z_r and z' are $b \times d_z$ shape predicted latent vectors. The third output y_r is $b \times d_f$ shape real feature vectors. Unlike the other two outputs, the feature vectors y_r are the

Algorithm 4 Algorithm to train PVDGAN

Require: D^*, G, Z, X, b, v

▷ update D^*

$z \leftarrow sample(Z, b)$
$x \leftarrow sample(X, b)$
$s \leftarrow \frac{\sqrt{d_z v}}{\|\sqrt{v}\|_2}$

$a_r, z_r, y_r \leftarrow D^*(x)$
$a_f, z', \_ \leftarrow D^*(G(z \circ s))$
$L^d_{adv} \leftarrow A_d(a_r, a_f)$
$L_{enc} \leftarrow avg\left(((z - z') \circ s)^2\right)$

$L_d \leftarrow L^d_{adv} + \lambda_{enc} L_{enc}$
$D^* \leftarrow minimize(D^*, L_d)$

$v_1 \leftarrow z'^2$

▷ update G

$z_x \leftarrow z_r\left[: b/2\right] + sample(Z, b/2) \circ \sqrt{\max(1 - v, 0)}$
$z \leftarrow concat(z_x, sample(Z, b/2))$
$a_f, z', y'_r \leftarrow D^*(G(z \circ s))$
$L^g_{adv} \leftarrow A_g(a_f)$
$L_{enc} \leftarrow avg\left(((z\left[b/2 :\right] - z'\left[b/2 :\right]) \circ s)^2\right)$
$L_{rec} \leftarrow avg\left((y_r\left[: b/2\right] - y'_r\left[: b/2\right])^2\right)$

$L_g \leftarrow L^g_{adv} + \lambda_{enc} L_{enc} + \lambda_{rec} L_{rec}$
$G \leftarrow minimize(G, L_g)$

$v_2 \leftarrow (z'\left[b/2 :\right])^2$ ▷ update v
$v \leftarrow update(v, concat(v_1, v_2))$

intermediate layer outputs of the integrated discriminator D^*. "_" represents unused values. In line 6, A_d represents the adversarial function for the discriminator. Several adversarial loss functions [6] can be used for A_d. Encoder loss L_{enc} of DLSGAN (Eq. 33) is calculated in line 7. Line 8 shows discriminator loss of PVDGAN. In line 9, integrated discriminator D^* is updated with discriminator loss L_d.

Lines 11-18 show the update process of PVDGAN's generator G. In line 11, z_x represents VAE latent vectors. $z_r\,[:b/2]$ represents first $b/2$ samples of z_r. Therefore, z_x is $\frac{b}{2} \times d_z$ shape matrix. In general cases, all elements of latent variance vector v are less than or equal to 1, but for stability, we recommend using $\sqrt{\max(1-v, 0)}$ instead of $\sqrt{1-v}$. In line 12, *concat* represents the concatenation function. Therefore, z is $b \times d_z$ shape matrix, where the first $b/2$ samples are VAE latent vectors z_x, and the last $b/2$ samples are sampled from latent random variable Z. In line 13, y'_r represents predicted feature vectors. In line 14, A_g represents the adversarial function for the generator. In line 15, latent vectors directly sampled from Z are used for encoder loss. In line 16, feature vectors of real data and paired fake data are used for perceptual reconstruction loss. In line 18, generator G is updated with generator loss L_g. In line 20, the latent variance vector v is updated with v_1 from line 10 and v_2 from line 19.

In algorithm 4, there is no additional forward propagation & backpropagation to minimize L_{rec}. In lines 4 and 13, there is forward propagation of the integrated discriminator D^* for a_r, a_f, z_r, and z'. Since y_r and y'_r are intermediate layer outputs of the integrated discriminator D^*, forward propagation for y_r and y'_r is concurrent with forward propagation for a_r, a_f, z_r, and z'. Similarly, the backpropagation for L_{rec} on line 16 is incorporated into the backpropagation for losses L^g_{adv} and L_{enc}, which is starting from the output of the integrated discriminator D^*. Therefore, PVDGAN does not require additional computation compared to basic GAN or DLSGAN (to be precise, there are some additional d_f-dimensional vector operations to calculate L_{rec}, but it is very small compared to operations for forward propagation & backpropagation in most large deep learning models).

The overall algorithm of PVDGAN is similar to DLSGAN's. The difference between

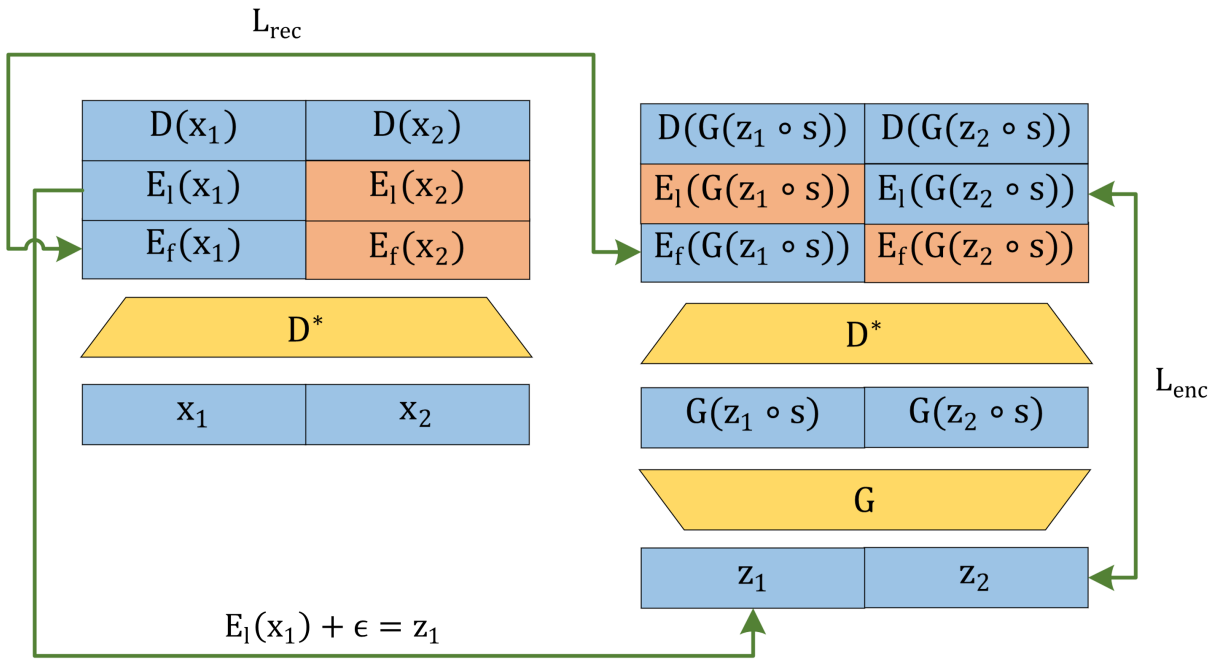

Figure 5: Visualization of reconstruction loss L_{rec} and encoder loss L_{enc} of PVDGAN with two data (x_1 and x_2) and two latent vectors (z_1 and z_2). z_1 is a VAE latent vector for both VAE and GAN training. Red blocks represent values not used in PVDGAN training.

DLSGAN and PVDGAN is that PVDGAN uses VAE latent vector z_x for training, and the generator of PVDGAN is trained to minimize the perceptual reconstruction loss L_{rec}.

The losses of PVDGAN are summarized as follows.

$$L_d = L_{adv}^d + \lambda_{enc} L_{enc} \tag{40}$$

$$L_g = L_{adv}^g + \lambda_{enc} L_{enc} + \lambda_{rec} L_{rec} \tag{41}$$

Fig. 5 visualize reconstruction loss L_{rec} and encoder loss L_{enc} of PVDGAN with two data (x_1 and x_2) and two latent vectors (z_1 and z_2). In Fig. 5, ϵ represents $sample(Z, b/2) \circ \sqrt{1-v}$.

Unlike VAEGAN [68] or VAE [2], PVDGAN does not use VAE prior loss and encoder variance estimation. Since training encoder with encoder loss L_{enc} is a maximum likelihood estimation, it can replace VAE prior loss. For the same reason, latent variance vector v replaces encoder variance estimation. Also, VAEGAN uses 3 models (encoder, decoder,

discriminator) for training, while PVDGAN uses only 2 models. Using only 2 models in PVDGAN is computationally efficient and expected to produce a better perceptual loss. This is because the discriminator does not need all the information of input data to discriminate input data, so the discriminator of VAEGAN may lose the information of input data. Assume that the penultimate layer of the discriminator has only one unit (i.e., output dimension is 1), and using this penultimate layer output as feature encoder output in VAEGAN (i.e., $D(x) = E_f(x) \cdot w + b$, where w and b are both 1-dimensional trainable weight). In such a case, perceptual loss with this feature encoder of VAEGAN is almost meaningless since the feature encoder output is only one-dimensional. On the other hand, the dimension of the PVDGAN feature encoder output is at least $d_z + 1$. This is because PVDGAN's discriminator has a one-dimensional adversarial value output and d_z-dimensional latent encoder output. Also, the latent encoder E_l is trained not to lose the information of input fake data. Therefore, it is expected that PVDGAN's feature encoder will produce a better perceptual reconstruction loss since it contains more meaningful information than VAEGAN's.

PVDGAN can be utilized for most generative model inversion applications since it is an architecture-agnostic method. Simply, the encoder of PVDGAN can be used for embedding layers of most models. For example, InterFaceGAN [41] showed attributes of generated data can be linearly separable in the latent space. Adding a linear classifier to PVDGAN's encoder can make a deep classifier. Also, it is possible to perform continuous attribute edits of input data using the linear classifier and PVDGAN. PVDGAN can also perform out-of-distribution detection using AnoDLSGAN [52]. InterFaceGAN and AnoDLSGAN assume that the generative model inversion method transforms data distribution into latent distribution. Therefore, state-of-the-art StyleGAN-specific inversion methods [35, 36, 37] cannot be applied for these applications since StyleGAN-specific methods only can transform input images into style vectors, which are intermediate layer outputs.

Also, PVDGAN can be applied for other out-of-distribution detection methods [45, 46], data augmentation for better model performance [47], semantic segmentation [48], and other

applications using generative model inversion [18]. Furthermore, since the PVDGAN is basically an independent component analysis and representation (feature) learning method, it can be utilized for other applications using independent component analysis or representation learning.

3.5 Training Self-supervised Class-conditional GAN with Classifier Gradient Penalty and Dynamic Prior

In section 2.1.2, we discussed problems of previous unsupervised class-conditional data generation methods. In this section, we introduce Classifier Gradient Penalty GAN (CGPGAN) which can perform class-conditional data generation and clustering under more general conditions than previous works. CGPGAN can be used under the following very general conditions:

1. The labels of all data are unknown.
2. Optimal categorical latent distribution is unknown.
3. Metric to measure the distance between the data is unknown.

InfoGAN cannot be used under condition 2 because it requires an optimal categorical latent distribution and Elastic InfoGAN cannot be used under condition 3 because it requires a metric for identity-preserving transformation.

A CGPGAN consists of a discriminator D, classifier Q, and (class-conditional) generator G. The generator G takes d_z-dimensional continuous latent vector and d_c-dimensional categorical latent vector as inputs to generate class-conditional data. The classifier Q is trained to predict the label of the generated data, and the label of the real data predicted by the classifier is used for adversarial training of the discriminator and generator for class-conditional data generation. The generator G and discriminator D are trained with class-conditional GAN loss to generate class-conditional data. Instead of ACGAN loss [3], CGPGAN uses CAGAN loss [4], [7] for better generative performance.

The following equations show the losses for CGPGAN.

$$L_q = \lambda_{cls} L_{cls} + \lambda_{cgp} L_{cgp} \tag{42}$$

$$L_d = L_{adv}^d \tag{43}$$

$$L_g = L_{adv}^g \tag{44}$$

$$L_{cgp} = \mathbb{E}_{z,c_f} \left[\| \nabla_{G(z,c_f)} \left((1 - Q(G(z,c_f)) \cdot c_f)^2 \right) \|_2^2 \right] \tag{45}$$

$$L_{adv}^d = \mathbb{E}_{x,z,c_f} \left[A_d(D(x) \cdot argmax\ onehot(Q(x)), D(G(z,c_f)) \cdot c_f) \right] \tag{46}$$

$$L_{adv}^g = \mathbb{E}_{z,c_f} \left[A_g(D(G(z,c_f)) \cdot c_f) \right] \tag{47}$$

In Eq. 42, λ_{cgp} and L_{cgp} represent classifier gradient penalty loss weight and classifier gradient penalty loss, respectively. Classification loss L_{cls} is cross-entropy loss, which is the same as InfoGAN's classification loss (Eq. 12). In Eq. 44, one can see that there is no classification loss L_{cls} in generator loss L_g. Since CGPGAN's generator is trained with adversarial losses only, there is no conflict between L_{cls} and L_{adv}^g as in InfoGAN.

Eqs. 46 and 47 show CAGAN adversarial losses for CGPGAN. Since the true label c_f of the fake data $G(z, c_f)$ is known, the adversarial loss for fake data in CGPGAN is the same as the CAGAN loss. However, the label of the real data x is unknown. Instead, in CGPGAN, $argmax\ onehot(Q(x))$ is used as the label of the real data x. The $argmax\ onehot$ function replaces the maximum value of the vector with 1 and all other values with 0 (e.g., $argmax\ onehot([0.2, 0.5, 0.3]) = [0.0, 1.0, 0.0]$).

The classification loss L_{cls} can be minimized by simply classifying the generated data well, but it can also be minimized by increasing the slope of the decision boundary or by moving the decision boundary near the generated data with lower density. Therefore, when training the classifier with cross-entropy loss, we assumed that the decision boundary will naturally move to the local optima which minimizes the density of the generated data, and the slope

of the decision boundary will increase. If the slope of the decision boundary is very large and the probability density function of the generated data is lumpy, the decision boundary will converge to a local optima in a very narrow region that minimizes the density of the data.

To avoid classifier decision boundary converging local optima in a narrow region that minimizes the probability density of generated data $P(G(X, C))$, CGPGAN uses a classifier gradient penalty loss L_{cgp} in Eq. 45. By relaxing the slope of the classifier's decision boundary, the decision boundary can converge to a local optima in a wider region. One can see that the classifier gradient penalty loss L_{cgp} becomes higher when the generated data point $G(z, c_f)$ is near the decision boundary in Eq. 45. In the classifier, the larger the classifier gradient penalty loss L_{cgp}, the more the decision boundary converges to the local optima in a larger region. Therefore, CGPGAN can adjust the sensitivity of each category (cluster) through the weight of the classifier gradient penalty loss λ_{cgp}.

Fig. 6 shows an example of a classifier decision boundary moving to the local optima that minimizes classification loss L_{cls}. In Fig. 6, $P(x)$ represents the probability density function of generated data distribution. $Q(x)$ represents predicted probability of the classifier Q. To minimize classification loss L_{cls} (Eq. 12), $P(x)$ near the decision boundary should be minimized, and the slope of the classifier output near the decision boundary should be maximized. Therefore, we assumed that the decision boundary of the classifier will converge to a local optima near 2 that minimizes $P(x)$, and the classifier output slope near the decision boundary will be maximized. If classifier gradient penalty L_{cgp} is added to the classifier loss, the classifier output slope near the decision boundary softens, so the decision boundary can converge to a wider range of local optima that minimize $P(x)$ (near 1.5).

Additionally, CGPGAN updates the probability of the categorical latent distribution $P(C)$ during the training with $\mathbb{E}_x[Q(x)]$ (i.e., $P(C) \approx \mathbb{E}_x[Q(x)]$). Through this approximation, CGPGAN can approximate $P(C)$ without knowing the optimal prior probability. However, updating $P(C)$ early in the training can make CGPGAN converge to a trivial solution (i.e., one category has a probability of 1 and the other has a probability of 0). To avoid

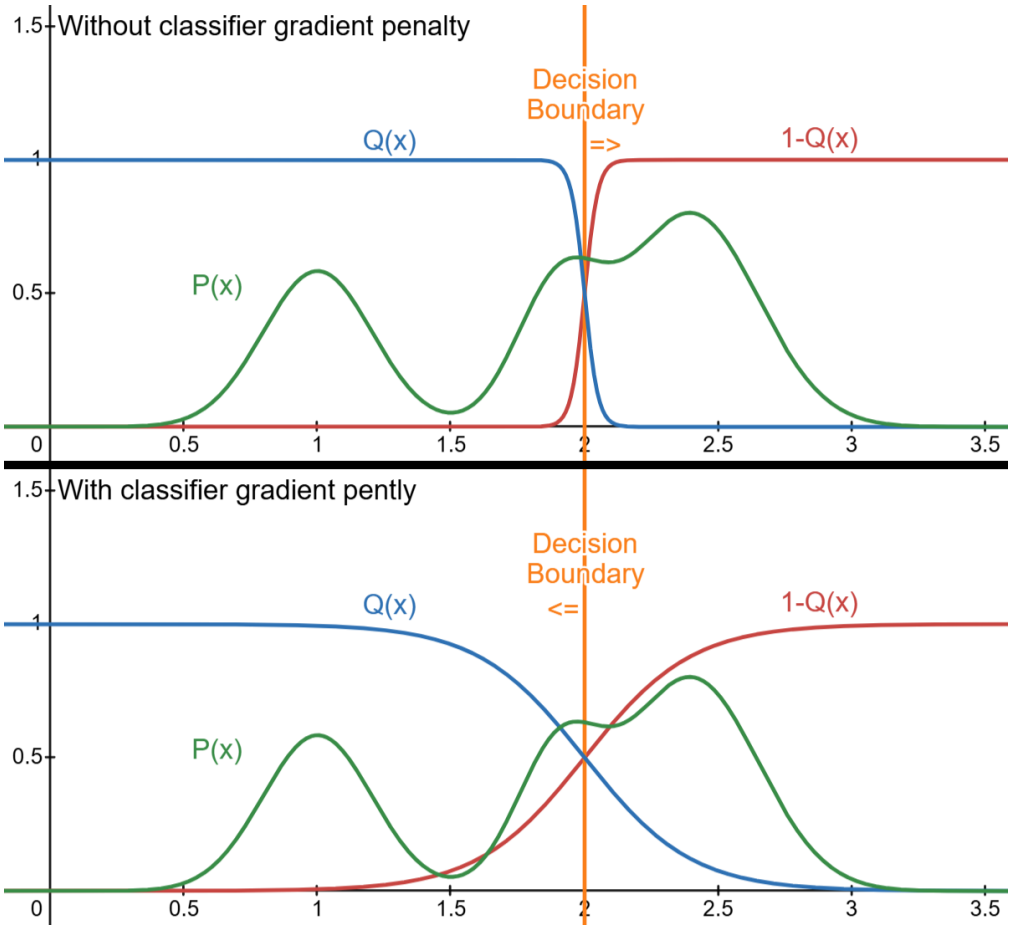

Figure 6: Example of a classifier's decision boundary moving. The top plot shows the classifier decision boundary moving to the right side when the classifier gradient penalty was not applied. The bottom plot shows the classifier decision boundary moving to the left side when the classifier gradient penalty was applied. In the top plot, the decision boundary of classifier Q moves to the right toward $x = 2.1$ to minimize the classification loss. On the other hand, in the bottom plot, the decision boundary moves to the left toward $x = 1.5$ to minimize the classification loss.

converging a trivial solution and ensure that the ratio of real to fake data in each category is similar, CGPGAN normalizes $Q(x)$ by the batch distribution only at the beginning of training.

Algorithm 5 shows the training step of CGPGAN.

The training step of CGPGAN requires X (data random variable), Z (continuous latent random variable), C (categorical latent random variable), D (discriminator), Q (classifier), and G (generator).

In lines 1-3, the *sample* function represents the sampling function from a random variable. x (real data point), z (continuous latent vector), and c_f (fake categorical latent vector) are sampled from X, Z, and C, respectively.

In line 4, G generates fake data x' with z and c_f. In lines 5 and 6, D and Q takes a fake data point x' as input and outputs a_f (fake adversarial vector) and c'_f (fake categorical latent vector prediction), respectively. Similarly, in lines 7-8, D and Q take a real data point x as input and outputs a_r (real adversarial vector) and c'_r (real categorical latent vector prediction).

In lines 9-11, the real categorical latent vector prediction c'_r is normalized for stable training only at the beginning of the training. In line 10, the *prob normalize* function forces the real categorical latent vector c_r to approach a uniform distribution. This ensures that the ratio of real data to fake data in each category is similar, allowing for stable training in the early stages of CGPGAN training.

$$prob\ normalize(\mathbf{c}) = \mathbf{c} - batch\ average(\mathbf{c}) + \frac{1}{d_c} \tag{48}$$

Eq. 48 shows the function to normalize the categorical latent vectors $\mathbf{c}$, where $\mathbf{c}$ is a $b \times d_c$ matrix, and b represents the batch size. *batch average* is a function that computes the element-wise average vector of $\mathbf{c}$. Therefore, $batch\ average(\mathbf{c})$ is d_c-dimensional vector. One can see that

$batch\ average(prob\ normalize(\mathbf{c}))$ is always $[\frac{1}{d_c}, \frac{1}{d_c}, \ldots, \frac{1}{d_c}]$. However, the *prob normalize*

Algorithm 5 Algorithm to train CGPGAN

Require: X, Z, C, D, Q, G

▷ update D and Q

$x \leftarrow sample(X)$
$z \leftarrow sample(Z)$
$c_f \leftarrow sample(C)$
$x' \leftarrow G(z, c_f)$

$a_r \leftarrow D(x)$
$c'_r \leftarrow Q(x)$
$a_f \leftarrow D(x')$
$c'_f \leftarrow Q(x')$

if *early in training* **then**
    $c'_r = prob\ normalize(c'_r)$
end if
$c_r \leftarrow argmax\ onehot(c'_r)$

$L_{cls} \leftarrow -c_f \cdot \log(c'_f)$
$L_{cgp} \leftarrow \|gradient((1 - c'_f \cdot c_f)^2, x')\|_2^2$

$L_d \leftarrow A_d(a_r \cdot c_r, a_f \cdot c_f)$
$L_q \leftarrow \lambda_{cls} L_{cls} + \lambda_{cgp} L_{cgp}$

$D \leftarrow minimize(D, L_d)$
$Q \leftarrow minimize(Q, L_q)$

▷ update G

$z \leftarrow sample(Z)$
$c_f \leftarrow sample(C)$
$x' \leftarrow G(z, c_f)$
$a_f \leftarrow D(x')$

$L_g \leftarrow A_g(a_f \cdot c_f)$

$G \leftarrow minimize(G, L_g)$

$P(C) \leftarrow update(P(C), c'_r)$ ▷ update $P(C)$

function restricts the representation of the real categorical latent vector c_r, so it is disabled after some training. In line 12, the real categorical latent vector c_r is calculated from c'_r.

In line 13, L_{cls} represents classification loss for c_f prediction. L_{cls} is categorical cross-entropy loss. In line 14, $gradient(y, x)$ function calculates gradient dy/dx.

In lines 15 and 16, L_d and L_q represent discriminator loss and classifier loss, respectively. A_d represents GAN adversarial loss functions for the discriminator. When training GAN with R1 or R2 regularization [7], we recommend using R2 regularization because the true label of generated data is known, unlike real data.

In lines 17 and 18, discriminator D and classifier Q are updated to minimize its loss. In lines 19-24, generator G is updated to minimize its loss.

In line 25, $P(C)$ is updated with predicted real categorical latent vector c'_r. The *update* function can be a simple moving average, an exponential moving average, or other scheduling functions. In CGPGAN, $P(C)$ is initialized with uniform distribution ($P(C) = [\frac{1}{d_c}, \frac{1}{d_c}, ..., \frac{1}{d_c}]$), and c'_r is normalized at the beginning of the training (line 10 in algorithm 5). Thus, at the beginning of training, $P(C)$ will always be $[\frac{1}{d_c}, \frac{1}{d_c}, ..., \frac{1}{d_c}]$. This makes $P(C)$ to not converge to a trivial solution.

Fig. 7 shows overall training process of CGPGAN.

3.5.1 Training Classifier Gradient Penalty GAN with Codebook Architecture

In sections 1.4.5 and 2.4, we discussed models that use discrete latent random variables. In this section, we propose an architecture that uses codebooks to improve the categorical generation performance of CGPGAN and generate high-quality data. The proposed architecture simply replaces the categorical latent vector c_f of CGPGAN with the trainable page vector of the codebook corresponding to the index of the categorical latent vector, instead of directly inputting the categorical vector c_f to the generator. For example, if $c_f = [0.0, 1.0, 0.0]$, the page vector of the 2nd index of the codebook would be input to the generator instead of the categorical latent vector c_f. The codebook is updated with generator losses during train-

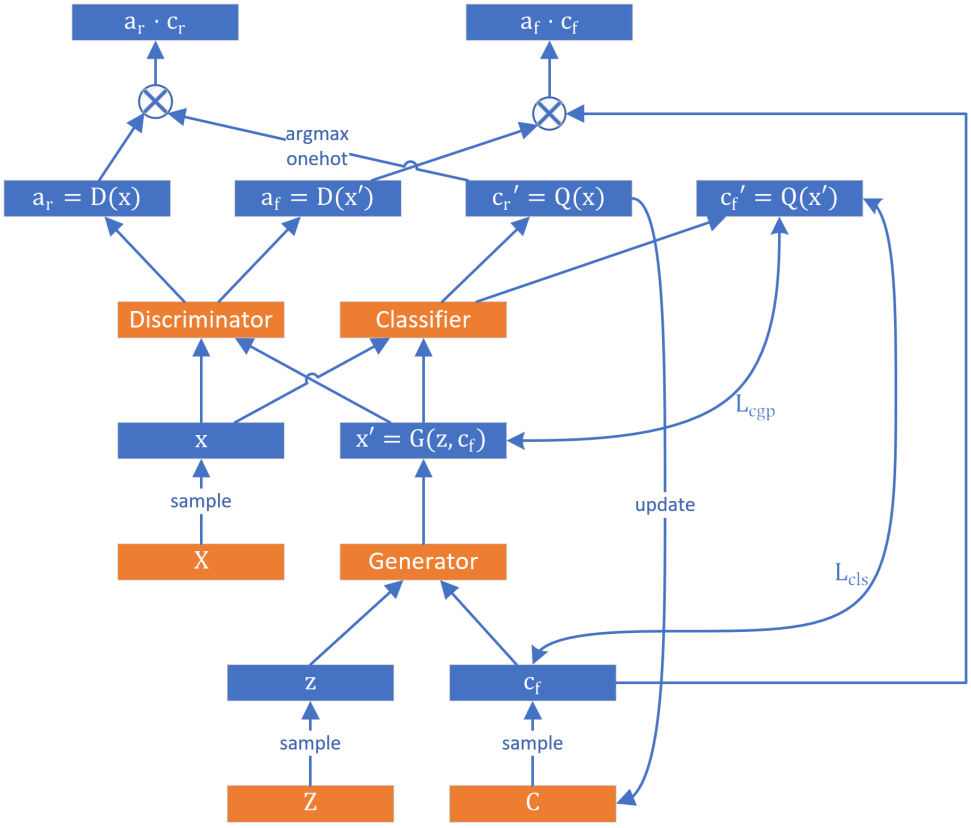

Figure 7: Flowchart showing the training process of CGPGAN. Values $a_r \cdot c_r$ and $a_f \cdot c_f$ are used for adversarial training. "$\otimes$" represents the inner product.

ing, just like other trainable parameters of the generator. Therefore, one can think that the codebook is the trainable parameter of the generator. The proposed codebook architecture enforces CGPGAN's discrete representation and quality of generated data because it is a kind of memory architecture, like VQGAN [83] or VQVAE [84] using vector quantization. The difference between the proposed codebook architecture and other vector quantization generative models is that the proposed codebook architecture does not encode the train data through an encoder. Therefore, no additional autoencoder training is required. We found this simple method improves CGPGAN's generated data quality (precision) and class-conditional data generation & clustering performance, like other vector quantization generative models.

4 Experiments

In this section, we present experiments to validate our contributions. In general, the experiments were conducted using RTX 3090 and RTX 4090 GPUs.

4.1 Conditional Activation GAN and Mixed Batch Training

4.1.1 Conditional Activation GAN

In this experiment, we trained revised ACGAN and CAGAN to generate Gaussian clusters and MNIST handwritten digits dataset [61], and compared the generative performance of each method. The left plot of Fig. 1 shows samples from the Gaussian cluster dataset. The following hyperparameters were used for both Gaussian clusters and MNIST experiments.

$$
\begin{gathered}
Z \sim N\left(0, I_{256}\right) \\
\lambda_r = 1.0 \\
optimizer = AdamW \begin{pmatrix} learning\ rate = 0.001 \\ \beta_1 = 0.0 \\ \beta_2 = 0.99 \end{pmatrix} \\
trainable\ weights\ ema\ decay\ rate = 0.999
\end{gathered}
$$

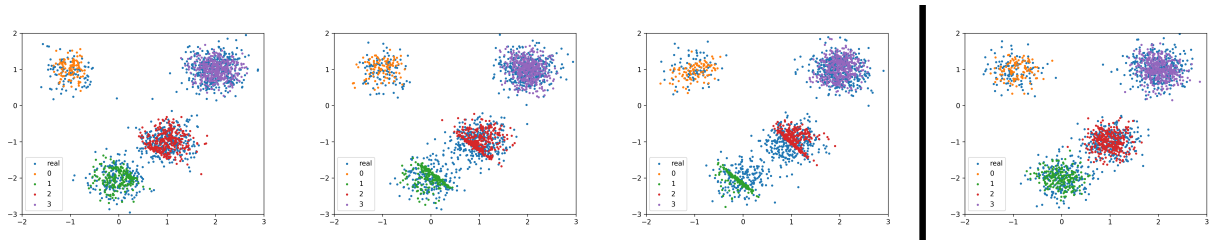

Figure 8: Generated samples in Gaussian clusters experiments. First plot: samples generated by revised ACGAN with $\lambda_{cls} = 0.01$. Second plot: samples generated by revised ACGAN with $\lambda_{cls} = 1.0$. Third plot: samples generated by revised ACGAN with $\lambda_{cls} = 100.0$. Fourth plot: samples generated by CAGAN.

$$batch\ size = 32$$

$$train\ step\ per\ epoch = 1000$$

$$epochs = 200$$

In Adam [92] and AdamW [93] optimizers, β_1 and β_2 control the exponential moving averages of the first-order (mean) and second-order (variance) moments of the gradients, respectively. A higher β_1 makes the optimizer rely more on past gradients, while β_2 stabilizes the learning by controlling the update scale based on gradient variance.

NSGAN loss [1] with R1+R2 [7] regularization was used for adversarial loss. Since real data and fake data are mixed in each batch in the subsequent mixed batch training experiment, we used R1+R2 regularization for the efficient experiment. λ_r represents R1+R2 regularization loss weight ($\lambda_r L_r = \lambda_r(\mathbb{E}_x\left[\|\nabla_x D(x)\|_2^2\right] + \mathbb{E}_{x'}\left[\|\nabla_{x'} D(x')\|_2^2\right])$).

The leaky ReLU function was used for the activation function. Equalized learning rate [8] was used for all trainable weights. For the Gaussian clusters experiments, the model consists of fully connected layers with 512 units. Full codes for the experiments are available in https://github.com/jeongik-jo/CAGAN.

Fig. 8 shows samples generated by revised ACGAN and CAGAN. First, in the first three plots, revised ACGAN does not generate samples near the classifier decision boundary between label 1 and label 2. This is because the generator tries to avoid generating samples near the decision boundary to minimize classification loss. For the same reason, the generated

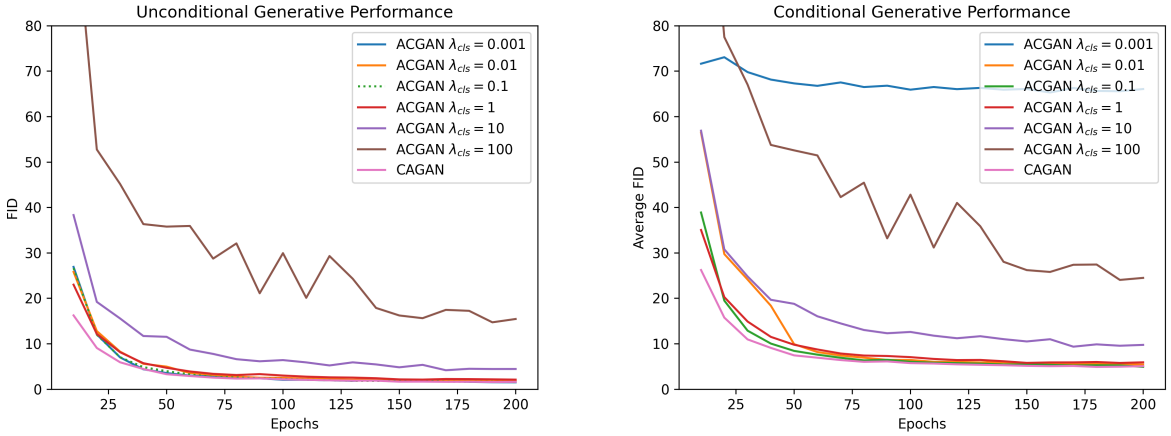

Figure 9: Performance of revised ACGAN and CAGAN for each epoch. In the first plot, ACGAN with $\lambda_{cls} = 0.1$ is shown as a dashed line for visibility. Left plot: FID between training data and generated data. Right plot: Average of FIDs for each class.

sample moves further away from the decision boundary as the classification loss weight λ_{cls} increases. On the other hand, there was a natural division between label 1 and label 2 of CAGAN in the fourth plot. This is because CAGAN's generator is trained with adversarial loss only, so there is no conflict between adversarial loss and classification loss.

In MNIST experiments, we used simple model architecture consisting of CNN layers. We used fréchet inception distance (FID) to evaluate generative performance of the model. $32,000$ train images and generated images were used for FID [9] evaluation. Models were evaluated every 10 epoch.

Fig. 9 shows the performance of revised ACGAN and CAGAN for each epoch. The first plot in Fig. 9 shows the unconditional generative performance of the model, and the second plot shows the conditional generative performance of the model. FID between real data and fake data, regardless of class, was used for unconditional performance evaluation. And average FID for each class was used for conditional performance evaluation. In Fig. 9, one can see that the performance of revised ACGAN depends on the hyperparameter λ_{cls}. When λ_{cls} is too large ($\lambda_{cls} = 100$), the model's unconditional performance and conditional generative performance were low because of the high classification loss. When λ_{cls} is too low

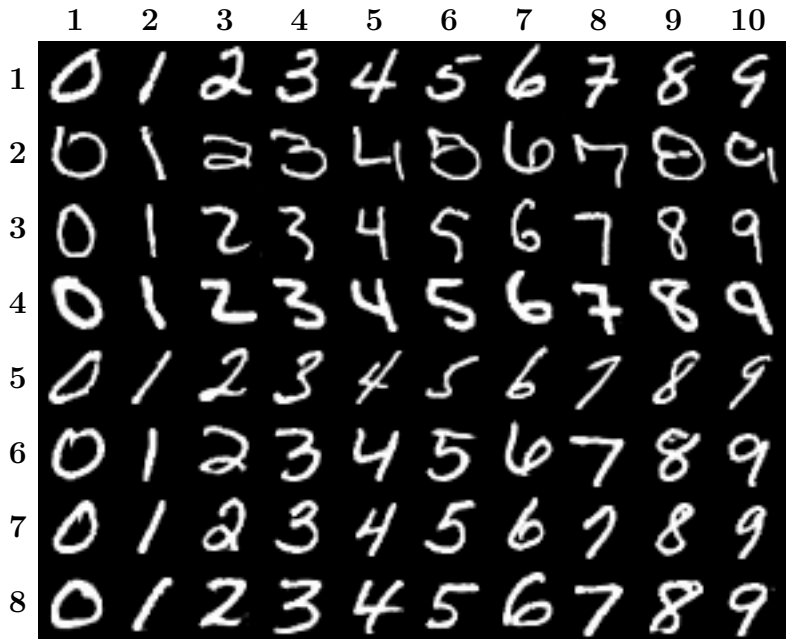

Figure 10: Generated samples of CAGAN in MNIST experiments. Each row shares the same latent vector, and each column shares the same conditional vector. Row and column indices are added for better readability.

($\lambda_{cls} = 0.001$), revised ACGAN could generate unconditional data well, but failed to generate class-conditional data.

Fig. 10 shows generated samples of CAGAN in MNIST experiments, complementing the quantitative results shown in Fig. 9.

4.1.2 Mixed Batch Training

In mixed batch training experiments, we changed several settings from 4.1.1. We used batch standard deviation layer [8] as batch-wise operation for mixed batch training experiments. We inserted the batch standard deviation layer into the penultimate layer of the hidden layers. We also modified the conditional distribution of the real data to make the conditional latent distribution different from the conditional distribution of the real data. $mix\ rate\ per\ epoch = 0.005$ was used for mixed batch training. It means that $real\ data : fake\ data$ gradually changes to the target ratio by $0.5\%p$ for epochs $1 \sim 100$. Then, $real\ data : generated\ data = 50 : 50$ is used for epochs $101 \sim 200$.

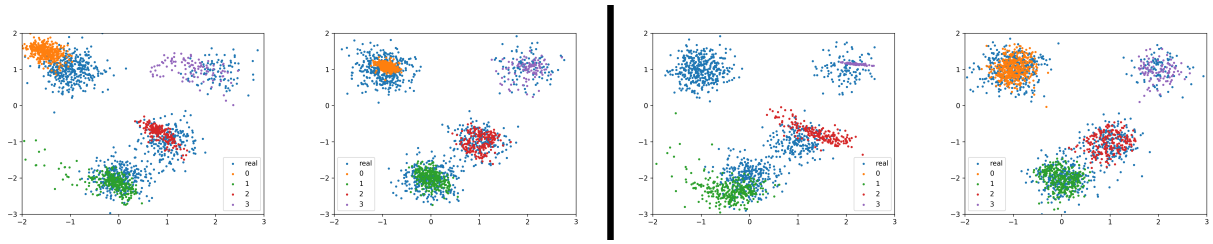

Figure 11: Generated samples with real conditional distribution ($P(C_r) = [0.4, 0.3, 0.2, 0.1]$) in Gaussian clusters experiments. First plot: ACGAN without mixed batch training. Second plot: ACGAN with mixed batch training. Third plot: CAGAN without mixed batch training. Fourth plot: CAGAN with mixed batch training.

In Gaussian clusters experiments, we changed real categorical distribution to $[0.4, 0.3, 0.2, 0.1]$. Therefore, categorical latent distribution of GAN is $P(C_f) = [0.1, 0.2, 0.3, 0.4]$, and real categorical distribution is $P(C_r) = [0.4, 0.3, 0.2, 0.1]$. Other settings are the same as section 4.1.1.

Fig. 11 shows the data generated by ACGAN and CAGAN trained on the conditional latent distribution $P(C_f)$. One can see that both revised ACGAN and CAGAN show an improvement in generative performance when mixed batch training is applied. In particular, both ACGAN and CAGAN generated centroids for each cluster well when mixed batch training was applied.

In MNIST experiments, we changed the real categorical distribution to $[0.55, 0.05, 0.05, \ldots, 0.05]$. Therefore, categorical latent distribution is $P(C_f) = [0.1, 0.1, \ldots, 0.1]$, and real categorical distribution is $P(C_r) = [0.55, 0.05, 0.05, \ldots, 0.05]$.

Figs. 12-15 show generated samples of revised ACGAN and CAGAN without and with mixed batch training. One can see that both revised ACGAN and CAGAN ignore the input condition vector and generate the digit 0 when mixed batch training is not applied. Since there are many digits 0 in real data, and there are batch-wise operations in the discriminator, ignoring input conditions and generating more digits 0 can minimize adversarial loss for the generator. Because of this, the generator ignores the input condition vector and generates digit 0. On the other hand, when mixed batch training is applied, one can see that both

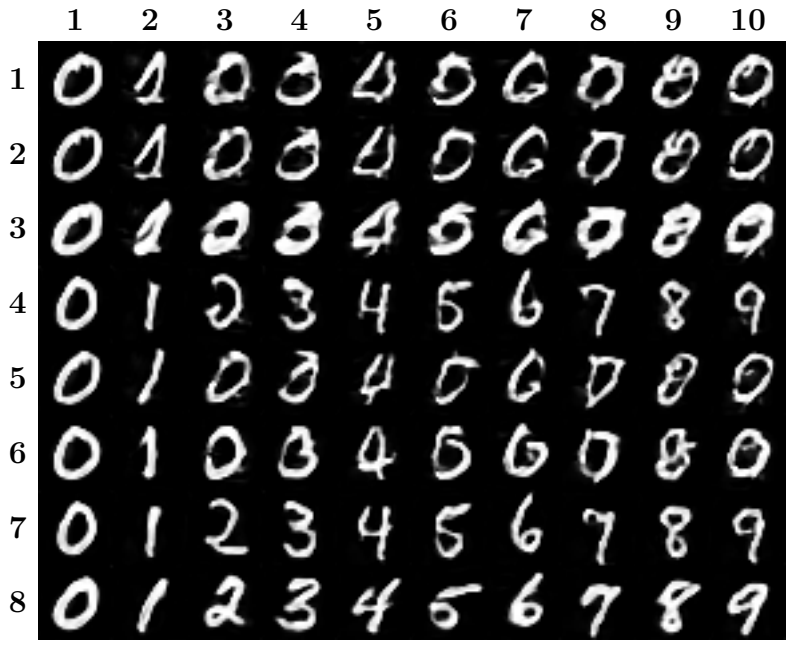

Figure 12: Generated samples of ACGAN without mixed batch training.

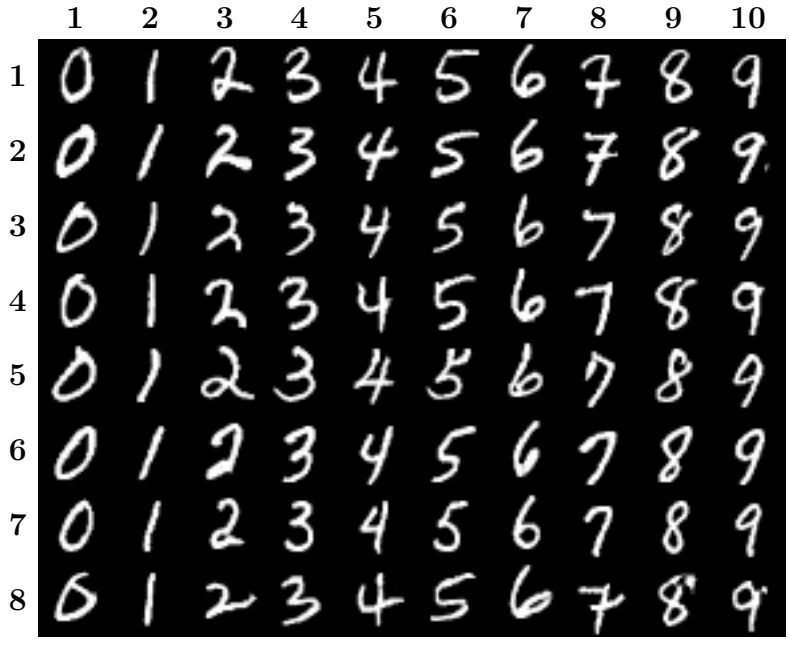

Figure 13: Generated samples of ACGAN with mixed batch training.

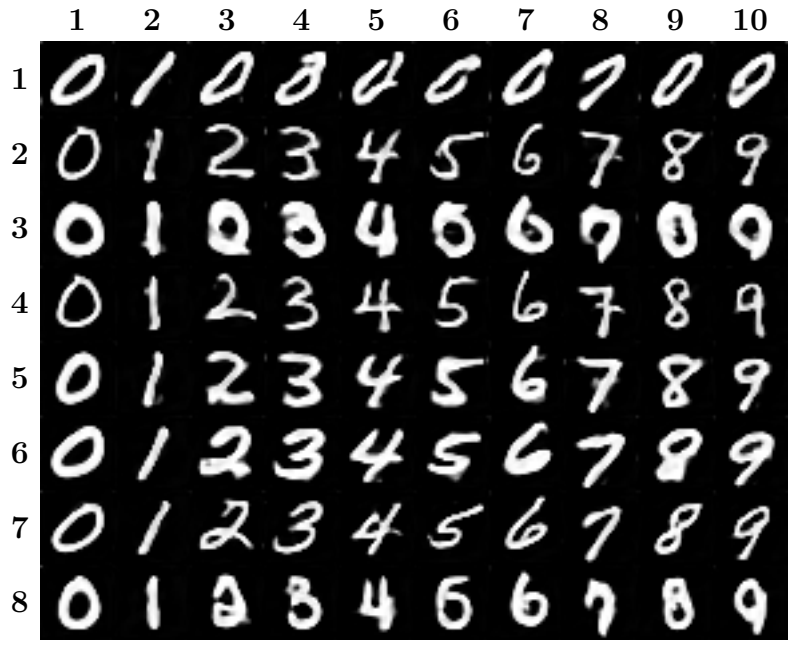

Figure 14: Generated samples of CAGAN without mixed batch training.

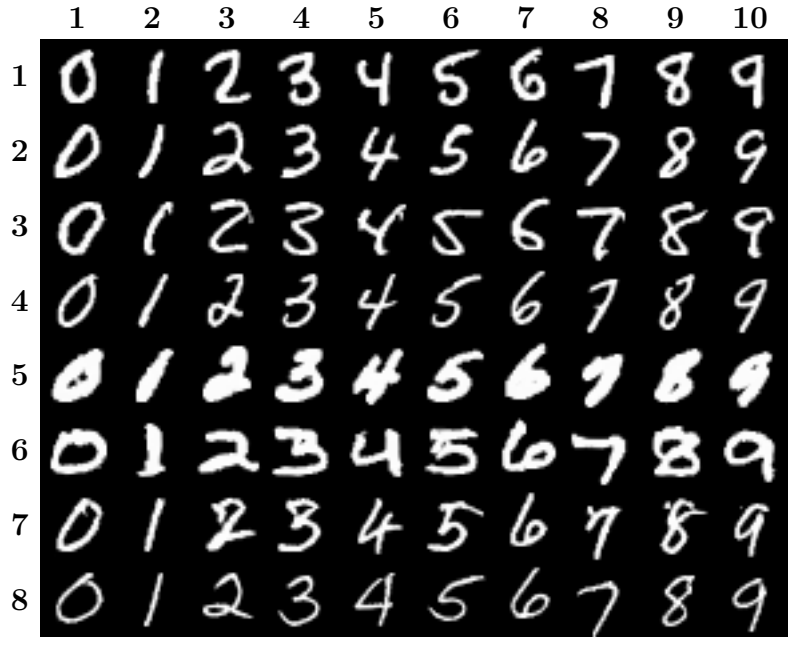

Figure 15: Generated samples of CAGAN with mixed batch training.

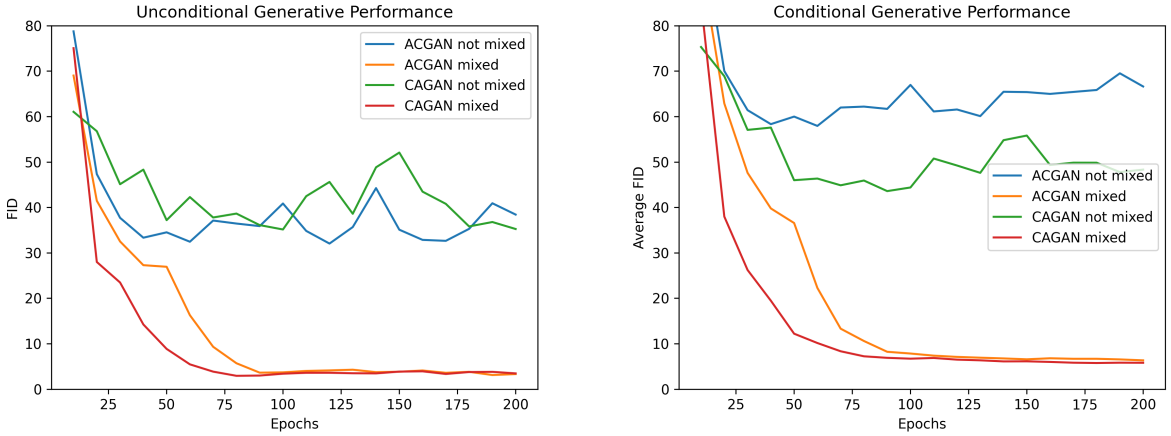

Figure 16: Mixed batch training comparison graph.

revised ACGAN and CAGAN correctly generate class-conditional data.

Fig. 16 shows the performance of revised ACGAN and CAGAN with and without mixed batch training for each epoch in MNIST experiments. One can see that mixed batch training clearly improves the generative performance of models when real conditional distribution and fake conditional distribution are different.

4.1.3 Summary

In the first contribution, we introduced a novel class-conditional GAN that uses only multiple adversarial loss instead of classification loss. CAGAN is a composite of multiple GANs, where each GAN is trained to generate each class. Since each GAN shares all hidden layers, it can be considered a single model. Because CAGAN only uses adversarial loss, it has fewer hyperparameters than ACGAN. Also, CAGAN does not use classification loss, so there is no conflict between adversarial loss and classification loss as there is with ACGAN. Therefore, CAGAN has better performance than ACGAN. In the experiments in section 4.1.1, we showed that CAGAN has a faster training speed, and better convergence compared to ACGAN.

We also propose mixed batch training to train conditional GAN, when the conditional distribution is biased and there are batch-wise operations in the discriminator. When train-

ing conditional GAN, real conditional distribution and fake conditional distribution can be different. If the discriminator has batch-wise operations, it can use the conditional distribution of the input batch for real/fake discrimination, which lowers the generative performance of the model. Mixed batch training is configuring the ratio of real data to fake data in each batch to be always the same. It prevents the discriminator from relying on the conditional distribution of the input batch to distinguish between real and fake samples. In the experiments in section 4.1.2, we showed that mixed batch training enables the generator to generate class-conditional data on an imbalanced dataset correctly.

4.2 Dynamic Latent Scale GAN

4.2.1 Experiments Settings

In the DLSGAN experiment, we compared our method with several variations of MSE loss methods. Many papers are using several variations of MSE loss to predict latent vectors (e.g., [31], [32], [33], [34], [39], [45]). Our experiment can be seen as a comparison between our methods and theirs. We know that there are several StyleGAN-specific inversion methods (e.g., [35], [36], [37]), but our method is an architecture-agnostic method, so we did not compare our method with them.

We compared the performance of baseline, DLSGAN(ours), MSEGAN, and InfoGAN. Baseline is training an encoder that inverts the generator of the GAN with MSE loss ($L_{enc} = \frac{1}{d_z}\|z - z'\|_2^2$, Eq. 15) after training the GAN without encoder loss. Therefore, the baseline requires additional computation compared to other methods. The encoder architecture for the baseline is the same as the discriminator architecture. MSEGAN is training a GAN with MSE loss during the training (Eqs. 15, 17, and 18). InfoGAN is trained with InfoGAN loss (Eqs. 16, 17, and 18).

We trained GAN to generate the FFHQ dataset [19] resized to 256×256 resolution. Of the 70k images of the FFHQ dataset, the first 50k images were used as training images, and the remaining 20k images were used as unseen images.

We used a simple model architecture consisting of convolution and skip connections. We used a skip connection directly connecting the input to improve the inversion performance of the model. We did not use StyleGAN architecture because some methods used in StyleGAN make GAN inversion difficult (e.g., low learning rate in mapping network) or impossible (e.g., mixing regularization). Furthermore, experiments with simple model architecture can show a more general performance of the proposed method.

NSGAN with R1 regularization [7] was used as an adversarial loss like StyleGAN.

The following hyperparameters were used for the experiments.

$$\lambda_{enc} = 1.0$$

$$\lambda_{r1} = 10.0$$

$$Z \sim N(0, I_{1024})$$

$$optimizer = AdamW \begin{pmatrix} learning\ rate = 0.003 \\ weight\ decay = 0.0001 \\ \beta_1 = 0.0 \\ \beta_2 = 0.99 \end{pmatrix}$$

$$trainable\ weights\ ema\ decay\ rate = 0.999$$

$$batch\ size = 8$$

$$epochs = 100$$

λ_{r1} is R1 regularization weight ($\lambda_{r1} L_{r1} = \lambda_{r1} \mathbb{E}_x \left[\|\nabla_x D(x)\|_2^2 \right]$).

We used an exponential moving average for trainable weights during the evaluation. *trainable weights ema decay rate* is a decay rate for generator and discriminator weights.

For the *update* function for DLSGAN, we used an exponential moving average with *decay rate* $= 0.999$. We also compared the effect of the encoder loss L_{enc} on generator loss L_g.

We used FID [9], Precision, and Recall [10] methods for generative performance evaluation. Among 70k images, the first 50k images were used as a training dataset, and the last

20k images were used as a test dataset. Pre-trained inception model and size of the neighborhood $k = 3$ were used for Precision and Recall evaluation. Average PSNR and average SSIM between 20k generated images and reconstructed images of generated images were used to evaluate the inversion performance of each method (i.e., the difference between $G(z)$ and $G(E(G(z)))$ where $z \sim Z$). The higher the PSNR and SSIM, the better the image reconstruction performance. The PSNR ranges from zero to infinity, while the SSIM ranges from zero to one. Average PSNR and average SSIM between test images and reconstructed images of training images were used to evaluate the comprehensive performance of GAN inversion (i.e., the difference between x and $G(E(x))$ where $x \sim X$). The real image reconstruction performance is high when both the generative performance and the inversion performance are high. Therefore, real image reconstruction performance shows the comprehensive performance of generative performance and inversion performance.

Full codes for the DLSGAN experiments are available at https://github.com/jeongik-jo/DLSGAN.

4.2.2 Dynamic Latent Scale GAN Experiment Results

Fig. 17 shows the generative performance of each method. First, the FID evaluation showed little difference in the generative performance among the methods. However, for the Precision & Recall evaluation, DLSGAN showed slightly higher recall.

Fig. 18 shows the inversion performance of each method. One can see that the inversion performance (fake data reconstruction performance) of DLSGAN is significantly higher than other methods.

Fig. 19 also clearly shows that DLSGAN has the best comprehensive performance (real data reconstruction performance).

Fig. 20 shows the average encoder loss L_{enc} according to the training methods for each epoch. One can see that encoder loss L_{enc} hardly changes from 1.0 except for DLSGAN. This shows that encoder E trained without dynamic latent scale fails to converge because

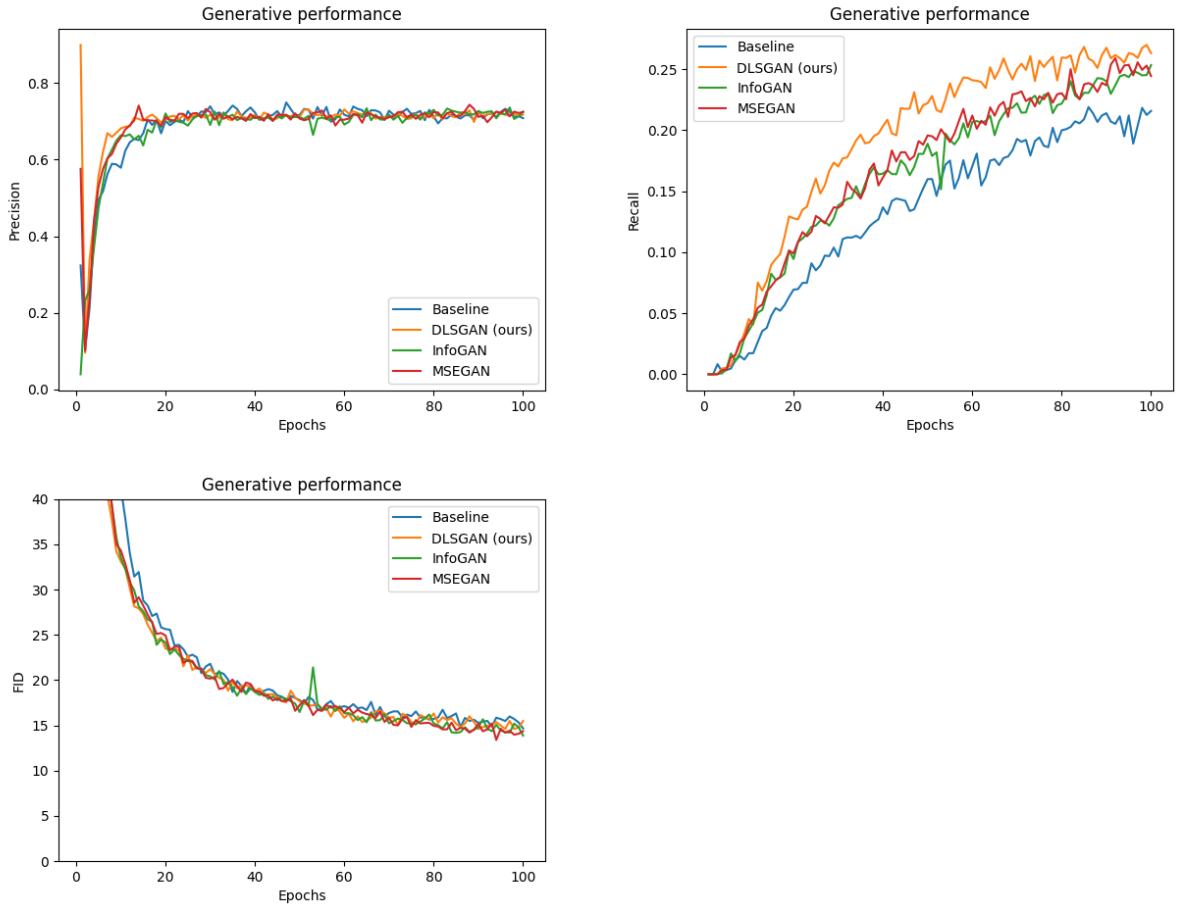

Figure 17: Generative performance for each epoch.

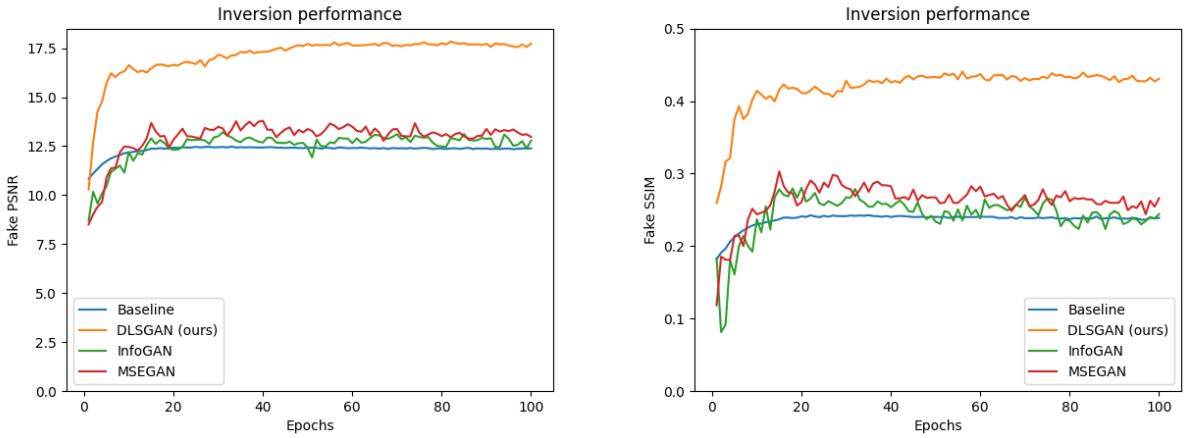

Figure 18: Inversion performance for each epoch.

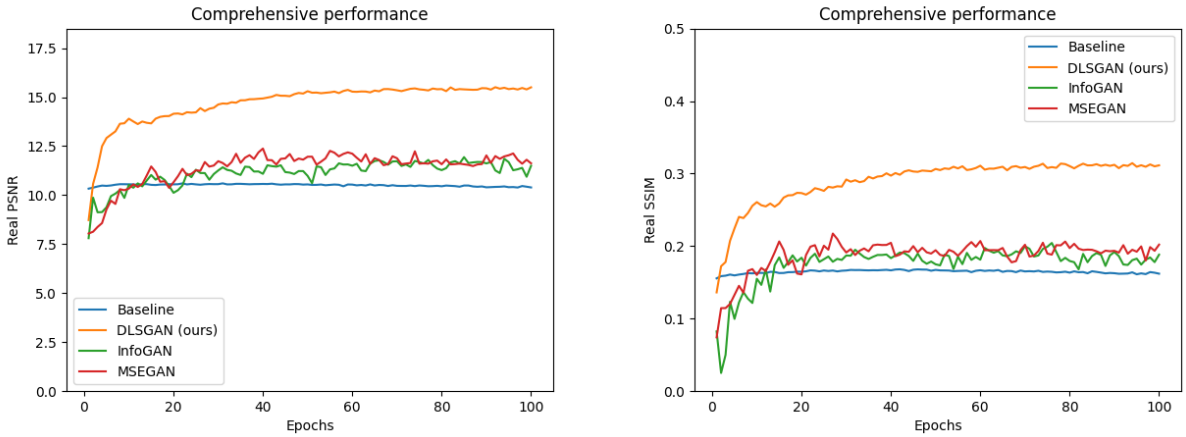

Figure 19: Comprehensive performance for each epoch.

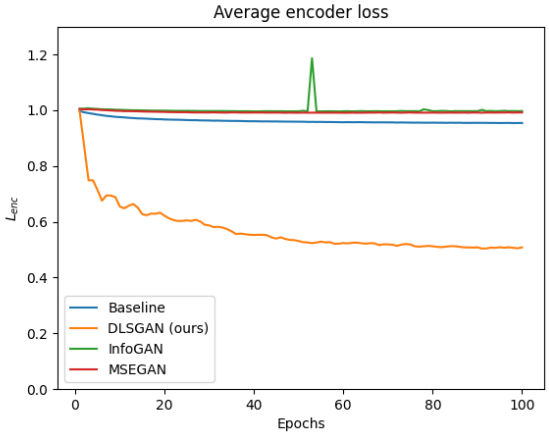

Figure 20: Average L_{enc} for each epoch.

generator G loses information of latent random variable Z.

In the encoder loss of InfoGAN (Eq. 16), the squared error is scaled by dividing it by $e^{E_{lv}(G(Z))}$. However, when this value becomes too small, it can cause numerical instability. Although we added a small constant $\epsilon = 10^{-7}$ to the denominator to mitigate this issue, instability still occurs when $e^{E_{lv}(G(Z))}$ is very small. This numerical instability results in the spike observed in the InfoGAN loss in Fig. 20.

Fig. 21 shows the result of reconstructing the unseen images for each method. One can see that DLSGAN performs significantly better than the other methods.

Fig. 22 shows the differential entropy of scaled latent random variable $Z \circ s$ with dynamic latent scale for each epoch. One can see that the latent entropy decreases as the training progresses with a dynamic latent scale. Like encoder loss L_{enc} of Fig. 20, one can see that the differential entropy of the scaled latent random variable $Z \circ s$ decreases faster when scaled encoder loss $\lambda_{enc} L_{enc}$ is added to the generator loss L_g. Note that differential entropy can be negative.

Figs. 23, 24, and 25 show the interpolating value of the specific dimension of the latent random variable from -2.0 to 2.0. Figure 3 shows the latent interpolation of the values of important dimensions in DLSGAN from -2 to 2.

The larger the value of the scale vector s, the more important (more informative) dimension. One can see that dimensions with large scale contain a lot of information.

4.2.3 Attribute Editing with Dynamic Latent Scale GAN

Since the FFHQ dataset does not have attribute labels, we used an aligned & cropped CelebA [42] dataset for AEDLSGAN experiments. Images of aligned & cropped CelebA dataset have 218×178 resolution, so we resized the images to 256×256 resolution. Model architecture and most hyperparameter settings are the same as section 4.2.1.

We trained AEDLSGAN with three attributes of the CelebA dataset: bangs, gender, and smile. All elements of linear classifier weights w are initialized with $N(0, 1^2)$, and we did not

Figure 21: Unseen real image reconstruction samples for each method. The third column is the input images.

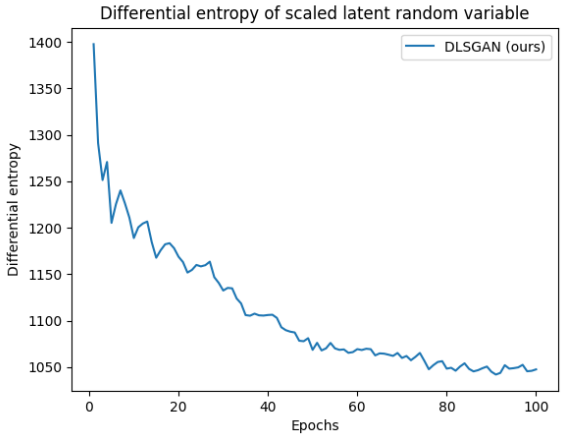

Figure 22: Differential latent entropy of scaled latent random variable $(Z \circ s)$.

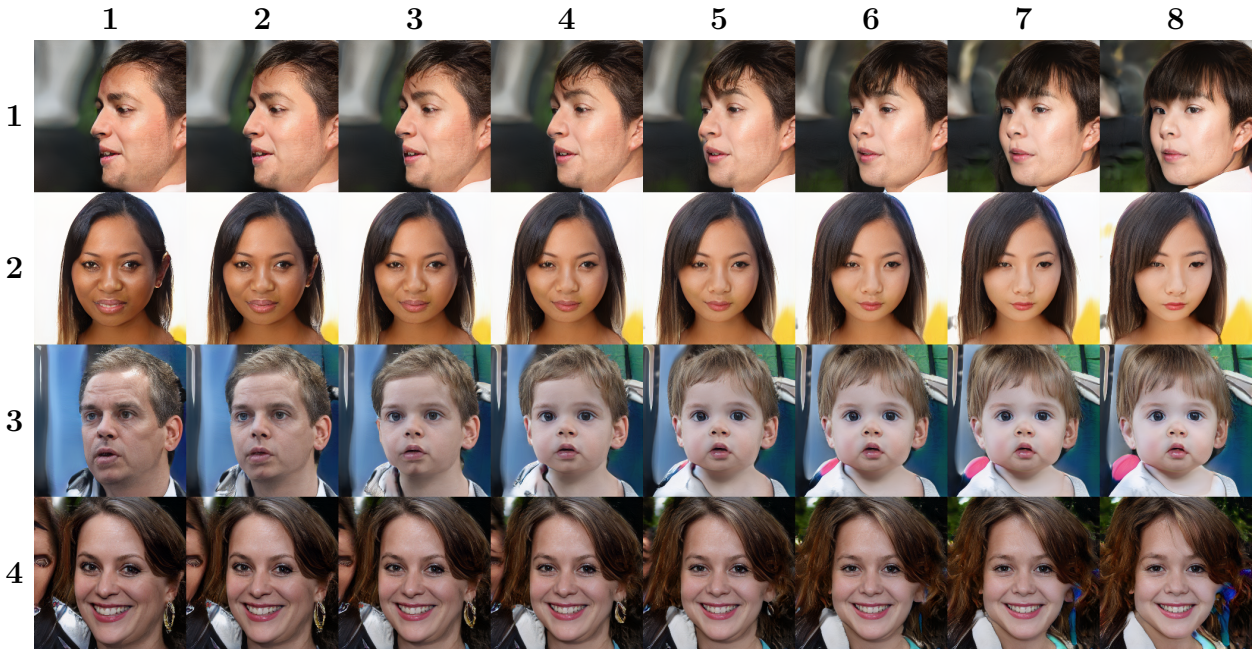

Figure 23: Latent interpolation on most important dimension.

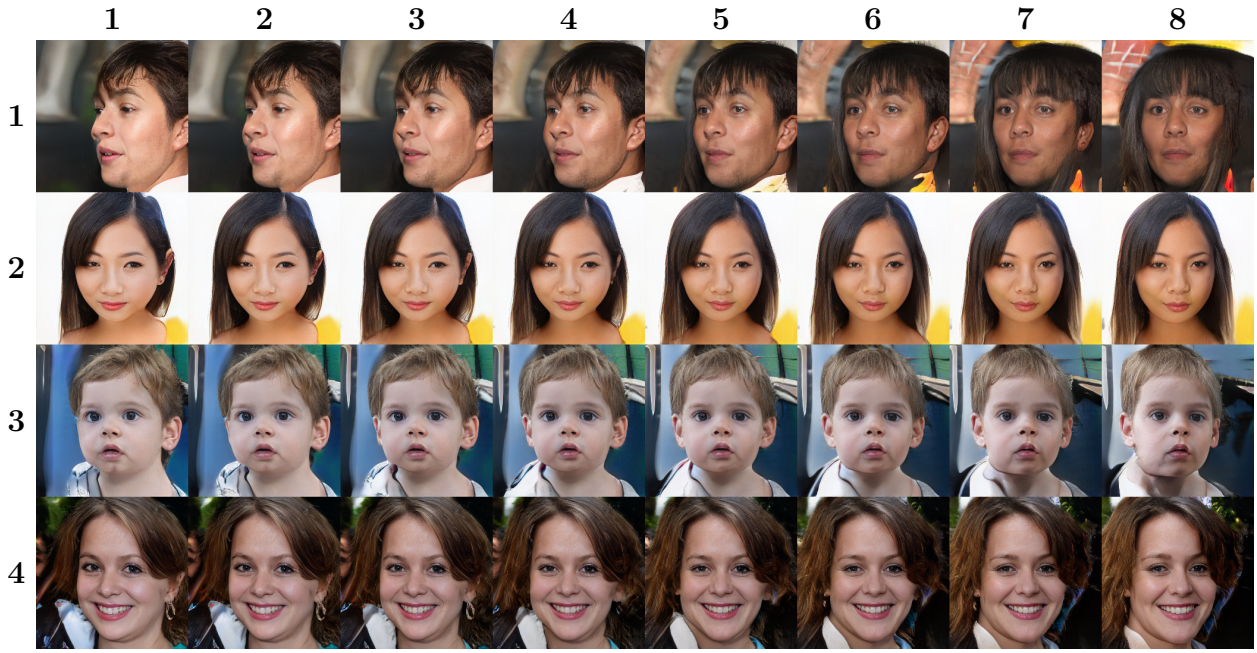

Figure 24: Latent interpolation on second most important dimension.

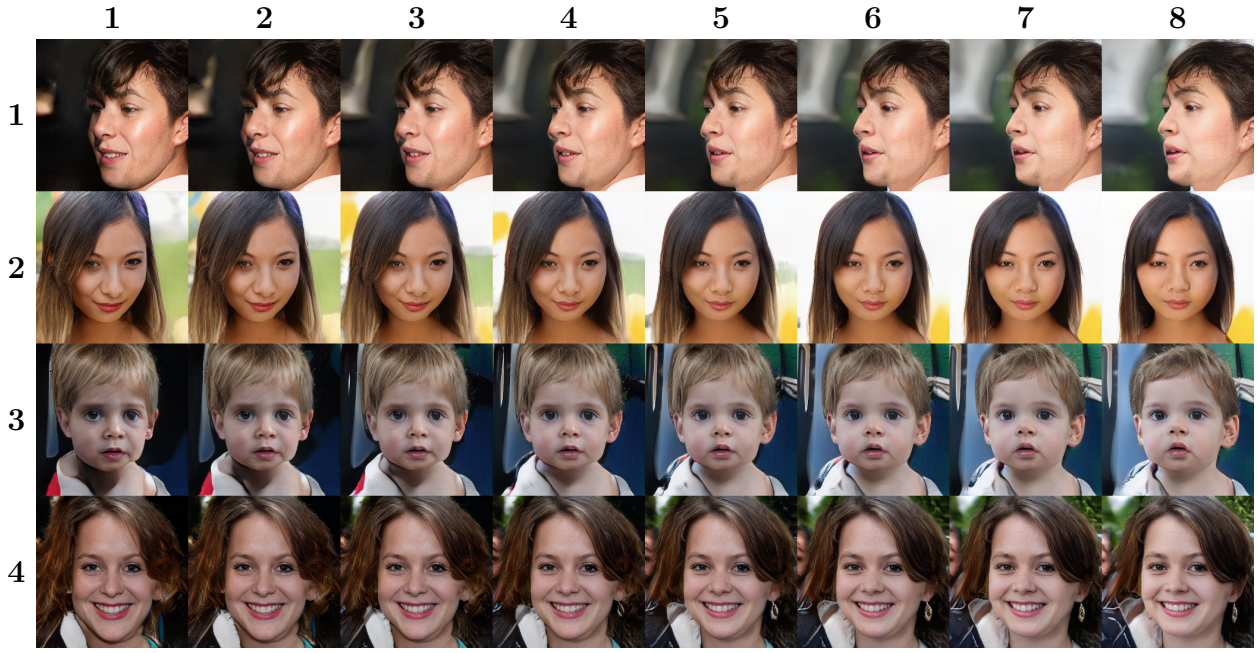

Figure 25: Latent interpolation on third most important dimension.

FID↓	10.7735
Precision↑	0.6743
Recall↑	0.3734
Fake PSNR↑	19.4202
Fake SSIM↑	0.6170
Real PSNR↑	15.0686
Real SSIM↑	0.4178
Last epoch average L_{enc}	0.1609

Table 4: Performance of AEDLSGAN. ↑ represents the higher the better, and ↓ represents the lower the better.

Figure 26: Attribute 'Bangs' transfer of unseen real images. The first column shows input images, and the second column shows reconstructed images. The right part shows continuous attribute edits of attribute 'Bangs'.

use b for simplicity.

Full codes for the AEDLSGAN experiments are available at https://github.com/jeongik-jo/AEDLSGAN.

Table 4 shows the performance of AEDLSGAN. About 20k unseen images and generated images were used for evaluation.

Figs. 26, 27, 28 shows continuous attributes transfer of unseen real images. The first column of figures shows the input images, and the second column shows the reconstructed images through predicted latent vectors. Based on the thick white line, the images on the right

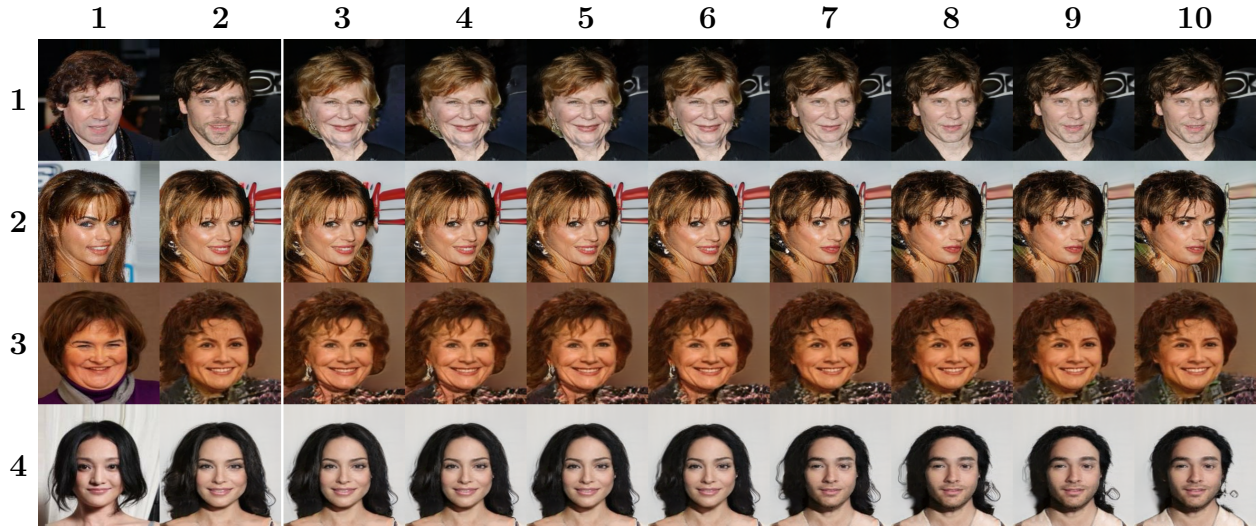

Figure 27: Attribute ‘Gender’ transfer of unseen real images. The first column shows input images, and the second column shows reconstructed images. The right part shows continuous attribute edits of attribute ’Gender’.

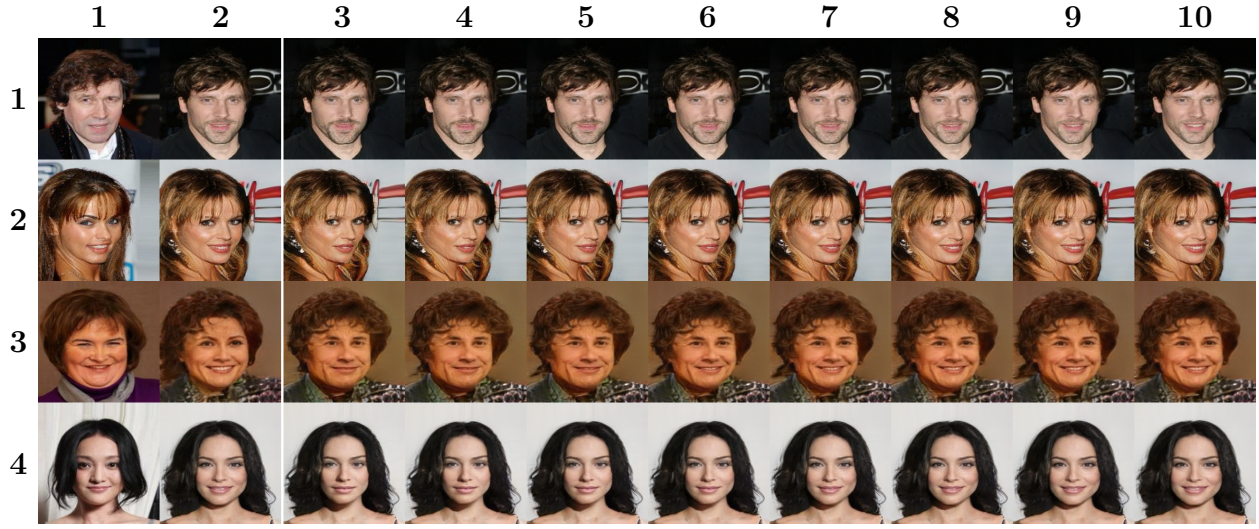

Figure 28: Attribute ‘Smile’ transfer of unseen real images. The first column shows input images, and the second column shows reconstructed images. The right part shows continuous attribute edits of attribute ’Smile’.

side show results of continuous attribute transfer. In the right part, the score $((z' \circ s)w/\sqrt{d_z})$ changes from -1 to 1. If the score is positive, the image has the attribute 'has bangs,' 'male,' and 'smile,' respectively. If the score is negative, the image has the attribute 'no bangs,' 'female,' and 'not smile,' respectively. One can see that AEDLSGAN can edit the attribute of the input image continuously.

4.2.4 Summary

In the second contribution, we introduced DLSGAN, a novel encoder-based GAN inversion method for better convergence than in a simple mean squared error. DLSGAN dynamically adjusts the scale of each element of the latent random variable to prevent the generator from losing information of the latent random variable. The entropy of the scaled latent random variable gradually decreases until it becomes suitable to represent the data. This allows the model to converge better, resulting in better inversion performance. The scale of the latent random variable is approximated by tracing the element-wise variance of the predicted latent random variable of the encoder from previous training steps. In the experiments in section 4.2.2, DLSGAN showed better performance than MSEGAN, InfoGAN, and the encoder trained with MSE loss.

Additionally, we propose AEDLSGAN which utilizes DLSGAN to continuously edit the attributes of input data. AEDLSGAN trains a class-conditional GAN with the proposed GAN inversion method to fit a fixed linear classifier. In section 4.2.3, we showed that AEDLSGAN can continuously modify the attribute of input data by progressively changing the predicted latent vector to alter the output of the fixed linear classifier.

4.3 Out-of-distribution detection with Dynamic Latent Scale GAN

To compare OOD detection performance, we trained five types of models: AnoDLSGAN, MSEGAN, InfoGAN [39], autoencoder, and classifier. Each model is trained only with the ID train dataset.

Training AnoDLSGAN is the same as training DLSGAN (Eqs. 33, 17, and 18). MSEGAN is trained with simple MSE losses (Eqs. 15, 17, and 18). And InfoGAN is trained with InfoGAN losses (Eqs. 16, 17, and 18). Autoencoder is trained with pixel-wise mean squared error reconstruction loss. The classifier is trained with cross-entropy loss. AnoDLSGAN, InfoGAN, and autoencoder were trained without labels, while the classifier was trained with labels. The same encoder and decoder architecture was used for training each method (the classifier uses only the encoder for training).

We used negative log probability OOD score and reconstruction OOD score for OOD detection with DLSGAN, InfoGAN, and MSEGAN. The negative log probability score of DLSGAN is the proposed AnoDLSGAN. Pixel-wise mean squared reconstruction error is used for the reconstruction OOD score. Also, reconstruction OOD score is used for autoencoder [65]. Energy score [57] with ReAct [58] is used for OOD detection with the classifier.

Training GANs and an autoencoder take more time than classifiers because those models use both an encoder and decoder for training. GAN and classifier require one encoder inference to classify one data, while autoencoder requires one encoder and decoder inference.

We experimented with two cases where the ID dataset is the MNIST handwritten digits dataset [61] or the CelebA dataset [42].

4.3.1 MNIST Experiment Settings

In MNIST experiments, the MNIST dataset [61] was used as the ID dataset, and Corrupted MNIST [62] (CMNIST), Fashion MNIST [63] (FMNIST), and Kuzushiji MNIST [64] (KMNIST) datasets were used as OOD dataset. All datasets are available through Tensorflow datasets [73]. For the preprocessing, we added padding to the images to make the resolution 32×32 and normalized the pixel values to be between -1 and 1. In the MNIST experiment, we used 7-fold cross-validation. Each MNIST dataset has $70,000$ images, so each ID train dataset has $60,000$ images, the ID test dataset has $10,000$ images, and each OOD test dataset has $10,000$ images in each fold.

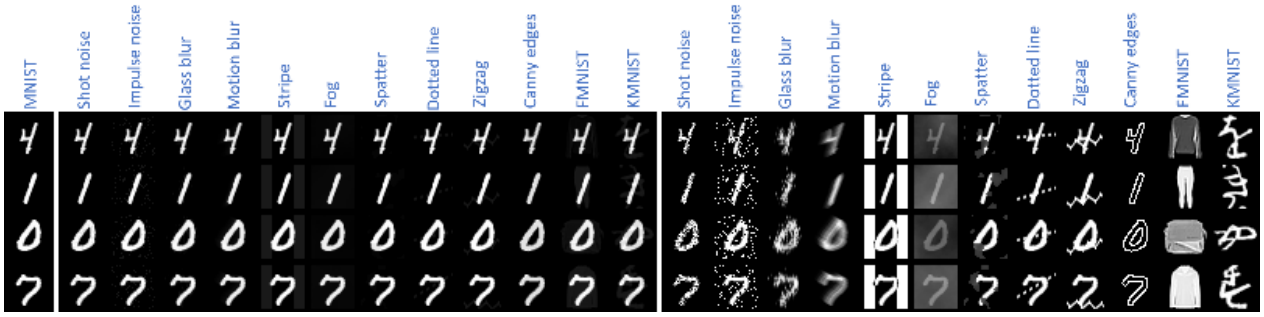

Figure 29: Sample ID and OOD images for MNIST experiments. Column 1: ID images. Columns 2-13: near OOD images. Columns 14-25: far OOD images. $OOD\ intensity = 0.1$ was used to generate near OOD images.

Fig. 29 shows samples of ID images and OOD images for the MNIST experiment. The first column of Fig. 29 shows ID images. Columns 13-23 (right part of the right white line) show far OOD images of the OOD dataset (CMNIST, FMNIST, KMNIST). Columns 2-12 (between the two white lines) show near OOD images. Near OOD images are generated by linear interpolation between far OOD images and ID images (i.e., $near\ OOD\ image = far\ OOD\ image \times OOD\ intensity + ID\ image \times (1 - OOD\ intensity)$). We used $OOD\ intensity = 0.1$ to generate near OOD images. The near OOD images are hard to distinguish for humans without looking very closely.

Images from the CMNIST dataset were generated by adding corruption to the images from the MNIST dataset. Therefore, each image from the CMNIST dataset has a corresponding original image from the MNIST dataset. When generating near OOD images with the CMNIST dataset, corresponding images from the MNIST dataset were used as ID images, not random images from the MNIST dataset.

The Following hyperparameters were used for training models.

$$batch\ size = 32$$
$$optimizer = Adam \begin{pmatrix} learning\ rate = 0.001 \\ \beta_1 = 0.0 \\ \beta_2 = 0.99 \end{pmatrix}$$

$$epoch = 30$$

$$learning\ rate\ decay\ rate\ per\ epoch = 0.95$$

learning rate decay rate per epoch is a value multiplied by the learning rate for each epoch.

The following hyperparameters were used for training GANs.

$$\lambda_{enc} = 1$$

$$\lambda_{r1} = 0.1$$

$$Z \sim N(0, I_{256})$$

λ_{enc} and λ_{r1} represents encoder loss weight and R1 regularization weight [7] ($\lambda_{r1} L_{r1} = \lambda_{r1} \|\nabla_x D(X)\|_2^2$), respectively. Also, v *decay rate* $= 0.999$ was used for DLSGAN. It represents the exponential moving average decay rate for variance vector v of DLSGAN. Equalized learning rate [8] was used for both the encoder and decoder. We simply used CNNs and fully connected layers for the model architecture. Full codes for the MNIST experiments are available at https://github.com/jeongik-jo/AnoDLSGAN

4.3.2 Experiment Results

Table 5 shows the OOD detection performance for each method. We used AUROC (area under ROC curve) to evaluate OOD detection performance. Threshold values to calculate AUROC are a set of every 10th element when OOD scores of ID data are sorted (i.e., set of 0-th, 10-th, 20-th, ..., elements).

Each value in Table 5 is the average AUROC multiplied by 100. "NLL" and "Rec" represent that the method used a negative log likelihood (probability) OOD score and reconstruction error OOD score (pixel-wise mean squared error), respectively. "NLL" with DLSGAN is the proposed AnoDLSGAN, and "Rec" with GANs is similar to [45] and [46].

In "Energy" of Table 5, "t" represents the temperature of the energy score [57], and "p" represents the activation percentage of ReAct [58]. When $p = 1.0$, it is the same as when ReAct was not applied.

AUROC (×100)		GAN						Auto-encoder	Classifier							
		DLSGAN [51]		InfoGAN [39]		MSEGAN		Reconst-ruction [65]	Energy score [57] with ReAct [58]							
		NLL (ours)	Rec	NLL	Rec	NLL	Rec	Rec	t=1.0 p=0.85	t=1.0 p=0.90	t=1.0 p=0.95	t=1.0 p=1.0	t=10.0 p=0.85	t=10.0 p=0.90	t=10.0 p=0.95	t=10.0 p=1.0
Near OOD	Shot noise	50.68	50.15	50.68	48.96	50.62	49.17	52.85	50.61	50.98	51.40	51.94	50.50	50.92	51.37	51.93
	Impulse noise	93.47	65.74	57.07	52.88	59.35	53.50	72.20	50.64	51.00	51.41	51.92	50.54	50.94	51.38	51.91
	Glass blur	66.60	57.45	52.58	47.40	52.53	48.00	47.96	50.90	51.85	52.95	54.28	50.64	51.70	52.88	54.26
	Motion blur	85.13	65.14	55.84	48.65	55.94	49.33	50.05	52.08	53.05	54.10	55.31	51.79	52.89	54.03	55.28
	Stripe	100.00	95.27	97.17	82.38	97.61	79.36	98.72	51.12	51.65	52.37	53.36	51.12	51.67	52.41	53.41
	Fog	99.44	89.74	67.29	53.54	68.34	54.37	74.53	52.21	53.74	55.45	57.39	51.84	53.54	55.36	57.37
	Spatter	70.81	64.16	52.48	49.83	52.74	49.99	52.10	50.34	50.54	50.76	51.05	50.29	50.51	50.75	51.05
	Dotted line	72.04	59.99	53.59	52.64	54.91	53.04	56.05	50.22	50.29	50.36	50.46	50.18	50.26	50.35	50.45
	Zigzag	90.99	73.19	61.02	54.82	63.77	55.89	64.13	50.44	50.58	50.74	50.93	50.37	50.54	50.72	50.92
	Canny edges	87.58	60.47	57.53	48.39	57.08	49.26	62.08	52.57	54.54	56.71	59.19	52.00	54.21	56.56	59.12
	FMNIST	99.85	87.92	73.30	55.19	73.78	57.78	80.65	53.57	55.37	57.33	59.56	52.95	55.01	57.15	59.48
	KMNIST	99.46	84.34	71.96	54.96	72.98	56.90	82.90	54.11	55.93	57.93	60.20	53.46	55.55	57.74	60.10
Far OOD	Shot noise	94.97	71.55	69.67	59.81	68.46	61.23	99.74	69.49	71.23	73.06	75.03	67.07	69.75	72.12	74.34
	Impulse noise	100.00	99.89	99.99	99.84	100.00	99.83	100.00	83.69	83.55	83.27	83.22	81.03	81.77	81.94	82.03
	Glass blur	99.89	72.23	93.39	58.85	93.48	60.84	99.80	82.57	86.73	89.95	91.75	78.08	84.34	88.60	90.76
	Motion blur	100.00	80.51	96.73	59.94	97.57	65.29	97.60	83.49	86.59	88.88	90.27	80.59	85.11	88.07	89.68
	Stripe	100.00	100.00	100.00	100.00	100.00	100.00	100.00	92.99	92.86	92.25	91.36	93.26	93.19	92.52	91.52
	Fog	100.00	97.85	99.97	96.78	99.98	95.57	99.99	99.27	99.65	99.79	99.82	96.50	98.53	99.22	99.40
	Spatter	99.54	87.41	89.73	75.56	91.11	75.96	99.41	63.21	64.48	65.69	66.90	61.82	63.60	65.12	66.47
	Dotted line	99.41	89.91	86.53	88.04	87.61	87.88	99.96	64.04	63.51	62.79	62.03	61.74	61.88	61.55	60.95
	Zigzag	99.89	95.12	94.27	95.34	95.01	95.14	99.99	71.91	71.01	69.63	67.84	68.53	68.46	67.53	65.93
	Canny edges	100.00	94.04	99.93	88.27	99.92	89.35	99.94	92.41	89.76	84.03	65.95	90.17	87.62	81.34	62.08
	FMNIST	100.00	97.64	99.96	93.96	99.98	94.01	99.98	96.37	97.06	97.46	97.60	94.20	95.87	96.72	97.00
	KMNIST	99.94	97.49	98.02	95.89	98.22	96.55	99.98	94.89	94.81	94.32	93.49	92.29	93.04	92.87	92.01

Table 5: OOD detection performance for each method in MNIST experiment.

In Table 5, one can see that the overall performance of AnoDLSGAN is the best.

Energy score with ReAct showed good performance in easy OOD of far OOD datasets detection (Fog, FMNIST, and KMNIST) but showed relatively poor performance in some CMNIST datasets (Dotted line, Spatter, Zigzag). Also, it could hardly detect the near OOD images.

Comparing p=1.0 with the others in the energy score, ReAct improved the performance on some far OOD datasets (Canny edges, Zigzag, Dotted line, and KMNIST), but for most datasets, ReAct degraded or did not improve the OOD detection performance.

OOD detection using reconstruction error (i.e., "Rec" with DLSGAN, InfoGAN, and Autoencoder in Table 5) showed good performance with far OOD datasets but not in near OOD datasets. This shows that the reconstruction-based OOD detection methods cannot properly detect OOD data when OOD data have low reconstruction error, even if the perceptual distance from ID data is far. Also, AnoDLSGAN showed significantly better performance with near OOD detection, even if the ID image reconstruction performance (PSNR and SSIM in Table 6) of the autoencoder was much better than AnoDLSGAN's.

	DLSGAN	InfoGAN	MSEGAN	Autoencoder	Classifier
FID	5.1614	6.9579	6.7274	-	-
PSNR	15.4209	14.2021	14.5149	28.9784	-
SSIM	0.5642	0.4764	0.4990	0.9627	-
Accuracy	-	-	-	-	0.9947

Table 6: Basic model performances in MNIST experiments.

AnoDLSGAN showed much better performance when compared to InfoGAN using the NLL OOD score. This indicates that AnoDLSGAN performs better than InfoGAN because ID data is densely mapped to the latent space due to the latent entropy optimality of DLSGAN.

All methods failed to detect shot noise near OOD images. Shot noise near OOD images are very difficult to detect, even for humans.

Table 6 shows the basic performance of each model. In table 6, PSNR and SSIM represent ID image reconstruction performance. The FID of GANs is below 10, within the range of 0 to 400, indicating good generative performance. For autoencoder, the PSNR is around 30 and the SSIM is close to 1, indicating high reconstruction performance. GANs show lower reconstruction performance compared to the autoencoder since they also have to perform the generation task. Also, the accuracy of the classifier is very high. The high performance of models indicates that the MNIST OOD detection experiments were performed properly.

4.3.3 CelebA Experiments

In CelebA experiments, we used CelebA [42] images as ID dataset, and train images of Coil100[74], Deep weeds[75], STL10 [76], Cassava[77], Colorectal histology [78], Malaria[79], Stanford dogs [80, 81], Stanford online products [82] as OOD datasets. We resized all images to 128×128 resolution. In CelebA experiments, we used $OOD\ intensity = 0.5$ to generate near OOD images.

Fig. 30 shows ID and OOD images for CelebA experiments.

In CelebA experiments, we used $10,000$ images of the CelebA test dataset as test ID

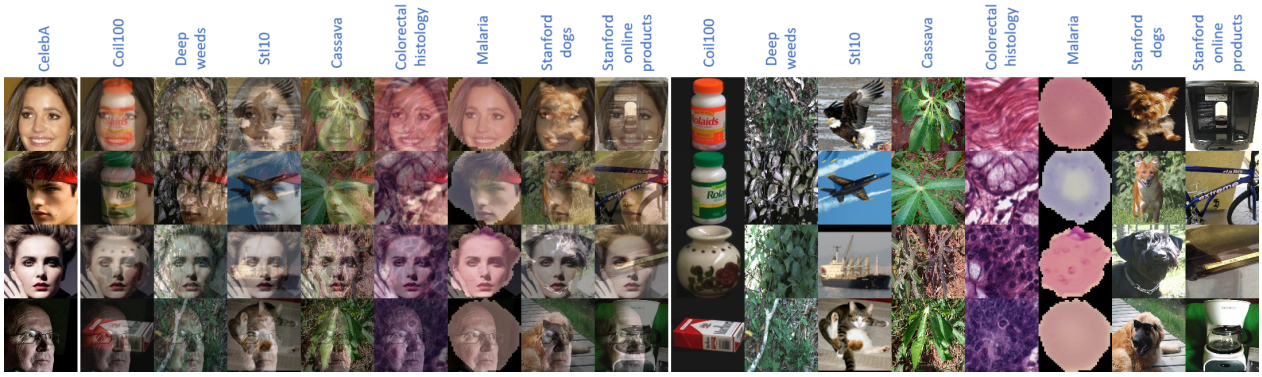

Figure 30: Sample ID and OOD images for CelebA experiments. Column 1: ID images. Columns 2-9: near OOD images. Columns 10-17: far OOD images. $OODintensity = 0.5$ was used to generate near OOD images.

OOD Dataset	Sample Size
Coil 100 [74]	7200
Deep weeds [75]	10000
STL 10 [76]	4992
Cassava [77]	5648
Colorectal histology [78]	4992
Malaria [79]	10000
Stanford dogs [80, 81]	10000
Stanford online products [82]	10000

Table 7: OOD sample sizes for each OOD dataset in CelebA experiments.

images. However, some datasets have fewer than $10,000$ images. To equalize the number of ID images and OOD images for each ID-OOD pair, we reduced the number of ID images to match the number of OOD images if the number of OOD images was less than $10,000$, and reduced the number of OOD images to match the number of ID images if it was greater than $10,000$. Table 7 shows the number of images used for each ID-OOD pair. For example, $7,200$ ID images and $7,200$ OOD images were used for Coil 100 evaluation.

The following hyperparameters were used for CelebA experiments.

$$\begin{gathered} batch\ size = 16 \\ optimizer = Adam \begin{pmatrix} learning\ rate = 0.001 \\ \beta_1 = 0.0 \\ \beta_2 = 0.99 \end{pmatrix} \\ epoch = 10 \\ ema\ decay\ rate = 0.999 \end{gathered}$$

In CelebA experiments, we used the exponential moving average for trainable weights. *ema decay rate* represents the exponential moving average decay rate for trainable weights. The generator (decoder) architecture was similar to StyleGAN2 [20], while the discriminator (encoder) architecture was simply consists of CNN layers and skip connections. Also, the following hyperparameters were used for training GANs.

$$\begin{gathered} \lambda_{enc} = 1 \\ \lambda_{r1} = 1.0 \\ Z \sim N(0, I_{512}) \end{gathered}$$

Full codes for the CelebA experiments are available at https://github.com/jeongik-jo/AnoDLSGAN_CelebA.

4.3.4 CelebA Results

Table 8 shows OOD detection performance for each method in CelebA experiments. Similar to the MNIST experiments, one can see that the overall performance of the proposed An-

AUROC(×100)		GAN						Auto-encoder	Classifier							
		DLSGAN [51]		InfoGAN [39]		MSEGAN		Reconst-ruction	Energy score [57] with ReAct [58]							
		NLL (ours)	Rec	NLL	Rec	NLL	Rec	Rec	t=1.0 p=0.85	t=1.0 p=0.90	t=1.0 p=0.95	t=1.0 p=1.0	t=10.0 p=0.85	t=10.0 p=0.90	t=10.0 p=0.95	t=10.0 p=1.0
Near OOD	Coil100	72.26	50.65	51.28	35.40	45.25	35.72	34.72	69.60	69.91	70.33	70.96	62.18	64.16	65.61	66.59
	Deep weeds	87.86	83.61	67.37	68.65	77.12	74.78	83.98	78.69	79.85	81.54	83.31	80.26	82.07	83.82	85.50
	STL10	74.06	74.23	51.82	63.82	52.89	63.90	58.92	79.52	79.98	80.53	81.17	73.60	75.29	76.48	77.04
	Cassava	87.77	87.45	81.36	74.30	81.26	76.57	82.68	80.19	81.56	83.31	85.46	79.33	81.66	83.82	85.95
	Colorectal histology	73.46	75.90	64.11	65.69	56.79	71.83	50.09	74.44	75.23	76.17	77.20	75.52	76.48	77.21	77.84
	Malaria	76.08	71.58	68.80	63.86	64.60	68.96	48.46	73.67	73.40	73.40	73.55	74.25	74.21	73.98	73.67
	Stanford dogs	78.76	79.06	53.35	69.05	55.94	69.33	68.32	78.67	79.56	80.63	81.88	72.98	75.06	76.67	77.77
	Stanford online	86.12	80.38	68.21	74.16	67.97	71.94	65.21	77.70	78.14	78.69	79.37	71.72	73.42	74.71	75.50
Far OOD	Coil100	97.54	80.36	86.49	69.27	86.75	70.31	65.26	70.72	70.41	69.34	65.77	65.91	68.01	68.75	66.30
	Deep weeds	99.46	98.06	97.61	94.95	99.09	95.52	99.22	72.75	74.36	76.54	83.42	83.98	86.84	89.87	94.34
	STL10	96.97	95.00	89.56	92.65	90.07	91.27	91.93	78.64	78.29	77.63	75.69	74.30	76.16	77.06	76.13
	Cassava	99.61	98.70	99.44	96.51	99.04	96.28	99.28	78.34	77.16	75.44	71.76	80.80	82.85	83.98	83.47
	Colorectal histology	96.27	96.65	80.48	90.20	79.11	91.10	75.58	78.49	78.85	79.26	78.33	80.41	82.05	83.66	85.35
	Malaria	99.91	97.25	99.86	94.77	99.81	96.85	86.37	82.83	80.60	76.95	72.93	78.13	79.59	79.97	81.68
	Stanford dogs	98.21	96.45	92.38	94.52	93.31	93.20	96.28	77.58	77.74	77.85	77.76	74.01	76.08	77.49	78.20
	Stanford online	99.21	96.52	95.27	95.44	95.77	94.39	91.58	76.76	76.26	75.33	73.02	72.58	74.42	75.31	74.63

Table 8: OOD detection performance for each method in CelebA experiments.

oDLSGAN is the best. For far OOD images, the proposed AnoDLSGAN performed the best. Similar to the MNIST experiments, ReAct increased model performance with some far OOD datasets (e.g., Coil100, STL10, Cassava, Malaria, Stanford online with $t = 1$), but decreased model performance with some other far OOD datasets (e.g., Deep weeds with $t = 1$ and $t = 10$, Cassava, Colorectal histology with $t = 10$).

For the Near OOD dataset, the proposed AnoDLSGAN and Energy score with $t = 1$ or $t = 10$, without ReAct, performed well. AnoDLSGAN significantly outperformed the Energy score on the near OOD Coil100 and Stanford online datasets, and there was no significant performance difference on the other near OOD datasets. DLSGAN with reconstruction OOD score showed good performance except for the Coil100 data.

Overall, the performance of AnoDLSGAN was the best.

Fig. 31 shows autoencoder reconstructed images of near and far OOD images. Since the autoencoder is trained to reconstruct the input, it tries to reconstruct the input image even if it is an OOD image. This means that the autoencoder can also reconstruct OOD images correctly, which is why the near OOD detection performance of the autoencoder with reconstruction score was not good.

Table 9 shows the basic performance of each model in CelebA experiments. The overall performance of the models is lower than that of the MNIST experiment (table 6) because

Figure 31: Autoencoder reconstructed samples in CelebA experiments. Row 1: near OOD images. Row 3: far OOD images. Row 2 and 4: reconstructed images of left images.

	DLSGAN	InfoGAN	MSEGAN	Autoencoder	Classifier
FID	8.1337	7.4739	7.7283	-	-
PSNR	15.8093	14.9347	14.3391	23.9455	-
SSIM	0.3727	0.3502	0.3268	0.6328	-
Accuracy	-	-	-	-	0.9267

Table 9: Basic model performances in CelebA experiments

the complexity and dimensionality of the dataset is higher than that of the MNIST dataset. Still, the GANs show good generative performance with FIDs below 10, and the autoencoder shows acceptable performance with PSNR above 20 and SSIM above 0.6. The accuracy of the classifier is also acceptable at over 90%. Therefore, as table 6, acceptable basic model performances show that the CelebA OOD detection experiments were performed correctly.

4.3.5 Summary

In the third contribution, we introduced AnoDLSGAN, a method for out-of-distribution detection using DLSGAN. In DLSGAN, the entropy of the scaled latent random variable becomes suitable for representing the data. Furthermore, since each dimension of the latent random variable is independent and follows a simple distribution, it is possible to calculate the probability of the input data. AnoDLSGAN performs out-of-distribution detection through the calculated log probability of the input data. In the experiments in sections 4.3.1-4.3.4, we showed that AnoDLSGAN has higher OOD detection performance than other methods on the MNIST dataset and CelebA dataset.

4.4 Dynamic Latent Scale GAN with Perceptual VAE loss

4.4.1 Experiment Settings

We compared PVDGAN with other architecture-agnostic encoder-based GAN inversion methods [32, 33, 34] (MSEGAN uses MSE encoder loss (Eq. 15) to train the encoder and the generator), InfoGAN [39], DLSGAN [51], VAEGAN [68]) in the experiments. Full codes for experiments are available in `https://github.com/jeongik-jo/PVDGAN`.

Flickr Faces HQ (FFHQ) dataset [19] and Animal Face HQ (AFHQ) dataset [21] resized to 256×256 resolution were used for the experiments. In the FFHQ dataset, $50,000$ images were used as the training dataset, and $20,000$ images were used as the test dataset among $70,000$ images. In the AHFQ dataset, all $16,130$ images were used as both training dataset and test dataset. Pixel values were normalized between -1 to 1.

Non-saturating GAN loss [1] was used as an adversarial loss function. The following equations show adversarial loss functions used in the experiments.

$$A_d(a_r, a_f) = \log(1 + e^{-a_r}) + \log(1 + e^{a_f}) \tag{49}$$

$$A_g(a_f) = \log(1 + e^{-a_f}) \tag{50}$$

Also, $\lambda_{r1} L_{r1}$ is added to discriminator loss for R1 regularization [7], where λ_{r1} represents R1 regularization loss weight. The following equations show R1 regularization loss.

$$L_{r1} = \|\nabla_x D(X)\|_2^2 \tag{51}$$

We used a simple model architecture consisting of only convolutional layers and skip connections. Equalized learning rate [8] was used for all trainable weights. All methods used the same model architecture except for the VAEGAN. VAEGAN requires two encoders (VAE encoder and discriminator) and one decoder (generator), while other methods require one encoder (discriminator) and decoder (generator). Therefore, the computation and memory required per each training step are higher than other methods. In our experiments, VAEGAN required approximately 30% additional computation time compared to other methods. There was no significant difference in computation time between MSEGAN, InfoGAN, DLSGAN, and PVDGAN. The penultimate layer output of the discriminator was used as the feature encoder output of PVDGAN and VAEGAN. The feature vector dimension d_f was $4 \times 4 \times 512 = 8,192$.

The following hyperparameters were used for experiments.

$$\lambda_{r1} = 3.0$$

$$d_z = 1024$$

$$Z \sim N(0, I_{d_z})$$

$$optimizer = Adam \begin{pmatrix} learning\ rate = 0.003 \\ \beta_1 = 0.0 \\ \beta_2 = 0.99 \end{pmatrix}$$

$$trainable\ weights\ ema\ decay\ rate = 0.999$$

$$latent\ variance\ vector\ ema\ decay\ rate = 0.999$$

$$batch\ size = 8$$

$$epochs = 100$$

MSEGAN, InfoGAN, PVDGAN, and DLSGAN used encoder loss weight $\lambda_{enc} = 1.0$. Also, PVDGAN and VAEGAN used reconstruction loss weight $\lambda_{rec} = 1.0$. VAEGAN used three different prior loss weights $\lambda_{prr} = 0$, 1, and 10.

We used Frechet Inception Distance (FID) [9], Precision & Recall [10] metrics with a pre-trained inception model for generative performance evaluation. Generative performance indicates how close the generated data distribution is to the target real data distribution. The lower the FID score, the higher the generative performance of the model. In the Precision & Recall evaluation, precision indicates the percentage of generated data that is close to the real data, and recall indicates the percentage of real data that is close to the generated data. Precision and recall have values between zero to one, with higher values indicating better generative performance. In general, a high precision indicates that the quality of generated data is high, and a high recall indicates that the diversity of data is high. Pre-trained inception model and size of the neighborhood $k = 3$ were used for both FID and Precision & Recall evaluation.

Inversion performance indicates how well the encoder can invert the generator. It is evaluated through real data and fake data reconstruction performance via the encoder and the generator. Average Peak Signal-to-noise Ratio (PSNR) and Structural Similarity (SSIM) were used for inversion performance evaluation as DLSGAN [51]. PSNR and SSIM are both metrics that indicate image reconstruction performance. PSNR ranges from 0 to infinity, with higher values indicating better reconstruction performance. SSIM ranges from -1 to 1,

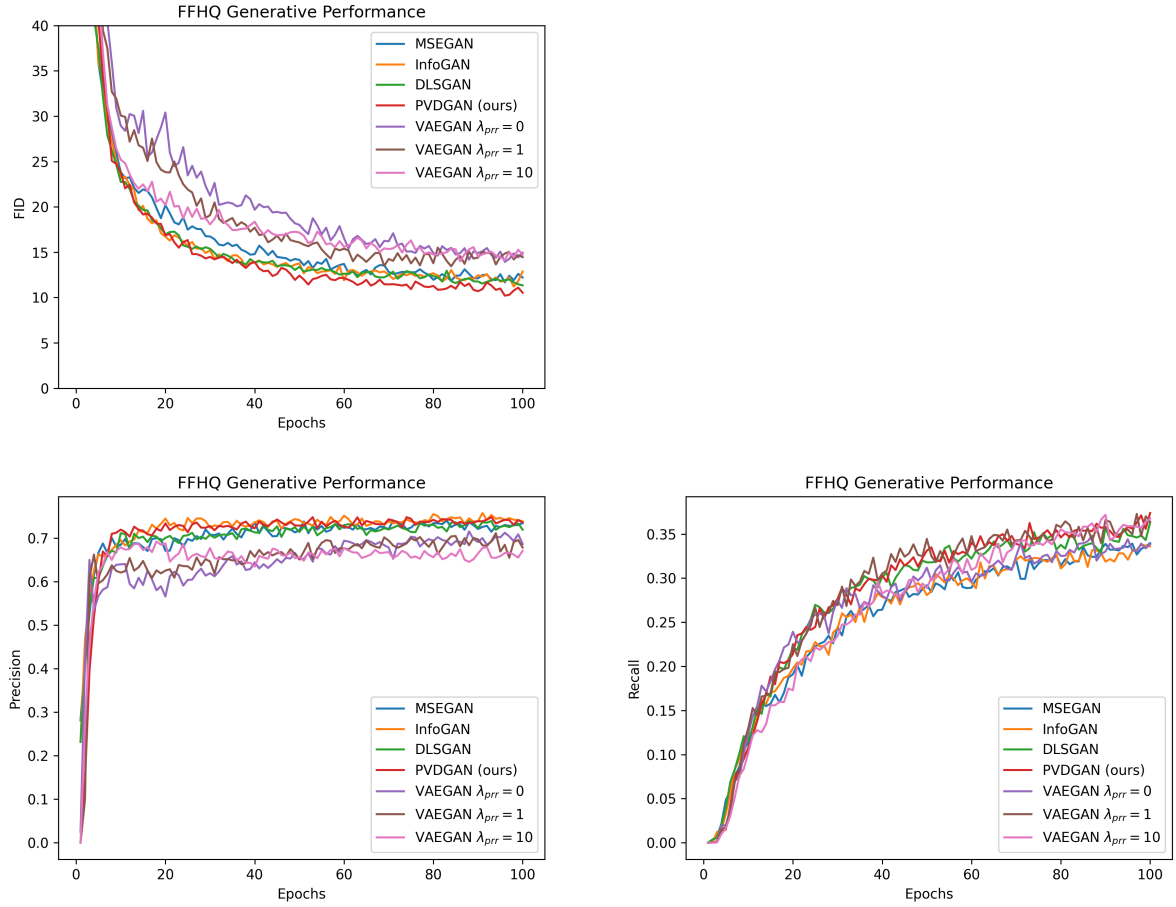

Figure 32: Generative performance graphs in FFHQ experiments.

with higher values indicating better reconstruction performance.

4.4.2 Experimental Results

Figs. 32 and 33 show the generative performance of each method in FFHQ and AFHQ experiments. In the FID evaluation in FFHQ and AFHQ experiments, PVDGAN performs similarly or better than MSEGAN, InfoGAN, and DLSGAN. This is because PVDGAN integrated perceptual VAE into DLSGAN, which allows generative models to generate diverse data more easily. This can be seen more clearly with the Precision & Recall evaluation. In both FFHQ and AFHQ experiments, PVDGAN shows similar precision to MSEGAN, InfoGAN, and DLSGAN. On the other hand, PVDGAN showed higher recall than those

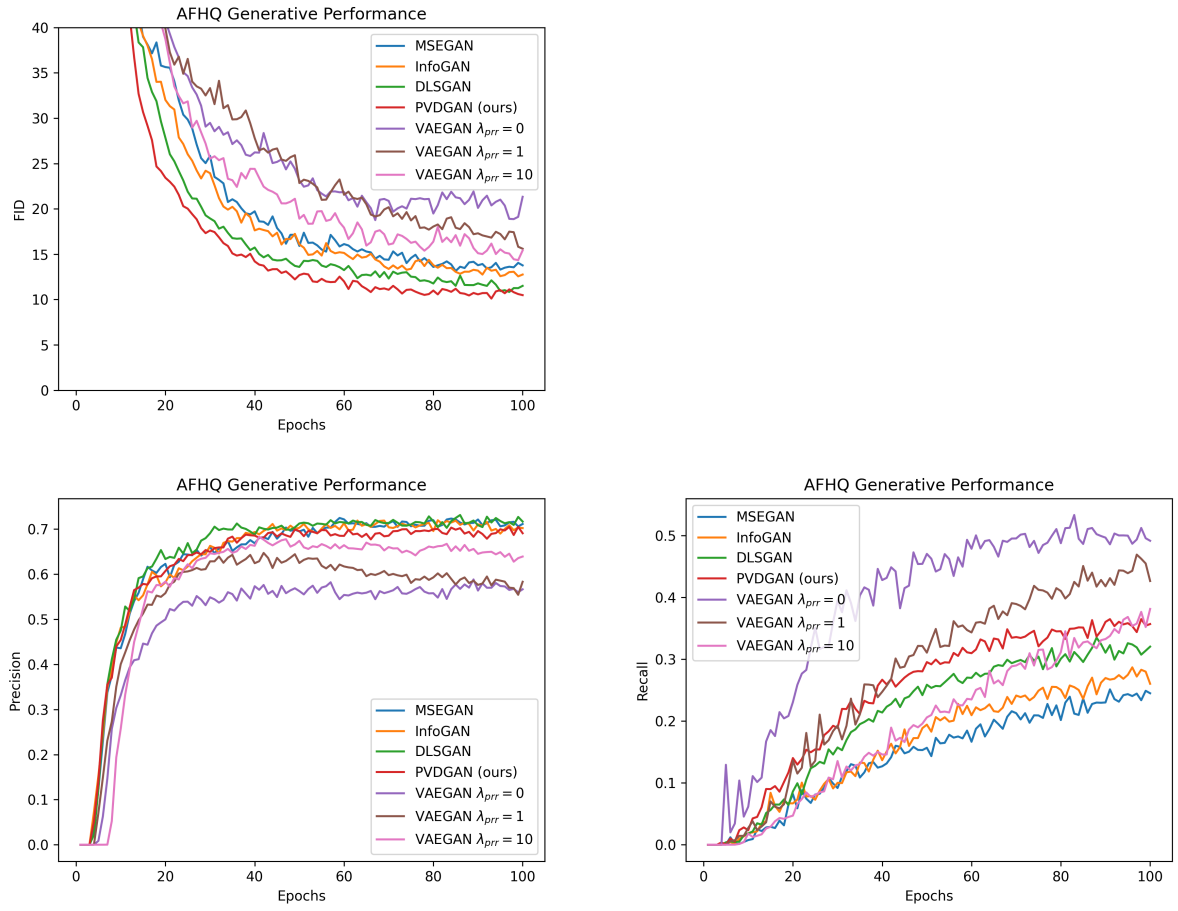

Figure 33: Generative performance graphs in AFHQ experiments.

methods. It shows that PVDGAN could generate more diverse data (recall) without reducing the quality of the data (precision), because it integrated perceptual VAE into GAN.

VAEGAN showed the worst generative performance in both experiments in FID evaluation. This is because Z_x (Eq. 19), which VAEGAN uses to train the GAN, does not follow Z, so the generator is trained with two different random variables. In the FID evaluation (Figs. 32 and 33), the generative performance of PVDGAN increases as λ_{prr} increases. This is because as λ_{prr} increases, Z_x gets closer to Z. In Precision & Recall evaluation, VAEGAN always shows low precision but high recall. VAEGAN uses VAE loss to achieve high recall, but the quality of each data decreased, resulting in low precision. In precision evaluation, especially AFHQ experiments (Fig. 33), the precision of VAEGAN increased when λ_{prr} increased like FID evaluation.

Figs. 34 and 35 show the inversion performance of each method in FFHQ and AFHQ experiments. In Figs. 34 and 35, Fake PSNR and Fake SSIM represent generated image reconstruction performance, and Real PSNR and Real SSIM represent real target image reconstruction performance. In both experiments, PVDGAN showed better inversion performance than MSEGAN, InfoGAN, and DLSGAN. VAEGAN with $\lambda_{prr} = 0$ showed the best real data reconstruction performance (Real PSNR and Real SSIM in Figs. 34 and 35). However, when $\lambda_{prr} = 0$, VAEGAN is equivalent to being trained with a perceptual autoencoder rather than a perceptual VAE. It means that the predicted latent distribution does not have useful features of latent distribution (i.e., each dimension follows an unknown complex distribution rather than simple distributions such as normal or uniform distribution, and each dimension is not independent). It can be seen more clearly through the inversion performance of the model (Fake PSNR and Fake SSIM in Figs. 34 and 35). VAEGAN with $\lambda_{prr} = 0$ has lower generative performance and generated data inversion performance but has higher real data inversion performance than PVDGAN. It indicates that the predicted latent distribution $E(X)$ of VAEGAN with $\lambda_{prr} = 0$ does not follow the latent distribution Z, but follows a complex unknown distribution.

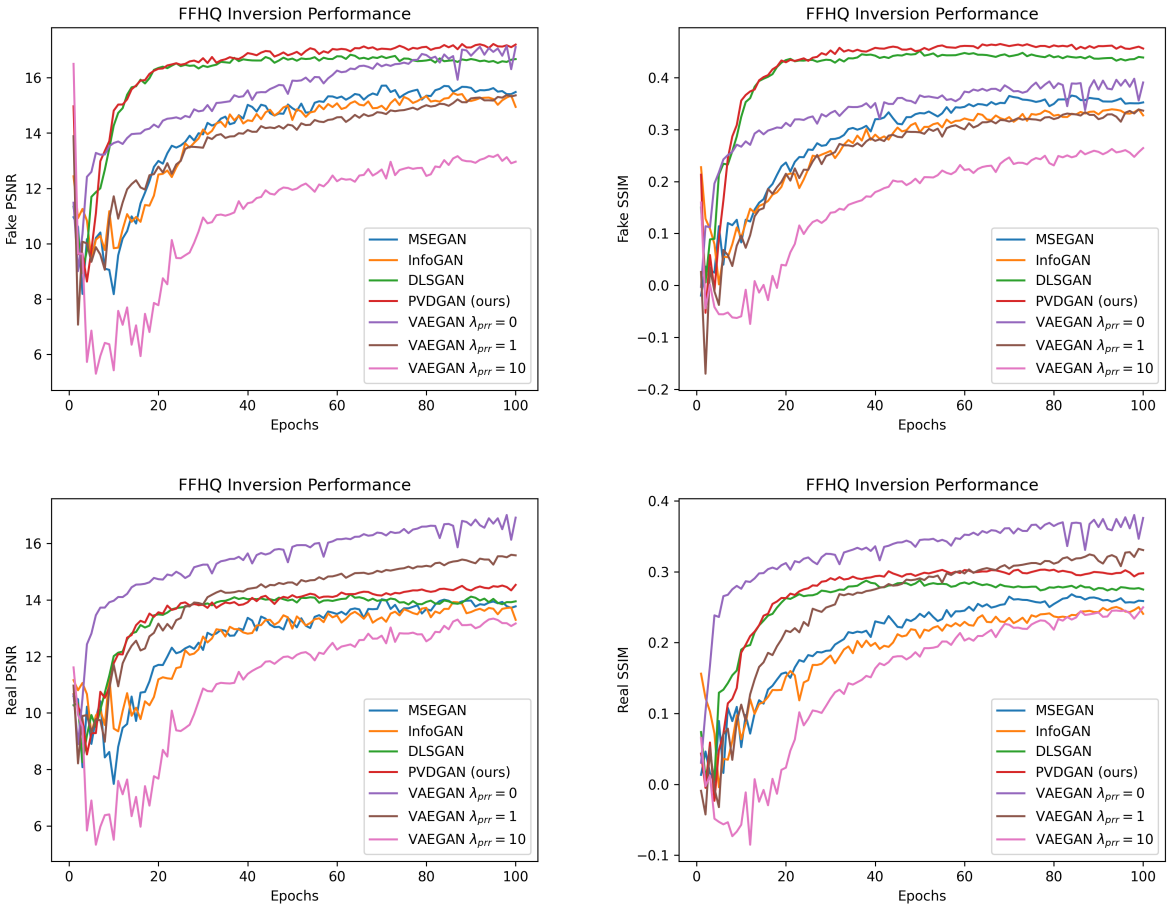

Figure 34: Inversion performance graphs in FFHQ experiments.

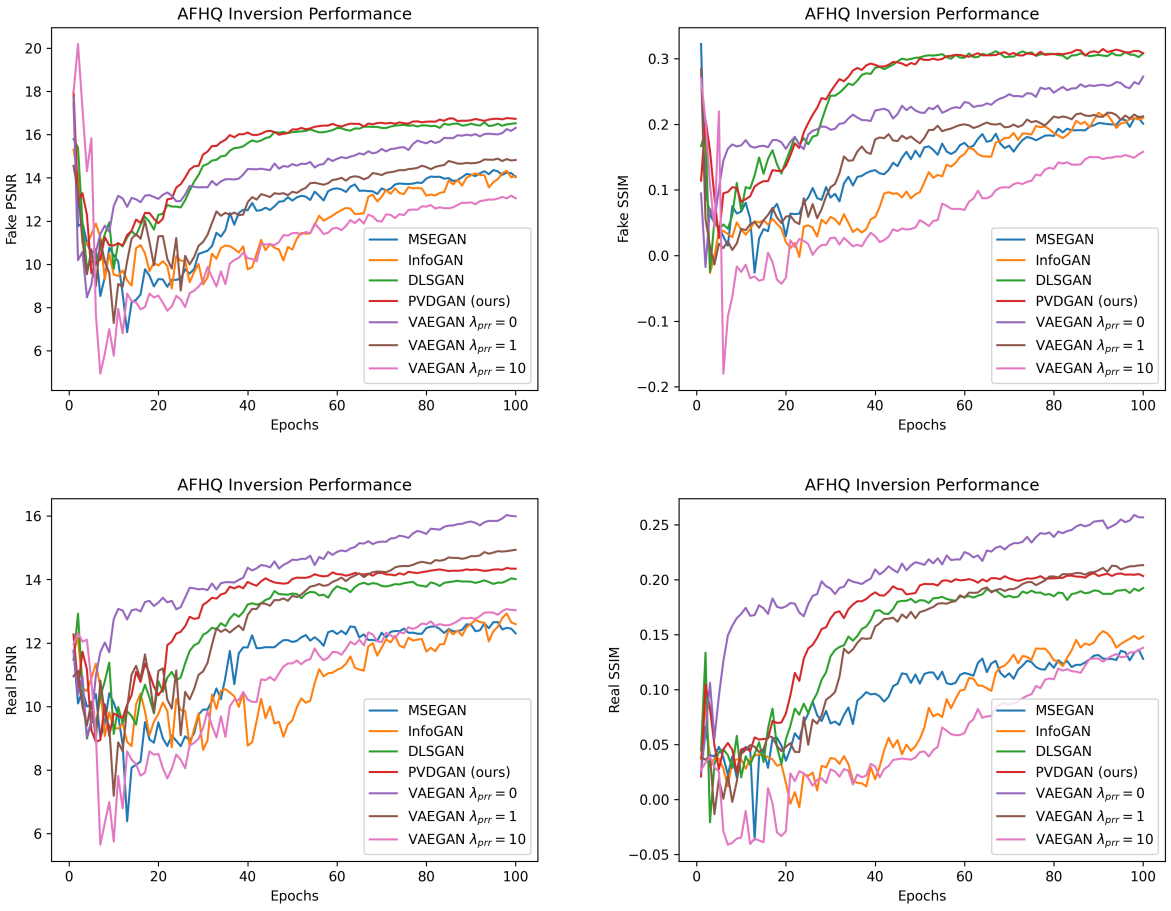

Figure 35: Inversion performance graphs in AFHQ experiments.

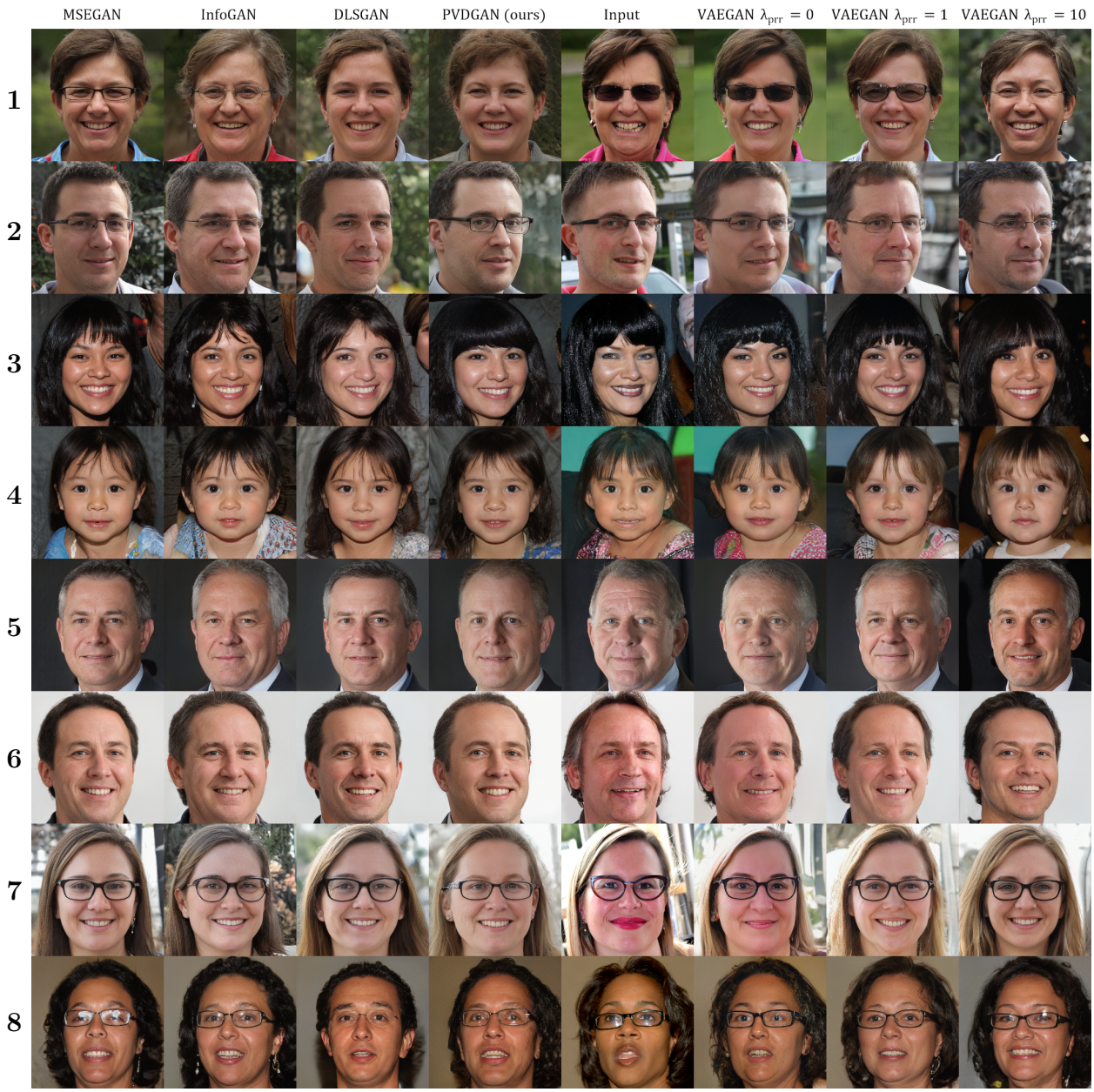

Figure 36: Unseen test image reconstruction examples in FFHQ experiments. Column 5 is input images.

Figure 37: Train image reconstruction examples in AFHQ experiments. Column 5 is input images.

Figs. 36 and 37 show the reconstruction performance of real data and generated data in FFHQ and AFHQ experiments. In both experiments, VAEGAN with low λ_{prr} shows better real data reconstruction than other methods as Fig. 34. However, as previously analyzed, the encoder output of a VAEGAN trained with low λ_{prr} does not follow the latent distribution, and thus does not have the useful properties of generative model inversion (i.e., each dimension is not independent, and each dimension follows complex distribution). On the other hand, VAEGAN's reconstruction performance decreases as λ_{prr} increases in both experiments.

PVDGAN showed better reconstruction performance compared to MSEGAN, InfoGAN, and DLSGAN in some samples in FFHQ experiments (Fig. 36). For example, PVDGAN showed better hair reconstruction in row 3 and background reconstruction in rows 6 and 8 than MSEGAN, InfoGAN, and DLSGAN.

AFHQ reconstruction samples (Fig. 37) show a clear difference between PVDGAN and other methods. In row 3, PVDGAN reconstructs the pattern of the cat better than MSEGAN, InfoGAN, and DLSGAN. Similarly, in row 7, PVDGAN reconstructs the pose of the cheetah better than MSEGAN, InfoGAN, and DLSGAN.

We trained the models to generate two-dimensional Gaussian clusters and MNIST dataset [61]. In Gaussian clusters experiments, we compare the performance of Vanilla GAN [1], InfoGAN [39], Elastic InfoGAN [11], and our proposed CGPGAN. In MNIST experiments, we compared the clustering of CGPGANs according to classifier gradient penalty loss weight λ_{creg}.

4.4.3 Summary

In the fourth contribution, we introduced PVDGAN, a novel method that combines perceptual VAE and DLSGAN to improve GAN inversion performance. When DLSGAN is trained with a normal latent random variable where each dimension is i.i.d., and the latent encoder is integrated into the discriminator, a sum of a predicted latent random variable of real data

and a scaled normal noise follows the normal random variable where each dimension is i.i.d.. Since this random variable is paired with real data and follows the latent random variable, it can be used for both VAE and GAN training. Furthermore, by considering the intermediate layer output of the discriminator as the feature encoder output, the VAE can be trained to minimize the perceptual reconstruction loss. In the experiments in section 4.4.2, we showed that integrating perceptual VAE into DLSGAN improved the generative and inversion performance of DLSGAN. PVDGAN also showed better overall performance than MSEGAN, InfoGAN, and VAEGAN.

4.5 Classifier Gradient Penalty GAN Experiments

In this section, we trained the models to generate two-dimensional Gaussian clusters, MNIST dataset [61], and AFHQ dataset [21]. In Gaussian clusters experiments, we compare the performance of Vanilla GAN [1], InfoGAN [39], Elastic InfoGAN [11], and our proposed CGPGAN. In MNIST and AFHQ experiments, we compared the clustering of CGPGAN according to classifier gradient penalty loss weight λ_{cgp}.

Full codes for Gaussian clusters and MNIST experiments in sections 4.5.1 and 4.5.2 are available at https://github.com/jeongik-jo/CGPGAN. Full codes for AFHQ generation experiments in section 4.5.3 are available at https://github.com/jeongik-jo/CGPGAN-HQ. Full codes for AFHQ generation codebook architecture experiments in section 4.5.4 are available at https://github.com/jeongik-jo/CGPGAN_codebook.

4.5.1 Gaussian Clusters Experiments

In Gaussian clusters experiments, we used the dataset consisting of four 2-dimensioanl Gaussian clusters as a training dataset. The left plot of Fig. 1 shows the target data distribution used in the experiments. One can see that there are four Gaussian clusters with different probabilities in data distribution. The generator, discriminator, and classifier consisting of four fully connected hidden layers with 512 units were used for training. The following

hyperparameters were used for experiments.

$$
\begin{gathered}
Z \sim N(0, I_{256}) \\
optimizer = AdamW \begin{pmatrix} learning\ rate = 0.0001 \\ weight\ decay = 0.0001 \\ \beta_1 = 0.0 \\ \beta_2 = 0.99 \end{pmatrix} \\
batch\ size = 16 \\
\lambda_{r2} = 1 \\
train\ step\ per\ epoch = 2000 \\
epoch = 100 \\
activation\ function = Leaky\ ReLU
\end{gathered}
$$

λ_{r2} represents R2 regularization [7] loss weight ($\lambda_{r2} L_{r2} = \lambda_{r2} \mathbb{E}_{z,c_f} \left[\| \nabla_{G(z,c_f)} (D(G(z, c_f)) \cdot c_f) \|_2^2 \right]$).

NSGAN loss [1] was used for adversarial loss. Classification loss weight $\lambda_{cls} = 1.0$ was used for InfoGAN, elastic InfoGAN, and CGPGAN. We used exponential moving average with $decay\ rate = 0.999$ as $update$ function for CGPGAN. In InfoGAN, Elastic InfoGAN, and CGPGAN, classifier Q and discriminator D do not share hidden layers. Equalized learning rate [8] was used for all trainable weights. Also, exponential moving average [8] with $decay\ rate = 0.999$ was used for generator weights. In CGPGAN and Elastic InfoGAN, $d_c = 16$ was used, and $P(C)$ was updated after epoch 30 (i.e., in CGPGAN, *early in training* in line 9 of algorithm 5 was $True$ until epoch 30). Since we assumed that there is no good metric to measure the distance between data, we did not use identity preserving transformations in Elastic InfoGAN. Only gradient descent on a categorical latent distribution with Gumbel softmax was used for Elastic InfoGAN.

Fig. 38 shows samples generated with vanilla GAN (trained only with adversarial loss). The left plot of Fig. 38 shows data generated by a vanilla GAN with a one-dimensional categorical latent distribution. Since there is no discrete dimension in the latent distribution, the vanilla GAN generates a lot of samples between clusters. This shows that when training a

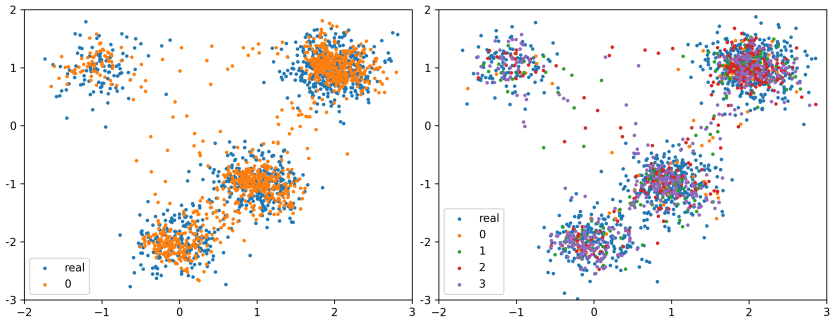

Figure 38: Vanilla GAN (trained only with adversarial loss) is trained with both continuous latent distribution and categorical latent distribution. Left plot: 1-dimensional categorical latent distribution ($P(C) = [1.0]$). Right plot: 4-dimensional optimal categorical latent distribution ($P(C) = [0.1, 0.2, 0.3, 0.4]$).

GAN with only continuous latent vectors, the latent space is entangled and it is not suitable for representing data with concave parts with low density.

The right plot of Fig. 38 shows data generated by a vanilla GAN trained with optimal categorical latent distribution ($P(C) = [0.1, 0.2, 0.3, 0.4]$). One can see that the generator of vanilla GAN did not use the information of categorical latent distribution, and training was exclusively performed only with a continuous latent distribution. Therefore, the generator output was also continuous, which caused a sample generation between each cluster. This means that even if the generator takes a discrete categorical latent distribution as input, additional loss is required to make the generator use categorical latent distribution meaningfully.

Fig. 39 shows data generated by InfoGAN trained with the optimal categorical latent distribution. Unlike the Vanilla GAN, one can see that the model generates class-conditional distribution with the categorical latent distribution. However, one can still see the problems of InfoGAN in this figure.

The first problem was that even though the categorical latent distribution was optimal, most class was not mapped to the correct cluster. We repeated the InfoGAN training eight times, and some classes were assigned correctly in some iterations, but never all classes were assigned correctly. For example, in the second and fourth plots, class 3 was assigned correctly.

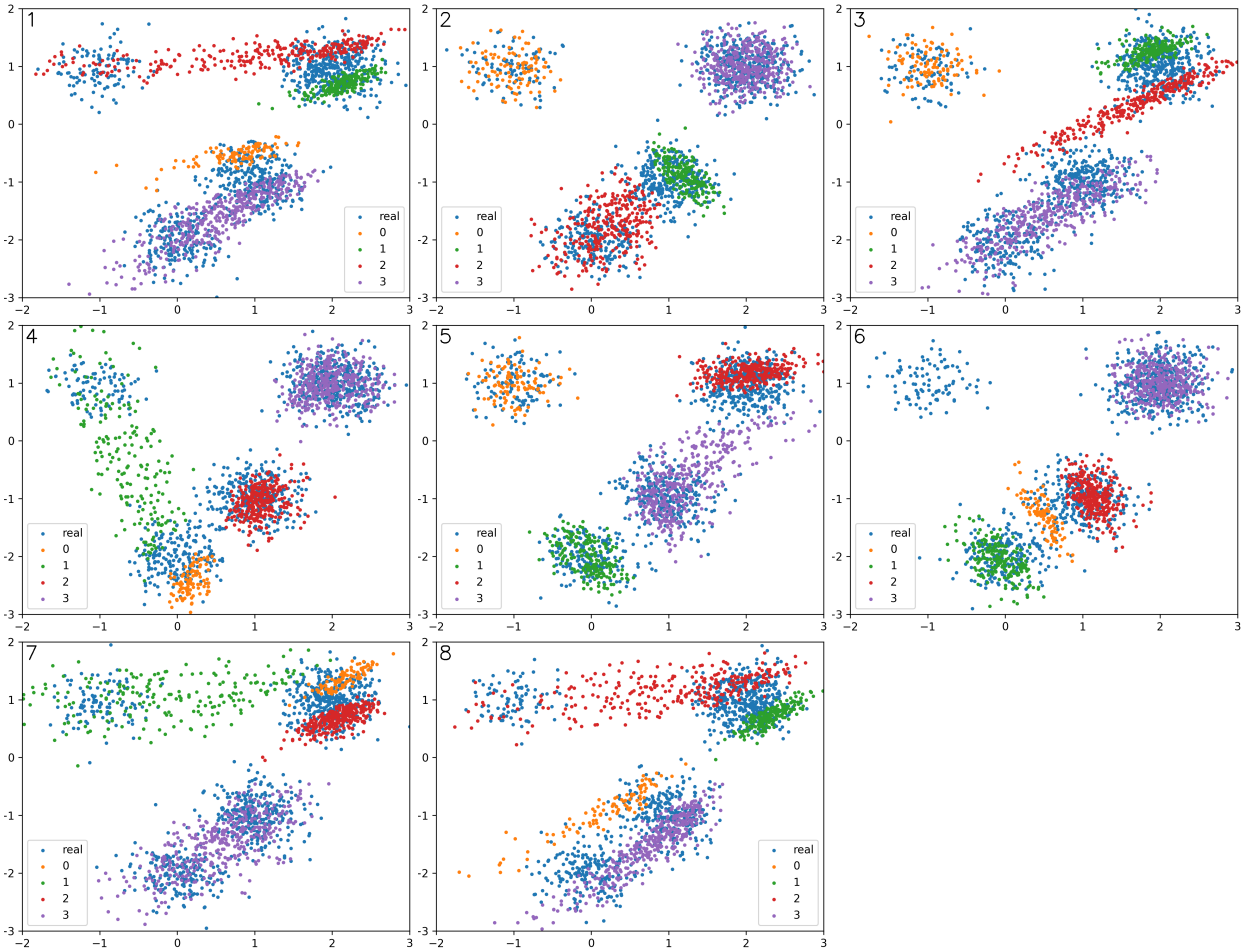

Figure 39: InfoGAN trained with 4-dimensional optimal categorical latent distribution ($P(C) = [0.1, 0.2, 0.3, 0.4]$). Eight times repeated.

In the fifth plot, classes 0 and 1 were assigned correctly. In the sixth plot, classes 1 and 3 were assigned correctly. However, there was never an iteration when all four classes were assigned correctly. This is because when the decision boundary of the InfoGAN classifier is not initialized ideally, the decision boundary of the classifier converges to local optima, resulting in inaccurate cluster assignments. And since the probability of the classifier's decision boundary being ideally initialized is very low, InfoGAN was not able to assign all classes correctly.

The second problem is that the generator does not generate data near the decision boundary of the classifier. In Fig. 39, one can see that the generator of InfoGAN does not generate data near the decision boundary of the classifier. This is because classification loss and adversarial loss conflict with the generator of InfoGAN.

Fig. 40 shows data generated by elastic InfoGAN. We tested several combinations of hyperparameters (temperature t and learning rate for the categorical latent distribution lr), but Elastic InfoGAN could not generate class-conditional data correctly.

Fig. 41 shows samples generated by CGPGAN trained with different λ_{cgp}. In the first and second plots of Fig. 41, each category is assigned to each cluster correctly. Because the probability density function of the generated data distribution is smooth due to the simplicity of the data and small model, each class is assigned correctly, even though there was no classifier gradient penalty in the first plot. Also, the probability of each category is very accurate, and there was a natural division between clusters, unlike InfoGAN. This is because CGPGAN's generator is only trained with adversarial losses of CAGAN, not classification loss, so there is no conflict between those losses. In the third and fourth plots, multiple clusters were assigned to the same category. This is because λ_{cgp} was too high, causing the classifier to converge to local optima in a wide region. This shows that CGPGAN can adjust the sensitivity of each category via λ_{cgp}.

Fig. 42 shows the results of eight iterations of CGPGAN training with $\lambda_{cgp} = 0.1$. One can see that CGPGAN is generating class-conditional data correctly over multiple iterations.

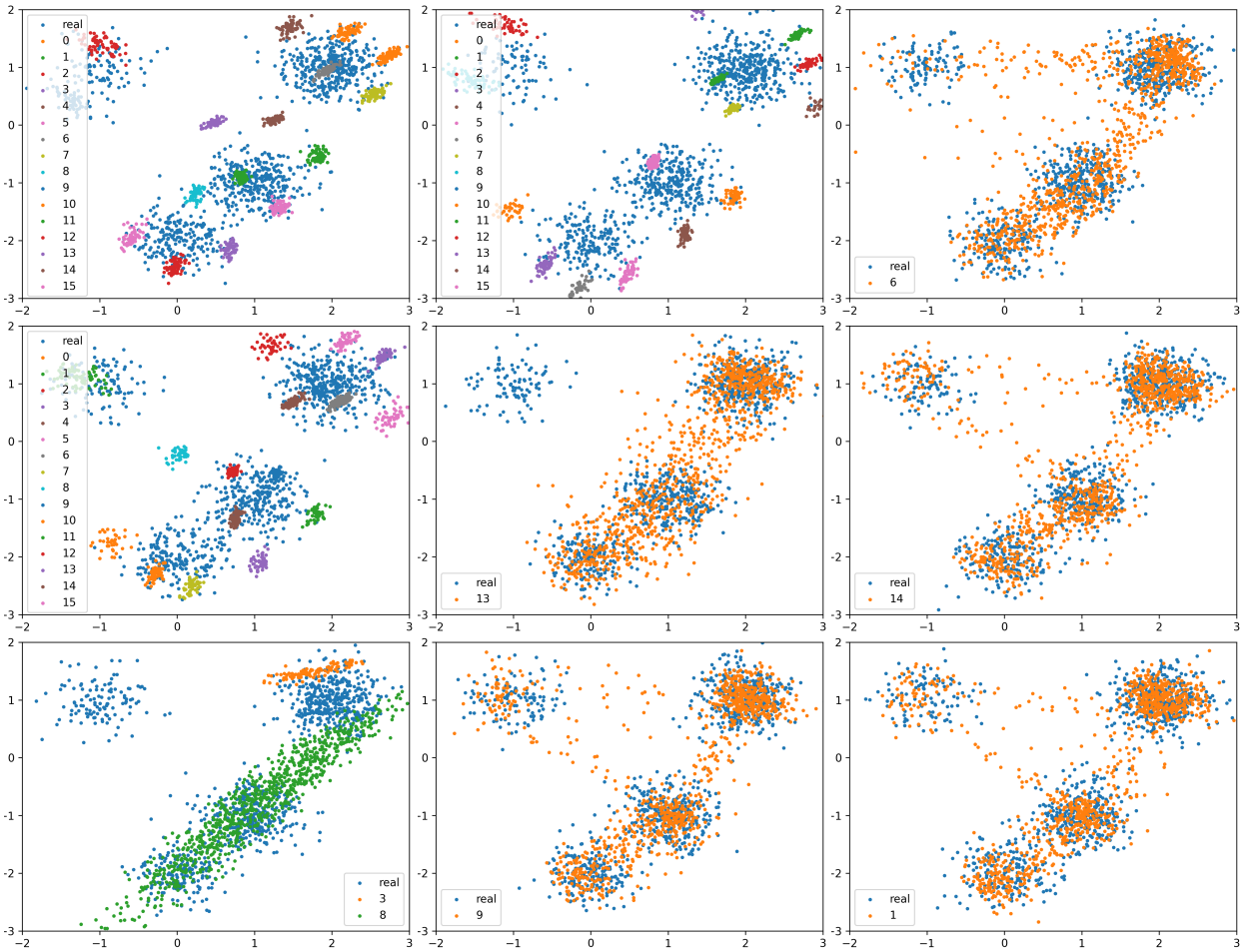

Figure 40: Elastic InfoGAN trained with different temperatures and learning rates. Contrastive loss was not used. Row 1: $t = 0.1$, Row 2: $t = 0.3$, Row 3: $t = 1.0$, Column 1: $lr = 0.0001$, Column 2: $lr = 0.0003$, Column 3: $lr = 0.001$. Categories with a probability of less than 1% were omitted.

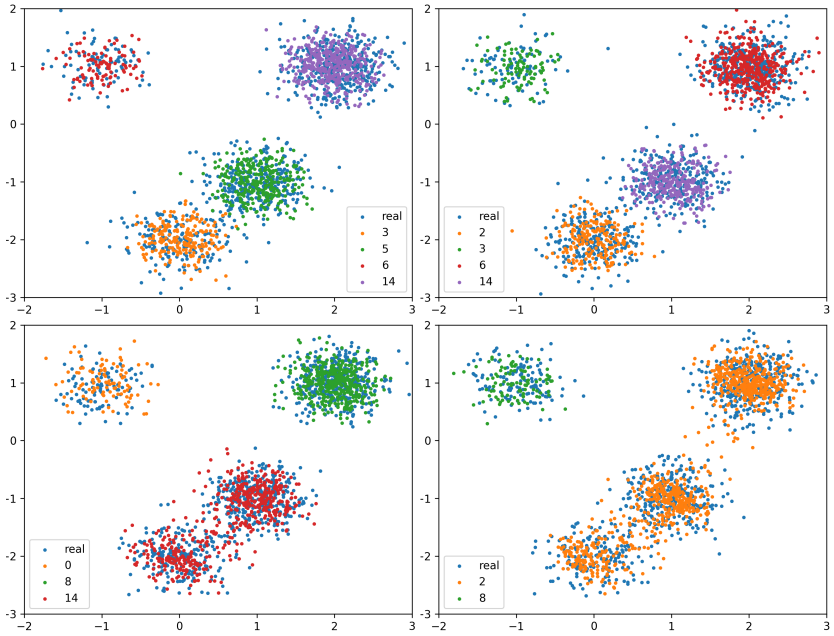

Figure 41: CGPGAN trained with different λ_{cgp}. Categories with a probability of less than 1% were omitted. Top left plot: $\lambda_{cgp} = 0.0$. Categorical probability after training was $P(C) = [0.2003, 0.3029, 0.0994, 0.3973]$. Top right plot: $\lambda_{cgp} = 1.0$ Categorical probability after training was $P(C) = [0.1973, 0.0956, 0.4054, 0.3018]$. Bottom left plot: $\lambda_{cgp} = 10.0$. Categorical probability after training was $P(C) = [0.0992, 0.4002, 0.5006]$. Bottom right plot: $\lambda_{cgp} = 100.0$. Categorical probability after training was $P(C) = [0.9002, 0.0998]$.

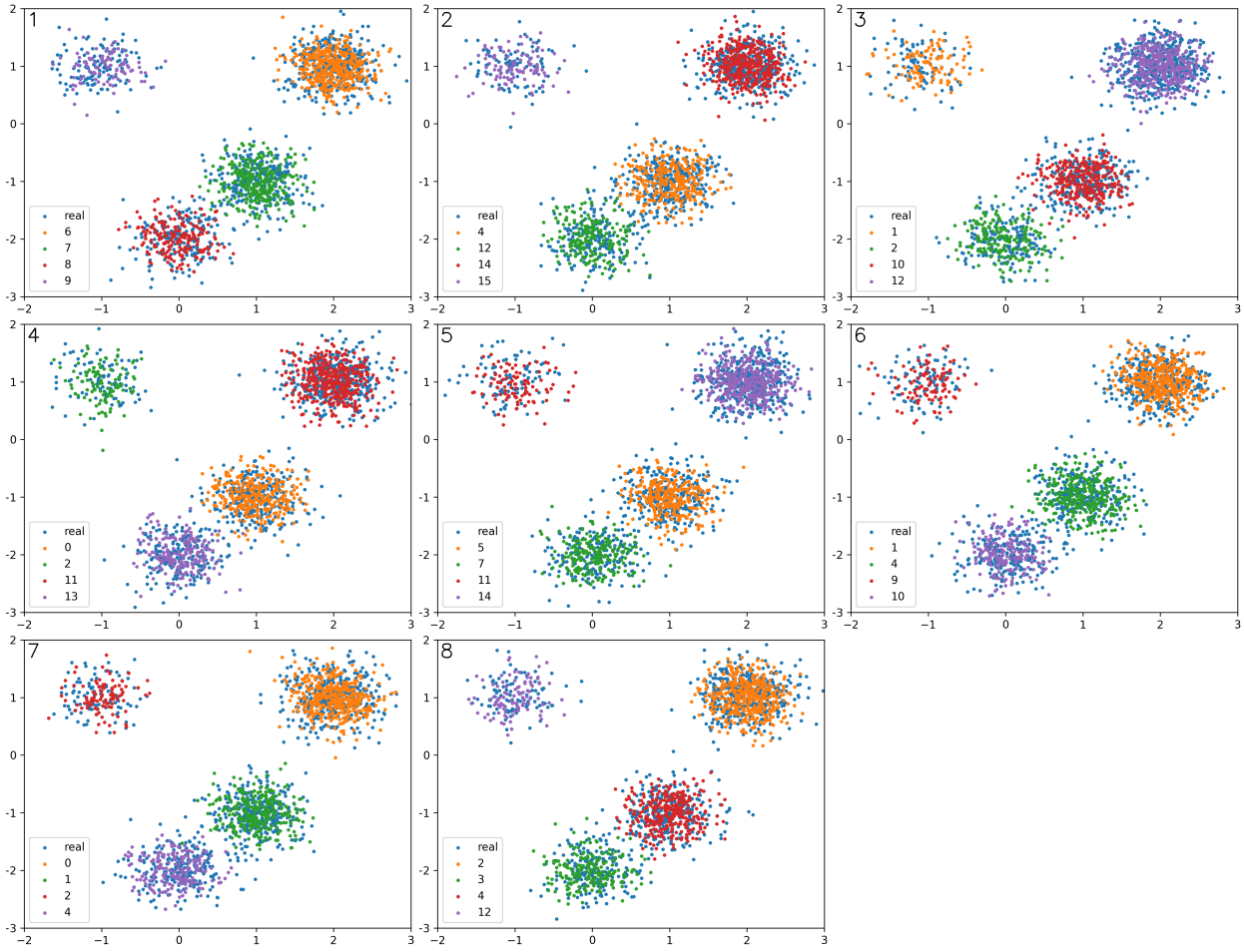

Figure 42: CGPGAN trained with $\lambda_{cgp} = 0.1$. Categories with a probability of less than 1% were omitted. Eight times repeated. Each run converges differently because the model's weights are initialized differently for each run, and latent vectors for training are different each time. The numbers in the upper left of each plot indicate the number of each experiment number.

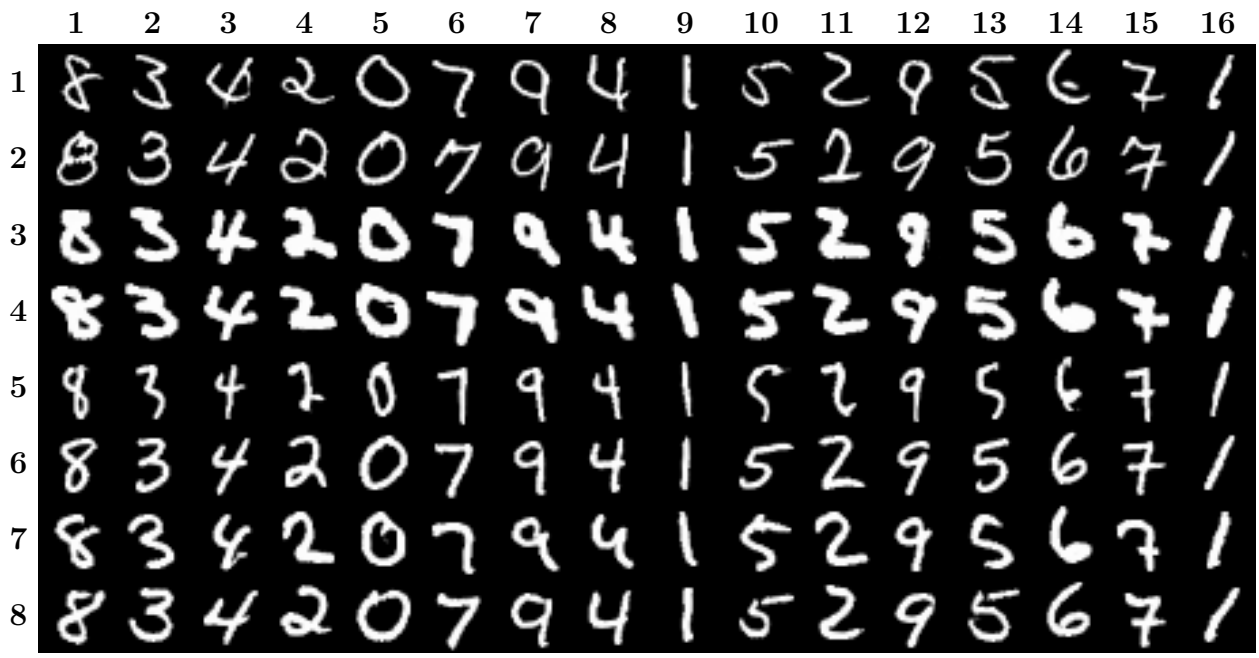

Figure 43: MNIST generated data with $\lambda_{cgp} = 50$. Each row has the same continuous latent vector, and each column has the same categorical latent vector. Out of d_c categories, those with a probability less than 1% were omitted. The probabilities for each category are $[0.0975, 0.1059, 0.0384, 0.0740, 0.0974, 0.0902, 0.0395, 0.0601, 0.0657, 0.0385, 0.0274, 0.0563, 0.0504, 0.1002, 0.0138, 0.0447]$. FID: 1.3140, precision: 0.8158, recall: 0.6763.

4.5.2 MNIST Experiments

In this experiment, we trained CGPGAN to generate the MNIST handwritten digits dataset [61]. The MNIST dataset consists of 10 digits from 0 to 9, with each digit representing about 10% of the total.

The generator, discriminator, and classifier simply consist of CNNs. $learning\ rate = 0.001$, $d_c = 32$, $epoch = 300$ were used for the experiments. The categorical latent distribution of CGPGAN was updated after epoch 100. Other hyperparameters are the same as in section 4.5.1. We used FID [9], precision & recall [10] for generative performance evaluation. All evaluation methods used the Inception model. 32k training samples were used for generative performance evaluation.

Figs. 43-45 show the difference in class-conditional data generation of CGPGAN according to λ_{cgp}.

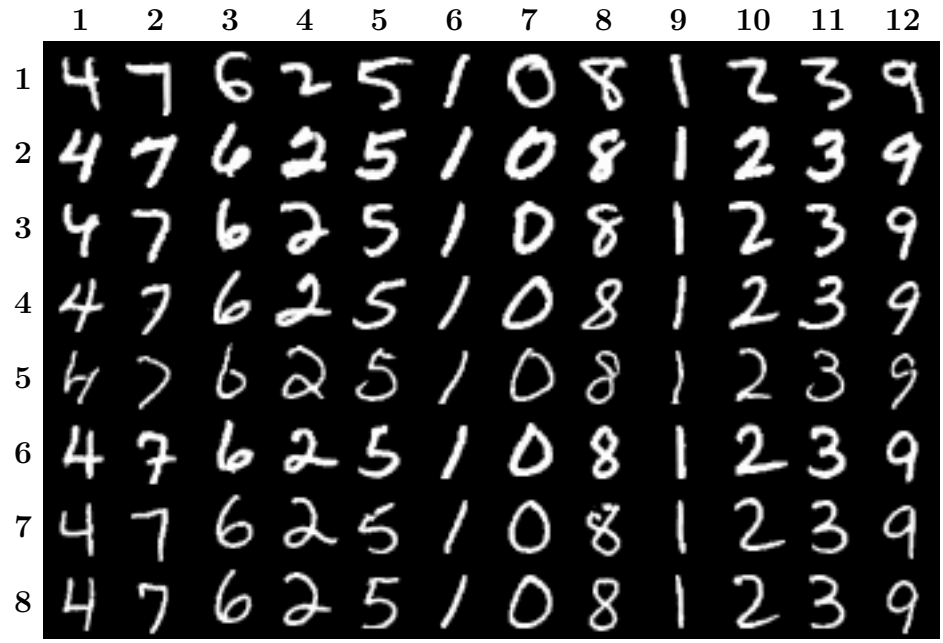

Figure 44: MNIST generated data with $\lambda_{cgp} = 70$. Each row has the same continuous latent vector, and each column has the same categorical latent vector. Out of d_c categories, those with a probability less than 1% were omitted. The probabilities for each category are $[0.0975, 0.1051, 0.0993, 0.0765, 0.0901, 0.0454, 0.0988, 0.0964, 0.0668, 0.0270, 0.1012, 0.0957]$. FID: 1.4724, precision: 0.8122, recall: 0.6728.

First, in Fig. 43, since λ_{cgp} was too low, the classifier decision boundary converged on a local optima in a narrow region. Thus, some digits were split into multiple categories. In Fig. 43, the digit 2 was divided into categories 4 (column 4) and 11 (column 11), and digit 7 was divided into categories 6 and 15. However, the same digit splitting into multiple categories does not mean that CGPGAN performed an incorrect class-conditional data generation. If we ignore the human knowledge of each digit, digit 2 with a loop and without a loop, and digit 7 with a horizontal line in the center and without a horizontal line can be considered different categories. One can see that the digit 2 in category 4 has a loop, but category 11 has no loop. And the sum of the probabilities of those two categories is $0.074 + 0.0274 = 0.1014$, which is similar to the proportion of the digit 2 in the MNIST digits dataset (about 10%). Likewise, the digit 7 in category 6 does not have a horizontal line, but category 15 has it. The sum of the probability of those two categories is $0.0902 + 0.0138 = 0.1040$, which is about 10%. These splits show that the optimal clustering (and class-conditional data generation)

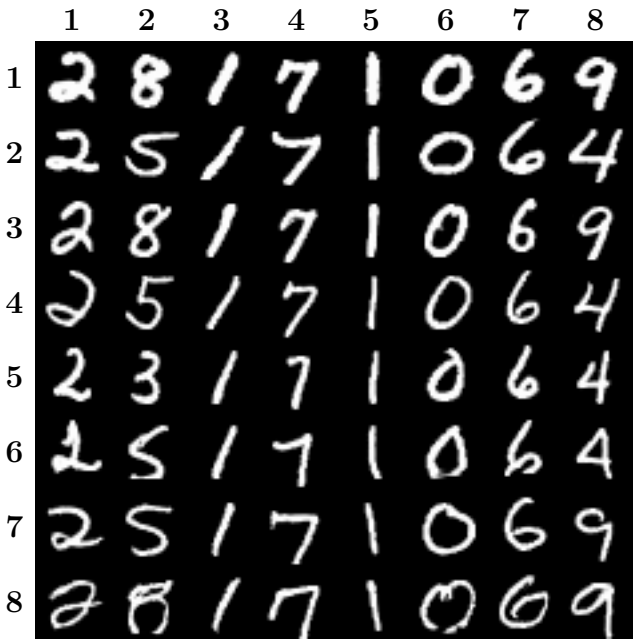

Figure 45: MNIST generated data with $\lambda_{cgp} = 120$. Each row has the same continuous latent vector, and each column has the same categorical latent vector. Out of d_c categories, those with a probability less than 1% were omitted. The probabilities for each category are $[0.1013, 0.2940, 0.0491, 0.0908, 0.0586, 0.1004, 0.0947, 0.2110]$. FID: 1.6374, precision: 0.8076, recall: 0.6734.

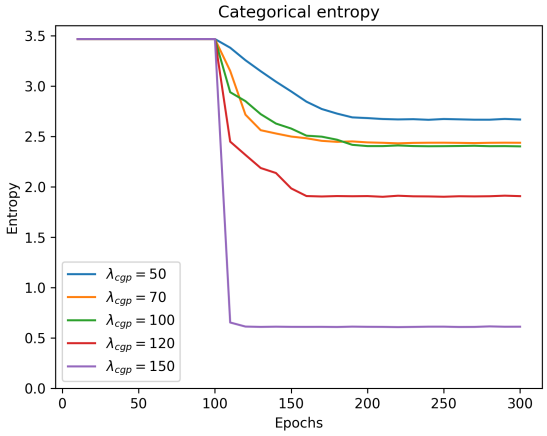

Figure 46: The entropy of a categorical latent distribution over epochs in MNIST experiments.

can depend on the sensitivity of each cluster.

As λ_{cgp} increases, the classifier decision boundary converges to the local optima over a wider region. In Fig. 45, $\lambda_{cgp} = 120$ was used for training. One can see that multiple digits are clustered into one category, except for digit 1. For example, digits 3, 5, and 8 were clustered in category 2, and digits 4 and 9 were clustered in category 8. It means that the distance between those digits is closer than other digits for the model. The digit 1 was divided into two categories based on whether it was tilted or not in Figs. 43, 44, 45. It means that for the model, it is easy to distinguish between a tilted digit 1 and a vertical digit 1.

Fig. 46 shows the entropy of a categorical latent distribution over epochs. The entropy was calculated every 10 epochs. In Fig. 46, one can observe that as λ_{cgp} increases, the lower the entropy of the categorical latent distribution. One can also see that the decreased entropy converges stably. This shows that CGPGAN can adjust the sensitivity of each category via λ_{cgp}.

In this experiment, we showed that CGPGAN can properly generate the class-conditional data of the MNIST handwritten digit by adjusting λ_{cgp}. In particular, when λ_{cgp} is low so the sensitivity of each cluster is high, CGPGAN found that there are different patterns within some digits (e.g., digits 1, 2, and 7 in Fig. 43). When λ_{cgp} is high so the sensitivity of each cluster is low, CGPGAN found that some digits have a similar pattern (e.g., digits 3, 5, 8, and 4, 9). This means that CGPGAN can generate class-conditional data by adjusting the sensitivity of each cluster according to λ_{cgp}. Separately, all three CGPGANs have good unconditional generative performance from FID and precision & recall.

4.5.3 AFHQ Experiments

In this section, we trained CGPGAN to generate AFHQ dataset [21]. The AFHQ dataset consists of animal face images. We resized the images to 256×256 resolution and used them for training.

The following hyperparameters were used to train the model.

$$Z \sim N(0, I_{1024})$$

$$optimizer = AdamW \begin{pmatrix} learning\ rate = 0.003 \\ weightdecay = 0.0001 \\ \beta_1 = 0.0 \\ \beta_2 = 0.99 \end{pmatrix}$$

$$batch\ size = 8$$

$$\lambda_{r2} = 10.0$$

$$epoch = 150$$

$$d_c = 16$$

$$decay\ rate = 0.999$$

The model simply consists of CNNs with skip connections, and $P(C)$ was updated after epoch 50. Other settings are the same as section 4.5.1.

Figs. 47-49 show generated samples of CGPGAN trained with the AFHQ dataset. In Figs. 47-49, each row has the same continuous latent vector, and each column has the same categorical latent vector. One can see that each column in the figures has similar patterns. For example, in Fig. 48, animals in the third column have black noses. Animals in the fourth and sixth columns have white noses with black mask patterns, and the fifth column has black fur. The other figure's animal faces also show a similar pattern if they are in the same column.

Fig. 50 shows the entropy of categorical latent distribution in the AFHQ experiment. In the AFHQ experiment, categorical entropy did not consistently decrease as λ_{cgp} increased. This is because the dataset is high dimensional and complex, so there are many possible converged patterns of the decision boundaries. Thus, the initial separation by the classifier has a strong influence on the converged decision boundary.

Nevertheless, one can still see that the categorical entropy converges stably. Except when $\lambda_{cgp} = 180$, the categorical entropy of C decreases from epoch 51, when $P(C)$ starts to be updated, to approximately epoch 100. After that, the categorical entropy doesn't change

Figure 47: AFHQ dataset generated with CGPGAN when $\lambda_{cgp} = 80$. Each row has the same continuous latent vector, and each column has the same categorical latent vector. Out of d_c categories, those with a probability less than 1% were omitted. The probabilities for each category are $[0.1022, 0.4298, 0.0521, 0.2549, 0.0837, 0.0240, 0.0400]$. FID: 9.9898, precision: 0.7296, recall: 0.3059.

Figure 48: AFHQ dataset generated with CGPGAN when $\lambda_{cgp} = 120$. Each row has the same continuous latent vector, and each column has the same categorical latent vector. Out of d_c categories, those with a probability less than 1% were omitted. The probabilities for each category are $[0.5311, 0.2330, 0.0144, 0.0150, 0.0139, 0.0160, 0.1660]$. FID: 10.4323, precision: 0.7300, recall: 0.3051.

Figure 49: AFHQ dataset generated with CGPGAN when $\lambda_{cgp} = 160$. Each row has the same continuous latent vector, and each column has the same categorical latent vector. Out of d_c categories, those with a probability less than 1% were omitted. The probabilities for each category are $[0.0141, 0.0730, 0.0968, 0.0792, 0.0911, 0.0148, 0.6268]$. FID: 9.7840, precision: 0.7397, recall: 0.3219.

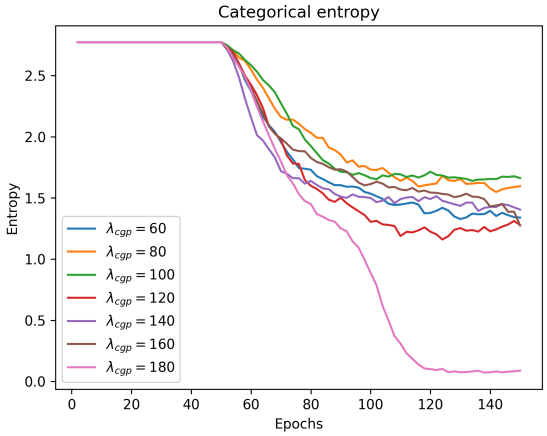

Figure 50: The entropy of a categorical latent distribution over epoch in AFHQ experiments.

much. This shows that the decision boundary of CGPGAN has stably converged to the local optima. For $\lambda_{cgp} = 180$, all categories were merged into single category around epoch 120.

4.5.4 Classifier Gradient Penalty GAN with Codebook Experiments

In this section, we compared the performance of Vanilla GAN and CGPGAN with and without codebook architecture. AFHQ dataset [21] resized to 256×256 resolution was used as the training dataset.

Models are trained with NSGAN adversarial loss [1] (i.e., A_d and A_g) with R2 regularization [7]. Equalized learning rate [8] was used for all trainable parameters. Also, an exponential moving average of generator parameters with $decay\ rate = 0.999$ was used for generative performance evaluation. The model architecture is simply composed of CNNs.

Vanilla GAN is a GAN that simply uses only adversarial losses.

In without codebook architecture, the d_z-dimensional continuous latent vector and $d_l \times d_c$ shape multiple categorical latent vectors are the input to the generator, where d_l and d_c represent label dimension and category dimension. The label represents the number of categorical vectors. That is, for d_l labels d_c categories distribution, the class-conditional vector consists of d_l one-hot vectors, where the dimension of each one-hot vector is d_c. Then the input latent vector is projected to the $4 \times 4 \times 1024$ shape feature maps.

In with codebook architecture, d_z-dimensional continuous latent vector is projected to $4 \times 4 \times 512$ shape feature maps, then $4 \times 4 \times 512$ shaped page vectors are concatenated. The codebook is $d_l \times d_c \times d_p$ shape trainable matrix, where d_p represents the dimension of the page vector. It means that there are $d_l \times d_c$ page vectors, and $d_p = 4 \times 4 \times 512/d_l$.

The following hyperparameters were used for the experiments:

$$
\begin{gathered}
Z \sim U(-\sqrt{3}, \sqrt{3})^{1024} \\
optimizer = AdamW \begin{pmatrix} learning\ rate = 0.003 \\ weight\ decay = 0.0001 \\ \beta_1 = 0.0 \\ \beta_2 = 0.99 \end{pmatrix} \\
batch\ size = 8 \\
\lambda_{r2} = 10 \\
epoch = 150 \\
activation\ function = Leaky\ ReLU
\end{gathered}
$$

For CGPGAN, $P(C)$ starts to be updated after epoch 50 (i.e., the *probnormalize* function is disabled from epoch 50), and $\lambda_{cls} = 1, \lambda_{cgp} = 10$ were used for model training.

FID [9] and precision & recall [10] were used for generative performance evaluation.

Figs. 51-54 show the generated samples of Vanilla GAN and CGPGAN, with and without codebook architecture when label dimension $d_l = 4$ and categorical vector dimension $d_c = 4$. In Figs. 51-54, each row has the same continuous latent vector, and each column has the same categorical latent vector. The categorical vector corresponding to each column was sampled from a categorical latent distribution.

Fig. 51 shows generated samples of Vanilla GAN without codebook architecture. Thus, the categorical latent vector is concatenated with the continuous latent vector and input into the generator. In Fig. 51, changes in the categorical latent vector changed some features of the samples. For example, in row 2, the categorical vector determines whether the animal is a cat or a tiger. In row 4, the categorical vector determines the pattern of the cat. However,

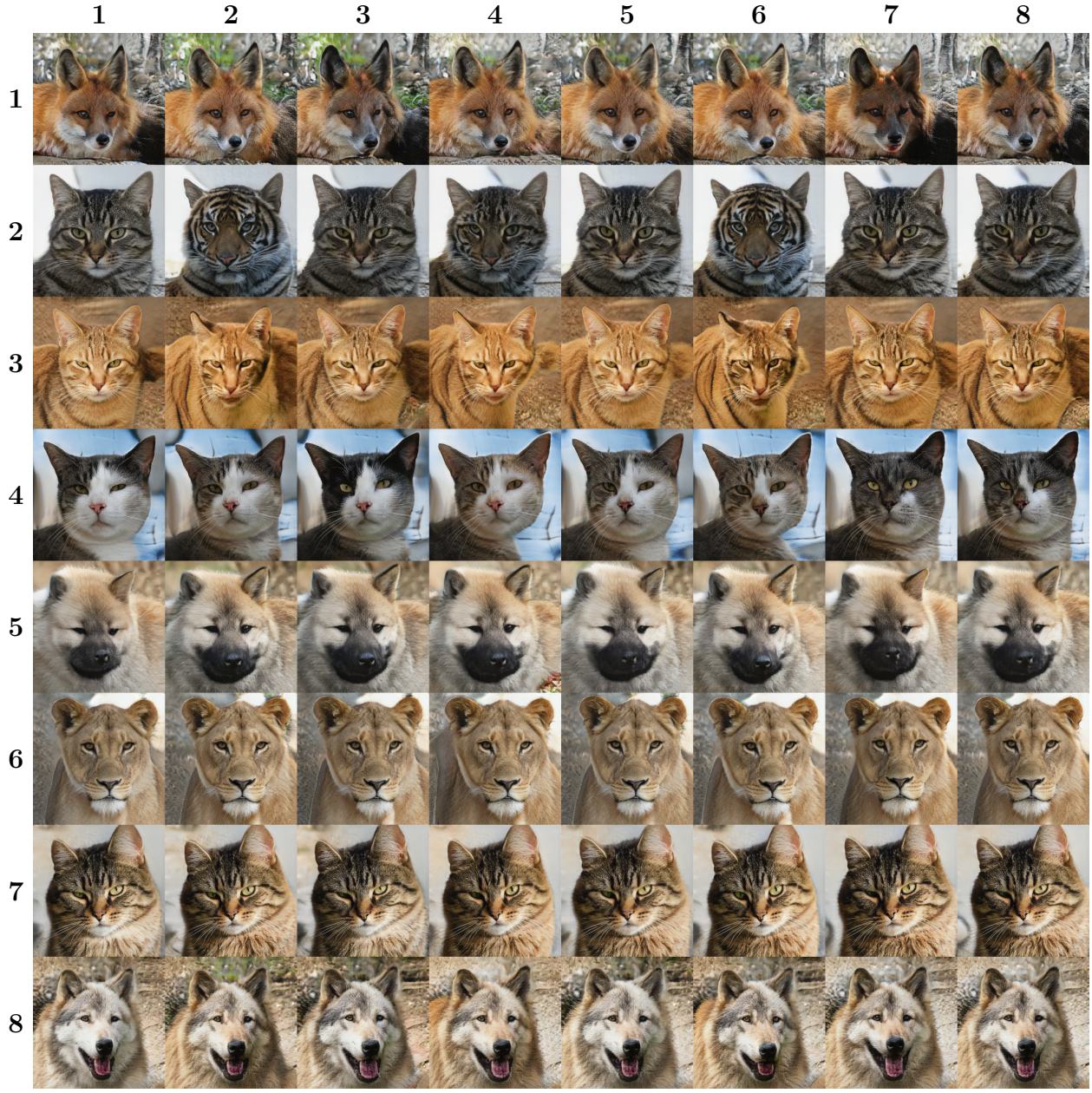

Figure 51: Generated samples of Vanilla GAN without codebook architecture. Each row has the same continuous latent vector, and each column has the same categorical latent vector. $d_l = 4$, $d_c = 4$. FID: 10.6690, precision: 0.7349, recall: 0.2742.

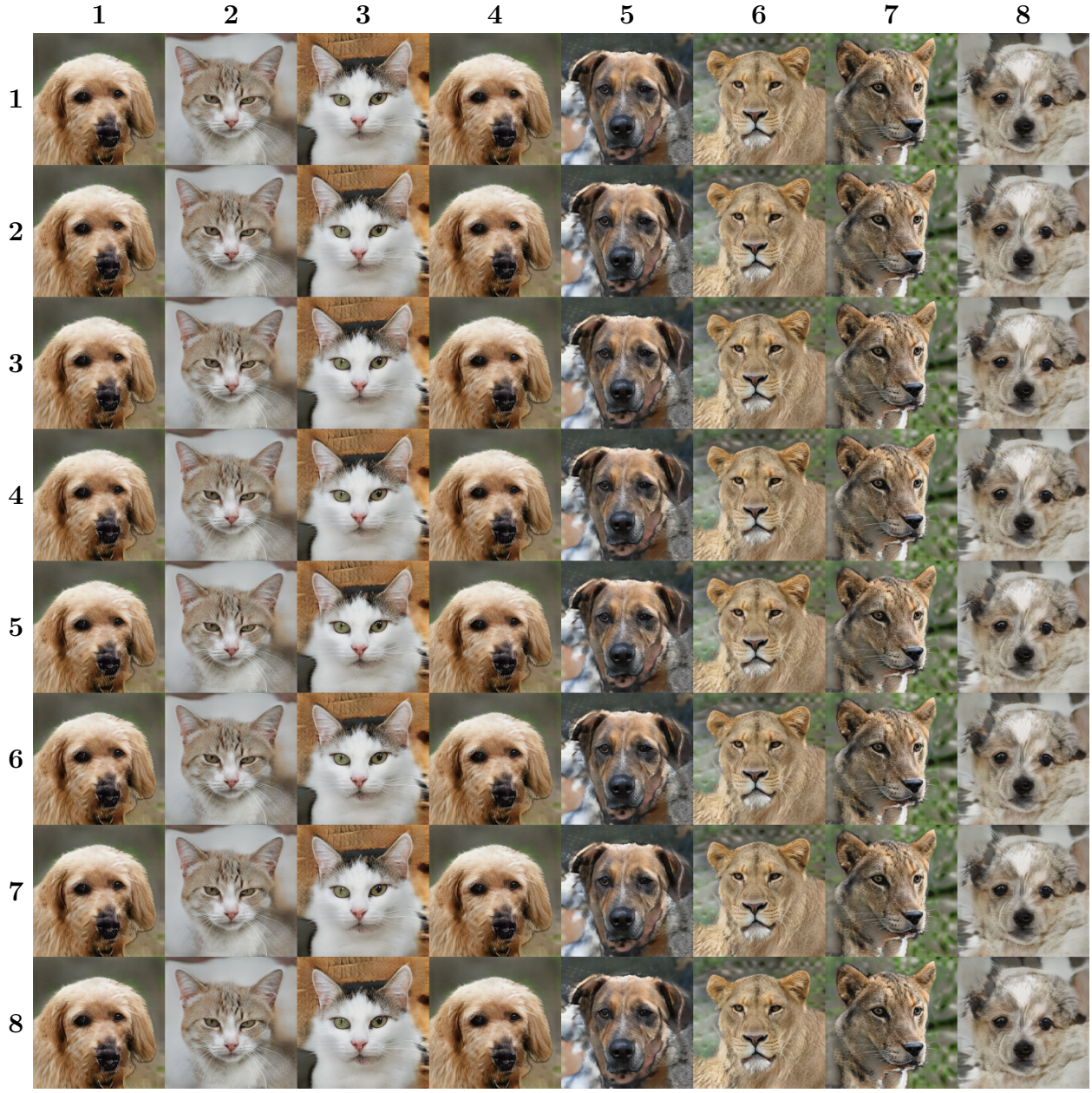

Figure 52: Generated samples of Vanilla GAN with codebook architecture. Each row has the same continuous latent vector, and each column has the same categorical latent vector. $d_l = 4$, $d_c = 4$. FID: 74.2704, precision: 0.6324, recall: 0.0000.

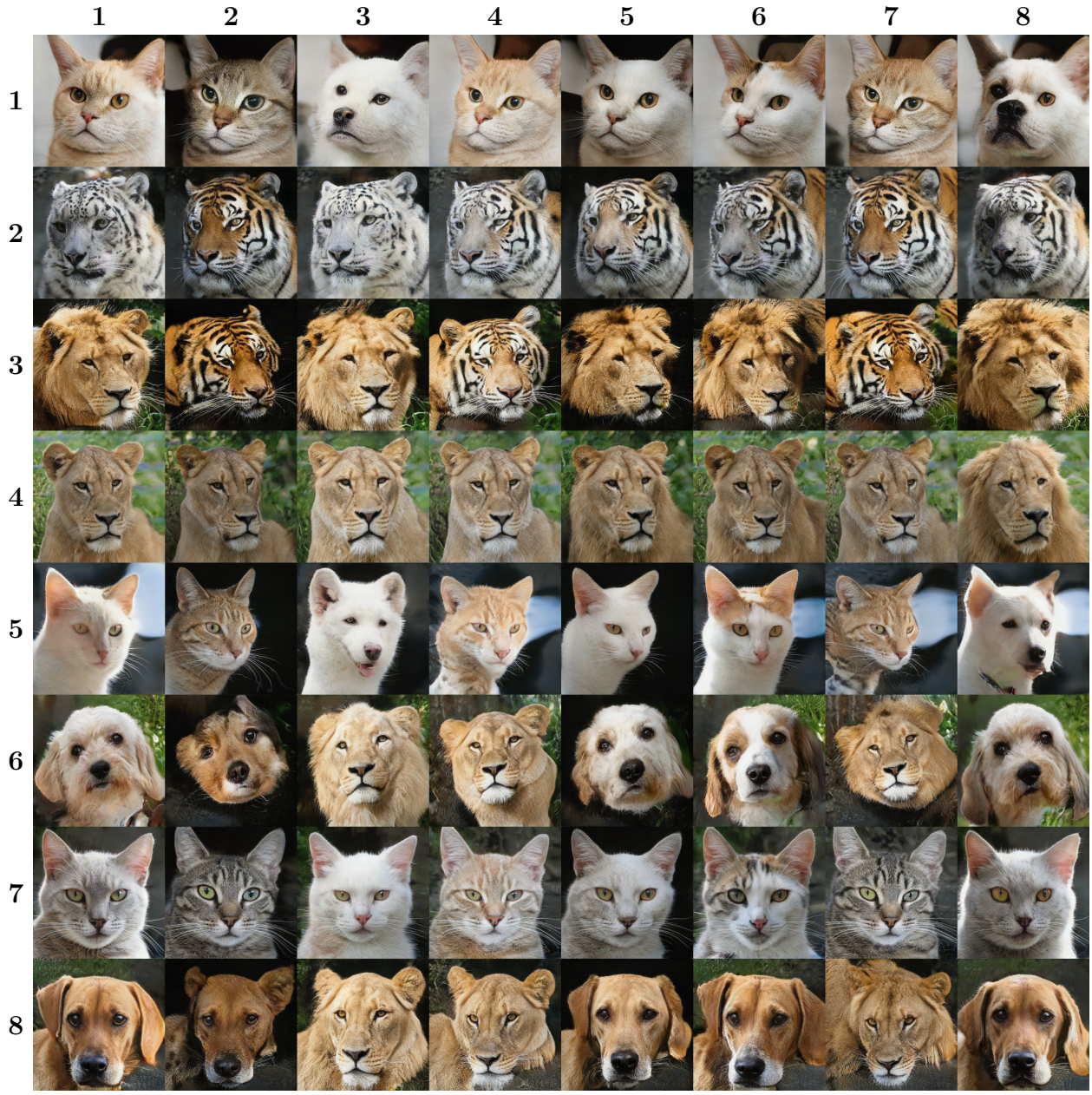

Figure 53: Generated samples of CGPGAN without codebook architecture. Each row has the same continuous latent vector, and each column has the same categorical latent vector. $d_l = 4$, $d_c = 4$. FID: 10.6059, precision: 0.7077, recall: 0.3242.

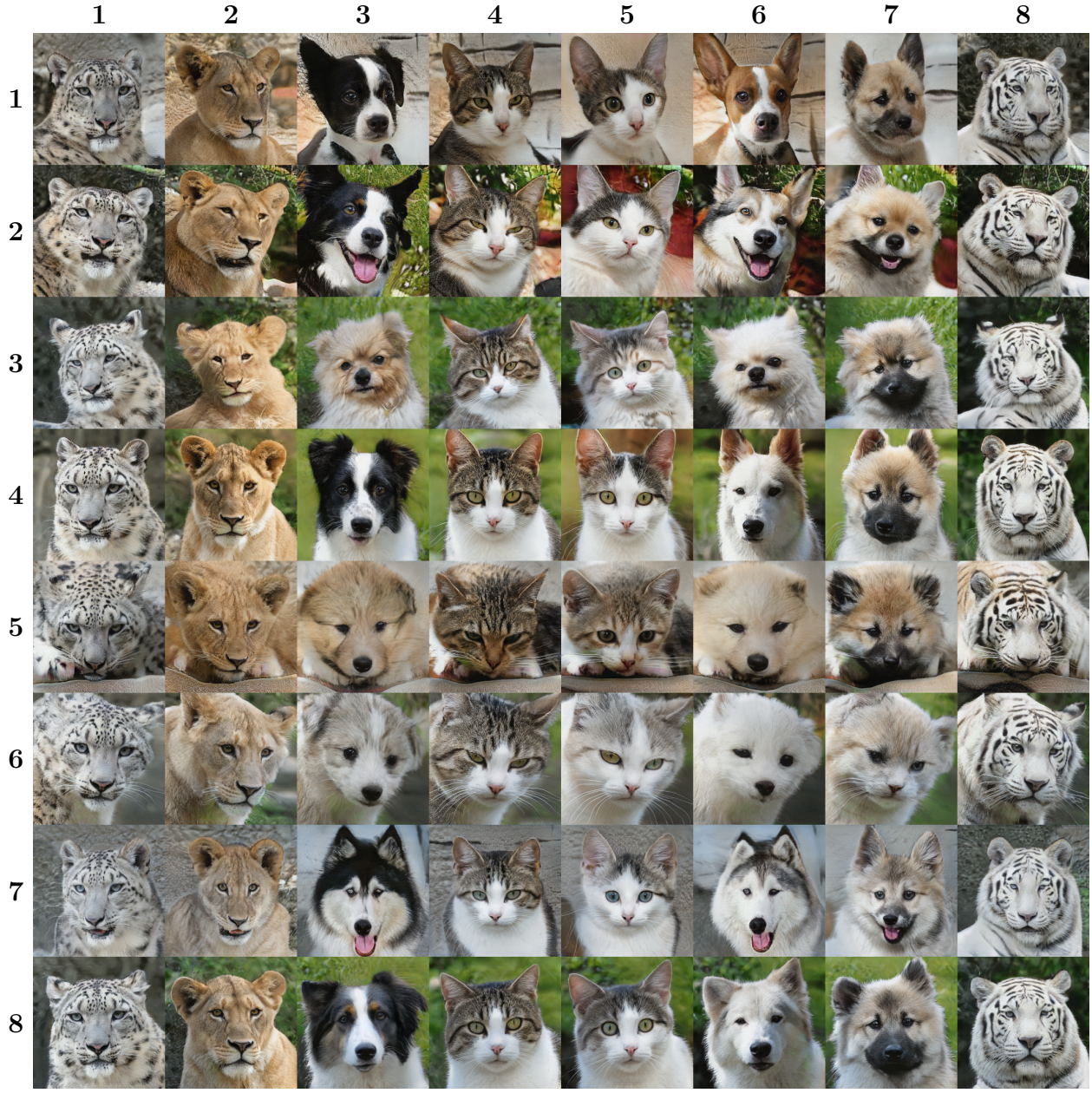

Figure 54: Generated samples of CGPGAN with codebook architecture. Each row has the same continuous latent vector, and each column has the same categorical latent vector. $d_l = 4$, $d_c = 4$. FID: 10.1116, precision: 0.7981, recall: 0.2012.

overall, the impact of categorical vectors was not significant. For example, in rows 5 and 6, the categorical vector had no effect on the sample generated.

Fig. 52 shows generated samples of Vanilla GAN with codebook architecture. One can see that Vanilla GAN with codebook architecture had mode collapse for each categorical vector. Therefore, there is no diversity of images for each class. This results in very low generative performance for the Vanilla GAN with codebook architecture. One can see that the FID of the model is 74.2704, which is quite high compared to other models, and the recall of the model is 0.0000, which means that there is little diversity in the generated data.

Fig. 53 shows generated samples of CGPGAN without codebook architecture. Compared to Vanilla GAN without codebook architecture (Fig. 51), the categorical vector in CGPGAN without codebook architecture has a greater impact on the generated image. For example, in rows 1, 3, 5, 6, and 8, one can see that the species of animal changes according to the categorical latent vector. In rows 2 and 7, the pattern of the animal changed according to the categorical latent vector. However, it is difficult for humans to interpret the features of each category, and the differences in features between categories are not significant. The generative performance of CGPGAN without codebook architecture was similar to Vanilla GAN without codebook architecture. Compared to Vanilla GAN without codebook architecture, CGPGAN without codebook architecture showed better image diversity (recall), but slightly lower image quality (precision). Overall, the FIDs were similar.

Fig. 54 shows generated samples of CGPGAN with codebook architecture. Compared to CGPGAN without codebook architecture (Fig. 53), one can see that there are significant feature differences in each class in CGPGAN with codebook architecture. For example, the images in column 2 are lionesses, and the images in columns 4 and 5 are cats with different patterns. Also, one can see that the continuous latent vector is used to represent continuous features like pose or mouth openness. For example, the animals are facing at the same angle and tilting their heads at the same angle in all rows. Overall, one can see that CGPGAN with codebook architecture uses continuous latent vectors to represent continuous features

like pose, and categorical latent vectors to represent discrete features like species or pattern. The generative performance of CGPGAN with codebook architecture was similar to other methods. The precision of CGPGAN with codebook architecture is 0.7981, which is higher than Vanilla GAN without codebook architecture at 0.7349 and CGPGAN without codebook architecture at 0.7077. However, CGPGAN with codebook architecture had a recall of 0.2012, lower than Vanilla GAN without codebook architecture at 0.2742 and CGPGAN without codebook architecture at 0.3242. Overall, the FID for CGPGAN with codebook architecture was similar to Vanilla GAN without codebook architecture and CGPGAN without codebook architecture.

Figs. 54-57 show the difference of CGPGAN with codebook architecture according to label dimension d_l and categorical latent dimension d_c. One can see that as d_l and d_c increase, the diversity of images for each class decreases. For example, in Fig. 54 with $d_l = 4$ and $d_c = 4$, the image changes significantly depending on the continuous latent vector, but in Fig. 57 with $d_l = 32$ and $d_c = 32$, the image changes little even if the continuous latent vector is varied. This is because as d_l and d_c increase, the number of real data belonging to each class decreases.

In summary, the codebook architecture in CGPGAN enforces a discrete representation of the data being generated so that the features of each class of data are clearly distinguishable. This makes the class-conditional data more interpretable. The codebook architecture in CGPGAN reduced the diversity of the generated data (recall), but increased the quality (precision).

4.5.5 Summary

In the fifth contribution, we introduced CGPGAN, a novel GAN that allows the model to perform self-supervised class-conditional data generation and clustering without knowing labels, optimal categorical probability, or metric function. When training class-conditional GAN with the classifier, the decision boundary of the classifier falls to the local optima

Figure 55: Generated samples of CGPGAN with codebook architecture. Each row has the same continuous latent vector, and each column has the same categorical latent vector. $d_l = 4$, $d_c = 8$. FID: 9.7836, precision:0.7805, recall:0.2593.

Figure 56: Generated samples of CGPGAN with codebook architecture. Each row has the same continuous latent vector, and each column has the same categorical latent vector. $d_l = 16$, $d_c = 16$. FID: 10.8957, precision: 0.7636, recall: 0.2619.

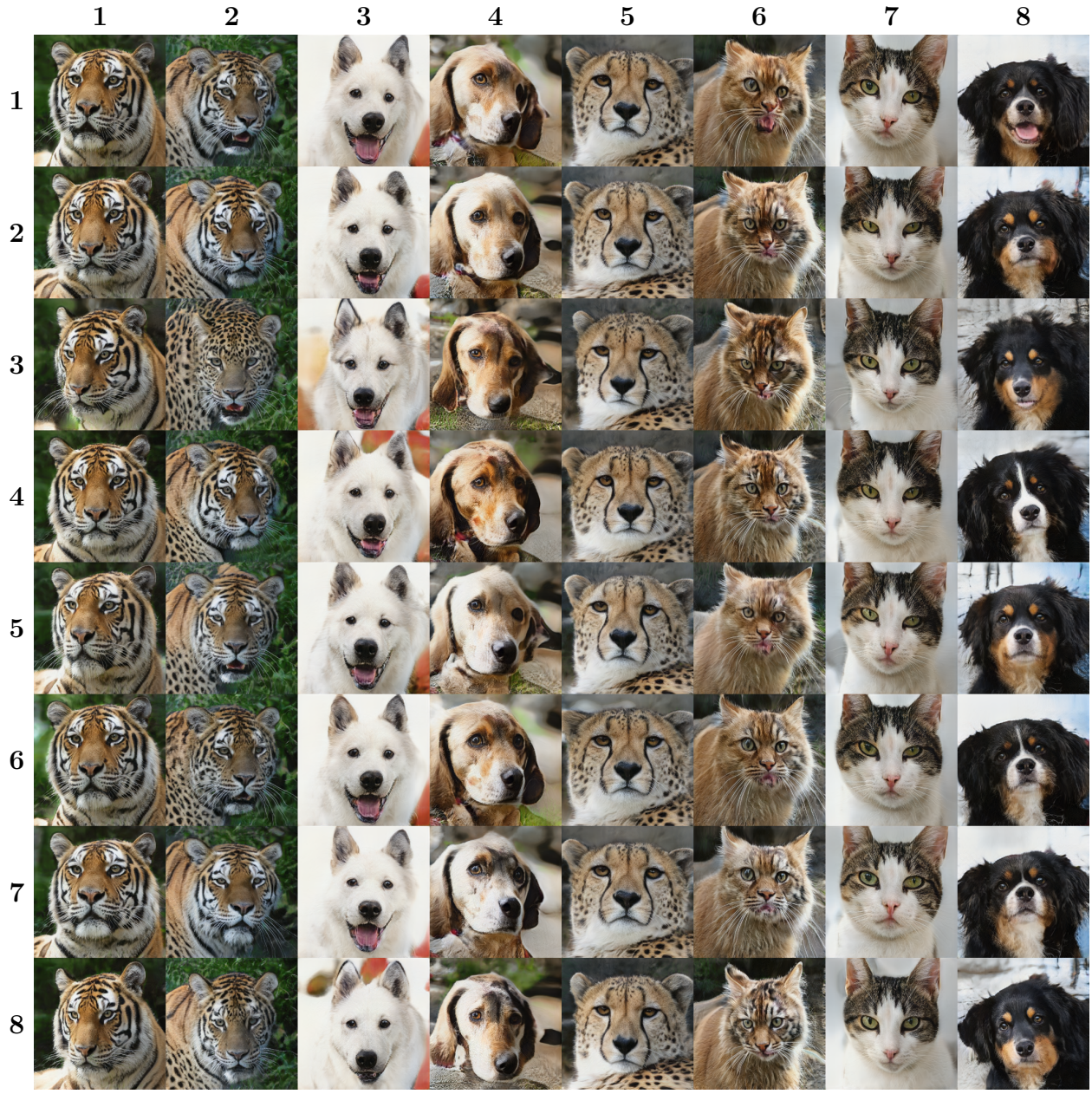

Figure 57: Generated samples of CGPGAN with codebook architecture. Each row has the same continuous latent vector, and each column has the same categorical latent vector. $d_l = 32$, $d_c = 32$. FID: 10.0439, precision: 0.7840, recall: 0.2487.

where the density of the data is minimized. CGPGAN adds a classifier gradient penalty loss to the classifier loss to prevent the classifier's decision boundary from falling into a narrow range of local optima. The gradient penalty regulates the gradient of the classifier's output to prevent the gradient near the decision boundary from becoming too large. Additionally, CGPGAN updates the categorical latent distribution with the categorical probability of real data predicted by the classifier. As training progresses, the entropy of the categorical latent distribution decreases and converges according to the classifier gradient penalty loss weight. In the experiments in sections 4.5.1-4.5.3, we showed that CGPGAN can perform self-supervised class-conditional data generation, without any prior probability or metric function.

We also proposed codebook architecture for CGPGAN to strengthen the discrete representation and make it easier to interpret the discrete representation of the model. Instead of directly inputting a one-hot categorical latent vector into the generator, the codebook architecture inputs the trainable page vector of the corresponding index of the categorical latent vector. In the experiments in sections 4.5.4, we showed that the codebook architecture improved the quality of CGPGAN's generated data and made each class distinct and interpretable.

5 Conclusions and Future Works

In this thesis, we made several contributions to the architecture-agnostic algorithms of generative models and their inversions, as well as applications utilizing these methods.

In the first contribution, we introduced a novel class-conditional GAN that uses only multiple adversarial loss instead of classification loss. Since CAGAN only uses adversarial loss, it has fewer hyperparameters than ACGAN. Also, CAGAN does not use classification loss, so there is no conflict between adversarial loss and classification loss as there is with ACGAN. We also propose mixed batch training to train conditional GAN, when the condi-

tional distribution is biased and there are batch-wise operations in the discriminator. Mixed batch training prevents the discriminator from relying on the conditional distribution of the input batch to distinguish between real and fake samples.

In the second contribution, we introduced DLSGAN, a novel encoder-based GAN inversion method for better convergence than in a simple mean squared error. DLSGAN dynamically adjusts the scale of each element of the latent random variable based on the variance of encoder output to prevent the generator from losing information of the latent random variable. This allows the model to converge better, resulting in better inversion performance. Additionally, we propose AEDLSGAN which utilizes DLSGAN to continuously edit the attributes of input data. AEDLSGAN trains a class-conditional DLSGAN to fit a fixed linear classifier. It enables continuous modification of input data attributes by progressively changing the predicted latent vector to alter the output of the fixed linear classifier.

In the third contribution, we introduced AnoDLSGAN, a method for out-of-distribution detection using DLSGAN. In DLSGAN, the entropy of the scaled latent random variable becomes suitable for representing the data. Furthermore, since each dimension of the latent random variable is independent and follows a simple distribution, it is possible to calculate the probability of the input data.

In the fourth contribution, we introduced PVDGAN, a novel method that combines perceptual VAE and DLSGAN to improve GAN inversion performance. When DLSGAN is trained with a normal latent random variable where each dimension is i.i.d., and the latent encoder is integrated into the discriminator, a sum of a predicted latent random variable of real data and a scaled normal noise follows the normal random variable where each dimension is i.i.d.. Since this random variable is paired with real data and follows the latent random variable, it can be used for both VAE and GAN training. Furthermore, by considering the intermediate layer output of the discriminator as the feature encoder output, the VAE can be trained to minimize the perceptual reconstruction loss.

In the fifth contribution, we introduced CGPGAN, a novel GAN that allows the model

to perform self-supervised class-conditional data generation and clustering without knowing labels, optimal categorical probability, or metric function. When training class-conditional GAN with the classifier, the decision boundary of the classifier falls to the local optima where the density of the data is minimized. CGPGAN adds a classifier gradient penalty loss to the classifier loss to prevent the classifier's decision boundary from falling into a narrow range of local optima. The gradient penalty regulates the gradient of the classifier's output to prevent the gradient near the decision boundary from becoming too large. Additionally, CGPGAN updates the categorical latent distribution with the categorical probability of real data predicted by the classifier. We also proposed codebook architecture for CGPGAN to strengthen the discrete representation and make it easier to interpret the discrete representation of the model. Instead of directly inputting a one-hot categorical latent vector into the generator, the codebook architecture inputs the trainable page vector of the corresponding index of the categorical latent vector.

For each contribution, we use the same model architecture and compare our method against other architecture-agnostic state-of-the-art algorithms. The results demonstrate that the proposed algorithm outperforms existing methods in at least one of the following aspects: generative performance (measured by FID), inversion performance (measured by PSNR and SSIM), or OOD detection performance (measured by AUROC).

In summary, we have introduced several architecture-agnostic deep learning algorithms that enable bidirectional transformation between data distributions and latent distributions. Since the proposed algorithms are architecture-agnostic, those algorithms can be widely used in most deep learning applications, not only for data generation and data manipulation, but also for general tasks such as data processing, data augmentation, and feature representation. We also introduced several architecture-agnostic applications using proposed methods, such as data domain transfer, anomaly detection, and clustering. These applications demonstrate that the proposed algorithms can be widely used in various applications.

Compared to diffusion models, GANs have the advantage of lower latent dimensionality

and easier inversion. However, the generative performance of GANs is generally lower than that of diffusion models, and they often suffer from unstable training and poor convergence. In contrast, diffusion models are known for their stable training and strong generative quality, but their multi-step sampling process makes inversion inherently difficult. Interestingly, a VAE can be seen as a one-step diffusion model, as it maps a latent distribution directly to data in a single decoding step. Based on this view, a promising direction is to combine the strengths of GANs and VAEs to improve both generative performance and invertibility. We also think that better regularization for the discriminator and incorporating memory-based architecture could be fundamental ways to improve the generative performance of GANs.

Another promising direction is to extend the codebook architecture of CGPGAN to large language models (LLMs). CGPGAN is an unsupervised model that learns to embed data into a codebook, which functions as an explicit memory structure. By integrating this mechanism into LLMs, we aim to address the long-term memory limitations of current models. We believe that a self-supervised codebook architecture could serve as a memory module that helps LLMs retain and recall information over longer contexts.

References

[1] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, "Generative Adversarial Nets," in Commun. ACM, vol. 63, no. 11, pp. 139-144, Nov. 2020. https://doi.org/10.1145/3422622

[2] Diederik P Kingma, Max Welling, "Auto-Encoding Variational Bayes," in arXiv preprint, Dec 2013, arXiv:1312.6114. https://arxiv.org/abs/1312.6114v11

[3] Augustus Odena, Christopher Olah, Jonathon Shlens, "Conditional Image Synthesis with Auxiliary Classifier GANs," in proceedings of the 34th International Conference on Machine Learning, PMLR 70:2642-2651, 2017. https://proceedings.mlr.press/v70/odena17a.html

[4] Jeongik Cho, Kyoungro Yoon, "Conditional Activation GAN: Improved Auxiliary Classifier GAN," In IEEE Access, vol. 8, pp. 216729-216740, 2020. https://doi.org/10.1109/ACCESS.2020.3041480

[5] Ming-Yu Liu, Xun Huang, Arun Mallya, Tero Karras, Timo Aila, Jaakko Lehtinen, Jan Kautz, "Few-Shot Unsupervised Image-to-Image Translation," in Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2019, pp. 10551-10560. https://openaccess.thecvf.com/content_ICCV_2019/html/Liu_Few-Shot_Unsupervised_Image-to-Image_Translation_ICCV_2019_paper.html

[6] Mario Lucic, Karol Kurach, Marcin Michalski, Sylvain Gelly, Olivier Bousquet, "Are GANs Created Equal? A Large-Scale Study," in Advances in Neural Information Processing Systems (NIPS), 2018. https://papers.nips.cc/paper/2018/hash/e46de7e1bcaaced9a54f1e9d0d2f800d-Abstract.html

[7] Lars Mescheder, Andreas Geiger, Sebastian Nowozin, "Which Training Methods for GANs do actually Converge?" in Proceedings of Machine Learning Research (PMLR), 2018. https://proceedings.mlr.press/v80/mescheder18a.html

[8] Tero Karras, Timo Aila, Samuli Laine, Jaakko Lehtinen, "Progressive Growing of GANs for Improved Quality, Stability, and Variation," in International Conference on Learning Representations (ICLR), Vancouver, Canada, Apr. 30-May 3, 2018. https://openreview.net/forum?id=Hk99zCeAb

[9] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, Sepp Hochreiter, "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium," in Advances in Neural Information Processing Systems (NIPS), 2017. https://papers.nips.cc/paper/2017/hash/8a1d694707eb0fefe65871369074926d-Abstract.html

[10] Tuomas Kynkäänniemi, Tero Karras, Samuli Laine, Jaakko Lehtinen, Timo Aila, "Improved precision and recall metric for assessing generative models," in Advances in Neural Information Processing Systems (NIPS) proceedings, 2019. https://proceedings.neurips.cc/paper/2019/hash/0234c510bc6d908b28c70ff313743079-Abstract.html

[11] Utkarsh Ojha, Krishna Kumar Singh, Cho-Jui Hsieh, Yong Jae Lee, "Elastic-InfoGAN: Unsupervised Disentangled Representation Learning in Class-Imbalanced Data," in Advances in Neural Information Processing Systems (NIPS), 2020. https://proceedings.neurips.cc/paper/2020/hash/d1e39c9bda5c80ac3d8ea9d658163967-Abstract.html

[12] Zhisheng Xiao, Karsten Kreis, Arash Vahdat, "Tackling the Generative Learning Trilemma with Denoising Diffusion GANs", in International Conference on Learning Representations (ICLR) 2022. https://arxiv.org/abs/2112.07804

[13] Ting Chen, Simon Kornblith, Mohammad Norouzi, Geoffrey Hinton, "A simple framework for contrastive learning of visual representations," in Proceedings of the 37th International Conference on Machine Learning (ICML), 2020. https://dl.acm.org/doi/abs/10.5555/3524938.3525087

[14] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros, in IEEE / CVF Computer Vision and Pattern Recognition Conference (CVPR), 2017. https://openaccess.thecvf.com/content_cvpr_2017/papers/Isola_Image-To-Image_Translation_With_CVPR_2017_paper.pdf

[15] Boris Knyazev, Harm de Vries, Cătălina Cangea, Graham W. Taylor, Aaron Courville, Eugene Belilovsky. "Generative compositional augmentations for scene graph prediction," in Proceedings of the IEEE/CVF International Conference on Computer Vision. 2021.

[16] Eric Jang, Shixiang Gu, Ben Poole, "Categorical reparameterization with gumbel-softmax," in International Conference on Learning Representations (ICLR), 2017. https://openreview.net/pdf?id=rkE3y85ee

[17] Chris J. Maddison, Andriy Mnih, Yee Whye Teh, "The concrete distribution: A continuous relaxation of discrete random variables," in International Conference on Learning Representations (ICLR), 2017. https://openreview.net/forum?id=S1jE5L5gl

[18] Weihao Xia, Yulun Zhang, Yujiu Yang, Jing-Hao Xue, Bolei Zhou, Ming-Hsuan Yang, "GAN Inversion: A Survey," in IEEE Transactions on Pattern Analysis and Machine Intelligence, 2022, doi: 10.1109/TPAMI.2022.3181070. https://ieeexplore.ieee.org/abstract/document/9792208

[19] Tero Karras, Samuli Laine, Timo Aila, "A Style-Based Generator Architecture for Generative Adversarial Networks," in 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019, pp. 4396-4405. https://doi.org/10.1109/CVPR.2019.00453

[20] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, Timo Aila, "Analyzing and Improving the Image Quality of StyleGAN," in 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020, pp. 8107-8116. https://doi.org/10.1109/CVPR42600.2020.00813

[21] Yunjey Choi, Youngjung Uh, Jaejun Yoo, Jung-Woo Ha, "StarGAN v2: Diverse Image Synthesis for Multiple Domains," in 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). https://openaccess.thecvf.com/content_CVPR_2020/html/Choi_StarGAN_v2_Diverse_Image_Synthesis_for_Multiple_Domains_CVPR_2020_paper.html

[22] Matthew D. Zeiler, Rob Fergus, "Visualizing and Understanding Convolutional Networks," in European Conference on Computer Vision (ECCV) 2014. https://link.springer.com/chapter/10.1007/978-3-319-10590-1_53

[23] Raymond A. Yeh, Chen Chen, Teck Yian Lim, Alexander G. Schwing, Mark Hasegawa-Johnson, Minh N. Do, "Semantic Image Inpainting with Deep Generative Models," in Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 6882-6890. https://doi.org/10.1109/CVPR.2017.728

[24] Zachary C. Lipton, Subarna Tripathi, "Precise Recovery of Latent Vectors from Generative Adversarial Networks," In International Conference on Learning Representations (ICLR) workshop, Toulon, France, Apr. 24-26, 2017. https://openreview.net/forum?id=HJC88BzFl

[25] Antonia Creswell, Anil Anthony Bharath, "Inverting the Generator of a Generative Adversarial Network," in IEEE Transactions on Neural Networks and Learning Systems, vol. 30, no. 7, pp. 1967-1974, July 2019. https://doi.org/10.1109/TNNLS.2018.2875194

[26] David Bau, Jun-Yan Zhu, Jonas Wulff, William Peebles, Hendrik Strobelt, Bolei Zhou, Antonio Torralba, "Seeing What a GAN Cannot Generate," in Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2019, pp. 4502-4511. https://openaccess.thecvf.com/content_ICCV_2019/html/Bau_Seeing_What_a_GAN_Cannot_Generate_ICCV_2019_paper.html

[27] David Bau, Hendrik Strobelt, William Peebles, Jonas Wulff, Bolei Zhou, Jun-Yan Zhu, Antonio Torralba, "Semantic photo manipulation with a generative image prior," in ACM Transactions on Graphics (TOG), Volume 38, Issue 4, Jul. 2019. https://doi.org/10.1145/3306346.3323023

[28] Vincent Dumoulin, Ishmael Belghazi, Ben Poole, Alex Lamb, Martin Arjovsky, Olivier Mastropietro, Aaron Courville, "Adversarially Learned Inference," in International Conference on Learning Representations (ICLR), Toulon, France, Apr. 24-26, 2017. https://openreview.net/forum?id=B1ElR4cgg

[29] Jeff Donahue, Philipp Krähenbühl, Trevor Darrell, "Adversarial Feature Learning," in International Conference on Learning Representations (ICLR) conference, Toulon, France, Apr. 24-26, 2017. https://openreview.net/forum?id=BJtNZAFgg

[30] Jeff Donahue, Karen Simonyan, "Large Scale Adversarial Representation Learning," in Advances in Neural Information Processing Systems 32 (NeurIPS 2019). https://papers.nips.cc/paper/2019/hash/18cdf49ea54eec029238fcc95f76ce41-Abstract.html

[31] Guim Perarnau, Joost van de Weijer, Bogdan Raducanu, Jose M. Álvarez, "Invertible Conditional GANs for image editing," in arXiv preprint, 2016, arXiv: 1611.06355. https://arxiv.org/abs/1611.06355

[32] Lucy Chai, Jonas Wulff, Phillip Isola, "Using latent space regression to analyze and leverage compositionality in GANs," in International Conference on Learning Representations (ICLR) conference, Vienna, Austria, May 4, 2021. https://openreview.net/forum?id=sjuuTm4vj0

[33] Jiapeng Zhu, Yujun Shen, Deli Zhao, Bolei Zhou, "In-Domain GAN Inversion for Real Image Editing," in European Conference on Computer Vision (ECCV), 2020. https://doi.org/10.1007/978-3-030-58520-4_35

[34] Dmitry Ulyanov, Andrea Vedaldi, Victor Lempitsky, "It Takes (Only) Two: Adversarial Generator-Encoder Networks," in Proceedings of the AAAI Conference on Artificial Intelligence, 2018. https://ojs.aaai.org/index.php/AAAI/article/view/11449

[35] Elad Richardson, Yuval Alaluf, Or Patashnik, Yotam Nitzan, Yaniv Azar, Stav Shapiro, Daniel Cohen-Or, "Encoding in Style: A StyleGAN Encoder for Image-to-Image Translation," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2021, pp. 2287-2296. https://openaccess.thecvf.com/content/CVPR2021/html/Richardson_Encoding_in_Style_A_StyleGAN_Encoder_for_Image-to-Image_Translation_CVPR_2021_paper.html

[36] Rameen Abdal, Yipeng Qin, Peter Wonka, "Image2StyleGAN: How to Embed Images Into the StyleGAN Latent Space?," in Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), pp. 4432-4441, 2019. https://openaccess.thecvf.com/content_ICCV_2019/html/Abdal_Image2StyleGAN_How_to_Embed_Images_Into_the_StyleGAN_Latent_Space_ICCV_2019_paper.html

[37] Hyunsu Kim, Yunjey Choi, Junho Kim, Sungjoo Yoo, Youngjung Uh, "Exploiting Spatial Dimensions of Latent in GAN for Real-Time Image Editing," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2021, pp. 852-861. https://openaccess.thecvf.com/content/CVPR2021/html/Kim_Exploiting_Spatial_Dimensions_of_Latent_in_GAN_for_Real-Time_Image_CVPR_2021_paper.html

[38] Mehdi Mirza, Simon Osindero, "Conditional Generative Adversarial Nets," in arXiv preprint, 2014, arXiv:1411.1784. https://arxiv.org/abs/1411.1784

[39] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, Pieter Abbeel, "InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets," in Advances in Neural Information Processing Systems 29 (NIPS 2016). https://papers.nips.cc/paper/2016/hash/7c9d0b1f96aebd7b5eca8c3edaa19ebb-Abstract.html

[40] Mario Lucic, Karol Kurach, Marcin Michalski, Sylvain Gelly, Olivier Bousquet, "Are GANs Created Equal? A Large-Scale Study," in Advances in Neural Information

Processing Systems 31 (NeurIPS 2018). https://papers.nips.cc/paper/2018/hash/e46de7e1bcaaced9a54f1e9d0d2f800d-Abstract.html

[41] Yujun Shen, Jinjin Gu, Xiaoou Tang, Bolei Zhou, "Interpreting the Latent Space of GANs for Semantic Face Editing," in 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020, pp. 9240-9249. https://doi.org/10.1109/CVPR42600.2020.00926

[42] Ziwei Liu, Ping Luo, Xiaogang Wang, Xiaoou Tang, "Deep Learning Face Attributes in the Wild," in International Conference on Computer Vision (ICCV), 2015. https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

[43] Yunjey Choi, Minje Choi, Munyoung Kim, Jung-Woo Ha, Sunghun Kim, Jaegul Choo, "StarGAN: Unified Generative Adversarial Networks for Multi-Domain Image-to-Image Translation," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 8789-8797. https://openaccess.thecvf.com/content_cvpr_2018/html/Choi_StarGAN_Unified_Generative_CVPR_2018_paper.html

[44] Zhenliang He, Wangmeng Zuo, Meina Kan, Shiguang Shan, Xilin Chen, "AttGAN: Facial Attribute Editing by Only Changing What You Want," in IEEE Transactions on Image Processing, vol. 28, no. 11, pp. 5464-5478, Nov. 2019. https://doi.org/10.1109/TIP.2019.2916751

[45] Thomas Schlegl, Philipp Seeböck, Sebastian M. Waldstein, Georg Langs, Ursula Schmidt-Erfurth, "f-AnoGAN: Fast unsupervised anomaly detection with generative adversarial networks," in Medical Image Analysis, vol. 54, pp. 30-44, 2019. https://doi.org/10.1016/j.media.2019.01.010

[46] Pouya Samangouei, Maya Kabkab, Rama Chellappa, "Defense-GAN: Protecting Classifiers Against Adversarial Attacks Using Generative Models," in International Confer-

ence on Learning Representations (ICLR), 2018. https://openreview.net/forum?id=BkJ3ibb0-

[47] Lucy Chai, Jun-Yan Zhu, Eli Shechtman, Phillip Isola, Richard Zhang, "Ensembling With Deep Generative Views," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2021, pp. 14997-15007. https://openaccess.thecvf.com/content/CVPR2021/html/Chai_Ensembling_With_Deep_Generative_Views_CVPR_2021_paper.html

[48] Rameen Abdal, Peihao Zhu, Niloy J. Mitra, Peter Wonka, "Labels4Free: Unsupervised Segmentation Using StyleGAN," in Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2021, pp. 13970-13979. https://openaccess.thecvf.com/content/ICCV2021/html/Abdal_Labels4Free_Unsupervised_Segmentation_Using_StyleGAN_ICCV_2021_paper.html

[49] Nontawat Tritrong, Pitchaporn Rewatbowornwong, Supasorn Suwajanakorn, "Repurposing GANs for One-Shot Semantic Part Segmentation," In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2021, pp. 4475-4485. https://openaccess.thecvf.com/content/CVPR2021/html/Tritrong_Repurposing_GANs_for_One-Shot_Semantic_Part_Segmentation_CVPR_2021_paper.html

[50] Edo Collins, Raja Bala, Bob Price, Sabine Susstrunk, "Editing in Style: Uncovering the Local Semantics of GANs," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020, pp. 5771-5780. https://openaccess.thecvf.com/content_CVPR_2020/html/Collins_Editing_in_Style_Uncovering_the_Local_Semantics_of_GANs_CVPR_2020_paper.html

[51] Jeongik Cho, Adam Krzyżak, "Dynamic Latent Scale for GAN Inversion," in Proceedings of the 11th ICPRAM, 2022, pp.221-228. https://www.scitepress.org/Link.aspx?doi=10.5220/0010816800003122

[52] Jeongik Cho, Adam Krzyżak, "Self-supervised Out-of-distribution Detection with Dynamic Latent Scale GAN," in Structural, Syntactic, and Statistical Pattern Recognition (S+SSPR), 2022. https://link.springer.com/chapter/10.1007/978-3-031-23028-8_12

[53] Jeongik Cho, Adam Krzyżak, "Efficient integration of perceptual variational autoencoder into dynamic latent scale generative adversarial network." in Expert Systems (2024), e13618. https://onlinelibrary.wiley.com/doi/full/10.1111/exsy.13618

[54] Ian J. Goodfellow, Jonathon Shlens, Christian Szegedy, "Explaining and harnessing adversarial examples," in arXiv preprint, 2014, arXiv:1412.6572. https://arxiv.org/abs/1412.6572

[55] Kimin Lee, Kibok Lee, Honglak Lee, Jinwoo Shin, "A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks," in Advances in Neural Information Processing Systems (NIPS) 31, 2018. https://proceedings.neurips.cc/paper/2018/hash/abdeb6f575ac5c6676b747bca8d09cc2-Abstract.html

[56] Shiyu Liang, Yixuan Li, R. Srikant, "Enhancing The Reliability of Out-of-distribution Image Detection in Neural Networks," in International Conference on Learning Representations (ICLR), 2018. https://openreview.net/forum?id=H1VGkIxRZ

[57] Weitang Liu, Xiaoyun Wang, John Owens, Yixuan Li, "Energy-based Out-of-distribution Detection," in Advances in Neural Information Processing Systems (NIPS) 33, 2020. https://proceedings.neurips.cc/paper/2020/hash/f5496252609c43eb8a3d147ab9b9c006-Abstract.html

[58] Yiyou Sun, Chuan Guo, Yixuan Li, "ReAct: Out-of-distribution Detection With Rectified Activations," in Advances in Neural Information Processing Systems (NIPS) 34, 2021. https://proceedings.neurips.cc/paper/2021/hash/01894d6f048493d2cacde3c579c315a3-Abstract.html

[59] Jihun Yi, Sungroh Yoon, "Patch SVDD: Patch-level SVDD for Anomaly Detection and Segmentation," in Proceedings of the Asian Conference on Computer Vision (ACCV), 2020. https://openaccess.thecvf.com/content/ACCV2020/html/Yi_Patch_SVDD_Patch-level_SVDD_for_Anomaly_Detection_and_Segmentation_ACCV_2020_paper.html

[60] Xudong Yan, Huaidong Zhang, Xuemiao Xu, "Learning Semantic Context from Normal Samples for Unsupervised Anomaly Detection," in Proceedings of the AAAI Conference on Artificial Intelligence, 2021. https://ojs.aaai.org/index.php/AAAI/article/view/16420

[61] Yann LeCun, Corinna Cortes, Christopher J.C. Burges, "THE MNIST DATABASE of handwritten digits," ATT Labs, 2010. http://yann.lecun.com/exdb/mnist/

[62] Norman Mu, Justin Gilmer, "MNIST-C: A Robustness Benchmark for Computer Vision," in arXiv preprint, 2019, arXiv:1906.02337. https://arxiv.org/abs/1906.02337

[63] Han Xiao, Kashif Rasul, Roland Vollgraf, "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms," in arXiv preprint, 2017, arXiv:1708.07747. https://arxiv.org/abs/1708.07747

[64] Tarin Clanuwat, Mikel Bober-Irizar, Asanobu Kitamoto, Alex Lamb, Kazuaki Yamamoto, David Ha, "Deep Learning for Classical Japanese Literature," in arXiv preprint, 2018, arXiv:1812.01718. https://arxiv.org/abs/1812.01718

[65] Tensorflow autoencoder tutorial. https://www.tensorflow.org/tutorials/generative/autoencoder. Accessed on: Jan. 24, 2023.

[66] Tero Karras, Miika Aittala, Janne Hellsten, Samuli Laine, Jaakko Lehtinen, Timo Aila, "Training Generative Adversarial Networks with Limited Data," in Advances in Neural Information Processing Systems (NIPS) 33, 2020. https://proceedings.neurips.cc/paper/2020/hash/8d30aa96e72440759f74bd2306c1fa3d-Abstract.html

[67] Sergey Ioffe, Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in Proceedings of the 32nd International Conference on Machine Learning, PMLR 37:448-456, 2015. https://proceedings.mlr.press/v37/ioffe15.html

[68] Anders Boesen Lindbo Larsen, Søren Kaae Sønderby, Hugo Larochelle, Ole Winther, "Autoencoding beyond pixels using a learned similarity metric" in Proceedings of The 33rd International Conference on Machine Learning (PMLR), 2016. http://proceedings.mlr.press/v48/larsen16.html

[69] Alireza Makhzani, Jonathon Shlens, Navdeep Jaitly, Ian Goodfellow, Brendan Frey, "Adversarial Autoencoders," in arXiv preprint, 2015. https://arxiv.org/abs/1511.05644

[70] Jonathan Ho, Ajay Jain, Pieter Abbeel, "Denoising Diffusion Probabilistic Models," in Part of Advances in Neural Information Processing Systems 33 (NeurIPS 2020). https://proceedings.neurips.cc/paper/2020/hash/4c5bcfec8584af0d967f1ab10179ca4b-Abstract.html

[71] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, Björn Ommer, "High-Resolution Image Synthesis With Latent Diffusion Models," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2022, pp. 10684-10695. https://openaccess.thecvf.com/content/CVPR2022/html/Rombach_High-Resolution_Image_Synthesis_With_Latent_Diffusion_Models_CVPR_2022_paper.html

[72] Rewon Child, "Very Deep VAEs Generalize Autoregressive Models and Can Outperform Them on Images," in International Conference on Learning Representations (ICLR) 2021 Conference, 2021. https://openreview.net/forum?id=RLRXCV6DbEJ

[73] Tensorflow datasets. https://www.tensorflow.org/datasets

[74] S. A Nene, S. K Nayar, H. Murase and others, "Columbia object image library (coil-20)," in Technical report CUCS-005-96. https://www.cs.columbia.edu/CAVE/software/softlib/coil-100.php

[75] Alex Olsen, Dmitry A. Konovalov, Bronson Philippa, Peter Ridd, Jake C. Wood, Jamie Johns, Wesley Banks, Benjamin Girgenti, Owen Kenny, James Whinney, Brendan Calvert, Mostafa Rahimi Azghadi, Ronald D. White, "DeepWeeds: A Multi-class Weed Species Image Dataset for Deep Learning," in Scientific Reports, 2019. https://doi.org/10.1038/s41598-018-38343-3

[76] Adam Coates, Honglak Lee, Andrew Y. Ng, "An Analysis of Single Layer Networks in Unsupervised Feature Learning," in AISTATS, 2011. https://ai.stanford.edu/~acoates/stl10/

[77] Ernest Mwebaze, Timnit Gebru, Andrea Frome, Solomon Nsumba, Jeremy Tusubira, "iCassava 2019 Fine-Grained Visual Categorization Challenge," in arXiv preprint, 2019, arXiv:1908.02900. https://arxiv.org/abs/1908.02900

[78] Jakob Nikolas Kather, Frank Gerrit Zöllner, Francesco Bianconi, Susanne M Melchers, Lothar R Schad, Timo Gaiser, Alexander Marx, Cleo-Aron Weis, "Multi-class texture analysis in colorectal cancer histology," in Scientific reports, 2016. https://zenodo.org/record/53169#.XGZemKwzbmG

[79] Sivaramakrishnan Rajaraman, Sameer K. Antani, Mahdieh Poostchi, Kamolrat Silamut, Md. A. Hossain, Richard J. Maude, Stefan Jaeger, George R. Thoma, "Pre-trained convolutional neural networks as feature extractors toward improved malaria parasite detection in thin blood smear images," in PeerJ, 2018.

[80] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, Fei-Fei Li, "Novel Dataset for Fine-Grained Image Categorization," in First Workshop on Fine-Grained Visual

Categorization, IEEE Conference on Computer Vision and Pattern Recognition, 2011. http://vision.stanford.edu/aditya86/ImageNetDogs/main.html

[81] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, Li Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2009. http://vision.stanford.edu/aditya86/ImageNetDogs/main.html

[82] Hyun Oh Song, Yu Xiang, Stefanie Jegelka, Silvio Savarese, "Deep Metric Learning via Lifted Structured Feature Embedding," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016. https://cvgl.stanford.edu/projects/lifted_struct/

[83] Patrick Esser, Robin Rombach, Björn Ommer, "Taming Transformers for High-Resolution Image Synthesis," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2021, pp. 12873-12883. https://openaccess.thecvf.com/content/CVPR2021/html/Esser_Taming_Transformers_for_High-Resolution_Image_Synthesis_CVPR_2021_paper.html?ref=

[84] Aaron van den Oord, Oriol Vinyals, koray kavukcuoglu, "Neural Discrete Representation Learning," in Part of Advances in Neural Information Processing Systems 30 (NIPS 2017), 2017. https://proceedings.neurips.cc/paper/2017/hash/7a98af17e63a0ac09ce2e96d03992fbc-Abstract.html

[85] Alina Braun, Michael Kohler, Jeongik Cho, Adam Krzyżak, "Analysis of the rate of convergence of two regression estimates defined by neural features which are easy to implement," in Electronic Journal of Statistics 18.1 (2024): 553-598. https://projecteuclid.org/journals/electronic-journal-of-statistics/volume-18/issue-1/Analysis-of-the-rate-of-convergence-of-two-regression-estimates/10.1214/23-EJS2207.full

[86] Jeongik Cho, "Training Self-supervised Class-conditional GANs with Classifier Gradient Penalty and Dynamic Prior," in viXra preprint, 2024, viXra:2307.0121. https://vixra.org/abs/2307.0121

[87] Jeongik Cho, "Training Classifier Gradient Penalty GAN with Codebook Architecture," in viXra preprint, 2024, viXra:2409.0063. https://vixra.org/abs/2409.0063

[88] Jeongik Cho, "Dynamic Latent Scale GAN for GAN Inversion," in viXra preprint, 2022, viXra:2109.0028. https://vixra.org/abs/2109.0028

[89] Thomas M. Cover, Joy A. Thomas, "Elements of Information Theory," 2nd ed., John Wiley & Sons, 2006. ISBN: 9780471241959. DOI: 10.1002/047174882X.

[90] Ashish Bora, Ajil Jalal, Eric Price, Alexandros G. Dimakis, "Compressed Sensing using Generative Models," in Proceedings of the 34th International Conference on Machine Learning (PMLR) 70:537-546, 2017. https://proceedings.mlr.press/v70/bora17a.html

[91] Augustus Odena, Jacob Buckman, Catherine Olsson, Tom Brown, Christopher Olah, Colin Raffel, Ian Goodfellow, "Is Generator Conditioning Causally Related to GAN Performance?," in Proceedings of the 35th International Conference on Machine Learning (PMLR) 80:3849-3858, 2018. https://proceedings.mlr.press/v80/odena18a.html

[92] Diederik P. Kingma, Jimmy Ba, "Adam: A method for stochastic optimization," in arXiv preprint, 2014. https://arxiv.org/abs/1412.6980

[93] Ilya Loshchilov, Frank Hutter, "Decoupled Weight Decay Regularization," in International Conference on Learning Representations (ICLR) 2019 Conference, 2019. https://openreview.net/forum?id=Bkg6RiCqY7