CHOICE CELL ARCHITECTURE AND STABLE NEIGHBOR MATCH TRAINING TO INCREASE INTERPRETABILITY AND STABILITY OF DEEP GENERATIVE MODELING

By

ZHUXI YANG

Bachelor of Engineering in Civil Engineering
Central South University
Changsha, Hunan, China
2009

Master of Science in International Economics and Finance
Valparaiso University
Valparaiso, Indiana
2012

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
December, 2020

CHOICE CELL ARCHITECTURE AND STABLE NEIGHBOR MATCH TRAINING TO INCREASE INTERPRETABILITY AND STABILITY OF DEEP GENERATIVE MODELING

Dissertation Approved:

Dr. Douglas Heisterkamp

Dissertation Adviser

Dr. Blayne Mayfield

Dr. Christopher Crick

Dr. Ye Liang

ACKNOWLEDGMENTS

I would like to take this opportunity to thank many people who helped me along the path of completing my doctoral degree.

First and foremost, my deep gratitude goes to my dissertation advisor, Dr. Douglas Heisterkamp. I would like to thank him for taking the time out of his busy schedule to meet with me almost every week to discuss this dissertation project. I thank him for offering constructive feedback on this dissertation and encouraging me to submit chapters to conferences and journals.

My indebtedness also goes to my committee members, Dr. Blayne Mayfield, Dr. Christopher Crick, and Dr. Ye Liang. Their support during the pursuit of my PhD has been invaluable and encouraging to me. Their comments on my dissertation are inspiring.

Finally, I want to thank my wife Annie Zhao for her support, love, and patience.

Acknowledgments reflect the views of the author and are not endorsed by committee members or Oklahoma State University.

Name: ZHUXI YANG

Date of Degree: DECEMBER, 2020

Title of Study: CHOICE CELL ARCHITECTURE AND STABLE NEIGHBOR MATCH TRAINING TO INCREASE INTERPRETABILITY AND STABILITY OF DEEP GENERATIVE MODELING

Major Field: COMPUTER SCIENCE

Abstract: Although Generative Adversarial Networks (GANs) have achieved much success in various unsupervised learning tasks, their training is unstable. Another limitation of GAN and deep neural networks in general is their lack of interpretability. To help address these gaps, we aim to improve training stability of GAN and interpretability of deep learning models. To improve stability of GAN, we propose a Stable Neighbor Match (SNM) training. SNM searches for a stable match between generated and real samples, and then approximates a Wasserstein distance based on the stable match. Our experimental results show that SNM is a stable and effective training method for unsupervised learning. To develop more explainable neural components, we propose an interpretable architecture called the Choice Cell (CC). An advantage of CC is that its hidden representation can be reduced to intuitive interpretation of probability distribution. We then combine CC with other subgenerators to build the Choice Generator (CG). Experimental results indicate that CG is not only more explainable but also maintains comparable performance with other popular generators. In addition, to help subgenerators of CG learn more homogeneous representations, we apply within and between subgenerator regularization to the training of CG. We find that regularization improves the performance of CG in learning imbalanced data. Finally, we extend CC to an interpretable conditional model called the Conditional Choice Cell (CCC). The results indicate the potential of CCC as an effective conditional model with an advantage of being more explainable.

TABLE OF CONTENTS

Chapter Page

Chapter Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER I

INTRODUCTION

1.1 Motivation

We are in the age of big data. The Indexed Web contained at least 5.53 billion pages as of September, 2020 [1]; Facebook generated 4 petabytes of data per day as of October, 2014 [2]; more than 500 hours of video were uploaded to YouTube every minute as of May, 2019 [3]; on Twitter, there were 500 million tweets generated per day and around 200 billion tweets generated per year as of August, 2013 [4].

This tremendous amount of data calls for the development of automatic tools to discover patterns from data, which is what machine learning can deliver. Machine learning algorithms have produced promising results in various areas, especially the ones where humans lack the knowledge to devise efficient algorithms [5]. These areas include but are not limited to document classification, email spam filtering, face detection and recognition, as well as handwriting recognition [6, 5, 7].

Traditional machine learning algorithms require a significant amount of domain knowledge and labor to extract a good representation from raw data, which enables the learning system to perform well [7]. To reduce labor, deep learning algorithms, also called deep neural networks, have been proposed to automatically discover suitable representations from raw data [7, 8]. More specifically, a deep neural network consists of multiple neural layers, each of which transforms a representation at a lower level into a representation at a higher level that is more suitable for solving tasks at hand [7]. The key appealing feature of neural networks is that these layers do not re-

quire human labor and are learned automatically from raw data [7]. Neural networks have achieved start-of-the-art performance in a wide range of machine learning tasks, such as image recognition, speech recognition, and machine translation [7, 8, 9, 10, 11].

In spite of the ground-breaking success that deep learning algorithms have achieved, one limitation is that they usually require a large amount of labeled data to achieve high performance [8]. Labeled data, however, are more scarce than unlabeled data. It is expensive to obtain labeled data because they usually require much human labor [8]. Thus, it is desirable to reduce the amount of labeled data needed for neural networks so that these networks can be applied to a wider range of applications [8]. This is the promise of unsupervised learning.

Unsupervised learning is usually formalized as modeling the joint distribution of input data [6]. When the input data are high dimensional, unsupervised learning is confronted with two challenges: a statistical challenge and a computational challenge [8]. It is statistically challenging because the number of configurations that the model needs to distinguish grows exponentially as the number of dimensions grows. Computationally, it is challenging because the number of computations needed for learning and inference grows exponentially with the number of dimensions. One way to overcome these two challenges is to approximate high dimensional distribution. Another way is to design models that avoid the explicit computation of high dimensional distribution [8]. These models are very appealing because they do not require expensive computation. Generative Adversarial Networks (GANs) proposed in [12] are designed in this spirit.

GANs have achieved impressive results in various unsupervised learning tasks, such as image generation and image super-resolution [13, 14, 15]. However, GANs' minimax formulation introduces new issues, most notable of which are vanishing gradients, training instability, and mode dropping [8, 16]. To address vanishing gradients, [17] propose the Wasserstein GAN, which is based on a Wasserstein distance rather

than the original Jensen-Shannon divergence. The Wasserstein GAN has improved the original GAN significantly, but the instability of GAN training remains an issue, largely due to its minimax formulation [16]. To improve the stability of GAN while following the idea of the Wasserstein distance, [16] proposes the Sliced Wasserstein Generator (SWG). SWG approximates a Wasserstein distance directly from samples and formulates GAN as a single minimization instead of a minimax optimization. To continue this line of research, in Chapter 2 of this dissertation, we propose a Stable Neighbor Match (SNM) training. SNM searches for a stable match between generated and real samples, and then approximates a Wasserstein distance based on the stable match.

Another limitation of deep learning algorithms is their "black box" nature, which has prevented researchers and engineers from understanding their internal computation thoroughly. The lack of interpretability limits users' trust in them, preventing them from broader adoptions. Researchers have increasingly realized the value of interpretability of neural networks. They have been working on improving their explainability [18, 19, 20]. Programs like Explainable Artificial Intelligence (XAI) explicitly pursue this goal of creating "a suite of machine learning techniques that: produce more explainable models, while maintaining a high level of learning performance" [21]. The field of interpretable Neural Network (NN) can be divided into two areas. The first area mainly seeks to visualize hidden representations learned inside pre-trained neural networks, especially representations learned in filter maps. With the help of direct visualization of hidden layers, people can perceive internal states of neural networks to better understand their internal computation. Different from visualizing hidden representations in pre-trained neural networks, the second area tends to focus on developing explainable networks that can directly learn interpretable representations during the training. Our work in Chapter 3 of this dissertation is a continuation of the second line of research. We propose an interpretable neural architecture whose

internal representation can be reduced to a more intuitive interpretation of probability distribution. We coin this new neural architecture the Choice Cell (CC). Our work is also inspired by gated units in Long Short-Term Memory (LSTM), whose function is to control and manipulate information flowing through them. Additionally, our idea of CC is influenced by a recent development of Attention Model (AM), which is in turn inspired by human attention. More details about LSTM and AM can be found in Literature Review of Chapter 3.

Similar to standard neural networks, CC relies on the assumption of independence among training samples [22]. If data points are related in time or space, this assumption has its limitations. Thus, it is desirable to equip CC with additional capability of modeling dependency among input samples. To this end, in Chapter 4, we use CC as building blocks to develop a conditional model, and we coin this new network the Conditional Choice Cell (CCC). An advantage of CCC is that it is more explainable and interpretable.

1.2 Roadmap of the Dissertation

This dissertation is organized as follows. In Chapter 2, we present the SNM training. This chapter also includes information on GAN, Wasserstein GAN, and SWG. In addition, to demonstrate the stability and effectiveness of SNM, we conduct thorough experiments to compare its performance with other related generative models. In Chapter 3, we present an interpretable architecture we developed called CC. We also review its relevant literature in that chapter. The review covers topics on interpretable NN, LSTM, and AM. In addition, we demonstrate the effectiveness of CC with results from various experiments on synthetic and real world datasets. Chapter 4 extends CC to a conditional model namely CCC, to model dependency among input data points. Literature related to CCC and experimental results showing its effectiveness are also presented in Chapter 4. In Chapter 5, we conclude with a summary of main

contributions of this work and directions for future research.

1.3 Notation

The following notational convention will be used in this study. A normal face lowercase letter, a, represents a scalar. A boldfaced lowercase letter, $\boldsymbol{a}$, represents a vector. A boldfaced uppercase letter, $\boldsymbol{M}$, represents a matrix. $\mathbb{A}$ represents a set. $\mathbb{R}$ represents the set of real numbers. $\mathbb{G}$ represents the set of generated samples. $\mathbb{X}$ represents the set of real samples. $\sigma(x)$ represents logistic sigmoid, $\frac{1}{1+e^{-x}}$. $\mathcal{N}(\mu, \sigma^2)$ represents a Gaussian distribution with mean μ and variance σ^2. $a \sim P$ represents that a random variable has a distribution P. $\mathbb{E}_{x \sim P_x}[f(x)]$ represents the expectation of f(x) with respect to P(x).

In addition, the abbreviation of an experiment uses a scheme of *network : problem : data*. For example, an experiment that learns distribution of synthetic data using a Choice Cell is denoted as *CC:LD:SD*.

CHAPTER II

STABLE NEIGHBOR MATCH TRAINING

2.1 Introduction

Deep learning algorithms have achieved impressive success in a wide range of applications, such as image recognition, speech recognition, and machine translation [7, 8, 9, 10, 11]. One limitation of deep learning algorithms, however, is that a large amount of labeled data should be provided, which requires expensive human labor to obtain in order to achieve high performance [8]. We live in the world where unlabeled data are far more abundant than labeled data. Thus, it is desirable to advance deep learning algorithms to achieve more success in unsupervised learning tasks where only unlabeled data are required. Unsupervised learnings are usually formulated as modeling the joint distribution of input data [6]. When input data are high dimensional, the tasks are confronted with two challenges: a statistical challenge and a computational challenge [8]. It is statistically challenging because the number of configurations that model need to distinguish grows exponentially with the number of dimensions [8]. It is computationally challenging because the number of computations needed for learning and inference grows exponentially with the number of dimensions [8]. One way to overcome these two challenges is to approximate high dimensional distribution. Another way is to design models that avoid the explicit computation of high dimensional distribution [8].

Models that follow the second approach are very appealing, because they do not require expensive computation. Generative Adversarial Networks (GANs) are pro-

posed in this spirit [12]. GANs have been widely used in various areas, such as image generation, image super-resolution, etc [13, 14, 15]. GANs are formulated as a minimax game between two players. One player is called discriminator whose job is to distinguish real samples from fake samples, and the other player is called generator whose job is to fool the discriminator by generating samples as real as possible. After GANs are trained to reach a Nash equilibrium, the generator generates samples that the discriminator is unable to distinguish [23].

Although GANs are able to produce sharp images, even in very complex data distribution, GANs' saddle-point formulation inherent to minimax optimization has caused some issues [8, 16, 23]. Some of the most pressing issues are vanishing gradients, training instability, and mode dropping [8, 16]. To address the issue of vanishing gradients, [17] proposes the Wasserstein GAN, which is based on a Wasserstein distance rather than the original Jensen-Shannon divergence. Although impressive results have been obtained, the instability of GAN training remains an issue, largely due to its minimax optimization nature [16]. Numerous studies have been conducted to improve the stability of GAN training [16, 24, 25]. For example, the Sliced Wasserstein Generator (SWG) proposed in [16] approximates a Wasserstein distance directly from samples based on random projections of samples, and the paper shows that the approximated distance provides a tight upper bound for a Wasserstein distance. Also, SWG formulates GAN training as a single minimization instead of minimax optimization and demonstrates the improvement of the training stability. Following the effort of improving the stability of GANs, in this chapter, we propose a training method called Stable Neighbor Matching (SNM). SNM preprocesses generated and real samples to output a stable match, and then approximates a Wasserstein distance based on the stable match. Our main contributions of this chapter are:

- We propose a Stable Neighbor Matching (SNM) training method for unsupervised learning. Our goal is to demonstrate that SNM is an effective approxima-

tion for a Wasserstein distance, and it exhibits training stability.

- We demonstrate the robustness of the SNM training by comparing its performance with relevant generative models on various synthetic and real world datasets.

Chapter 2 is organized as follows. In Section 2.2, we review previous studies related to the SNM training. The review includes GAN, Wasserstein GAN, and SWG. Then, we formally introduce the SNM training in Section 2.3. In Section 2.4, we present experiments of the SNM training on a 2D synthetic dataset to demonstrate its training stability. In Section 2.5, we show the robustness of the SNM training from three perspectives: stability under different hyper-parameter settings, correlation between image quality and generator loss, and correlation between a latent variable and generated samples. Finally, Section 2.6 concludes with a summary of results and suggests future research directions.

2.2 Literature Review

Generative Adversarial Network (GAN) GAN is a generative model originally proposed in [12]. Its main goal is to bypass expensive computations of high dimenstional distribution [8]. GAN can be viewed as a minimax game between two players [23]. One player is called the generator, which generates samples as if they come from the same distribution as the real dataset. The other player is called the discriminator, which predicts whether samples come from the real dataset or the generator. More formally, a generator is a differentiable function, $G_{\boldsymbol{\theta}}(\boldsymbol{z})$ that transforms the latent variable $\boldsymbol{z}$ coming from a known distribution $P_{\boldsymbol{z}}$ into artificial samples. The generator is implemented as a multilayer perceptron with parameters $\boldsymbol{\theta}$. The discriminator $D_{\boldsymbol{\phi}}(\boldsymbol{x})$ takes either generated samples or real samples as its input and outputs a single scalar, which represents the probability that $\boldsymbol{x}$ comes from the real data. The discriminator is implemented as a multilayer perceptron with parameter $\boldsymbol{\phi}$. To perform

the classification, the discriminator minimizes the negative log-likelihood, i.e. $-D_{\boldsymbol{\phi}}$ on real data and $-log(1 - D_{\boldsymbol{\phi}})$ on generated samples. The generator tries to fool the discriminator by maximizing the negative log-likelihood, i.e. $-log(1 - D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(z))$. Together, GANs play the following minimax game

$$\max_{\boldsymbol{\theta}} \min_{\boldsymbol{\phi}} \mathbb{E}_{\boldsymbol{x} \sim P_{\boldsymbol{x}}}[-logD_{\boldsymbol{\phi}}(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim P_{\boldsymbol{z}}}[-log(1 - D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(\boldsymbol{z}))] \tag{2.1}$$

The Wasserstein GAN and the Sliced Wasserstein Generator (SWG) Although GAN avoids explicit computation of distribution of high dimensional data and empirically produces impressively high quality images, the minimax optimization brings some issues. Vanishing gradients, training instability, and mode dropping are among the most pressing issues [8, 23, 16]. [17] proposes the Wasserstein GAN to primarily address the vanishing gradients issue. The main idea of the Wasserstein GAN is to replace the Jensen-Shannon divergence in the original GAN framework with the Wasserstein distance, which is a metric for measuring distance between two distributions. The minimax formulation in Equation 2.1 then becomes

$$\max_{\boldsymbol{w}} \min_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{x} \sim P_{\boldsymbol{x}}}[f_{\boldsymbol{w}}(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{z} \sim P_{\boldsymbol{z}}}[f_{\boldsymbol{w}}(G_{\boldsymbol{\theta}}(\boldsymbol{z}))], \tag{2.2}$$

where $f_{\boldsymbol{w}}$ is a 1-Lipschitz function: $\mathbb{X} \to \mathbb{R}$ and is typically implemented by a neural network parameterized by $\boldsymbol{w}$.

Although the Wasserstein GAN addresses some issues of GAN, especially vanishing gradients, the instability of training GAN remains challenging because of the saddle-point objective [16]. Inspired by the Wasserstein GAN but motivated by the stability argument, [16] proposes an approach that approximates a Wasserstein distance directly from the samples. The approximation of the Wasserstein distance is based on random projections of samples, which leads to the sliced Wasserstein distance. Besides the different method for computing the Wasserstein distance, SWG

formulates the training as searching for a global minimizer to avoid instability of the saddle-point optimization.

To understand SWG, it is helpful to consider the Wasserstein distance between two sample sets. Let $\mathbb{X}$ be a set of real data and $\mathbb{G}$ be a set of data artificially generated by a generator G. Let $\mathbb{X}_i$ and $\mathbb{G}_i$ be i_{th} element in $\mathbb{X}$ and $\mathbb{G}$, respectively. The Wasserstein distance between $\mathbb{X}$ and $\mathbb{G}$, can be defined as

$$W(\mathbb{X}, \mathbb{G}) = \frac{1}{|\mathbb{X}|} \min_{\tau \in \Sigma_{\mathbb{X}}} \sum_{i=1}^{|\mathbb{X}|} \|\mathbb{X}_{\tau(i)} - \mathbb{G}_i\|, \tag{2.3}$$

where $\Sigma_{\mathbb{X}}$ is the set of all permutations of elements in $\mathbb{X}$. Equation 2.3 shows that to find a minimal Wasserstein distance between $\mathbb{X}$ and $\mathbb{G}$, we can search for the optimal permutation $\tau^\star$ such that the bijective mapping between $\mathbb{X}_{\tau^\star(i)}$ and $\mathbb{G}_i$ for all i's results in the minimal accumulated distance.

Searching for $\tau^\star$ can be found by solving a linear program and the problem can be solved in time complexity of $O(|\mathbb{X}|^{2.5} log(|\mathbb{X}|))$ with a dedicated linear program solver. This time complexity is prohibitively expensive for learning algorithms where it is common to train models with hundreds of thousands of iterations.

To address this complexity issue, the sliced Wasserstein distance is proposed [17]. The main observation is that when $\mathbb{X}$ and $\mathbb{G}$ are 1-dimensional, the optimal mapping between them can be obtained easily. Let $\tau_{\mathbb{X}}$ and $\tau_{\mathbb{G}}$ be permutations of $\mathbb{X}$ and $\mathbb{G}$ such that

$$\mathbb{X}_{\tau_{\mathbb{X}}(1)} \leq \mathbb{X}_{\tau_{\mathbb{X}}(2)} \leq ... \leq \mathbb{X}_{\tau_{\mathbb{X}}(|\mathbb{X}|)}$$

$$\mathbb{G}_{\tau_{\mathbb{G}}(1)} \leq \mathbb{G}_{\tau_{\mathbb{G}}(2)} \leq ... \leq \mathbb{G}_{\tau_{\mathbb{G}}(|\mathbb{G}|)}$$

Then, the Wasserstein distance between the 1-dimensional $\mathbb{X}$ and the 1-dimensional

$\mathbb{G}$, $W_{1d}(\mathbb{X}, \mathbb{G})$, can be easily computed as

$$W_{1d}(\mathbb{X}, \mathbb{G}) = \frac{1}{|\mathbb{X}|} \sum_{i=1}^{|\mathbb{X}|} \|\mathbb{X}_{\tau_{\mathbb{X}}(i)} - \mathbb{G}_{\tau_{\mathbb{G}}(i)}\| \tag{2.4}$$

Equation 2.4 can be proved inductively. Thus, the optimal mapping between $\mathbb{X}$ and $\mathbb{G}$ for calculating their Wasserstein distance can be found in $O(|\mathbb{X}|log(|\mathbb{X}|))$ by sorting $\mathbb{X}$ and $\mathbb{G}$.

The sliced Wasserstein distance can be built upon Equation 2.4 by randomly projecting original datasets $\mathbb{X}$ and $\mathbb{G}$ onto 1-dimensional space. Let $\boldsymbol{u}$ be a random unit vector representing projection direction. Let $\mathbb{X}^{\boldsymbol{u}}$ be the set formed by projecting data points in $\mathbb{X}$ unto $\boldsymbol{u}$, i.e. $\mathbb{X}^{\boldsymbol{u}} = \{\boldsymbol{u}^T\boldsymbol{x} | \boldsymbol{x} \in \mathbb{X}\}$. Let $\mathbb{G}^{\boldsymbol{u}}$ be the set formed by projecting data points in $\mathbb{G}$ unto $\boldsymbol{u}$, i.e. $\mathbb{G}^{\boldsymbol{u}} = \{\boldsymbol{u}^T\boldsymbol{g} | \boldsymbol{g} \in \mathbb{G}\}$. The sliced Wasserstein distance between $\mathbb{X}$ and $\mathbb{G}$, $W_s(\mathbb{X}, \mathbb{G})$, can then be defined as

$$W_s(\mathbb{X}, \mathbb{G}) = \frac{1}{|\mathbb{U}|} \sum_{\boldsymbol{u} \in \mathbb{U}} W_{1d}(\mathbb{X}^{\boldsymbol{u}}, \mathbb{G}^{\boldsymbol{u}}), \tag{2.5}$$

where $\mathbb{U}$ is a set of random unit directions, usually sampled from the Gaussian distribution at every iteration. Building upon Equation 2.5, the optimization procedure can be formulated as minimizing the sliced Wasserstein distance between real samples $\mathbb{X}$ and generated samples $\mathbb{G}$ through adjusting the set of parameters $\boldsymbol{\theta}$ in the generator, i.e.

$$\min_{\boldsymbol{\theta}} W_s(\mathbb{X}, \mathbb{G})$$

Similar to SWG, the SNM training that we propose also aims to improve the stability of GAN and approximate the Wasserstein distance from samples, but there is an important difference. SWG approximates the Wasserstein distance by projecting original datasets into 1-dimensional space and then calculating average 1-dimensional Wasserstein distance between projected sets. SNM, however, approxi-

mates the Wasserstein distance by searching for a good approximation for the optimal permutation $\tau^\star$ and then calculating the distance based on the approximation.

2.3 The Stable Neighbor Match Training Method

In this section, we describe our Stable Neighbor Match (SNM) training method. As mentioned in Section 2.2, the minimum Wasserstein distance between two sets of samples can be found by searching for the optimal mapping $\tau^\star$ between two sets, but it is computationally prohibitive for any typical learning algorithm to be trained efficiently, which usually involves more than hundreds of thousands of iterations [16]. Thus, we propose to find a stable match between two sets, which can be computed more efficiently than the optimal mapping. The Wasserstein distance is then calculated based on the stable match.

Intuitively, a match between two sets of samples is stable, if for every pair in the match at least one of its partner is satisfied with the pairing. Formally, stable match is defined below.

Definition II.1 *Let $\mathbb{X}$ be a set of real samples and $\mathbb{G}$ be a set of generated samples, and we assume that $|\mathbb{G}| \leq |\mathbb{X}|$. Let $\boldsymbol{g} \in \mathbb{G}$ and $\boldsymbol{x} \in \mathbb{X}$. Let $d(\boldsymbol{g}, \boldsymbol{x})$ be the distance between samples $\boldsymbol{g}$ and $\boldsymbol{x}$. A match $\mathbb{M}$ between $\mathbb{G}$ and $\mathbb{X}$ is a set of $(\boldsymbol{g}, \boldsymbol{x})$ pairs such that any $\boldsymbol{g}$ and $\boldsymbol{x}$ appears at most once in $\mathbb{M}$. If a sample $\boldsymbol{g}$ and a sample $\boldsymbol{x}$ forms a matched pair in $\mathbb{M}$, we say $\mathbb{M}(\boldsymbol{g}) = \boldsymbol{x}$ or $\mathbb{M}(\boldsymbol{x}) = \boldsymbol{g}$. A pair $\boldsymbol{g}$ and $\boldsymbol{x}$ forms a blocking pair in $\mathbb{M}$, if $\mathbb{M}(\boldsymbol{g}) \neq \boldsymbol{x}$ and $d(\boldsymbol{g}, \boldsymbol{x}) < d(\boldsymbol{g}, \mathbb{M}(\boldsymbol{g}))$ and $d(\boldsymbol{g}, \boldsymbol{x}) < d(\mathbb{M}(\boldsymbol{x}), \boldsymbol{x})$. A match $\mathbb{M}$ is stable if there is no blocking pair. Denote a stable match as $\mathbb{M}^s$.*

The main idea of SNM is to search for a stable match $\mathbb{M}^s$ between $\mathbb{G}$ and $\mathbb{X}$ by searching for nearest $(\boldsymbol{g}, \boldsymbol{x})$ pairs in a greedy manner. More specifically, we start the search with empty $\mathbb{M}$, i.e., $\mathbb{M} = \{\}$. At each subsequent iteration, we search for a nearest pair $(\boldsymbol{g}, \boldsymbol{x})$ that is not in $\mathbb{M}$. Then we add the pair $(\boldsymbol{g}, \boldsymbol{x})$ into $\mathbb{M}$, if neither $\boldsymbol{g}$ nor $\boldsymbol{x}$ is already in $\mathbb{M}$. We repeat this process until every sample $\boldsymbol{g} \in \mathbb{G}$ appears in the

match $\mathbb{M}$. It can be shown that the final match $\mathbb{M}$ of SNM is stable by observing that $\mathbb{M}$ is stable at the end of each iteration. Since we search for a nearest pair at every iteration of constructing a stable match $\mathbb{M}^s$, we name the process Stable Neighbor Match.

After applying SNM to find a stable match $\mathbb{M}^s$ between the generated samples $\mathbb{G}$ and the real samples $\mathbb{X}$, we calculate the generator loss between $\mathbb{G}$ and $\mathbb{X}$ below.

$$L(\mathbb{G}, \mathbb{X}) = \frac{1}{|\mathbb{G}|} \sum_{(\boldsymbol{g}, \boldsymbol{x}) \in \mathbb{M}^s} loss(\boldsymbol{g}, \boldsymbol{x}), \tag{2.6}$$

where *loss* is a loss function that measures loss between two samples.

2.4 Experiment: *Learning Encodings of Synthetic Data Using Generators Trained with SNM (SNM:LE:SD)*

The main purpose of the experiment *SNM:LE:SD* is to demonstrate the effectiveness of the SNM training by using a synthetic dataset. More specifically, we train a generator using the SNM training on a 2D mixture of 8 Gaussian distributions, and we expect that the generator can learn to generate samples as if they come from true distributions. In addition, we compare the performance of SNM with that of GAN and SWG to show its effectiveness.

Dataset Drawing upon the literature [24, 26], we design the dataset of the experiment *SNM:LE:SD*. The samples in the dataset are drawn from a mixture of 8 Gaussian distributions. Each Gaussian distribution in the dataset is characterized by a mean matrix and a diagonal covariance matrix. The samples drawn from this mixture of 8 Gaussian can be found in Figure 2.1.

Network Architecture As a fair comparison, we use the same generator structure in all models. Each generator is a 4-layer fully connected network. The latent

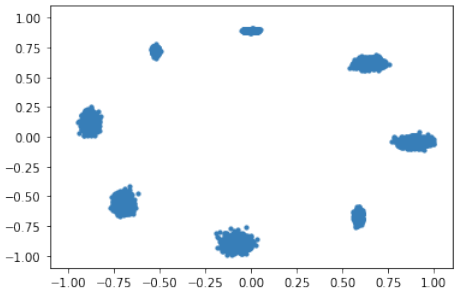


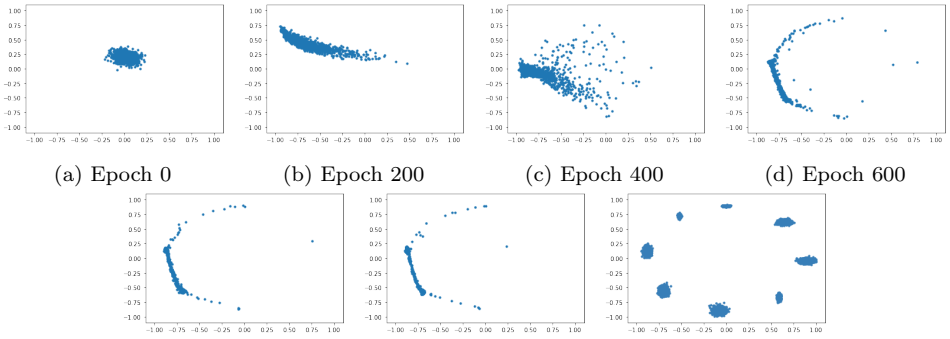
Figure 2.1: The 8 Gaussian dataset. Each cluster contains 1250 data points.

dimension is 20. Each hidden layer contains 128 nodes and uses the rectifier as an activation function. The output layer has dimensionality of 2 and uses *tanh* function as an activation function. For GAN, the discriminator is also a fully connected network, and the minimax optimization follows the original formulation.

In terms of generator losses, GAN uses Cross Entropy loss as the origin setting. Both SNM and SWG use L1-based Mean Absolute Error (MAE), so that their approximated Wasserstein distances are comparable. In addition, for the generator trained with SNM, SNM is first applied to generated and real samples to output a stable match $\mathbb{M}^s$. The generator loss is then computed based on the match $\mathbb{M}^s$.

Hyper-parameters We use Adam optimizer with learning rate equal to 0.001 through out the experiment. Each value in latent variable $\boldsymbol{z}$ comes from a Gaussian $\mathcal{N}(0, 1)$. Batch size is set to 512. We train each model on the dataset for 1000 epochs.

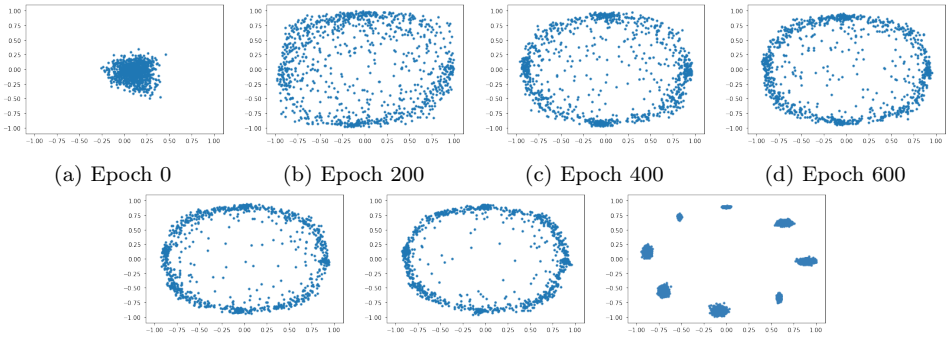
Result We first look at samples generated by GAN and SWG. The results of GAN are shown in Figure 2.2, where we plot the samples generated by GAN in every 200 epochs. As we can see from the figure, after 200 epochs, samples generated by GAN move towards the upper-left corner of the space. After 400 epochs, GAN biases towards the clusters in the left half of the space, and this trend maintains until the end of the training. At the end, GAN roughly learns five of eight Gaussians in the left half of the space, but we can clearly observe the common phenomenon of mode



(a) Epoch 0 (b) Epoch 200 (c) Epoch 400 (d) Epoch 600

(e) Epoch 800 (f) Epoch 1000 (g) 8 Gaussian

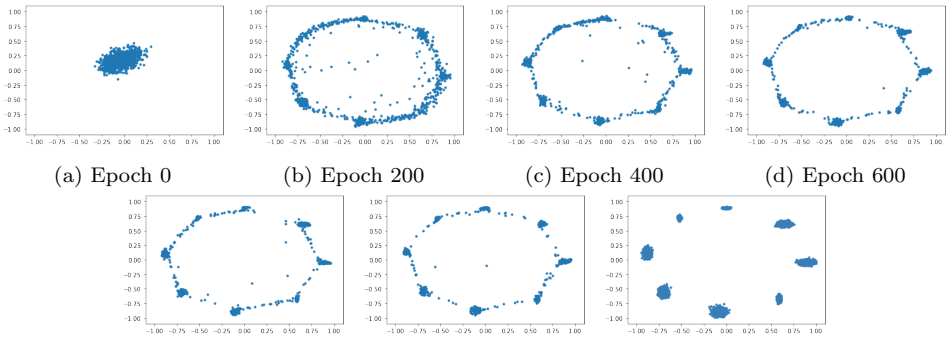
Figure 2.2: Evolution of samples generated by GAN on the 8 Gaussian dataset



(a) Epoch 0 (b) Epoch 200 (c) Epoch 400 (d) Epoch 600

(e) Epoch 800 (f) Epoch 1000 (g) 8 Gaussian

Figure 2.3: Evolution of samples generated by SWG on the 8 Gaussian dataset



(a) Epoch 0 (b) Epoch 200 (c) Epoch 400 (d) Epoch 600

(e) Epoch 800 (f) Epoch 1000 (g) 8 Gaussian

Figure 2.4: Evolution of samples generated by the generator trained with SNM on the 8 Gaussian dataset

dropping in GAN. Mode dropping is a severe and well-documented issue of GAN [23, 27]. It refers to the phenomenon where the generator disregards some modes of data, and the discriminator is not able to detect it [23, 27]. In our case, the distribution that the generator generates concentrates on the five clusters in the left half of the space, but the discriminator fails to detect the fact that this distribution comes from the generator. Many researchers have attempted to explain the mode dropping issue, and they suggest that the issue is caused by a combination of factors, such as the polynomial capacity of the discriminator and the optimization of KL-divergence [27, 28, 29]. Next, we move to samples generated by SWG, and the results are plotted in Figure 2.3. As we can see from the figure, after 200 epochs, samples spread over the space. After 400 epochs, SWG learns concentric circles which roughly correspond to the orbit of eight true clusters. At the end of the training, we can see that SWG roughly learns 8 true Gaussians. However, we observe a blending phenomenon of SWG: there are many points lying between adjacent clusters learned by SWG, and there are some points in the middle of the circle. We suspect that this blending is due to the nature of random projections used in SWG. For some projections, adjacent clusters are projected into the inseparable vectors, which causes data points to be pulled between them. For some projections, some clusters are projected into opposite directions, which causes data points to be pulled into the middle of the circle.

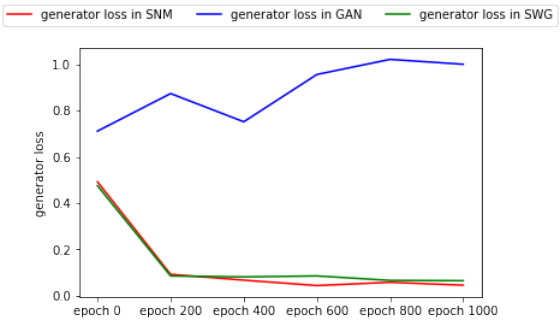


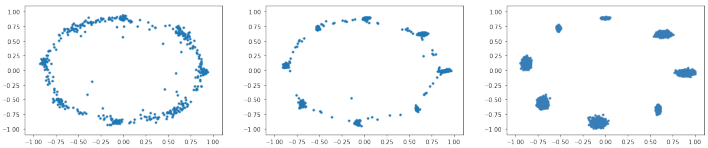
Figure 2.5: Evolution of generator losses on 8 Gaussian

We next look at samples generated by the generator trained with SNM, and the results are shown in Figure 2.4. We can observe a few improvements of the SNM training, compared with GAN and SWG. First, SNM learns the true distribution faster than GAN and SWG. After only 200 iterations, we observe that the generator learns a pattern of concentric circles where eight true clusters reside. With the same amount of training, GAN only learns one cluster at the upper-left corner, and SWG only learns a rectangular spreading over the space. Second, the generator trained by SNM learns the true distribution more accurately than GAN and SWG. At the end of the training, GAN only learns five clusters. SWG learns eight clusters, but there are many points lying between clusters. Although there are still a few points lying between adjacent clusters, the generator trained by SNM learns all eight clusters at the end of the training. Finally, the SNM training does not suffer the mode dropping issue of GAN, and its blending phenomenon is much less severe than SWG.

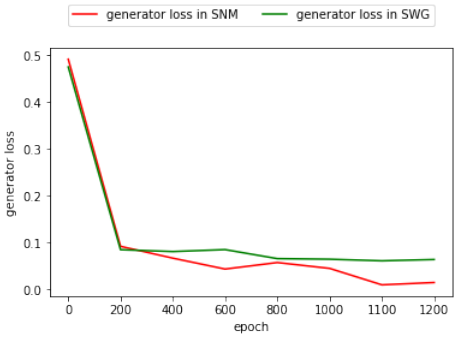
Next, we look at the evolution of the generator loss of the generator trained with SNM. We plot the generator loss of the SNM training in Figure 2.5. We can observe a few good indicators for the stability of the SNM training. First, its generator loss steadily decreases (from loss value of 0.498 all the way to the loss value of 0.045), as the training proceeds. This behavior of generator loss is similar to that of SWG. Also, the generator loss of SNM converges as fast as SWG, both of which take only 200 epochs to reduce their generator losses to around 0.09. The generator loss in GAN,

however, fluctuates throughout the training process, because it depends on both the quality of the discriminator and the quality of samples. Second, from Figures 2.4 and 2.5, we can see that sample quality of the SNM training correlates well with its generator loss. When the samples are clustered around the origin prior to the training, the loss value is at its peak which is 0.498. When the generator loss is reduced to 0.09, a pattern of concentric circles is formed. As the generator loss further decreases to 0.065, generated samples form eight clusters that agree with the true distributions, but there are many points lying between clusters. As the loss decreases below 0.05 at the end, the generator trained by SNM learns all eight Gaussians, and there are only a few points between adjacent clusters. Similar training stability from this perspective is also observed in the training of SWG. As the generator loss of SWG decreases, the quality of samples generated by SWG improves. GAN, however, does not exhibit the training stability from this perspective. It is more difficult to observe correlated patterns between sample quality and generator loss, because the quality of discriminator also plays a significant role. Finally, we also want to highlight the quality of the approximation of the true Wasserstein distance calculated based on SNM. [16] shows that SWG provides a tight upper bound for the Wasserstein distance. As we can see from Figure 2.5, empirically, SNM provides an upper bound as tight as that of SWG.

Finally, some data points generated by the SNM training are scattered towards adjacent clusters. We conjecture that this is due to the mismatch between the frequency of generated samples and that of real samples. We therefore conduct post-training with $|\mathbb{G}| < |\mathbb{X}|$ to train all generated samples to their closest clusters. More specifically, after the 1000 epochs of training, we train the generator with SNM for another 200 epochs with $|\mathbb{G}| = 256$ and $|\mathbb{X}| = 512$. We also train SWG for another 200 epochs to compare the results, which are shown in Figure 2.6. The empirical evidence supports our conjecture. As we can see from Figure 2.6b, the generator learns clusters



(a) SWG epoch 1200 (b) SNM epoch 1200 (c) 8 Gaussian



(d) Evolution of generator losses in post-training

Figure 2.6: Effects of post-training for the generator trained with SNM using $|\mathbb{G}| < |\mathbb{X}|$

that are more compact and closer to the original distribution. In addition, as shown in Figure 2.6d, training for additional 200 epochs does not reduce the generator loss of SWG any further, but SNM with $|\mathbb{G}| < |\mathbb{X}|$ reduces its generator loss significantly and yields a tighter upper bound for the Wasserstein distance.

2.5 Experiments: *Learning Encodings of Real World Data Using Generators Trained with SNM (SNM:LE:RD)*

The goal of this section is to demonstrate the stability of generators trained with SNM. We analyze its stability by conducting various experiments, which are presented in the following four subsections. In the first subsection, we show the training stability of SNM across different hyper-parameter settings. In the second subsection, an experiment showing correlation between latent variable and generated samples is presented to indicate the stability of generators trained with SNM. In the third experiment, we demonstrate the robustness of the SNM training by showing correlation between sample quality and generator loss. In the last subsection, we present an experiment that attempts to further improve the SNM training.

2.5.1 Experiment: Showing Training Stability of SNM across Different Hyper-parameter Settings *(SNM:HP:RD)*

In this experiment, we demonstrate the robustness of the SNM training across various hyper-parameter settings. We also compare its performance with that of GAN and SWG. The experiment is performed on the MNIST and Fashion-MNIST datasets [30, 31]. The comparison method is inspired by [16, 24].

Dataset The datasets used in this experiment are the MNIST and Fashion-MNIST datasets. The sizes of training data in both datasets are 60K.

Network Architecture To have a fair comparison, all models adopt the same generator archetype. The basic generator archetype is Deep Convolutional GAN (DCGAN) [13]. Each layer in the generator, except the last one, applies a leaky rectifier. The last layer uses sigmoid as an activation function. In addition, following the literature [24] that compares stability, we adopt two types of generator architectures. The first one has batch normalization in the generator (BN for short), and the other one excludes batch normalization in the generator (NoNB for short). Also, following the argument that selecting optimizer is critical to the modeling performance [17], each generator type has two different optimizers: Adam [32] and RMSprop [33]. Thus, in total we compare the SNM training with GAN under four different settings: BN with Adam, NoNB with Adam, BN with RMSprop, and NoBN with RMSprop.

Hyper-parameters For each training setting, all models use the same hyper-parameters. The details of hyper-parameters are summarized in Table 2.1. Batch size is set to 256. We train each model under each setting for 100 epochs as a fair comparison.

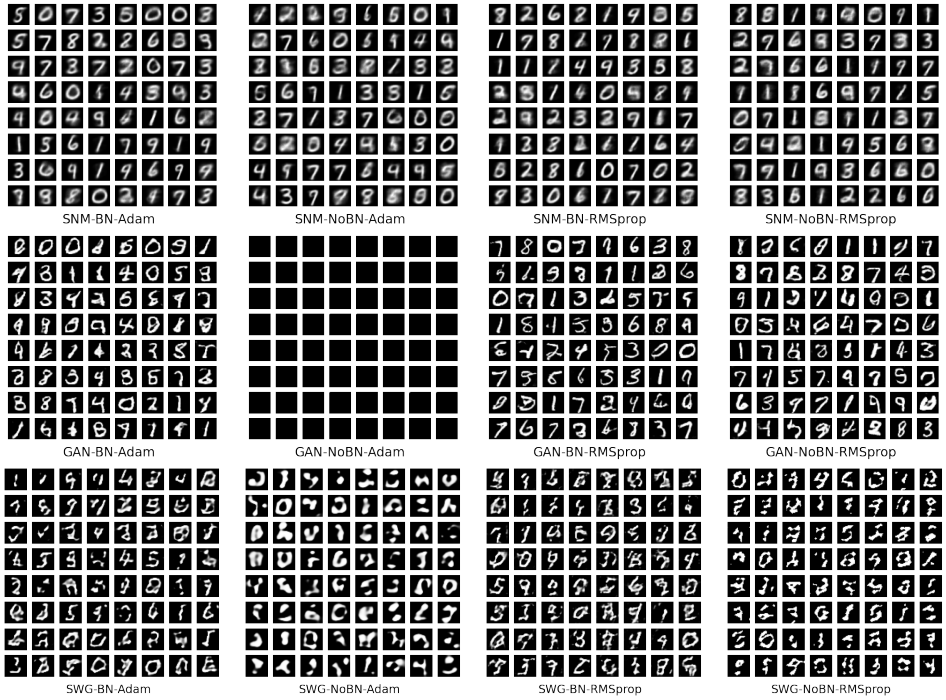


Figure 2.7: Comparing qualities of MNIST digits generated by SNM, GAN, and SWG using different regularizations and optimizations. Samples are obtained from training each model under each hyper-parameter setting for 100 epochs.

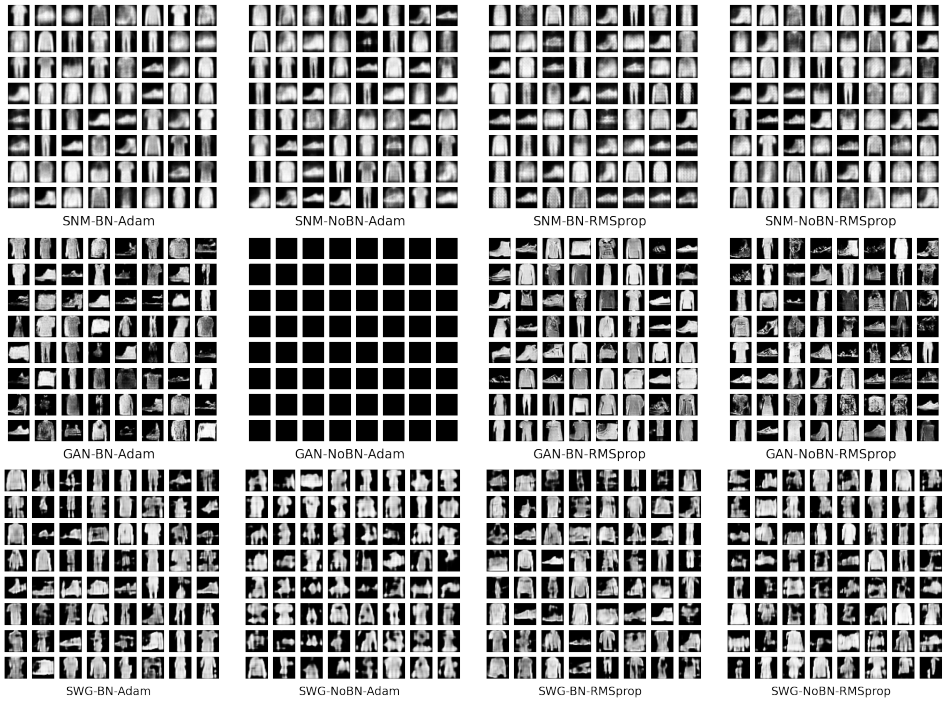


Figure 2.8: Comparing qualities of Fashion-MNIST samples generated by SNM, GAN, and SWG using different regularizations and optimizations. Samples are obtained from training each model under each hyper-parameter setting for 100 epochs.

	BN-Adam	NoNB-Adam	BN-RMSprop	NoBN-RMSprop
learning rate	0.0001	0.0001	0.001	0.001

Table 2.1: Details of the hyper-parameter setting in the experiment *SNM:HP:RD*

Result We present qualities of images produced by generators trained with SNM under different hyper-parameter settings in Figures 2.7 and 2.8. As we can see from these figures, GAN fails to generate any meaningful images in both the MNIST and Fashion-MNIST datasets under the NoNB-Adam setting. SWG indicates its strong stability by being able to produce meaningful digits and apparels under all hyper-parameter settings. SNM performs as stable as SWG because it also succeeds in generating meaningful images for both datasets under all hyper-parameter settings.

2.5.2 Experiment: Showing Robustness of Generators Trained SNM by Correlation between a Latent Variable and Generated Samples *(SNM:LV:RD)*

The goal of this experiment is to demonstrate the stability of the SNM training by showing a correlation between a latent variable and generated images. The main design of the experiment is to assess whether samples generated from the same latent locations are consistent during the training. Following the procedure used in the previous experiments, we also compare results with GAN and SWG on real world data.

Dataset The datasets used in this experiment is MNIST, where the size of training data is 60K.

Network Architecture The basic generator archetype is DCGAN. All models follow the same generator archetype. Each layer in the generator, except the last one, applies batch normalization and leaky rectifier. The last layer uses sigmoid activation.

Hyper-parameters All models use the same hyper-parameters. We use Adam as the optimizer and set the learning rate to 0.0001. Batch size is set to 256. We train

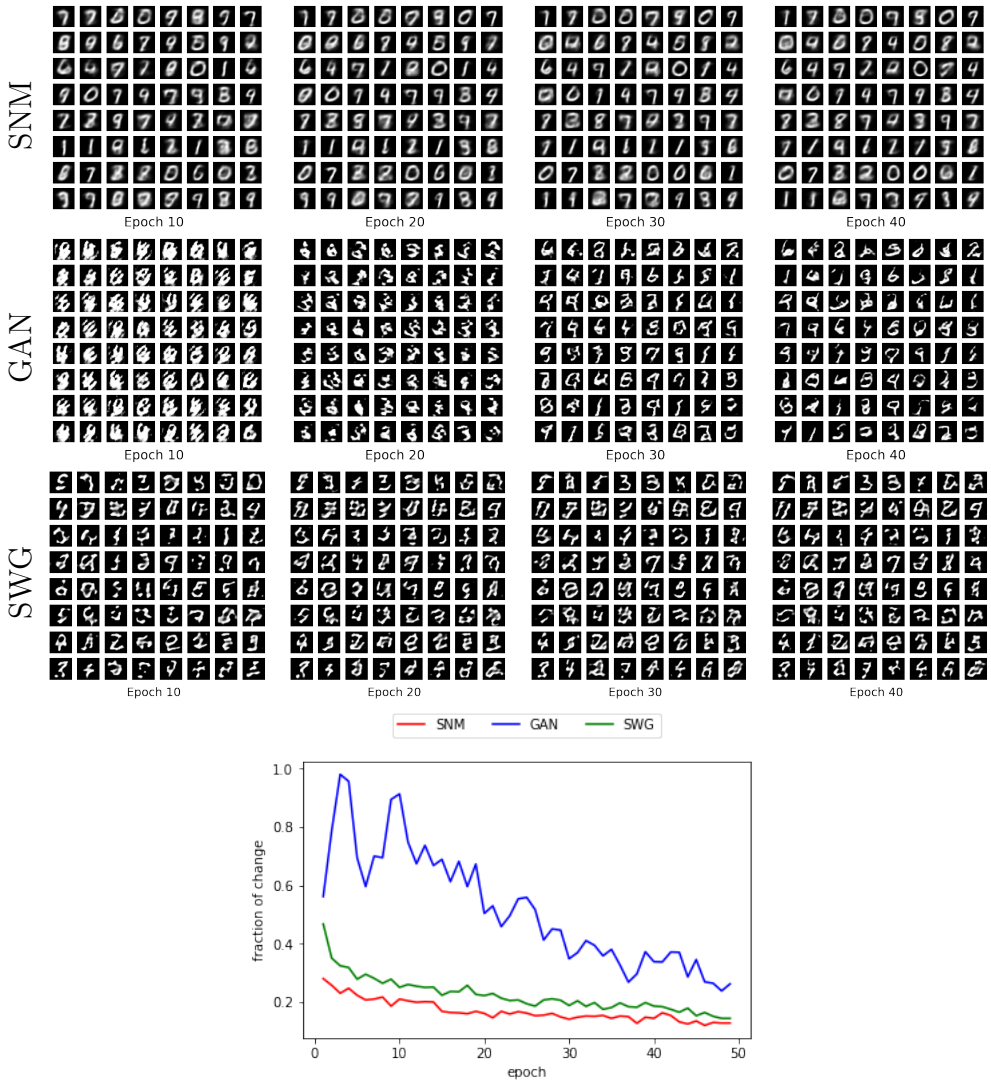


Figure 2.9: Comparing correlation between latent variable and generated samples during trainings of SNM, GAN, and SWG

each model for 50 epochs.

Result To quantify the correlation between the latent variable and generated samples, we define Fraction of Change (FoC) to measure the fraction of samples with changed labels. FoC is calculated as follows. First, we pre-train a classifier to predict labels of generated samples, and we specify a test latent variable with 1000 fixed locations. At the end of each epoch, we generate 1000 samples using the test latent variable. Then we use the classifier to predict their class labels and calculate the fraction of samples whose labels differ from the previous epoch. We calculate FoC from

epoch to epoch. The results are shown in Figure 2.9. As we can see from the figure, GAN performs well. Although its FoC fluctuates throughout its training process, the overall trend is decreasing. SWG demonstrates a strong stability. It maintains a low FoC score throughout its training. Also, its FoC drops from around 50% at the first epoch to below 15% at the last epoch. SNM is as consistent as SWG. Its FoC decreases from around 30% to below 13%, which means that more than 70% of samples generated from the same latent locations remain consistent in nature.

2.5.3 Experiment: Demonstrating Stability of SNM by Correlation between Sample Quality and Generator Loss *(SNM:GL:RD)*

In this experiment, we aim to demonstrate the robustness of the SNM training from another perspective, i.e., correlation between sample quality and generator loss. Following the previous experiments, we also compare its performance with that of GAN and SWG.

Dataset The experiment is performed on the MNIST dataset. The size of training data is 60K.

Network Architecture The basic generator archetype is DCGAN. All models adopt the same generator archetype. Each layer in the generator, except the last one, applies batch normalization and leaky rectifier. The last layer uses sigmoid activation.

Hyper-parameters All models use the same hyper-parameters. The optimizer used is the Adam optimizer and the learning rate is set to 0.0001. Batch size is set to 256. We train each model for 100 epochs.

Result Figure 2.10 shows the correlation between image quality and generator loss during the SNM training. As we can see from this figure, the generator loss of GAN fluctuates throughout the training process. Although the improvement of its sample

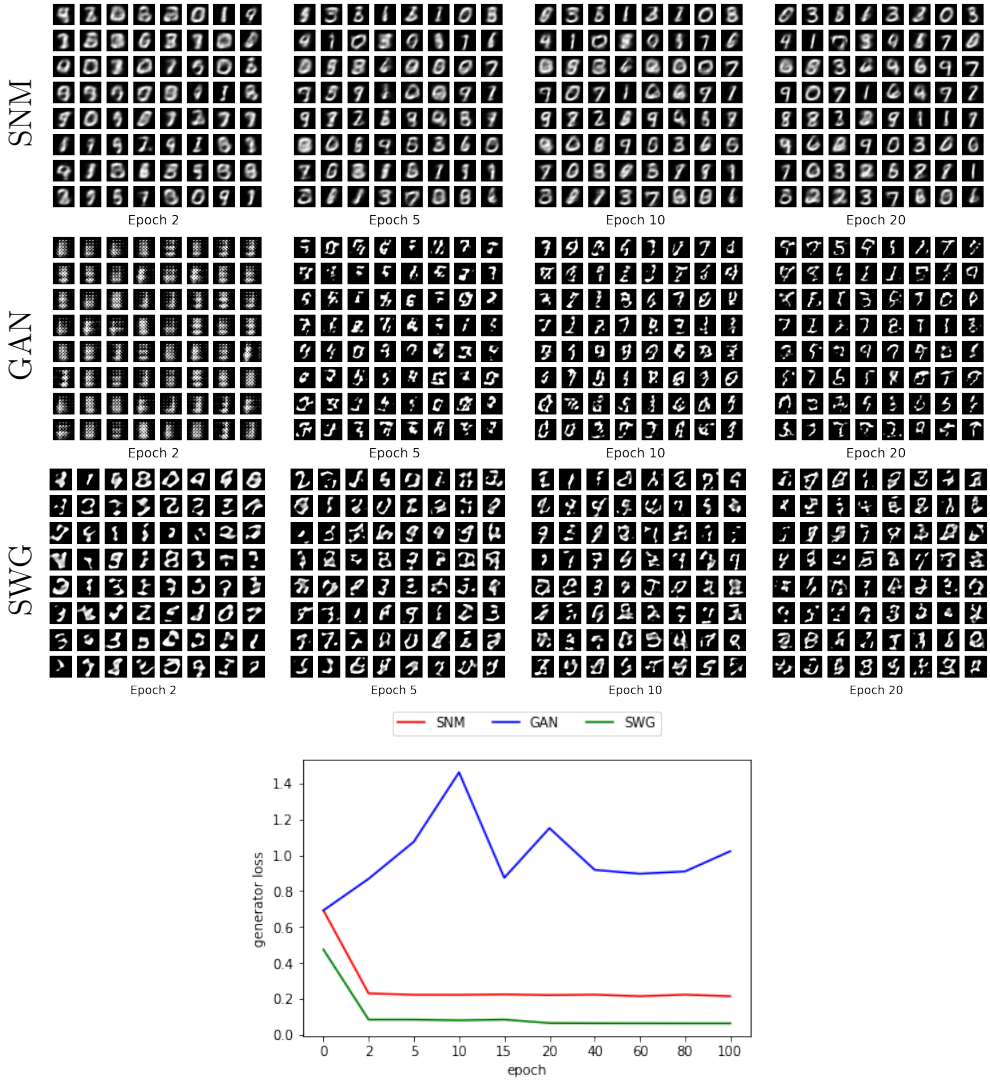


Figure 2.10: Comparing correlations between image quality and generator loss during trainings of SNM, GAN, and SWG

quality is clearly observed, it is hard to observe a correlated pattern between the generator loss and sample quality. SWG indicates its robustness during the training. Its generated images become sharper and better, as its generator loss decreases. SNM performs as robust as SWG, and a similar pattern can be observed from Figure 2.10.

2.5.4 Experiment: Attempts to Further Improve the SNM Training

In this final experiment, we explore possibilities of improving the SNM training in high dimensional space. The first attempt is Semantic SNM (S-SNM). The S-SNM training is similar to SNM, except the dissimilarity of samples is measured based on their hidden representations instead of raw data. After applying SNM to hidden representations, the loss is then applied on the original data space. To obtain semantic representation of samples, we first pre-train a classifier on the training data. The generated samples and the true samples are then fed to the classifier, and the representations in the second last layer are used as their semantic representations. For the classifier, all layers except the last two layers in the classifier use $rectifiers$ as activation functions. Its second last layer uses *sigmoid* function as the activation function, and its last layer uses a $softmax$ activation function. More details about the classifier can be found in Figure A.1 in the Appendix. The second attempt is to use the SNM training with ($|\mathbb{G}| < |\mathbb{X}|$), which is introduced in the experiment *SNM:LE:SD*.

The experimental results are shown in Figure 2.11. Based on the results, we can see that the improvement brought by semantic distance is not so obvious for SNM. However, we can observe improvements brought by training with $|\mathbb{G}| < |\mathbb{X}|$ for SNM. First, SNM with $|\mathbb{G}| < |\mathbb{X}|$ finds better matching between generated and real samples, which is indicated by the reduced loss values. Second, because of the better matching, its sample quality improves. More specifically, SNM with $|\mathbb{G}| < |\mathbb{X}|$ produces images that are sharper than those produced by the generator trained with the regular SNM.

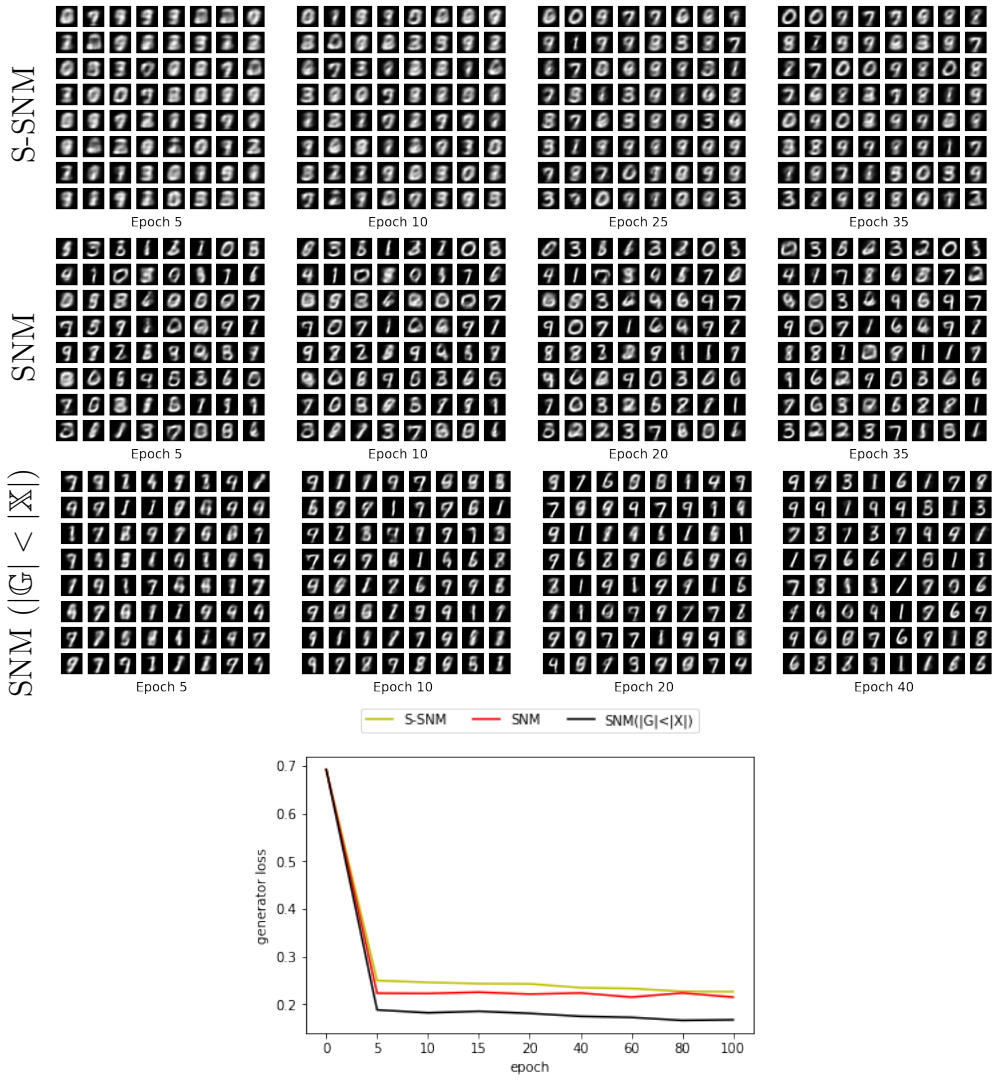


Figure 2.11: Attempts for improving SNM with semantic distance and training with $|\mathbb{G}| < |\mathbb{X}|$

2.6 Conclusion

In this chapter, we propose the SNM training method for unsupervised learning. We demonstrate that SNM is an effective and stable training method. Detailed analysis on its performance of synthetic data shows that SNM is effective. We demonstrate its robustness by three experiments on real world data. First, we show its robust performance across different hyper-parameter settings. Second, the stability of the SNM training is also indicated by a strong correlation between the latent variable and generated samples. Lastly, its robustness is suggested by the correlation between sample quality and generator loss. In future work, we would like to make SNM more efficient. Future studies should also investigate the combination of the SNM training and the adversarial training.

CHAPTER III

CHOICE CELL ARCHITECTURE

3.1 Introduction

Deep learning algorithms have been successfully applied to various applications, such as speech recognition, machine translation, object recognition, and drug discovery [7, 8, 9]. It is, however, challenging to understand the computation hidden inside these deep neural networks [18]. Thankfully, in recent years, researchers have increasingly realized the significant value of highly interpretable structure within the network, and significant efforts have been made to improve the interpretability of neural networks [18, 19, 20, 21]. The goal of this chapter is to improve interpretability of neural networks. We propose an explainable neural architecture and coin the newly developed architecture Choice Cell (CC). An advantage of CC is that its internal representations have an explainable meaning of probability distribution. In addition, we train CC with Stable Neighbor Match (SNM) introduced in Chapter II to demonstrate its effectiveness in learning various synthetic and real world datasets. The main contributions of this chapter are:

- We develop a new neural architecture called CC. The internal representation of CC has an intuitive interpretation of probability distribution.
- We train CC with SNM on various datasets, and the results demonstrate that CC can effectively learn distribution and encodings of input data. We also show that even in some cases of extremely imbalanced data, CC trained with SNM still shows its effectiveness in learning distribution of input data.

- We use regular generators as subnetworks and combine them through CC to form Choice Generator (CG). Our experimental results show that CG is not only a more interpretable generator but also generates meaningful data with comparable qualities with other popular generators. In addition, CG has an advantage of having its sub-generators learn a small set of classes of objects, which in turn enhances the transparency of its sub-generators.

- We also apply regularization to CG and show that regularization can make sub-generators of CG more homogeneous representations and further improve their transparency.

- The CC architecture that we propose in this chapter makes it much easier to quantify interpretability because its hidden representation can be reduced to probabilistic interpretation.

The organization of this chapter is as follows. In Section 3.2, we present previous studies related to CC. Then, we formally introduce CC architecture in Section 3.3. In Section 3.4, we report experimental findings of CC on synthetic datasets, which successfully demonstrates the effectiveness of the model that we build. We then further demonstrate the performance of CC by applying it to real world data in Section 3.5. The results show that CC can effectively learn distribution and encodings of real world data. Finally, we conclude with a summary of results in Section 3.6.

3.2 Literature Review

3.2.1 Interpretable Neural Network (NN)

The first line of research relevant to our work is interpretable NN. Significant progress has been made to understand NN representations and develop more interpretable NN architectures [18, 19, 20, 34, 35]. Interpretable NN can be divided into the following categories.

Understanding hidden representation of pre-trained NN The first category is the visualization of hidden representations of NNs, especially those representations encoded in filters. This is the most direct method that has been used to understand meanings inside blackbox neural units. There are different types of visualization techniques. The most frequently used approach is the gradient-based method [36, 37, 38], which computes gradients of filter units with respect to input pixels and then uses the gradient information to estimate input image. Another popular visualization technique is to invert visual representations with CNN. For example, [39] trains a network to predict the weighted average of all natural images which could have produced the given feature vector.

Explainable network component The second category is to build explainable network components whose goal is to learn meaningful and interpretable representations during the training of NNs. One of the most important studies in this direction is interpretable CNN [19]. The main rationale behind this approach is that each filter in the convolution layer is expected to be activated only by a certain part of objects of a certain class. Let $\boldsymbol{F}$ a feature map, which is a $n \times n$ matrix with $\boldsymbol{F}_{i,j} > 0$. During the forward pass, the CNN computes a feature map $\boldsymbol{F}$ of a filter f on a given image $\boldsymbol{I}$. The interpretable CCN estimates the position, μ, of the feature map $\boldsymbol{F}$ with the strongest activation. Then, a mask is assigned to $\boldsymbol{F}$ to produce a masked feature map $\boldsymbol{F}_{mask}$ based on the position μ. Since there are n^2 possibilities for μ, the paper designs a set of n^2 templates $\mathbb{T} = \{\boldsymbol{T}_{\mu_1}, \boldsymbol{T}_{\mu_2}, ..., \boldsymbol{T}_{\mu_{n^2}}\}$, where each template $\boldsymbol{T}_{\mu_i}$ is a $n \times n$ matrix representing the ideal distribution of activation for the feature map $\boldsymbol{F}$ whose (i, j) position has the strongest activation.

During the back-propagation, the author designs an additional regularization loss, L_f, to push a filter f towards representing a specific object part of a specific class c. The regularization works as follows. Let $\mathbb{I}$ be a set of training images and $\mathbb{I}_c$ be a set of training images of a specific class c. Let $\mathbb{F} = \{\boldsymbol{F} | \boldsymbol{F} = f(\boldsymbol{I}), \boldsymbol{I} \in \mathbb{I}\}$

be a set of feature maps of f for different training examples. Given an image $\boldsymbol{I} \in \mathbb{I}_c$, the feature map $\boldsymbol{F}$ is expected to be activated exclusively at a certain object part. In addition, the author adds a negative template $\boldsymbol{T}^-$ to the original templates become $\mathbb{T} = \{\boldsymbol{T}_{\mu_1}, \boldsymbol{T}_{\mu_2}, ..., \boldsymbol{T}_{\mu_{n^2}}, \boldsymbol{T}^-\}$. When $\boldsymbol{I} \notin \mathbb{I}_c$, the author hopes that the feature map x would match with the negative template $\boldsymbol{T}^-$. The regularization loss, L_f, is formulated as negative mutual information between $\mathbb{F}$ and $\mathbb{T}$.

Although the CC architecture that we develop is a continuation of this line of research, our work does have a few unique features. First, its internal encoding can explicitly be reduced to probability interpretation. Second, CC can be used in either supervised learning tasks or unsupervised learning tasks, while most of the other interpretable architectures are only designed for supervised learning.

3.2.2 Long Short-Term Memory

Our work also draws upon the concept of Long Short-Term Memory (LSTM), which is the modern architecture for Recurrent Neural Network (RNN). LSTM was introduced by Hochreiter and Schmidhuber to primarily overcome the problem of vanishing gradients [40]. Its main feature is that it adds a memory cell to the hidden layer in RNN. A memory cell consists of five basic elements: input node, input gate, internal state, forget gate, and output gate.

- input node (g): This is a standard neural unit that takes an input at current time step $\boldsymbol{x}_t$ and the hidden layer from the previous step $\boldsymbol{h}_{t-1}$. Typically, the activation function is *tanh* [22].

- input gate (i): This is also a neural unit that takes an input at current time step $\boldsymbol{x}_t$ and the hidden layer from the previous step $\boldsymbol{h}_{t-1}$, but its activation function is *sigmoid*. It is a *gate* because its value is used to modify other nodes. When its value is one, the value of other nodes can pass through the gate. When

its value is zero, it cuts off the flow of other nodes. As the name suggests, it modifies the value of input node.

- internal state (s): This is where the information about the memory cell retains. It is a state because information can flow across time steps. It is the main mechanism for multi-gating gradient vanishing and exploding.

- forget gate (f): It was introduced by [41]. Its main purpose is to control the amount of information that can be forgotten in the internal state. Similar to input gate, it uses sigmoid as its activation function. When its value is one, it allows the information of the internal state to pass through. When its value is zero, it cuts off the flow of the internal state.

- output gate (o): The output value of the memory cell is typically an element-wise product of internal state with *tanh* activation and output gate. Thus, this unit is termed output gate.

Thus, at any given time step t, a whole memory cell can be summarized below. Bias terms are omitted for brevity.

$$\boldsymbol{g}_t = tanh(\boldsymbol{U}_{gh}\,\boldsymbol{h}_{t-1} + \boldsymbol{W}_{gx}\,\boldsymbol{x}_t) \tag{3.1}$$

$$\boldsymbol{i}_t = \sigma(\boldsymbol{U}_{ih}\,\boldsymbol{h}_{t-1} + \boldsymbol{W}_{ix}\,\boldsymbol{x}_t) \tag{3.2}$$

$$\boldsymbol{f}_t = \sigma(\boldsymbol{U}_{fh}\,\boldsymbol{h}_{t-1} + \boldsymbol{W}_{fx}\,\boldsymbol{x}_t) \tag{3.3}$$

$$\boldsymbol{o}_t = \sigma(\boldsymbol{U}_{oh}\,\boldsymbol{h}_{t-1} + \boldsymbol{W}_{ox}\,\boldsymbol{x}_t) \tag{3.4}$$

$$\boldsymbol{s}_t = \boldsymbol{g}_t \odot \boldsymbol{i}_t + \boldsymbol{s}_{t-1} \odot \boldsymbol{f}_t \tag{3.5}$$

$$\boldsymbol{h}_t = tanh(\boldsymbol{s}_t) \odot \boldsymbol{o}_t\,, \tag{3.6}$$

where the vector h_t is the value of the hidden layer and the output value of the memory cell at time t.

3.2.3 Attention Model

A third area of research related to our work is Attention Model (AM) [42]. AM can be viewed as an extension of LSTM. The definitions of $\boldsymbol{s}_t$ and $\boldsymbol{h}_t$ remain the same, while the definitions of $\boldsymbol{g}_t, \boldsymbol{i}_t, \boldsymbol{f}_t, \boldsymbol{o}_t$ are extended to include a context term $\boldsymbol{z}_t$. So Equations 3.1 - 3.6 become

$$\begin{aligned}
\boldsymbol{g}_t &= tanh(\boldsymbol{U}_{gh}\,\boldsymbol{h}_{t-1} + \boldsymbol{W}_{gx}\,\boldsymbol{x}_t + \boldsymbol{W}_{gz}\,\boldsymbol{z}_t) \\
\boldsymbol{i}_t &= \sigma(\boldsymbol{U}_{ih}\,\boldsymbol{h}_{t-1} + \boldsymbol{W}_{ix}\,\boldsymbol{x}_t + \boldsymbol{W}_{iz}\,\boldsymbol{z}_t) \\
\boldsymbol{f}_t &= \sigma(\boldsymbol{U}_{fh}\,\boldsymbol{h}_{t-1} + \boldsymbol{W}_{fx}\,\boldsymbol{x}_t + \boldsymbol{W}_{fz}\,\boldsymbol{z}_t) \\
\boldsymbol{o}_t &= \sigma(\boldsymbol{U}_{oh}\,\boldsymbol{h}_{t-1} + \boldsymbol{W}_{ox}\,\boldsymbol{x}_t + \boldsymbol{W}_{oz}\,\boldsymbol{z}_t) \\
\boldsymbol{s}_t &= \boldsymbol{g}_t \odot \boldsymbol{i}_t + \boldsymbol{s}_{t-1} \odot \boldsymbol{f}_t \\
\boldsymbol{h}_t &= tanh(\boldsymbol{s}_t) \odot \boldsymbol{o}_t
\end{aligned}$$

Intuitively, $\boldsymbol{z}_t$ dynamically represents the relevant parts of images with respect to output $\boldsymbol{h}_t$ at time t. It can be computed from the annotation vectors $\boldsymbol{a}_i$, $i = 1, 2, ..., L$, which corresponds to the features extracted from raw input, and a positive score α_i representing relative importance of feature $\boldsymbol{a}_i$ for predicting next output $\boldsymbol{h}_t$. The set of annotation vectors $\{\boldsymbol{a}_1, \boldsymbol{a}_2, ..., \boldsymbol{a}_L\}$ are extracted from input data using lower layers of CNN. The relevance score at time t, α_{ti}, is computed as softmax of outputs, e_{ti} for $i = 1, ..., L$, of an AM, f_{att}. The output, e_{ti}, is defined as

$$e_{ti} = f_{att}(\boldsymbol{a}_i, \boldsymbol{h}_{t-1}),$$

where f_{att} is implemented as a multilayer perceptron.

Then, the context vector $\boldsymbol{z}_t$ is computed as blending the set of relevance scores at

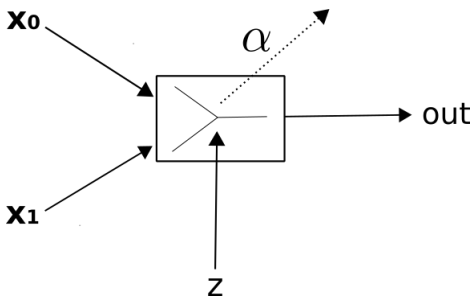


Figure 3.1: Abstract View of a Binary Choice Cell

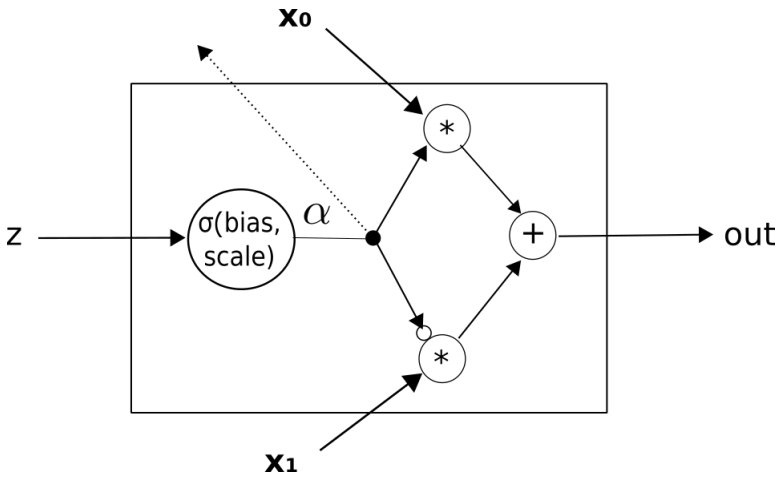


Figure 3.2: Detailed View of a Binary Choice Cell

time t and the set of annotation vectors together, i.e.,

$$\boldsymbol{z}_t = f_{ble}(\{\alpha_{t1}, \alpha_{t2}, ..., \alpha_{tL}\}, \{\boldsymbol{a}_1, \boldsymbol{a}_2, ..., \boldsymbol{a}_L\}).$$

A typical choice of blending function, f_{ble}, is to take $\boldsymbol{z}_t$ to be the weighted average of its inputs [43], i.e. $\boldsymbol{z}_t = \sum_{i=1}^{L}(\alpha_{ti}\boldsymbol{a}_i)$.

3.3 The Choice Cell Architecture

Intuitively, a Binary Choice Cell (BCC) behaves like the input gate, forget gate, and internal state introduced in Section 3.2. BCC is a gate-like mechanism that manipulates information flowing through it. A BCC can be represented graphically in Figure 3.1 and Figure 3.2. We now formally define BCC below.

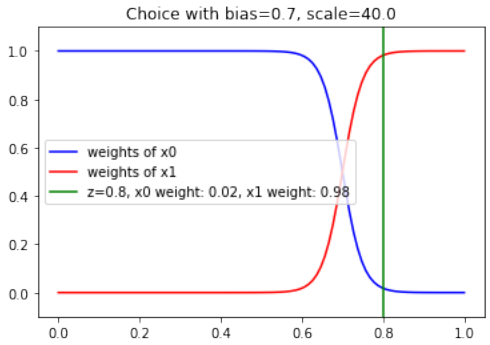


Figure 3.3: An example of a BCC

Definition III.1 *[Binary Choice Cell (BCC)] Let* $\boldsymbol{x}_0$, $\boldsymbol{x}_1$ *be input tensors and* z *be a scalar. Let bias and scale be the internal parameters. A BCC is a function*

$$bc(\boldsymbol{x}_0\,, \boldsymbol{x}_1\,, z; bias, scale) = \alpha * \boldsymbol{x}_0 + (1 - \alpha) * \boldsymbol{x}_1\,, \tag{3.7}$$

where

$$\alpha = \sigma((bias - z) * scale) \tag{3.8}$$

and α *can be outputted optionally.*

We design a BCC to behave like a fuzzy selection function that outputs a noisy version of one of its two input tensors $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$. To do so, additional constraints are imposed in BCC. First, the parameter *scale* is set to a relatively large number. Second, the random variable z is constrained to real numbers in the half-open unit interval $[0, 1)$. Figure 3.3 shows an example of BCC with $\sigma(0.7, 40)$ and $z \in [0, 1)$. The blue and red lines show the evolution of the weights of $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$ as z goes from 0 to 1. The green line shows a case where $z = 0.8$. In that case, the weight of $\boldsymbol{x}_0$, α, equals to 0.02, and the weight of $\boldsymbol{x}_1$, 1-α, equals to 0.98. The final output of the BCC is then $0.02 * \boldsymbol{x}_0 + 0.98 * \boldsymbol{x}_1$. As we can see from Figure 3.3, when $z < bias$ the BCC outputs a noisy version of $\boldsymbol{x}_0$, and when $z > bias$ it outputs a noisy version of $\boldsymbol{x}_1$. Since z is restricted to $[0, 1)$, *bias* also behaves like a probability threshold. The BCC has a probability of *bias* of "choosing" $\boldsymbol{x}_0$ and a probability of $1 - bias$ of

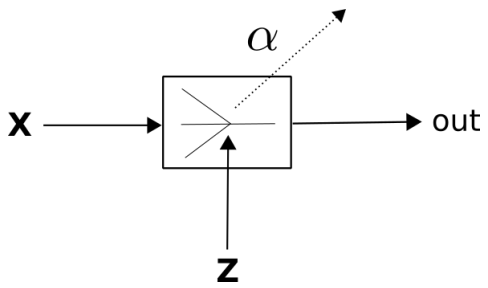


Figure 3.4: Abstract View of a Choice Cell

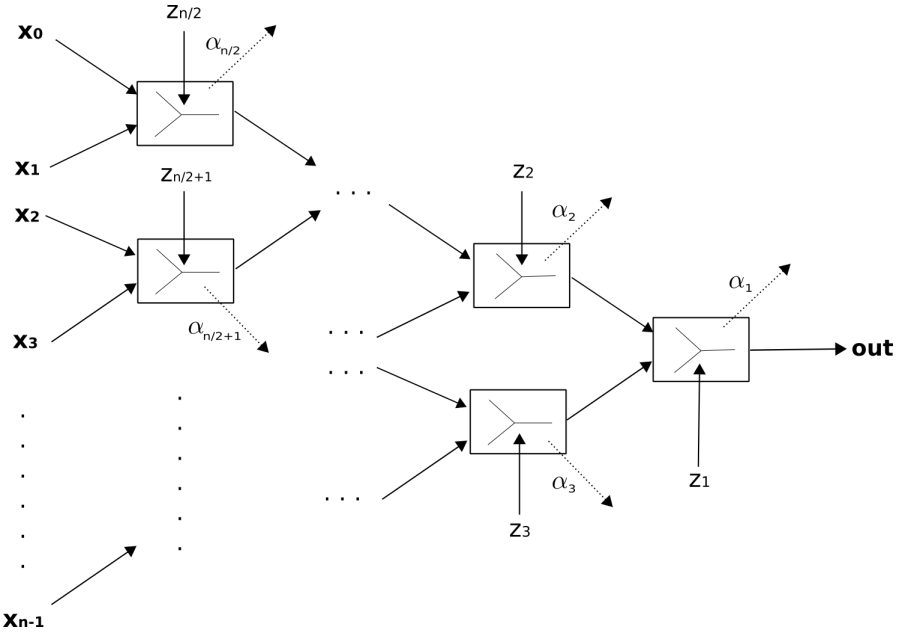


Figure 3.5: Detailed View of a Choice Cell

"choosing" $\boldsymbol{x}_1$. For this reason, the architecture is named Binary Choice Cell.

The formulation of BCC can be extended to the general case, Choice Cell (CC). A CC is a Complete Binary Tree where internal nodes provide a tournament of BCCs with input tensor placed in leaf nodes. Like BCC, it is also a gate-like mechanism that controls and modifies the information flowing through it, but it takes n input tensors instead of two. A CC can be represented graphically in Figures 3.4 and 3.5. We now define CC formally as below.

Definition III.2 *[Choice Cell (CC)] Let* $\{\boldsymbol{x}_0, \boldsymbol{x}_1, ..., \boldsymbol{x}_{n-1}\}$ *be a set of input tensors. A CC is a Complete Binary Tree with* $2n-1$ *nodes. Each input* $\boldsymbol{x}_i$ *is placed on the leaf node* $n+i$*. Each of its internal node* j *(*$j \leq n-1$*) represents*

a BCC $bc(\boldsymbol{out}_{2j}, \boldsymbol{out}_{2j+1}, z_j; bias_j, scale_j)$ *where* $\boldsymbol{out}_{2j}$ *and* $\boldsymbol{out}_{2j+1}$ *are outputs from nodes* $2j$ *and* $2j+1$*. The whole cell thus can be represented as a function* $cc(\boldsymbol{x}, \boldsymbol{z}; \boldsymbol{bias}, \boldsymbol{scale})$*, with* $\boldsymbol{\alpha}$ *as its optional output.*

Like BCC, we also design CC to behave like a fuzzy selection function that outputs a noisy version of one of its n input tensors. Thus, each internal parameter $scale_j$ is set to a relatively large number. Also, each value z in the random vector $\boldsymbol{z}$ is constrained to real numbers $[0, 1)$. Similar to BCC, the parameter vector $\boldsymbol{bias}$ forms a probability distribution. According to the distribution encoded internally, a CC outputs a noisy version of one of n input tensors. Formally, let $\mathbb{P}$ be a path from the root node of a CC to a leaf node $n+i$, excluding the leaf node $n+i$. Let $L(j), R(j)$ be a $0, 1$ indicators along the path, which indicate left branch and right branch, respectively. The probability of selecting the input tensor $\boldsymbol{x}_i$ is calculated as

$$P(\boldsymbol{x}_i) = \prod_{j \in \mathbb{P}} (\boldsymbol{bias}_j)^{L(j)} * (1 - \boldsymbol{bias}_j)^{R(j)} \tag{3.9}$$

CC is closely related to the gated mechanism LSTM and the attention mechanism in AM, but there are some fundamental differences. In LSTM, outputs of the input gate and forget gate are combined to produce the value in the internal state, and both gates can be on and off at the same time. In CC, however, we only allow one of its leaf nodes to be on at a time. In AM, the annotation vectors and their weighting scores are blended together to produce context vectors. CC, on the other hand, implements a fuzzy selection which produces a noisy version of one of outputs of its leaf nodes. Furthermore, both gated mechanism in LSTM and attention mechanism in AM are not transparent, while choice mechanism in CC can be reduced probability distribution which is human understandable.

3.4 Experiments of Choice Cell on Synthetic Data

In this section, we present experimental results from training CC on synthetic data. Experiments in this section are divided into four subsections. In the first subsection, we train CC to learn the distribution of input data. In the second subsection, we train CC to learn the encodings of input data. In the third subsection, we train CC to learn the distribution and the encodings of input data simultaneously. In the last two subsections, we combine CC with generators to form the Choice Generator (CG) where generators are placed on leaf nodes of CC as subnetworks. In the fourth subsection, we show that CG can learn encodings of 2D mixtures of Gaussian dataset. In the last subsection, we show that CG can learn both encodings and distribution of the 2D dataset.

3.4.1 Experiment: *Learning Distribution of Synthetic Data Using Choice Cell (CC:LD:SD)*

The goal of the experiment *CC:LD:SD* is to show that CC can learn the probability distribution of synthetic datasets. We assume that the generator knows the encodings of the input data before the training. Ideally, after the training, CC can learn to encode the probability distribution of input data in its internal nodes.

Dataset In the experiment *CC:LD:SD*, we use four synthetic datasets: 2E-1hot, 2E-Gen, 4E-1hot, and 4E-Gen. They are briefly described below.

- 2E-1hot: The sample space contains two one-hot encodings in 2D space, and the encodings obey a certain probability distribution. The dataset is formed by drawing encodings from the distribution.
- 2E-Gen: The sample space contains two general encodings in 2D space, and the encodings obey a certain probability distribution. The dataset is formed by drawing encodings from the distribution.

- 4E-1hot: The sample space contains four one-hot encodings in 4D space, and the encodings obey a certain probability distribution. The dataset is formed by drawing encodings from the distribution.

- 4E-Gen: The sample space contains four general encodings in 4D space, and the encodings obey a certain probability distribution. The dataset is formed by drawing encodings from the distribution.

Each dataset contains 6400 samples. Details about these datasets are summarized in Table A.1 in the Appendix.

Network architecture The network architecture of BCC follows Definition III.1. The network architecture of 4-nary CC follows Definition III.2 with $n = 4$. True encodings are placed on the leaf nodes of CCs. In all four cases, Cross Entropy is chosen as loss functions. Also, we adopt the SNM training method introduced in Chapter II to facilitate the training of CC. Thus, the SNM is first applied to generated samples and real samples to form matched pairs. The loss is then calculated based on matched samples according to Equation 2.6.

Hyper-parameters We use Adam optimizer with learning rate equal to 0.001 throughout the experiment *CC:LD:SD*. In addition, z is uniformly distributed over the half-open unit interval $[0, 1)$. Batch size is set to 64, and *scale* is set to 40. We train each network for 100 epochs.

Result We use Mean of Absolute Error (MAE) as a measurement for the performance of our CCs. The MAE is defined as

$$MAE(\hat{P}, P) = \frac{1}{|P|} \sum_{i=0}^{n-1} (\hat{P}_{\boldsymbol{x}_i} - P_{\boldsymbol{x}_i}), \tag{3.10}$$

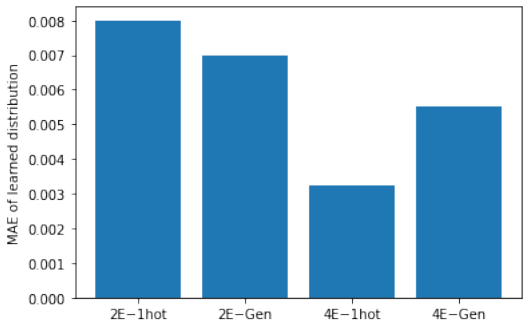


Figure 3.6: Errors of distribution learned by CC in the experiment *CC:LD:SD*. Each network is trained for 100 epochs.

where $\hat{P}$ and P are the estimated and true distributions, respectively, and $\hat{P}_{\boldsymbol{x}_i}$ is extracted from the ***bias*** in a CC according to Equation 3.9.

We first look at errors of learned distributions encoded in the internal nodes of CCs with the SNM training in Figure 3.6. As shown in the figure, in all four cases, CCs are able to learn the underlying probability distribution of the input data effectively. The MAEs of learned distributions in all four cases are below 0.1%. Considering the stochastic effect of the batch process, this number shows that the learned distribution is a fairly good approximation for the true distribution.

3.4.2 Experiment: *Learning Encodings of Synthetic Data Using Choice Cell (CC:LE:SD)*

The goal of the experiment *CC:LE:SD* is to show that CC can learn the encodings of synthetic datasets. We assume that encodings in the dataset are uniformly distributed, and the internal nodes of CC are fixed, not trainable during the training. Ideally, after the training, CC can learn to encode the encodings of input data in its leaf nodes.

Dataset Encodings in datasets used in this experiment are the same as those used in the experiment *CC:LD:SD*. However, all encodings are assumed to be uniformly distributed, and the main task of CC is to learn the encodings of input datasets.

Network architecture The network architecture of BCC follows Definition III.1. The network architecture of 4-nary CC follows Definition III.2 with $n = 4$. Instead of having true encodings placed on the leaf nodes, we place vector variables on the leaf nodes of CCs, and each of them is trained to learn the encoding of input data. Detailed information about network architectures can be found in Figure A.2 in the Appendix. SNM is applied to generated and real samples to produce the stable match $\mathbb{M}^s$, and then the final loss is calculated based on $\mathbb{M}^s$. Mean Absolute Error (MAE) is chosen as the *loss* function in Equation 2.6.

Hyper-parameters We use Adam as the optimizer, and the learning rate is set to 0.001 throughout the experiment. In addition, z is uniformly distributed over the half-open unit interval $[0, 1)$, and we set *scale* to 40. Batch size is set to 64, and we train each network for 100 epochs. Internal nodes are set to be non-trainable.

Result We use Relative Encoding Error (REE) as the performance measure for encoding learning. Since CCs do not control which vector variable would learn which target encoding, a mapping between learned encodings and true encodings is required in order to calculate error. We apply SNM to learned encodings and true encodings to form matched pairs. REE is defined based on the stable match as

$$REE(\boldsymbol{a}, \boldsymbol{b}) = \frac{\|\boldsymbol{a} - \boldsymbol{b}\|_2}{\|\boldsymbol{b}\|_2}, \tag{3.11}$$

where $\boldsymbol{a}, \boldsymbol{b}$ are learned and true encodings, respectively, and $(\boldsymbol{a}, \boldsymbol{b})$ is a pair in the stable match produced by SNM.

The experimental results for encoding learning are shown in Figure 3.7. The orange bar in the middle of each box refers to the median of REEs for each dataset. As shown in Figure 3.7, in all cases the CCs are able to learn encodings of input data successfully. For the 2E-Gen and 4E-Gen datasets, all REEs are below 0.1% of true

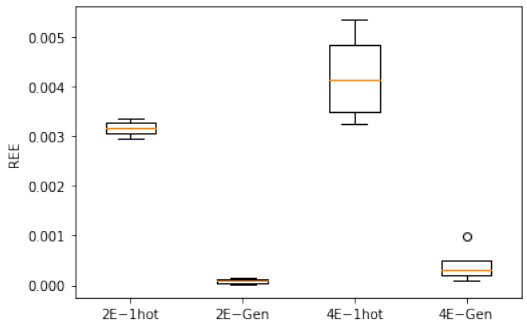


Figure 3.7: REEs in the experiment *CC:LE:SD*. Each network is trained for 100 epochs.

encodings. For the 2E-1hot and 4E-1hot datasets, the worst of REEs is about 0.5% of the target vector.

3.4.3 Experiment: *Learning Distribution and Encodings of Synthetic Data Using Choice Cell (CC:LDE:SD)*

The goal of the experiment *CC:LDE:SD* is to show that CC can simultaneously learn the probability distribution and encodings of synthetic input datasets. We assume that CC does not know encodings of the input data before the training. After the training, ideally the probability distribution of input data can be extracted and encoded in the internal nodes of CC, and the encodings of the input data can be learned in the leaf nodes of CC.

Dataset Datasets used in this experiment are the same as those used in the experiment *CC:LD:SD*. However, CCs do not know the true encodings of data in advance. Detailed information about the datasets can be found in Table A.1 in the Appendix.

Network architecture The network architecture of BCC follows Definition III.1. The network architecture of 4-nary CC follows Definition III.2 with $n = 4$. Instead of having true encodings placed on the leaf nodes, we place vector variables on the leaf nodes of CCs, and each of them is trained to learn the encoding of input data. Detailed information about network architectures can be found in Figure A.2 in the

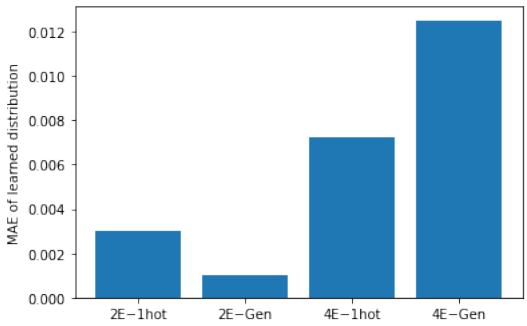


Figure 3.8: Errors of learned distribution in the experiment *CC:LDE:SD*. Each network is trained for 100 epochs.

Appendix. SNM is applied to match generated and real samples, and then the final loss is calculated based on matched samples. Mean Absolute Error (MAE) is chosen as the *loss* function in Equation 2.6.

Hyper-parameters We use Adam as the optimizer, and the learning rate is set to 0.001 throughout the experiment. In addition, z is uniformly distributed over the half-open unit interval $[0, 1)$, and we set *scale* to 40. Batch size is set to 64, and we train each network for 100 epochs.

Result Like the previous experiment, SNM is applied to match leaf nodes and true encodings before error calculations. We start with the performance of distribution learning of CCs. As in the experiment *CC:LD:SD*, we use MAE in Equation 3.10 as a measurement for distribution errors. The result is presented in Figure 3.8. As shown in the figure, in all four cases above, CCs are able to successfully learn the underlying probability distribution of the input data, even though now they are learning distributions and encodings of simultaneously. For the two binary datasets, the error rates are below 0.4%. For the 4E-1hot and 4E-Gen datasets, CCs can reach error rates around 0.8% and 1.2%, respectively.

Next, we turn to the result for encoding learning. Still, we use REE defined in Equation 3.11 as the performance measure. The experimental results are shown in Figure 3.9. As shown in the figure, in all cases CCs are able to learn encodings of

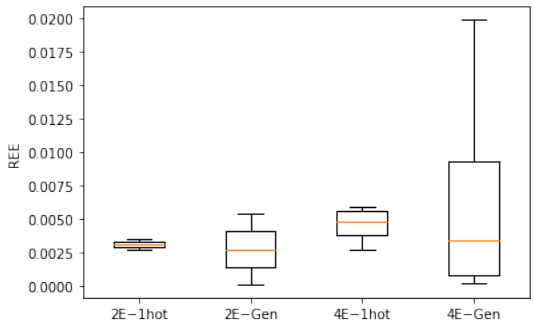


Figure 3.9: REEs in the experiment *CC:LDE:SD*. Each network is trained for 100 epochs.

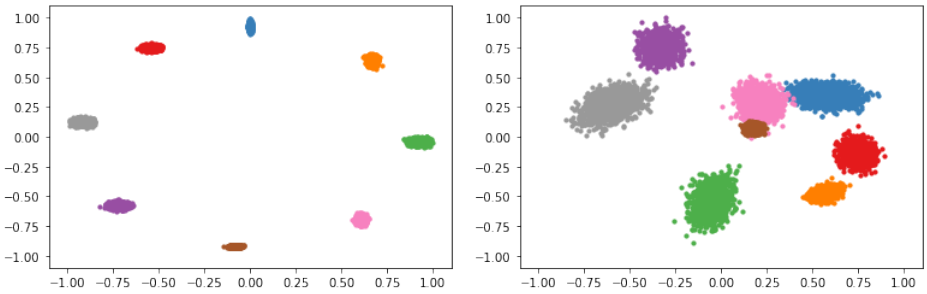
input data successfully. For all four datasets, the medians of REEs are around 0.5% of true encodings. The largest REE is about 2% of the target vector, which is observed in learning the more complicated 4E-Gen dataset.

3.4.4 Experiment: *Learning Encodings of 2D Synthetic Data Using Choice Generator (CG:LE:SD)*

The goal of this experiment is to show that CG with fixed internal nodes can learn encodings of 2D mixtures of Gaussian dataset. We assume that Gaussian clusters in the dataset are uniformly distributed and the internal nodes of CC are fixed, i.e., the internal nodes are not trainable during the training. Ideally, after the training, CC can learn to encode the encodings of all clusters in its leaf nodes.

Dataset We train CG to learn encodings of two synthetic datasets with different complexity: 8C-Ban-Sep and 8C-Ban-Insep. Each dataset contains 8 evenly distributed clusters, and each cluster consists of 2D data points drawn from a multivariate normal distribution characterized by a mean matrix and a diagonal covariance matrix. More detailed descriptions of 8C-Ban-Sep and 8C-Ban-Insep are presented below.

- 8C-Ban-Sep: This is a 8 Cluster dataset where clusters are balanced and separated. To help form separated clusters, its mean and covariance matrices are hand-coded. The design of this dataset is inspired by the literature [26].



(a) the 8C-Ban-Sep dataset. Each cluster contains 1250 data points.

(b) the 8C-Ban-Insep dataset. Each cluster contains 1250 data points.

Figure 3.10: Sample data used in the experiment *CG:LE:SD*.

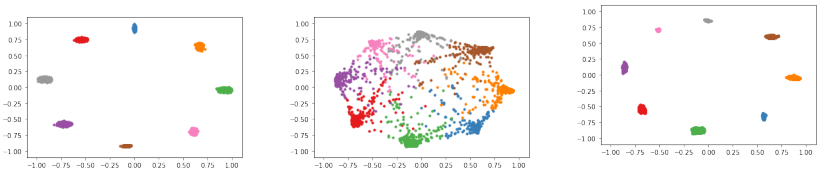
- 8C-Ban-Insep: This is a 8 Cluster dataset where clusters are balanced and some clusters are inseparable. In addition, we apply each cluster with a random rotation matrix to further increase the complexity of this dataset.

Sample data for the datasets are presented in Figure 3.10.

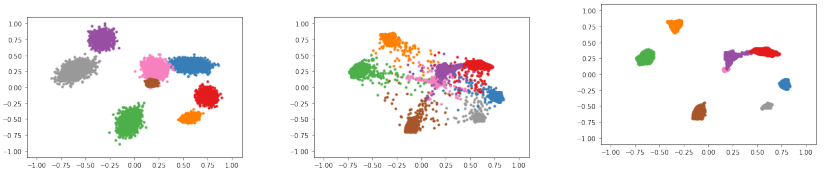
Networks We use a simple architecture for the generator of CG, following the literature [26]. The generator is a 4-layer fully connected network. The latent dimension is 20. Each hidden layer contains 48 nodes and uses the rectifier as activation function. The output layer has dimensionality of 2 and uses hyperbolic tanh function as its activation function. As presented in previous sections, the SNM algorithm is applied to generated and real samples to form a stable match $\mathbb{M}^s$, and then the final loss is calculated based on $\mathbb{M}^s$. The *loss* function in Equation 2.6 uses MAE.

Hyper-parameter We use Adam optimizer with learning rate equal to 0.001. In addition, z is uniformly distributed over the half-open unit interval $[0, 1)$, and we set *scale* to 40. Batch size is set to 512. We train each model on each dataset for 500 epochs.

Results We start with the qualitative measure of performance for CG. Generated samples are presented in Figure 3.11, where samples generated by CG are plotted in the last two columns, and samples generated by the same subnetwork are presented

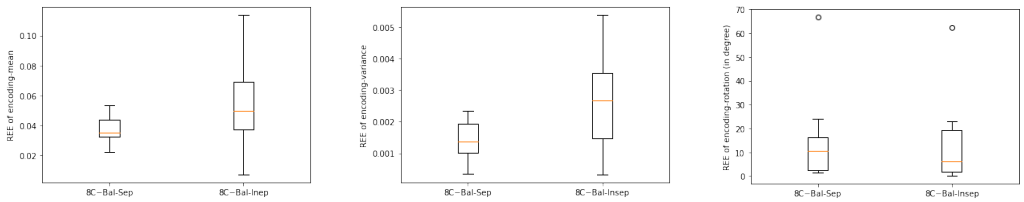


(a) 8C-Ban-Sep (b) samples from CG (c) samples from subnetworks of CG



(d) 8C-Ban-Insep (e) samples from CG (f) samples from subnetworks of CG

Figure 3.11: Samples generated by CG. (b)-(c): samples generated for 8C-Ban-Sep; (e)-(f): samples generated for 8C-Ban-Insep. Each of the aforementioned sub-figure contains 4096 generated samples. Each model is trained for 500 epochs.



(a) errors of encoding mean (b) errors of encoding variance (c) errors of encoding rotation

Figure 3.12: Quantitative error measure of learned encodings of CG. Encodings are from eight generators placed in leaf nodes of CG.

by the same color. From Figures 3.11b and 3.11e, we can see that CG can successfully learn encodings of all 8 clusters of data. In addition, in Figures 3.11c and 3.11f, we can see that each subnetwork of CG focuses on learning encoding of one cluster of samples. Even for the inseparable clusters (blue, pink, brown) for the 8C-Ban-Insep dataset, CG successfully makes each of the subnetworks (red, purple, pink) learn encoding of a specific cluster.

Next, we look at various quantitative error measures of learned encodings of CG. Since a specific goal is to have each subnetwork focus on a specific cluster, we incorporate the mismatch between subnetwork and cluster into our error measure. To do so, we first calculate centroids of generated samples and true samples. Then, we apply SNM to produce a stable match between learned and true centroids. After matching subnetworks with clusters, we calculate cluster mean error, cluster variance error,

and cluster rotation error. The measurement of cluster mean error is based on REE in Equation 3.11, where $\boldsymbol{a}$ refers to a centroid learned by a subnetwork and $\boldsymbol{b}$ refers to its partner in the stable match. To calculate cluster variance error and cluster rotation error, we first compute covariance matrices of generated and real samples. Then we apply Singular Value Decomposition to the covariance matrices to find the largest variances and the angle between axes associated with the largest variances. After that, we calculate MAE of the largest variances as variance error and MAE of the angle as rotation error.

The encoding errors are shown in Figure 3.12. We can see that CG successfully models the clusters in both datasets, as shown by low errors with respect to encoding means, variances, and rotations. The median errors of encoding means and variances for both datasets are below 5% and 0.003, respectively. The median MAEs of encoding rotation are around 10°.

3.4.5 Experiment: *Learning to Generate 2D Synthetic Data Using Choice Generator (CG:LDE:SD)*

In this experiment, we combine CC with generators to form CG where generators are placed on leaf nodes of CC as subnetworks. The goal of the experiment is to show that CG can learn to generate 2D synthetic data with increasing complexity. We would like CG to learn encodings and distribution of synthetic data simultaneously.

In addition, we would like each subnetwork of CG to generate a specific class or a small set of classes of samples in input data. However, since both generators in subnetworks of CC and internal nodes of CC have much extra freedom, additional regularization is needed to gear each generator towards a smaller set of classes of objects. We propose two regularization techniques to achieve this goal, and we also show the effects of adopting these regularizations. Note that the regularizations that we propose is only an initial exploration for regulating CG, and there are many

other possible regularization methods. Since our main focus of this chapter is not on regularization, we only show that regularization would help CG learn to model data more effectively and do not attempt to exhaustively test all regularizations to find the best one for CG.

The regularization consists of between-class regularization and within-class regularization, and we propose two ways to perform regularization. Two methods differ only in how we form matched samples for calculating regularization loss. The first matching method is defined below. Let $\mathbb{U}$ be a set of tuples $(\boldsymbol{a}, i)$, i.e. $\mathbb{U} = \{(\boldsymbol{a}, i)\}$, where $\boldsymbol{a}$ is a sample generated by a subnetwork of CG and i is the index of the subnetwork. Note that $\mathbb{U}$ contains samples from all subnetworks. Let $\mathbb{M}_T \subset \mathbb{U} \times \mathbb{U}$ be a temporary set of pairs after matching $\mathbb{U}$ with itself using SNM but excluding self-matching, i.e. $\mathbb{M}_T = \{((\boldsymbol{a}, i), (\boldsymbol{b}, j))|(\boldsymbol{a}, i) \in \mathbb{U} \wedge (\boldsymbol{b}, j) \in \mathbb{U} \wedge \boldsymbol{a} \neq \boldsymbol{b} \wedge \mathbb{M}^s(\boldsymbol{a}) = \boldsymbol{b}\}$. Then based on $\mathbb{M}_T$, we define $\mathbb{M}_{within}$ to be a set of sample pairs coming from the same subnetwork, i.e. $\mathbb{M}_{within} = \{(\boldsymbol{a}, \boldsymbol{b})|((\boldsymbol{a}, i), (\boldsymbol{b}, j)) \in \mathbb{M}_T \wedge i = j\}$. $\mathbb{M}_{between}$ is defined as a set of sample pairs coming from the different subnetworks, i.e. $\mathbb{M}_{between} = \{(\boldsymbol{a}, \boldsymbol{b})|((\boldsymbol{a}, i), (\boldsymbol{b}, j)) \in \mathbb{M}_T \wedge i \neq j\}$. Since the matching happens among all samples, we call the first method all-to-all (a2a) regularization.

The second matching method is defined below. The sample set $\mathbb{U}$ follows the same definition. Let $\mathbb{T}_{Wi}$ be a temporary set of all pairs within class i, excluding self pairs, i.e. $\mathbb{T}_{Wi} = \{(\boldsymbol{a}, \boldsymbol{b})|(\boldsymbol{a}, i) \in \mathbb{U} \wedge (\boldsymbol{b}, i) \in \mathbb{U} \wedge \boldsymbol{a} \neq \boldsymbol{b}\}$. Let $\mathbb{M}_{within}$ be the union of the sets of matched samples within a class using SNM, i.e. $\mathbb{M}_{within} = \bigcup_{i=0}^{n-1} \{(\boldsymbol{a}, \boldsymbol{b})|(\boldsymbol{a}, \boldsymbol{b}) \in \mathbb{T}_{Wi} \wedge \mathbb{M}^s(\boldsymbol{a}) = \boldsymbol{b}\}$. Let $\mathbb{T}_B$ be a temporary set of all between class pairs, i.e. $\mathbb{T}_B = \{(\boldsymbol{a}, \boldsymbol{b})|(\boldsymbol{a}, i) \in \mathbb{U} \wedge (\boldsymbol{b}, j) \in \mathbb{U} \wedge i \neq j\}$. Let $\mathbb{M}_{between}$ be the union of the sets of matched samples between classes using SNM, i.e. $\mathbb{M}_{between} = \{(\boldsymbol{a}, \boldsymbol{b})|(\boldsymbol{a}, \boldsymbol{b}) \in \mathbb{T}_B \wedge \mathbb{M}^s(\boldsymbol{a}) = \boldsymbol{b}\}$. Since the matching for between class happens between one subnetwork with all other subnetworks, we call the second method one-to-other (o2o) regularization. Notice the difference between the two matching methods. In the first method, every sample is

either in the $\mathbb{M}_{within}$ set or the $\mathbb{M}_{between}$ set, while in the second method, every sample is both the $\mathbb{M}_{within}$ set and the $\mathbb{M}_{between}$ set.

After the matching, the regularization loss is calculated based on matched samples. Let the distance between two vectors be $d(\boldsymbol{a}, \boldsymbol{b}) = \|\boldsymbol{a} - \boldsymbol{b}\|_1$. Define the within-class loss, L_{within}, as

$$L_{within} = \frac{1}{|\mathbb{M}_{within}|} \sum_{(\boldsymbol{a},\boldsymbol{b})\in\mathbb{M}_{within}} \frac{d(\boldsymbol{a}, \boldsymbol{b})}{d_{max}}, \tag{3.12}$$

where d_{max} is the maximum distance in $\mathbb{M}_{within}$. Define the between-class loss, $L_{between}$, as

$$L_{between} = \frac{1}{|\mathbb{M}_{between}|} \sum_{(\boldsymbol{a},\boldsymbol{b})\in\mathbb{M}_{between}} e^{-d(\boldsymbol{a},\boldsymbol{b})} \tag{3.13}$$

The final regularization loss, L_{reg}, is defined as

$$L_{reg} = \lambda_{between} * L_{between} + \lambda_{within} * L_{within}, \tag{3.14}$$

where $\lambda_{between}$ and λ_{within} are scale constants.

Dataset We train CG on four synthetic datasets with increasing complexity: 8C-Ban-Sep, 8C-Ban-Insep, 8C-Imban-Sep, and 8C-Imban-Insep. Each dataset contains 8 clusters of 2D data points, and each cluster is drawn from a multivariate normal distribution characterized by a mean matrix and a diagonal covariance matrix. To make the dataset imbalanced, the number of samples in each cluster is determined by a geometric distribution

$$n_c = n * p * (1 - p)^c, \tag{3.15}$$

where $c = 0, 1, 2, ...$ is a cluster or class label, and p is a scalar parameter. n is the number of samples in a dataset. To summarize, the four datasets have the following characteristics.

- 8C-Ban-Sep: This is a 8 Cluster dataset where clusters are balanced and sepa-

rated, and it is the same as the one in the previous subsection.

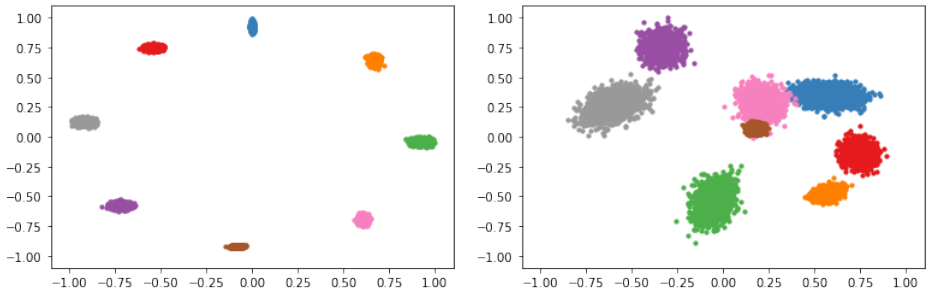
- 8C-Ban-Insep: This is a 8 Cluster dataset where clusters are balanced and some clusters are inseparable, and it is the same as the one in the previous subsection.

- 8C-Imban-Sep: This dataset is like 8C dataset but clusters are imbalanced. To make the dataset imbalanced, the number of samples in each cluster is determined by Equation 3.15. n is set to 10000, and p is set to $1/2$.

- 8C-Imban-Insep: This is a 8 Cluster dataset where clusters are imbalanced and some clusters are inseparable. Its mean and covariance matrices are drawn from random uniform distributions. To make the dataset imbalanced, the number of samples in each cluster is determined by Equation 3.15. n is set to 10000, and p is set to $1/2$. In addition, we apply each cluster with a random rotation matrix to further increase the complexity of this dataset.

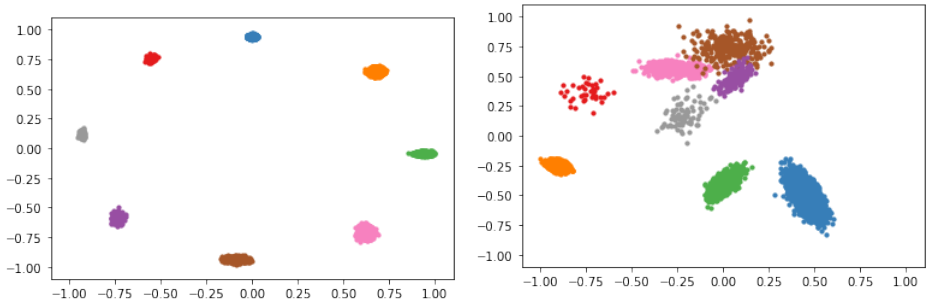
Sample data for the datasets are presented in Figure 3.13.

Networks We follows the literature [26] for the architecture of generators. For GAN, the generator is a 4-layer fully connected network. The latent dimension is 20. Each hidden layer contains 128 nodes and uses the rectifier as an activation function. The output layer has dimensionality of 2 and uses hyperbolic tanh function as activation function. In addition, the discriminator is also a fully connected network, and the minimax optimization follows the original formulation. For CG, generators are placed on the leaf nodes of CC. Each generator on the leaf node has the same structure as the generator in GAN, except that the number of nodes in each hidden layer is reduced to 48 so that the total number of parameters in generators of CG is comparable to the number of parameters in the generator of GAN. Like previous experiments, the SNM algorithm is applied to generated and real samples to output the stable match $\mathbb{M}^s$, and then the final loss is calculated based on the matched



(a) the 8C-Ban-Sep dataset. Each cluster contains 1250 data points.

(b) the 8C-Ban-Insep dataset. Each cluster contains 1250 data points.



(c) the 8C-Imban-Sep dataset. number of points in each cluster: [5000 (blue), 2500, 1250, 625, 312, 156, 78, 39 (red)]

(d) the 8C-Imban-Insep dataset. number of points in each cluster: [5000 (blue), 2500 (orange), 1250 (green), 625 (pink), 312 (brown), 156 (purple), 78 (grey), 39 (red)]

Figure 3.13: Sample data used in the experiment *CG:LDE:SD*.

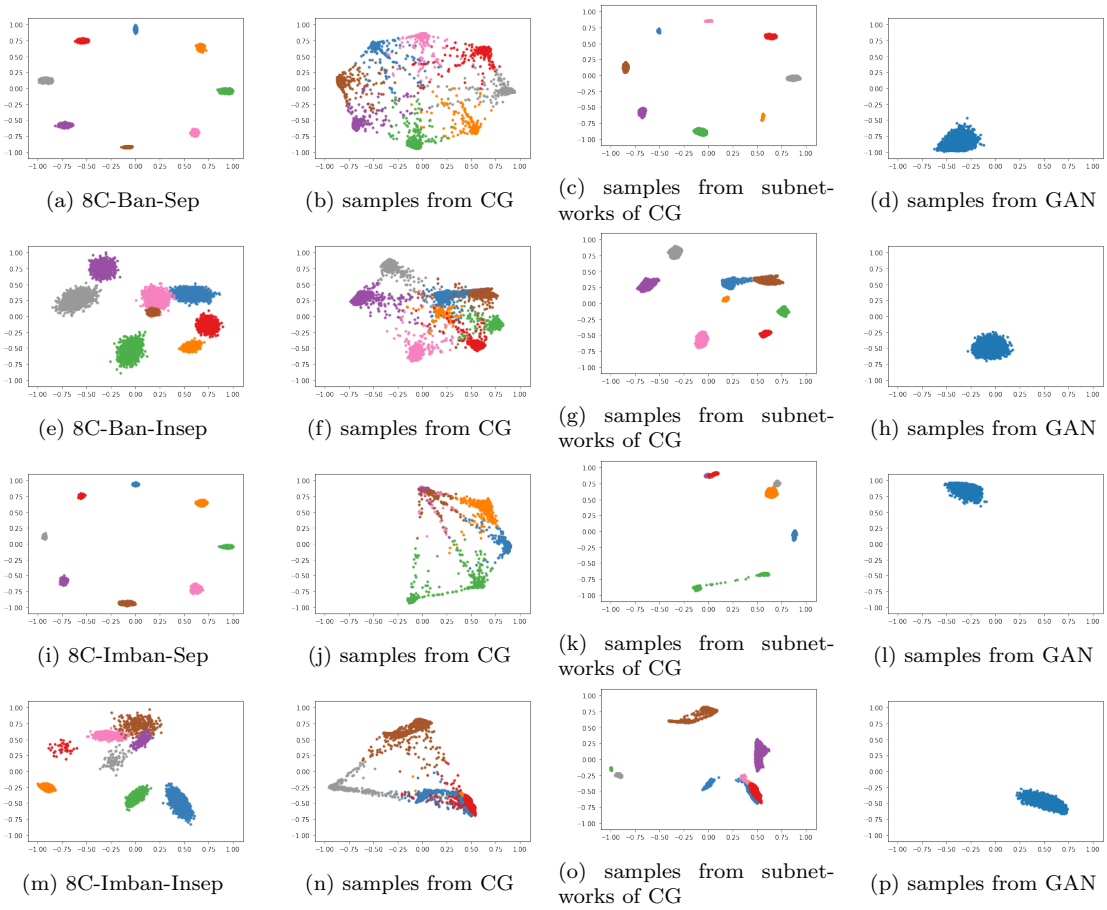
samples. MAE is used in the place of the *loss* function in Equation 2.6.

Hyper-parameter We use Adam optimizer with learning rate equal to 0.001 through out the experiment *CG:LDE:SD*. In addition, z is uniformly distributed over the half-open unit interval $[0, 1)$, and we set *scale* to 40. Batch size is set to 512. We train the each model on each dataset for 500 epochs. For regularization, $\lambda_{between}$ is set to 0.05, and λ_{within} is set to 0.5, so that the regularization does not overshadow the original loss.

Results without regularization We start with the qualitative measure of performance for CG and compare it with GAN. Generated samples are presented in Figure 3.14, where samples generated by CG are plotted in the two middle columns, and samples generated by the same subnetwork are presented by the same color. We first look at the results for the balanced datasets 8C-Ban-Sep and 8C-Ban-Insep, shown

in Figures 3.14a - 3.14h. From Figures 3.14b and 3.14f, we can see that CG can successfully generate all 8 clusters of data. In addition, in Figures 3.14c and 3.14g, we can see that each subnetwork of CG focuses on generating samples of one cluster. We argue that this is a great advantage of CG that its subnetwork tends to learn more homogeneous representations. On the other hand, the samples generated by GAN are presented in Figure 3.14d. We can see that GAN suffers the issue of mode dropping [8, 16] and only generates samples belonging to one cluster. Moving to the next case, the 8C-Imban-Sep dataset, the results are presented in Figures 3.14i - 3.14l. From Figure 3.14j, we can see that CG can successfully generate top five most frequent clusters but fails to generate the three least frequent ones. Looking closer at samples from its subnetworks in Figure 3.14k, we find that some subnetworks (e.g. green one) focus on multiple clusters while some subnetworks (e.g. orange and grey) conform to the same cluster. This is the experimental evidence that shows regularization is needed for subnetworks to learn more homogeneous representations. For GAN, again we can see from Figure 3.14l that GAN only generates samples belonging to one cluster. Further, notice that the cluster that GAN learns to generate is the most frequent cluster (blue cluster) among all. It means that its generator learns a cluster that is most likely to fool the discriminator. Lastly, the results of the most complicated case, the 8C-Imban-Insep dataset, are shown in Figures 3.14m - 3.14p. We observe a similar phenomenon as in the case of 8C-Imban-Sep, that is subnetworks successfully learn top five most frequent clusters (blue, orange, green, pink, brown) but fails to learn the three least frequent clusters (purple, grey, pink). For GAN, we observe the same result, that is it only learns to generate the most frequent cluster.

Next, we look at quantitative measures of performance of CG. We still apply SNM to subnetwork encodings and cluster encodings to build a stable match between subnetworks and clusters. The distribution error is measured by MAE in Equation 3.10 and is calculated based on the stable match. In addition, to better assess the perfor-



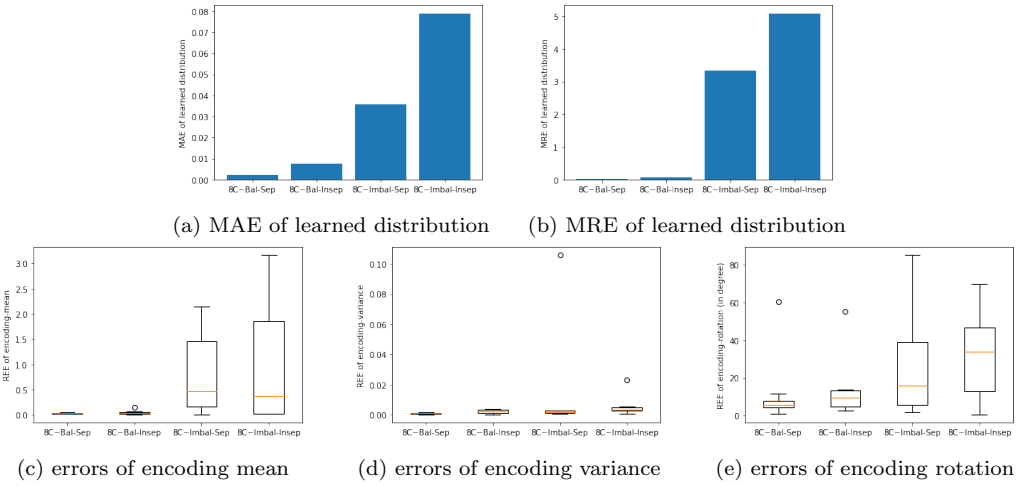
(a) 8C-Ban-Sep (b) samples from CG (c) samples from subnetworks of CG (d) samples from GAN

(e) 8C-Ban-Insep (f) samples from CG (g) samples from subnetworks of CG (h) samples from GAN

(i) 8C-Imban-Sep (j) samples from CG (k) samples from subnetworks of CG (l) samples from GAN

(m) 8C-Imban-Insep (n) samples from CG (o) samples from subnetworks of CG (p) samples from GAN

Figure 3.14: Samples generated by CG without regularization and GAN. (b)-(d): samples generated for 8C-Ban-Sep; (f)-(h): samples generated for 8C-Ban-Insep; (j)-(l): samples generated for 8C-Imban-Sep; (n)-(p): samples generated for 8C-Imban-Insep. Each of the aforementioned sub-figure contains 4096 generated samples. Each model is trained for 500 epochs.



(a) MAE of learned distribution (b) MRE of learned distribution

(c) errors of encoding mean (d) errors of encoding variance (e) errors of encoding rotation

Figure 3.15: Quantitative measures of the performance of CG without regularization

mance in imbalanced datasets, we introduce another error measure, Mean Relative Error (MRE), to measure distribution error. MRE is defined as

$$MRE(\hat{P}, P) = \frac{1}{|P|} \sum_{i=0}^{n-1} \frac{(\hat{P}_i - P_j)}{P_j}, \tag{3.16}$$

where $\hat{P}, P$ are the estimated and true distributions, respectively, and (i, j) is a pair in the stable match between subnetworks and clusters.

The results of distribution errors are shown in Figures 3.15a and 3.15a. As we can see from these figures, for the 8C-Ban-Sep and 8C-Ban-Insep datasets, CG learns the cluster distribution successfully, with MAE and MRE of distributions for the 8C-Ban-Sep dataset are around 0.22% and 1.9%. For the 8C-Ban-Insep dataset, MAE and MRE of distributions are around 0.7% and 6%, respectively. For the two imbalanced datasets, CG can learn the distribution in more frequent clusters, which is indicated by the moderate MAE of distribution. As indicated by MRE of distribution, CG fails to learn the distributions of less frequent clusters.

The REE for encodings are shown from Figure 3.15c to Figure 3.15e. From these three figures, we can see that CG accurately models the clusters in the 8C-Ban-Sep and 8C-Ban-Insep datasets. Errors with respect to mean, variance, and rotation are fairly low. For the 8C-Imban-Sep and 8C-Imban-Insep datasets, we can observe increases in all three error terms. We can also observe from Figure 3.15c that the most frequent clusters are learned pretty well by subnetworks, as indicated by errors under the orange bars. The increase of encoding errors is mainly driven by those less frequent clusters that are not learned by CG.

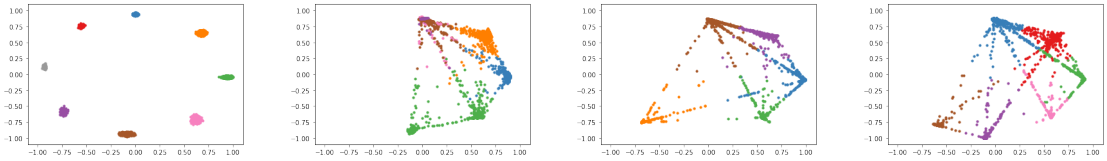
The experimental results show that the extremely imbalanced nature of the 8C-Imban-Sep and 8C-Imban-Insep datasets causes CG to fail to recognize the least frequent clusters, therefore increasing both distribution and encoding errors. Next, we present the results of applying regularization to the training of CG and show that

regularization helps subnetworks of CG learn more homogeneous representation and improve its performance.

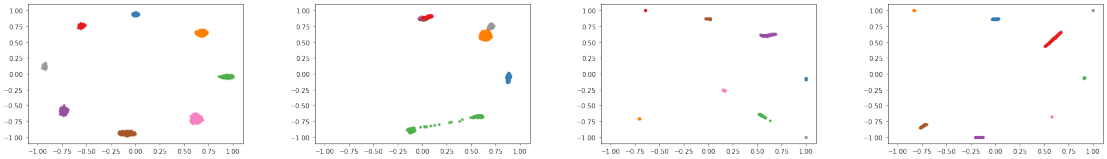
Results with regularization We now turn to the results of CG with regularization. Since results of the 8C-Ban-Sep and 8C-Ban-Insep datasets are fairly good, we only demonstrate regularization on the 8C-Imban-Sep and 8C-Imban-Insep datasets. Again, we start with sample quality of CG. The results are shown in Figure 3.16. In Figure 3.16h, we can see that the o2o regularization gives a significant performance improvement in learning the 8C-Imban-Sep dataset. Now six subnetworks have one-to-one match with the true cluster. One subnetwork (orange) is a bit off the target (grey), which is the second least frequent cluster. Only one subnetwork (grey) is off the target, and only one cluster (red), which has extremely low probability, is not learned by any subnetwork. The reason for the mismatch might be that the regularization error overshadows the original sample distance for the least frequent pairs.

For the task of learning the 8C-Imban-Insep dataset, samples generated by subnetworks are presented in Figure 3.16o and 3.16p. As we can see from the figures, both regularizations help make subnetworks more homogeneous. For a2a regularization, we can observe an improvement for having one-to-one learning between subnetworks and true clusters for the most frequent clusters (orange to blue, grey to orange, purple to green, red to pink). For o2o regularization, we can also observe an improvement for having one-to-one learning (blue-purple, purple-grey, green-orange, grey-green, orange pink-blue, red-brown pink) between subnetworks and true clusters. Only one cluster (red), which is the least frequent cluster, is not learned by any subnetwork, and only one subnetwork (brown) is off the target. As with the previous case, the reason might be that the regularization error overshadows the original samples distance for the least frequent pairs.

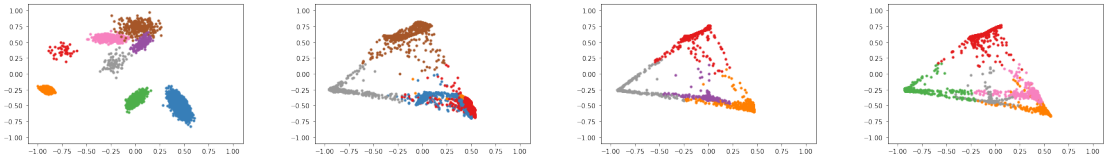
Next, we turn to quantitative measures for the performance of CG with regular-



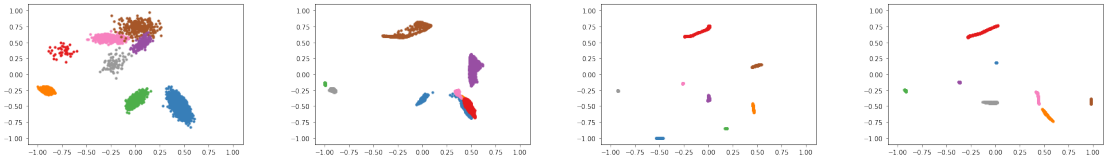
(a) 8C-Imban-Sep (b) samples from CG (c) samples from CG with a2a regularization (d) samples from CG with o2o regularization



(e) 8C-Imban-Sep (f) samples from subnetworks of CG (g) samples from subnetworks of CG with a2a regularization (h) samples from subnetworks of CG with o2o regularization

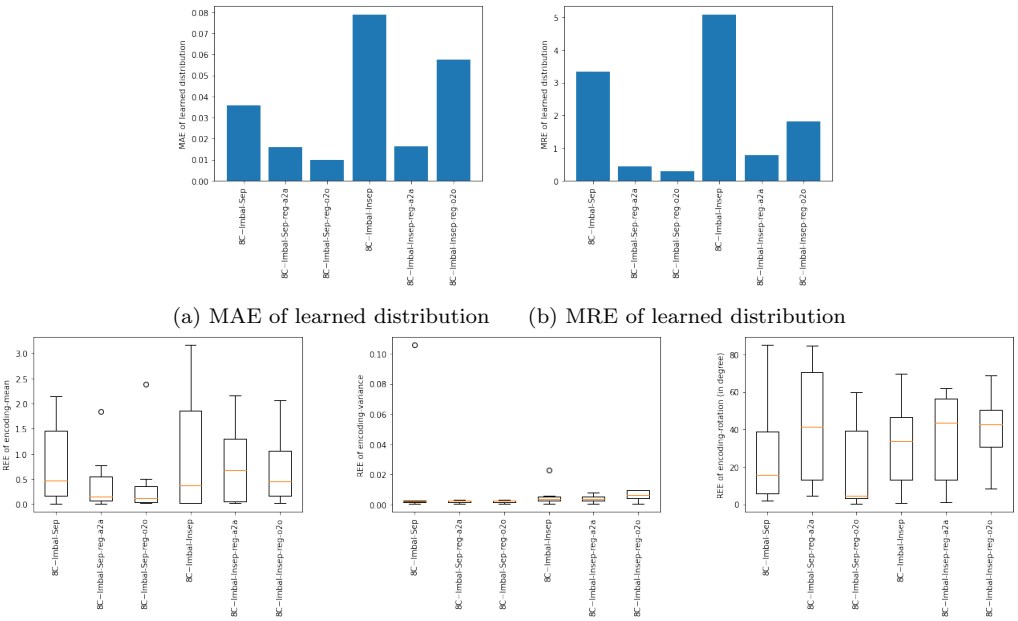


(i) 8C-Imban-Insep (j) samples from CG (k) samples from CG with a2a regularization (l) samples from CG with o2o regularization



(m) 8C-Imban-Insep (n) samples from subnetworks of CG (o) samples from subnetworks of CG with a2a regularization (p) samples from subnetworks of CG with o2o regularization

Figure 3.16: Samples generated by CG with/without regularization. (b)-(d),(f)-(h): samples generated for 8C-Imban-Sep. (j)-(l),(n)-(p): samples generated for 8C-Imban-Insep. Each of the aforementioned sub-figure contains 4096 generated samples. Each model is trained for 500 epochs.



(a) MAE of learned distribution (b) MRE of learned distribution

(c) errors of encoding mean (d) errors of encoding variance (e) errors of encoding rotation

Figure 3.17: Quantitative measures of the performance of CG with and without regularization

ization and the results are plotted in Figure 3.17. All metrics are the same as those used in cases without regularization. First, in the distribution errors shown in Figure 3.17a, we can see a significant improvement in performance brought by both regularization methods. The reduction of distribution can be observed more clearly in terms of MRE presented in Figure 3.17b. The main driving force is that the subnetworks of CG better focus on learning those more frequent clusters. Second, from the encoding mean errors presented in Figure 3.17c, we can also observe a dramatic reduction in the overall error bars. It means that each subnetwork better focuses on a specific cluster. More specifically, in the learning of the 8C-Imban-Sep dataset, only the least frequent cluster (red) is not learned by any subnetwork, and thereby the encoding error for this cluster remains high. For the learning of the 8C-Imban-Insep dataset, we also see the decrease of the whole error bars, which implies that subnetworks of CG are now targeting better at specific clusters. Finally, for errors measured by encoding variance and rotation, the improvement is not obvious, as shown in Figures 3.17d and 3.17e.

3.5 Experiments of Choice Cell on Real World Data

In this section, we demonstrate the effectiveness of CC with experiments on real world data. Experiments gradually increase in complexity. In the first subsection, we show that CC can effectively learn distribution of real world data, even when data is extremely imbalanced. In the second subsection, we combine CC with generators to form Choice Generator (CG) and demonstrate that the subnetworks of CG can learn the encodings of input data. In the third subsection, we show that CG can learn distribution and encodings of real world data simultaneously.

3.5.1 Experiment: *Learning Distribution of Real World Data Using CC with Pre-trained Subnetworks (CC:LD:RD)*

The goal of the experiment *CC:LD:RD* is to show that CC with pre-trained generators (CC_G) can learn the distribution of object classes in real world data. We first train generators independently, each of which is responsible for learning to generate objects of one class in the input data. Then these generators are placed on the leaf nodes of CC. Using these pre-trained generators, CC tries to learn the true probability distribution of classes of input data and encode it in its internal nodes. In addition, for high dimensional data, it would be beneficial to apply SNM on a hidden layer which carries more semantic meaning instead of raw input data. We call this training method Semantic-SNM (S-SNM). We show that S-SNM can further improve the performance of CC_G.

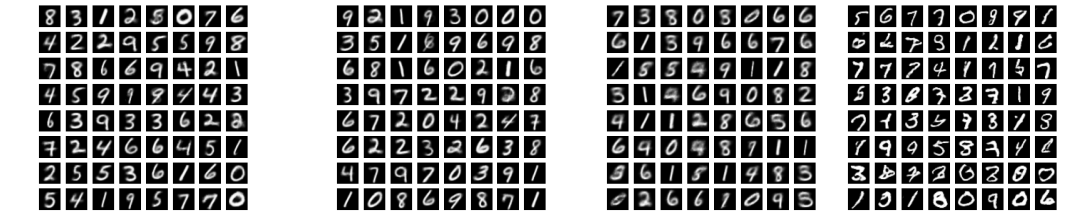
Dataset We train CC with pre-trained generators to learn distribution of classes on the MNIST dataset and its imbalanced version [30]. The size of training data for the balanced dataset is 60K. To make the dataset imbalanced, the number of samples in each class is determined by Equation 3.15. n is set to 60K, and p is set to 1/2. The size of imbalanced training data is 6149. For pre-training generators, we partition

data according to their classes and feed only one class of data into one generator.

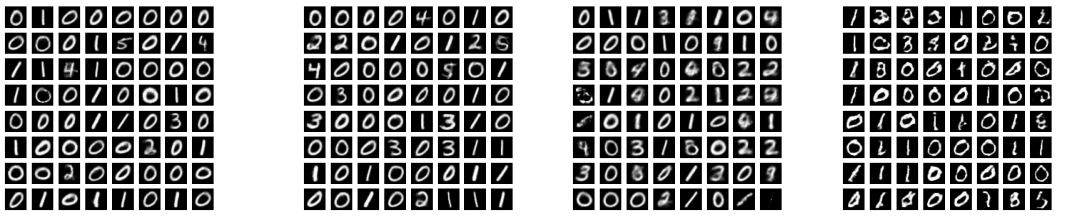
Networks The pre-trained generators are placed on leaf nodes of CC to form CC_G. We compare the performance CC_G with two other popular generative models, Variational Auto-encoder (VAE) [44] and GAN [12]. The generator archetype of CC_G follows VAE. The model archetype of GAN follows DCGAN [13]. Detailed information about network architectures of VAE can be found in Figure A.3 in the Appendix. In all models, we use Cross Entropy as loss functions. Like previous experiments, SNM is applied to generated samples and real samples to form matched samples, and then the loss function is calculated based on matched pairs.

The S-SNM training is similar to SNM, except the dissimilarity of samples is measured based on their hidden representations $h(\mathbf{g})$ and $h(\mathbf{x})$, i.e. $\|h(\mathbf{g})$ - $h(\mathbf{x})\|$, instead of raw data. After matching samples based on their semantic distance, the loss is applied on original data space. To get semantic representation of samples, we first pre-train a classifier on the training data. Then, the generated samples and the true data samples are fed to the classifier, and the representations in the second last layer are used as their semantic representations, $h(\mathbf{g})$ and $h(\mathbf{x})$, respectively. For the classifier, all layers except the last two layers in the classifier use $rectifier$ as activation functions. Its second last layer uses *sigmoid* function as the activation function, and its last layer uses a $softmax$ activation function. Detailed information about the structure of the classifier can be found in Figure A.1 in the Appendix.

Hyper-parameters We use Adam optimizer with learning rate equal to 0.001 throughout the experiment *CC:LD:RD*. In addition, z is uniformly distributed over the half-open unit interval $[0, 1)$. We set *scale* to 40 for training and 80 for reporting. Batch size is set to 512. We pre-train each subnetwork of CC_G for 1000 epochs and then train CC for 100 epochs. For VAE and GAN, we also train them for 1000 epochs for a fair comparison.



(a) CC_G with SNM on balanced MNIST (b) CC_G with S-SNM on balanced MNIST (c) VAE on balanced MNIST (d) GAN on balanced MNIST



(e) CC_G with SNM on imbalanced MNIST (f) CC_G with S-SNM on imbalanced MNIST (g) VAE on Unbalance MNIST (h) GAN on imbalanced MNIST

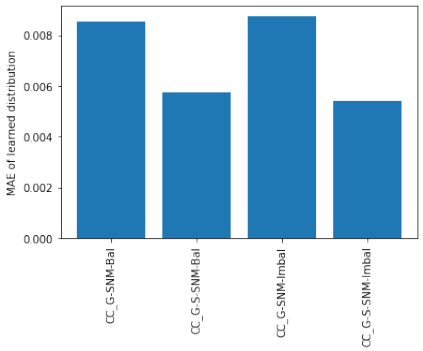
Figure 3.18: Comparing image qualities of CC_G with SNM, CC_G with S-SNM, VAE, and GAN on MNIST. (a)-(d) samples generated for the balanced MNIST. (e)-(h) samples generated for the imbalanced MNIST.

Result We start with quality of sample images produced by different network architectures for the MNIST dataset. Images are presented in Figure 3.18. First, We look at samples generated for the original MNIST dataset. As we can see from Figure 3.18a and Figure 3.18b, both CC_G with SNM and CC_G with S-SNM can produce high quality images. Compared with images generated by the original VAE shown in Figure 3.18c, digits generated by CC_G are even sharper. Compared with digits generated by GAN shown in 3.18d, digits generated by CC_G appear to be less sharper. However, most of the digits generated by CC_G are meaningful, while GAN is more likely to generate images that do not look like digits. Move on to the case for the imbalanced MNIST dataset. As we can see from Figure 3.18e and Figure 3.18f, both CC_G with SNM and CC_G with S-SNM can still produce high quality images even in the extremely imbalanced case. As shown in Figure 3.18g, the original VAE can also produce good quality images for the imbalanced MNIST, but its images are blurrier and it is more likely to produce images that are not meaningful. As we can see from Figure 3.18h, GAN can generate high quality and even sharper images for high frequent digits. However, for the low frequent digits, their quality is not appealing.

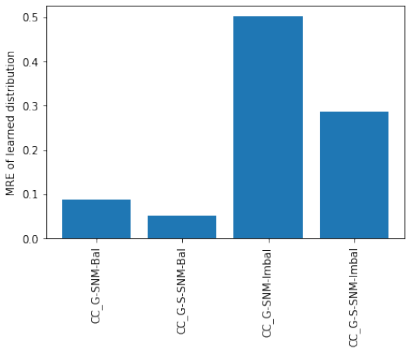
Next we turn to the quantitative analysis of the performance of distribution learning. We still use MAE and MRE defined in Equations 3.10 and 3.16 as the measurements of distribution errors. Since VAE and GAN have no explicit component that allows us to access the distribution of digits directly, we use the pre-trained classifier to estimate the distribution of digits generated by four models. Each model generates 512 sample images, and these sample images are classified by the classifier to produce an estimate of distribution of digits. Then MAE and MRE of distributions are estimated based on predictions produced by the classifier.

The results of distribution errors are shown in Figure 3.19. As shown in Figure 3.19a, in both balanced and imbalanced MNIST datasets, CC_G are able to learn the underlying distribution of digit classes successfully. The MAE of distributions for both balanced and imbalanced cases are about 0.85% for CC_G with SNM. With better matching brought by S-SNM, CC_G arrives at the MAE around only 0.55%. The benefits of S-SNM can be seen more clearly from the improvement of MRE for the imbalanced MNIST, as shown in Figure 3.19b. The MRE measure amplifies errors from low frequent digits. With S-SNM, the MRE of distribution is reduced from around 50% to below 30%. It demonstrates that semantic matching helps CC_G learn distributions in less frequent branches better.

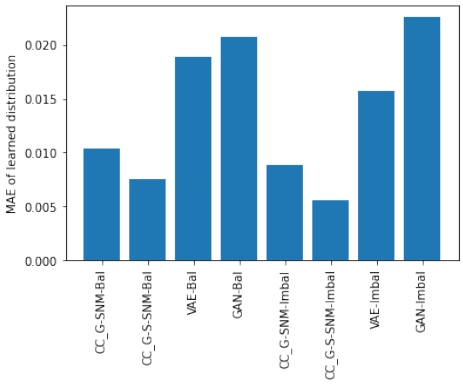
Next, we compare performance of CC_G to VAE and GAN. The results are shown in Figures 3.19c and 3.19d. As we can see from these figures, both CC_G with SNM and CC_G with S-SNM perform significantly better than VAE and GAN in both error measures, and CC_G with S-SNM has the best performance among all. In terms of MAE measure, its error rate is only about one-third of that of VAE and one-fourth of that of GAN. In terms of MRE measure, its error rate is also about one-third of that of VAE and one-third of that of GAN for the balanced MNIST. But for the imbalanced MNIST, its error rate is also about one-seventh of that of VAE and one-ninth of that of GAN. This result shows the advantage of CC_G in learning distribution



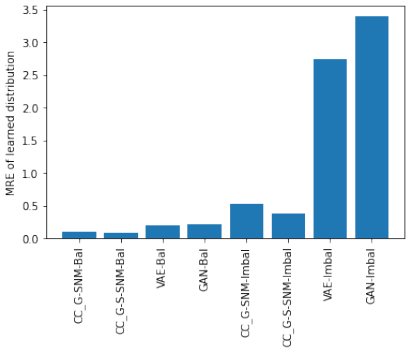
(a) MAE of distribution calculated from internal nodes of CC_G



(b) MRE of distribution calculated from internal nodes of CC_G



(c) MAE of distribution estimated by generated samples



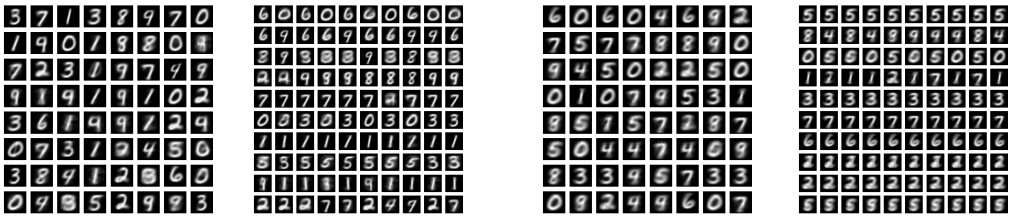
(d) MRE of distribution estimated by generated samples

Figure 3.19: Comparing distribution errors of CC_G with SNM, CC_G with S-SNM, VAE, and GAN on MNIST

for imbalanced datasets.

3.5.2 Experiment: *Learning Encodings of Real Data Using Choice Generator(CG:LE:RD)*

In the experiment *CG:LE:RD*, we combine CC with generators to form Choice Generator (CG) where generators are placed on leaf nodes of CC as its subnetworks. The goal of the experiment is to show that CG can learn encodings of real world data. We fix the internal nodes of CG and only train its subnetworks to learn encodings. After training, we would like each subnetwork to encode a specific class or a small set of classes of input data.



(a) samples from CG-SNM (b) samples from subnetworks of CG-SNM (c) samples from CG-S-SNM (d) samples from subnetworks of CG-S-SNM

Figure 3.20: Encodings learned by CG. In the second and fourth columns, each row represents samples generated by the same subnetwork of a CG. Each CG is trained for 100 epochs.

Dataset We train CG to learn encodings of MNIST [30], where the size of training data is 60K.

Network architecture We use decoders from VAE as generators and place them on leaf nodes of CC. The architecture of decoders can be found in Figure A.3 in the Appendix. We train CG with SNM (CG-SNM) and CG with S-SNM (CG-S-SNM) to compare results of encoding learning.

Hyper-parameters We use Adam optimizer with learning rate equal to 0.001 throughout the experiment *CG:LE:RD*. In addition, z is uniformly distributed over the half-open unit interval $[0, 1)$. We set *scale* to 40 for training and 200 when reporting results. Batch size is set to 128. We train each network for 100 epochs.

Result We first look at quality of sample MNIST images produced by CG. We present sample images generated by CG in Figure 3.20. As we can see from this figure, CG is able to learn meaningful encodings of input images. In addition, we also show MNIST encodings learned by individual subnetwork in Figure 3.20. Each row of subfigures in the second and last columns of Figure 3.20 represents samples generated from the same subnetwork of a CG. We can see that each subnetwork of CG learns encodings of a specific class or a small set of classes of input data. We also observe that encodings learned by CG-S-SNM is blurrier than CG-SNM, but its

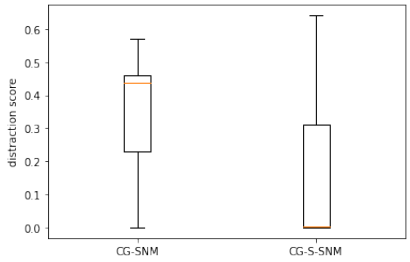


Figure 3.21: Distraction scores of subnetworks of CG

subnetwork concentrates more on a smaller set of classes objects.

Next, we turn to a quantitative measure of encodings learned by CG. We would like each subnetwork of CG to learn a small set of classes of objects. To better assess this performance, we define Distraction Score (DS). DS of a subnetwork i measures the percentage of non-dominant objects generated by the subnetwork, i.e.,

$$DS(i) = 1.0 - max(\hat{P}_i), \tag{3.17}$$

where i is the index of a subnetwork of CG, and $\hat{P}_i$ is its distribution over object classes, which is estimated by samples that it generates. The result is shown in Figure 3.21. As we can see from this figure, the median DS for CG-SNM is below 0.5, which means more than half of its subnetworks gear towards learning one dominant class of objects. CG-S-SNM performs even better. More than half of its subnetworks focus on learning one specific class of objects. We argue that this is another advantage of CG. Its subnetworks can focus on learning more homogeneous representations, which in turn makes subnetworks more transparent.

3.5.3 Experiment: *Learning Encodings and Distribution of Real Data Using Choice Generator (CG:LED:RD)*

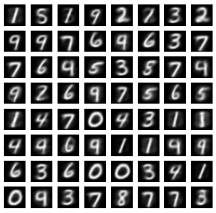
The experiment *CG:LED:RD* is to show that CG can learn real world data such that its internal nodes can learn the distribution of object classes, and its subnetwork generators can generate samples that are similar to input data. Our goal of this

experiments is two-fold. First, we aim to train CG to learn to generate input data. Second, we would also like each subnetwork to learn to generate a specific class or a small set of classes of objects.

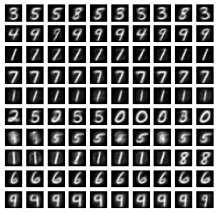
Dataset We apply CG to learn encodings and distributions on MNIST, as well as its imbalanced version. The size of training data for the balanced dataset is 60K. To make the datasets imbalanced, the number of samples in each class is determined by Equation 3.15. n is set to 60K, and p is set to $1/2$. The size of imbalanced training data is 6149.

Network architecture We use decoders from VAE as generators and place them on leaf nodes of CC. The architecture of decoders can be found in Figure A.3 in the Appendix. We train CG with SNM and S-SNM training methods to compare results. In addition, we train CG with a2a regularization and o2o regularization, both of which are introduced in the experiment *CG:LDE:SD*. Thus, in total we have four CG models: CG-SNM, CG-S-SNM, CG-a2a, and CG-o2o. Like previous experiments, we also compare performance of CG with two benchmark models: VAE and GAN.

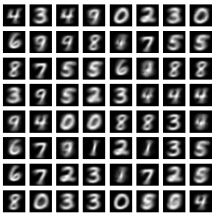
Hyper-parameters We use Adam optimizer with learning rate equal to 0.001 throughout the experiment. In addition, z is uniformly distributed over the half-open unit interval $[0, 1)$. We set *scale* to 40 for training and 200 when reporting results. Batch size is set to 128 for training balanced datasets. For imbalanced datasets, it is set to 512 to allow less frequent classes of images to show up during the training. For regularization, $\lambda_{between}$ is set to 1.0, and λ_{within} is set to 0.2. We also apply regularizations to the training of CG every five epochs. For balanced datasets, we train each network for 100 epochs. For imbalanced datasets, each network is trained for 200 epochs.



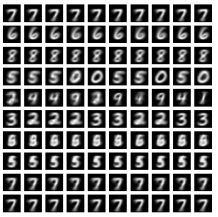
(a) samples from CG-SNM



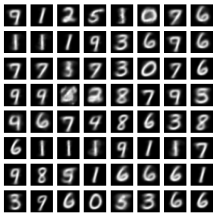
(b) samples from subnetworks of CG-SNM



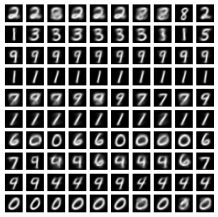
(c) samples from CG-S-SNM



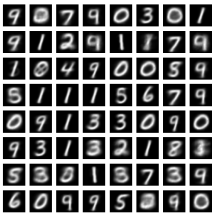
(d) samples from subnetworks of CG-S-SNM



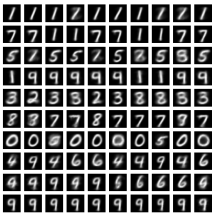
(e) samples from CG-a2a



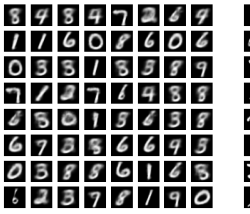
(f) samples from subnetworks of CG-a2a



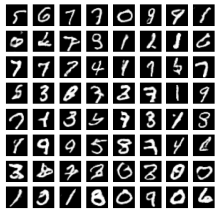
(g) samples from CG-o2o



(h) samples from subnetworks of CG-o2o

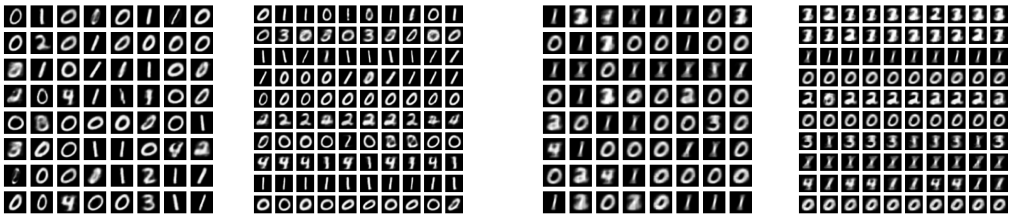


(i) samples from VAE

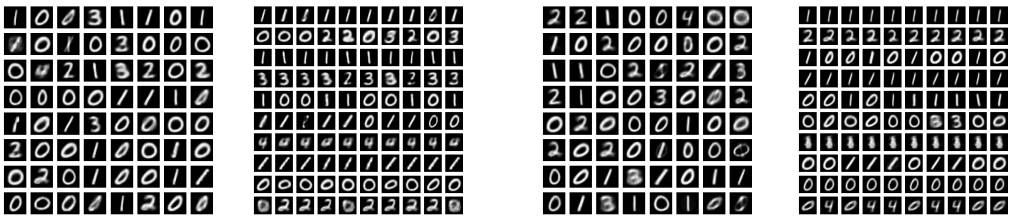


(j) samples from GAN

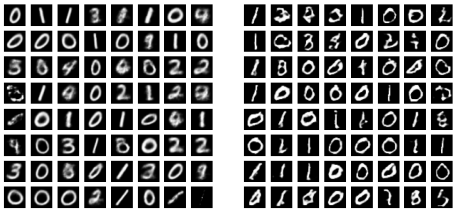
Figure 3.22: Comparing CG v.s. VAE, GAN on balanced MNIST. Each network is trained for 100 epochs. In (b),(d),(f),(h), each row represents samples generated by the same subnetwork of a CG.



(a) samples from CG-SNM (b) samples from subnetworks of CG-SNM (c) samples from CG-S-SNM (d) samples from subnetworks of CG-S-SNM

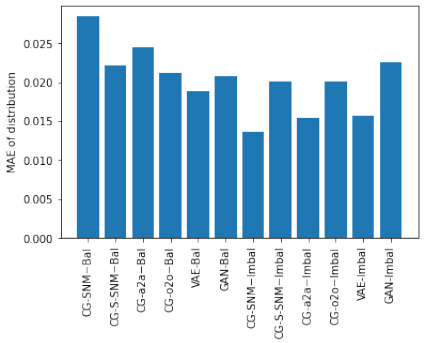


(e) samples from CG-a2a (f) samples from subnetworks of CG-a2a (g) samples from CG-o2o (h) samples from subnetworks of CG-o2o

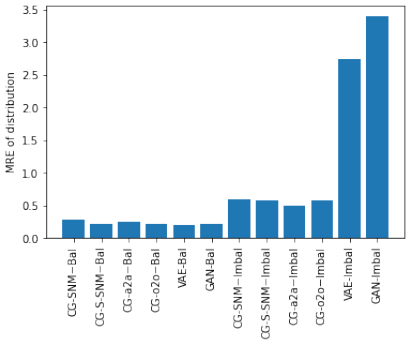


(i) samples from VAE (j) samples from GAN

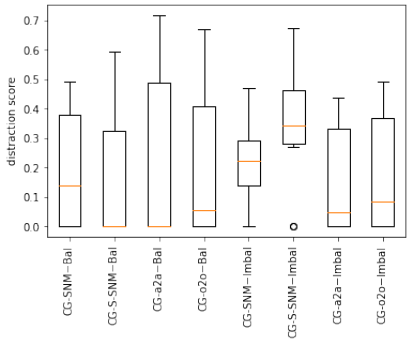
Figure 3.23: Comparing CG v.s. VAE, GAN on imbalanced MNIST. Each network is trained for 200 epochs. In (b),(d),(f),(h), each row represents samples generated by the same subnetwork of a CG.



(a) comparing MAE of distributions of CG v.s. VAE and GAN



(b) comparing MRE of distributions of CG v.s. VAE and GAN



(c) distraction scores of subnetworks of CG

Figure 3.24: Quantitative measures of performance of CG on MNIST. Results are obtained by training models for 100 and 200 epochs for balanced and imbalanced datasets, respectively.

Result We first look at the quality of sample MNIST images produced by CG. We present sample images generated by CG, VAE, and GAN in Figure 3.22 and Figure 3.23. We can observe a few things about the effectiveness of CG from these figures. First, compared with VAE and GAN, CG is able to generate digits with comparable quality. Second, as shown in Figure 3.23, diversity of digits learned by CG is comparable with VAE and better than GAN. Lastly, we also show MNIST samples generated from individual subnetwork in Figure 3.22 and Figure 3.23. Each row in the second and fourth columns contains samples generated from a single subnetwork. We can see that some subnetworks learn homogeneous representations. We argue that this is an advantage of CG. CG helps its subgenerators learn more homogeneous representations, making the subnetworks more transparent.

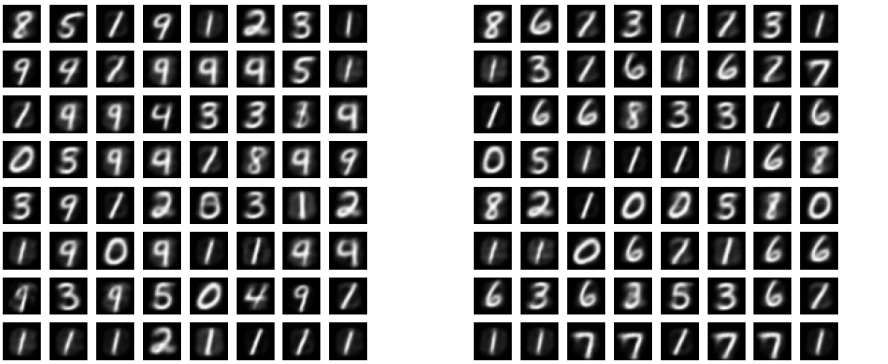
We now turn to the quantitative analysis of the performance of CG. As in previous sections, we use MAE and MRE defined in Equation 3.10 and Equation 3.16 as

error measurements. Following the method adopted in the experiment *CC:LD:RD*, we use a pre-trained classifier to estimate the distribution of object classes generated by four CG models and two benchmark models. Each model generates 512 sample images, and these sample images are classified by the classifier to estimate the distribution of object classes. MAE and MRE are then calculated based on the estimated distribution.

The quantitative results are shown in Figures 3.24a and 3.24b. As we can see from these two figures, CG can achieve comparable performance with VAE and GAN in the balanced dataset. For the imbalanced dataset, CG outperforms VAE and GAN in terms of MRE, as shown in Figure 3.24b. The high MRE errors for VAE and GAN mean that they do not learn distribution of less frequent classes well. CG, however, is able to achieve lower MRE errors. It is mainly because through its choice mechanism, its subgenerators learn more homogeneous representations, which helps CC detect less frequent classes of objects. Finally, we use DS defined in Equation 3.17 to evaluate subnetworks of CG, and we present results in Figure 3.24c. The results obtained so far show that regularizations help more subnetworks focus on a specific class of objects, but the overall benefits that they bring to CG have not been as impressive as what they have achieved in low dimensional space.

3.5.4 Experiment: Showing Flexibility of Choice Generator

In this experiment, we show that CG has a greater flexibility and finer control over subnetworks through controlling its ***bias*** parameters. More specifically, CG can control what kinds of objects it wants to generate by manipulating its internal ***bias***. For example, in Figure 3.25, we show that after training a CG, we are able to direct CG to generate samples excluding "6" and "7" (shown in Figure 3.25a) and samples excluding "4" and "9" (shown in Figure 3.25b).



(a) samples generated by CG without "6" and "7"

(b) samples generated by CG without "4" and "9"

Figure 3.25: Flexibility of CG in controlling its subnetworks

3.6 Conclusion

We introduce and formulate an interpretable neural architecture called CC. CC's internal representations can be reduced to an explainable interpretation of probability distribution. We also show that CC can effectively learn class distribution in both synthetic and real world data. Furthermore, CC can effectively learn class distribution even in the case of extremely imbalanced data. Last, we build CG by placing generators on the leaf nodes of CC as its subnetworks. Our experimental results show that CG is not only a more interpretable generator but also maintains a comparable performance with popular generators like VAE and GAN. In addition, it has an advantage of making its subnetworks learn more homogeneous representations, making them more transparent. We further demonstrate the flexibility of CG by controlling it to produce certain classes of objects through manipulating its internal ***bias***.

We also found that subnetworks and internal nodes of CG have too much freedom in learning encodings and distribution of input data. This freedom causes some within generators to learn heterogeneous representations and some between generators subnetworks to learn homogeneous representations. Thus, as a future study, we would like to explore the use of regularization to improve the performance of CG and have each of its subnetwork focus on learning a specific class of objects.

CHAPTER IV

CONDITIONAL CHOICE CELL ARCHITECTURE

4.1 Introduction

In Chapter III, we formulate Choice Cell (CC), and the results show that CC can effectively learn encodings and distribution of input data. In this chapter, we use CC as building blocks to develop a conditional network. We coin this newly built architecture Conditional Choice Cell (CCC). CCC not only inherits the interpretability of CC but also models order, relation, and dependency among events. The main contributions of this chapter are:

- We extend CC to build a new neural architecture CCC, which is a conditional model with an advantage of being more interpretable and transparent.
- We combine CCC with the Stable Neighbor Matching (SNM) training to show its effectiveness in learning conditional distribution and encodings of input data.

The chapter is organized as follows. In Section 4.2, we provide a literature review most relevant to CCC. The review covers information about RNN, Bidirectional RNN, Bidirectional LSTM, Conditional GAN, and Auxiliary Classifier GAN. In Section 4.3, we provide a formal introduction to CCC. In Section 4.4, we show the effectiveness of CCC in learning synthetic datasets. An experiment showing that CCC can effectively learn the conditional distribution of input data is presented in Subsection 4.4.1. With the goal of showing the effectiveness of CCC in learning both distribution and encodings, we conduct another experiment in Subsection 4.4.2. Finally, we summarize

main contributions of this chapter and suggest future work in Section 4.5.

4.2 Literature Review

RNN, Bidirectional RNN, and Bidirectional LSTM Recurrent Neural Network (RNN), a class of neural networks, is able to model data consisting of sequences of elements that are not independent [22]. Given an input sequence $(\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_T)$, an RNN computes current state h_t based on its current input and its past information h_{t-1}. It then produces current output y_t based on h_t. More specifically,

$$\boldsymbol{h}_t = \sigma(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1}) \tag{4.1}$$

$$\boldsymbol{y}_t = \boldsymbol{W}_{hy}\boldsymbol{h}_t, \tag{4.2}$$

where bias terms are omitted for brevity.

One notable issue of RNN is the vanishing gradient. To solve this issue, Hochreiter and Schmidhuber proposed Long Short-Term Memory (LSTM), which introduces memory cells to overcome difficulties encountered by the conventional RNN [40]. More details about LSTM can be found in Section 3.2. Another shortcoming of the conventional RNN is that it only exploits past information [45]. To help address this issue, Bidirectional RNN (BRNN) extends RNN to take into consideration the dependency on future information as well [46]. A BRNN extends Equation 4.1 to compute a forward hidden sequence $\overrightarrow{\boldsymbol{h}}$ and a backward hidden sequence $\overleftarrow{\boldsymbol{h}}$. Then it calculates the output sequence $\boldsymbol{y}$ based on $\overrightarrow{\boldsymbol{h}}$ and $\overleftarrow{\boldsymbol{h}}$. More specifically,

$$\begin{aligned}
\overrightarrow{\boldsymbol{h}_t} &= \sigma(\boldsymbol{W}_{x\overrightarrow{h}}\boldsymbol{x}_t + \boldsymbol{W}_{\overrightarrow{h}\overrightarrow{h}}\overrightarrow{\boldsymbol{h}}_{t-1}) \\
\overleftarrow{\boldsymbol{h}_t} &= \sigma(\boldsymbol{W}_{x\overleftarrow{h}}\boldsymbol{x}_t + \boldsymbol{W}_{\overleftarrow{h}\overleftarrow{h}}\overleftarrow{\boldsymbol{h}}_{t-1}) \\
\boldsymbol{y}_t &= \boldsymbol{W}_{\overrightarrow{h}y}\overrightarrow{\boldsymbol{h}_t} + \boldsymbol{W}_{\overleftarrow{h}y}\overleftarrow{\boldsymbol{h}_t},
\end{aligned}$$

Combining BRNN and LSTM yields Bidirectional LSTM, which solves the vanishing gradient issue and allows modeling long-range dependency in both input directions [45].

4.2.1 Attention Model

Another research area to CCC is Attention Model (AM) [42]. AM can be viewed as an extension of LSTM. It extends the formulation of LSTM to the following equations.

$$
\begin{aligned}
\boldsymbol{g}_t &= tanh(\boldsymbol{U}_{gh}\,\boldsymbol{h}_{t-1} + \boldsymbol{W}_{gx}\,\boldsymbol{x}_t + \boldsymbol{W}_{gz}\,\boldsymbol{z}_t) \\
\boldsymbol{i}_t &= \sigma(\boldsymbol{U}_{ih}\,\boldsymbol{h}_{t-1} + \boldsymbol{W}_{ix}\,\boldsymbol{x}_t + \boldsymbol{W}_{iz}\,\boldsymbol{z}_t) \\
\boldsymbol{f}_t &= \sigma(\boldsymbol{U}_{fh}\,\boldsymbol{h}_{t-1} + \boldsymbol{W}_{fx}\,\boldsymbol{x}_t + \boldsymbol{W}_{fz}\,\boldsymbol{z}_t) \\
\boldsymbol{o}_t &= \sigma(\boldsymbol{U}_{oh}\,\boldsymbol{h}_{t-1} + \boldsymbol{W}_{ox}\,\boldsymbol{x}_t + \boldsymbol{W}_{oz}\,\boldsymbol{z}_t) \\
\boldsymbol{s}_t &= \boldsymbol{g}_t \odot \boldsymbol{i}_t + \boldsymbol{s}_{t-1} \odot \boldsymbol{f}_t \\
\boldsymbol{h}_t &= tanh(\boldsymbol{s}_t) \odot \boldsymbol{o}_t\,,
\end{aligned}
$$

where $\boldsymbol{z}_t$ is a context vector. Intuitively, $\boldsymbol{z}_t$ dynamically represents the relevant parts of images with respect to output $\boldsymbol{h}_t$ at time t. It can be computed from the annotation vectors $\boldsymbol{a}_i$, $i = 1, 2, ..., L$, which corresponds to the features extracted from raw input, and a positive score α_i representing relative importance of feature $\boldsymbol{a}_i$ for predicting next output $\boldsymbol{h}_t$. The set of annotation vectors $\{\boldsymbol{a}_1\,, \boldsymbol{a}_2\,, ..., \boldsymbol{a}_L\}$ are extracted from input data using lower layers of CNN. The relevance score at time t, α_{ti}, is computed as softmax of outputs, e_{ti} for $i = 1, ..., L$, of an AM, f_{att}. The output, e_{ti}, is defined as

$$
e_{ti} = f_{att}(\boldsymbol{a}_i\,, \boldsymbol{h}_{t-1}),
$$

where f_{att} is implemented as a multilayer perceptron.

Then, the context vector $\boldsymbol{z}_t$ is computed as blending the set of relevance scores at time t and the set of annotation vectors together, i.e.,

$$\boldsymbol{z}_t = f_{ble}(\{\alpha_{t1}, \alpha_{t2}, ..., \alpha_{tL}\}, \{\boldsymbol{a}_1, \boldsymbol{a}_2, ..., \boldsymbol{a}_L\}).$$

A typical choice of blending function, f_{ble}, is to take $\boldsymbol{z}_t$ to be the weighted average of its inputs [43], i.e. $\boldsymbol{z}_t = \sum_{i=1}^{L}(\alpha_{ti}\boldsymbol{a}_i)$.

Conditional GAN and Auxiliary Classifier GAN Another line of research that is related to CCC is Conditional GAN, which is a variant of GAN. GAN represents a promising avenue for unsupervised learning, because it sidesteps the difficulty of approximating many intractable probabilistic computations [12, 47]. More information about GAN can be found in Section 2.2. As an unconditional generative model, GAN has no control on modes of the data being generated. To incorporate this capability into GAN, [47] proposes Conditional GAN to direct the data generation process by conditioning the GAN model on additional information [47]. Conditional GAN achieves this goal by feeding additional information $\boldsymbol{y}$ into the generator and the discriminator. On the generator side, extra information $\boldsymbol{y}$ is concatenated with the original latent vector $\boldsymbol{z}$ to form a joint hidden representation. This joint representation is then fed to the generator as its input. On the discriminator side, extra information $\boldsymbol{y}$ is concatenated with real sample $\boldsymbol{x}$ and they are presented as input to the discriminator. Thus, the minimax formulation in Equation 2.1 is modified to

$$\max_{\theta_G}\min_{\theta_D}\mathbb{E}_{\boldsymbol{x}\sim P_{\boldsymbol{x}}}[-logD_{\theta_D}(\boldsymbol{x}|\boldsymbol{y})] + \mathbb{E}_{\boldsymbol{z}\sim P_{\boldsymbol{z}}}[-log(1 - D_{\theta_D}(G_{\theta_G}(\boldsymbol{z}|\boldsymbol{y}))]$$

Auxiliary Classifier GAN (AC-GAN) represents an alternative way of utilizing side information in GAN [48]. In AC-GAN, the generator uses both class information and the latent variable to generate samples. The discriminator tries to predict both

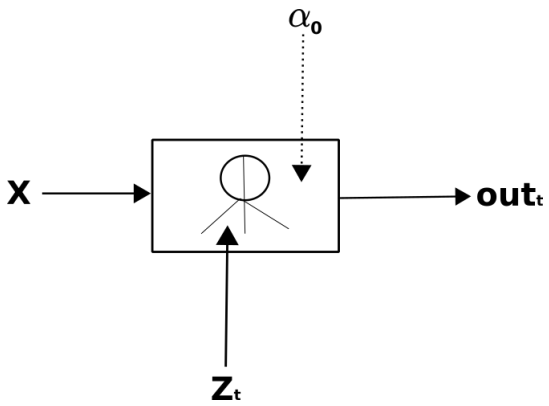


Figure 4.1: Abstract View of a Conditional Choice Cell

the probability distribution of sample sources and the probability distribution over sample classes. The objective of the discriminator is to maximize the log-likelihood of the correct source and the log-likelihood of the correct class. The objective of the generator is to maximize the log-likelihood of the correct class while minimizing the log-likelihood of the correct source.

4.3 The Conditional Choice Cell Architecture

As introduced in Section 3.3, a CC functions as a selection gate which outputs a noisy version of one of its input tensors placed on its leaf nodes. CCC uses a Fixed Choice Cell (FCC) to combine multiple CCs together to form a conditional network. A FCC behaves like CC, producing a noisy version of one of its input tensors placed on its leaf nodes, but its internal α values are pre-determined. Formally, FCC is defined as the following.

Definition IV.1 (Fixed Choice Cell (FCC)) *Let $\boldsymbol{x}$ be a set of input tensors and cc($\boldsymbol{x}$, $\boldsymbol{z}$; **bias**, and **scale**) be a CC. A FCC is a CC, except its $\boldsymbol{\alpha}$ is given instead of being determined by $\boldsymbol{z}$, **bias**, and **scale**, i.e.,*

$$\boldsymbol{y} = cc(\boldsymbol{x}, \boldsymbol{\alpha}) \tag{4.3}$$

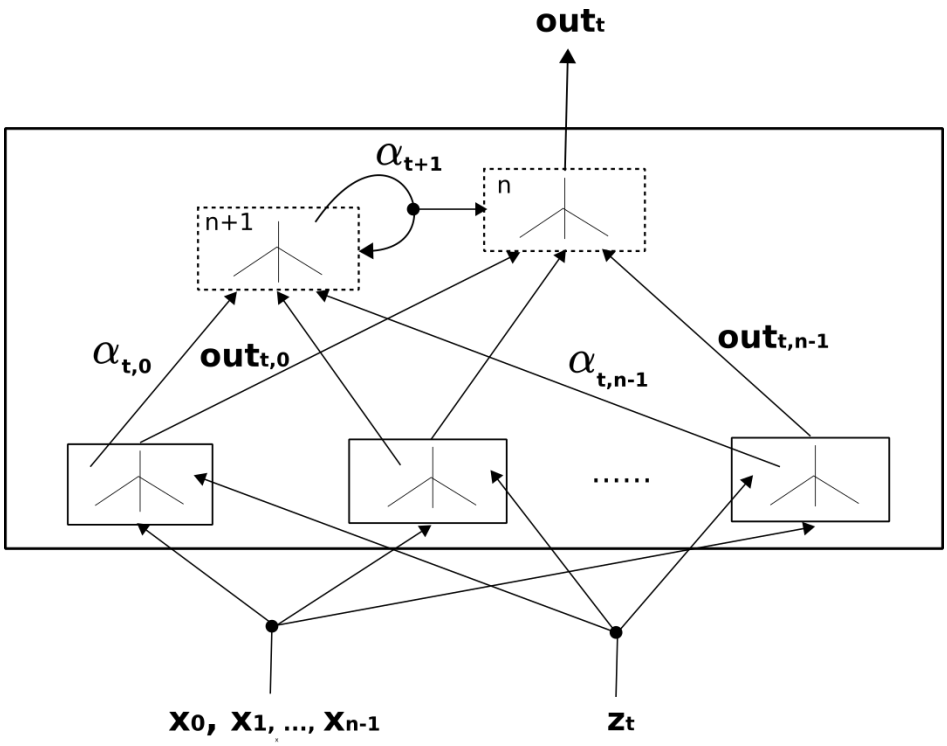


Figure 4.2: Detailed View of a Conditional Choice Cell

To distinguish it from an ordinary CC, denote FCC as $cc_f(\boldsymbol{x}, \boldsymbol{\alpha})$.

Using FCC, CCC combines multiple CCs together. Figures 4.1 and 4.2 show graphical representations of CCC. The top two nodes in Figure 4.2 are FCCs. Based on pre-determined $\boldsymbol{\alpha}$, the top-right node outputs a noisy version of one of outputs produced by the leaf nodes, and the top-left node outputs a noisy version of one of the optional outputs generated by the leaf nodes. The leaf nodes in Figure 4.2 are CCs that share a set of input tensors. Each of the leaf node models a separate conditional distribution $P(\boldsymbol{out}_t \,|\boldsymbol{out}_{t-1})$, where $\boldsymbol{out}_{t-1}$ is a noisy version of one of input tensor in the shared input set. Formally, CCC is defined as the following.

Definition IV.2 *[Conditional Choice Cell (CCC)] Let* $\boldsymbol{x} = \{\boldsymbol{x}_0, \boldsymbol{x}_1, ..., \boldsymbol{x}_{n-1}\}$ *be a set of input tensors. Let* $\{cc_0(\boldsymbol{x}, \boldsymbol{z}; \boldsymbol{bias}_0, \boldsymbol{scale}_0), cc_1(\boldsymbol{x}, \boldsymbol{z}; \boldsymbol{bias}_1, \boldsymbol{scale}_1), ...,$ $cc_{n-1}(\boldsymbol{x}, \boldsymbol{z}; \boldsymbol{bias}_{n-1}, \boldsymbol{scale}_{n-1})\}$ *be a set of CCs. A CCC is a n-nary tree with* $n+2$ *nodes, where a* cc_i *is placed on a leaf node i for i = 0, ..., n-1. Let* $\boldsymbol{out}_{t,i}$ *and* $\boldsymbol{\alpha}_{t,i}$ *be outputs produced by the leaf node i at step t. Two internal nodes* $n, n+1$ *of CCC*

contain FCCs, and their outputs at step t are

$$\boldsymbol{out}_t = cc_f(\{\boldsymbol{out}_{t,0}, ..., \boldsymbol{out}_{t,n-1}\}; \boldsymbol{\alpha}_t), \text{ for node } n \tag{4.4}$$

$$\boldsymbol{\alpha}_{t+1} = cc_f(\{\boldsymbol{\alpha}_{t,0}, ..., \boldsymbol{\alpha}_{t,n-1}\}; \boldsymbol{\alpha}_t), \text{ for node } n+1, \tag{4.5}$$

where $\boldsymbol{\alpha}_0$ *will be given at the initial step t=0, and* $\boldsymbol{out}_t$ *is the final output of the CCC at step t.*

We design CCC to be a conditional model. The overall desired behavior of CCC is to output a fuzzy version of one of input tensors of its leaf node according to conditional probability $P(\boldsymbol{out}_t | \boldsymbol{out}_{t-1})$, where $\boldsymbol{out}_{t-1}, \boldsymbol{out}_t \in \{\boldsymbol{x}_0, \boldsymbol{x}_1, ..., \boldsymbol{x}_{n-1}\}$. At each step t, a leaf node i produces one of the input tensors, $\boldsymbol{out}_{t,i}$, based on its internal probability distribution, and the FCC at node n then selects an output from outputs of its leaf nodes based on its previous selection $\boldsymbol{out}_{t-1}$. The FCC at node $n+1$ updates its $\boldsymbol{\alpha}_t$ to $\boldsymbol{\alpha}_{t+1}$ to prepare for the next selection.

4.4 Experiments of Conditional Choice Cell on Synthetic Data

In this section, we show the performance of CCC on synthetic datasets. Experiments in this section are divided into three subsections. In the first subsection, we show that CCC can learn the distribution of input data. In the second subsection, we demonstrate that CCC can learn the encodings of input data. In the last subsection, we present the effectiveness of CCC in learning the distribution and the encodings of input data simultaneously.

4.4.1 Experiment: *Learning Conditional Distribution of Synthetic Data Using Conditional Choice Cell (CCC:LD:SD)*

The goal of the experiment *CCC:LD:SD* is to show that CCC can learn conditional distribution of input data. We would like each leaf node of CCC to learn a sepa-

rate conditional distribution $P(\boldsymbol{out}_1 \,|\boldsymbol{out}_0 \,= \boldsymbol{x}_i\,)$. We assume that CCCs know the encodings of input datasets prior to the training, and we only train CCCs to learn conditional distribution of input data.

Dataset In this experiment, we use four synthetic datasets with increasing complexity: 2E-1hot-Con, 2E-Gen-Con, 4E-1hot-Con, and 4E-Gen-Con. To have the datasets exhibit conditional dependency, each dataset is formed by following the process below. First, a long sequence $\boldsymbol{S}$ is generated by drawing encodings from the set of true encodings, i.e., $\boldsymbol{S}_i \in \boldsymbol{x}$, based on the initial distribution and the conditional distribution. Then, a sliding window w slides through the sequence $\boldsymbol{S}$ to form a training dataset $\mathbb{S}$ with n samples, each of which is of length $|w|$. That is, the i_{th} sample in $\mathbb{S}$ is $\boldsymbol{S}_{i...i+|w|-1}$. Additional information about the datasets is briefly described below.

- 2E-1hot-Con: $\boldsymbol{x} \subset \mathbb{R}^2$ and $|\boldsymbol{x}| = 2$. Each encoding in $\boldsymbol{x}$ is one-hot encoded.
- 2E-Gen-Con: $\boldsymbol{x} \subset \mathbb{R}^2$ and $|\boldsymbol{x}| = 2$.
- 4E-1hot-Con: $\boldsymbol{x} \subset \mathbb{R}^4$ and $|\boldsymbol{x}| = 4$. Each encoding in $\boldsymbol{x}$ is one-hot encoded.
- 4E-Gen-Con: $\boldsymbol{x} \subset \mathbb{R}^4$ and $|\boldsymbol{x}| = 4$.

Each dataset contains 6400 samples, and $|w|$ is set to 2. More details about these datasets can be found in Table A.2 of the Appendix.

Network Architecture The basic architecture of CCC follows the formulation presented in Figures 4.1 and 4.2. To facilitate the learning of conditional distribution, we partition datasets and conduct training as follows. Let $\boldsymbol{s} \in \mathbb{S}$ be a training sample. Let $\boldsymbol{s}_0$ and $\boldsymbol{s}_1$ be the first and second encodings of $\boldsymbol{s}$. Let $\mathbb{X}$ be a batch of real samples. At each batch iteration, we first partition real samples according to their conditions $\boldsymbol{s}_0$. That is, $\mathbb{X}_{\boldsymbol{x}_i} = \{\boldsymbol{s}_1 \,|\boldsymbol{s}_0\, \boldsymbol{s}_1 \in \mathbb{X} \wedge \boldsymbol{s}_0 = \boldsymbol{x}_i\}$. For each partition $\mathbb{X}_{\boldsymbol{x}_i}$, we specify $\boldsymbol{\alpha}_0$ corresponding to $\boldsymbol{x}_i$. Then, using the pre-determined $\boldsymbol{\alpha}_0$, CCC generates a set

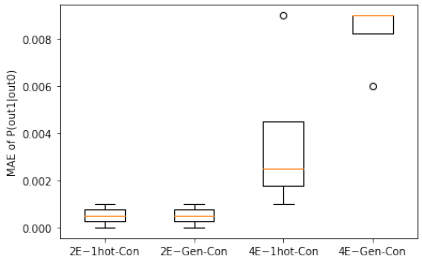


Figure 4.3: MAEs of $P(\boldsymbol{out}_1 | \boldsymbol{out}_0)$ learned by CCC in the experiment *CCC:LD:SD*. Each network is trained for 100 epochs.

of samples, $\mathbb{G}_{\boldsymbol{x}_i}$, according to Equation 4.4 with $t = 1$. After that, for each condition $\boldsymbol{x}_i \in \boldsymbol{x}$, we apply SNM to $\mathbb{G}_{\boldsymbol{x}_i}$ and $\mathbb{X}_{\boldsymbol{x}_i}$ to produce a stable match with respect to that condition. A loss, $L_{\boldsymbol{x}_i}$, is then calculated based on the stable match for each condition according to Equation 2.6, and MAE is used in the place of the *loss* function. The final loss is the average of $L_{\boldsymbol{x}_i}$ for $\boldsymbol{x}_i \in \boldsymbol{x}$.

Hyper-parameters We use Adam optimizer with learning rate equal to 0.001. In addition, z is uniformly distributed over the half-open unit interval $[0, 1)$ for all leaf nodes of CCC. We set *scale* to 40. Batch size is set to 512. We train each model on each dataset for 100 epochs.

Result To assess the quality of conditional distribution learned by each branch of CCC, we use MAE defined in Equation 3.10. Since each CCC has multiple branches of CC, we use a box plot to visualize the results in Figure 4.3. Each box in the plot represents the spread of MAE of conditional distributions learned by leaf nodes of CCC for a specific dataset. The orange bar in the middle of each box represents the median of MAE of conditional distributions learned by leaf node CCs. As we can see from Figure 4.3, CCC successfully learns conditional distribution of all datasets. The medians of MAEs of conditional distributions for 2E-1hot-Con and 2E-Gen-Con datasets are below 0.1%. For 4E-1hot-Con and 4E-Gen-Con datasets, they are only around 0.2% and 0.9%, respectively. The MAE of conditional distribution in the worst case is also below 1% error rate.

4.4.2 Experiment: *Learning Conditional Distribution and Encodings of Synthetic Data using Conditional Choice Cell (CCC:LDE:SD)*

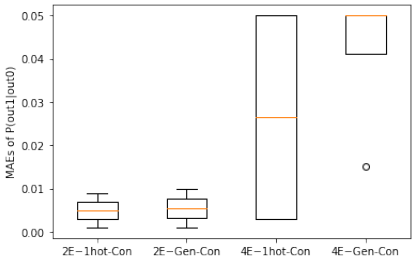
The goal of the experiment *CCC:LDE:SD* is to show that CCC can learn conditional distribution and encodings of input data simultaneously. We assume that CCCs know neither distributions nor encodings of input datasets prior to the training. After training, we expect that each CC at a leaf node should learn a separate conditional distribution $P(\boldsymbol{out}_1 | \boldsymbol{out}_0 = \boldsymbol{x}_i)$, without knowing true encodings in advance.

Dataset The datasets used in this experiment are the same as those used in the experiment *CCC:LD:SD*.

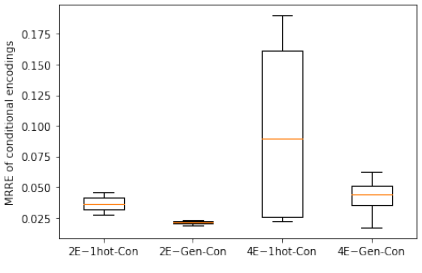
Network Architecture The basic architecture of CCC follows the formulation presented in Figures 4.1 and 4.2, except that for each CC on the leaf node of CCC, no input tensor is fed to it. Instead, each CC learns encodings using its hidden variables placed on its leaf nodes. Like the experiment *CCC:LD:SD*, we partition a batch of real samples into $\mathbb{X}_{\boldsymbol{x}_i}$ for $\boldsymbol{x}_i \in \boldsymbol{x}$. For each partition CCC uses $\boldsymbol{\alpha}_0$ specified $\boldsymbol{x}_i$ to generate samples $\mathbb{G}_{\boldsymbol{x}_i}$. SNM is then applied to $\mathbb{G}_{\boldsymbol{x}_i}$ and $\mathbb{X}_{\boldsymbol{x}_i}$ to produce stable matches under each condition $\boldsymbol{x}_i$. After that, a loss, $L_{\boldsymbol{x}_i}$ is calculated for each stable match according to Equation 2.6, and *loss* in the equation uses MAE. The final loss is then calculated based on average of $L_{\boldsymbol{x}_i}$.

Hyper-parameters We use Adam optimizer with learning rate equal to 0.001. In addition, z is uniformly distributed over half-open unit interval $[0, 1)$ for all branches of CCC. We set *scale* to 40. Batch size is set to 512. We train each model on each dataset for 400 epochs.

Result Like the experiment *CCC:LD:SD*, we use MAE to assess conditional distributions learned by CCC. Since CCC does not know true encodings, we apply SNM to



(a) MAEs of $P(\boldsymbol{out}_1 | \boldsymbol{out}_0)$



(b) MREEs of encodings learned by leaf nodes of CCC

Figure 4.4: Quantitative error measures for $P(\boldsymbol{out}_1 | \boldsymbol{out}_0)$ and encodings learned by leaf nodes of CCC in the experiment CCC:LDE:SD. Each network is trained for 400 epochs.

match encodings learned by a leaf node with true encodings following the same condition, $\boldsymbol{x}_i$, before calculating errors of conditional distribution. To assess the quality of encodings learned by each leaf node of CCC, we define Mean Relative Encoding Error (MREE) on top of Equation 3.11. MREE is defined as

$$MREE(\hat{\boldsymbol{x}}_{\boldsymbol{x}_i}, \boldsymbol{x}) = \frac{1}{|\boldsymbol{x}|} \sum REE(\boldsymbol{a}, \boldsymbol{b}), \tag{4.6}$$

where $\hat{\boldsymbol{x}}_{\boldsymbol{x}_i}$ are encodings learned by the leaf node i of CCC, $\boldsymbol{x}$ is the set of true encodings, and ($\boldsymbol{a}$, $\boldsymbol{b}$) is a pair in the stable match outputted by applying SNM to $\hat{\boldsymbol{x}}_{\boldsymbol{x}_i}$ and $\boldsymbol{x}$.

Results are shown in Figure 4.4. As shown in this figure, CCC can effectively learn the conditional distribution and encodings at the same time. For 2E-1hot-Con and 2E-Gen-Con datasets, both distribution errors learned by CCC are below 1%. The encoding errors are below 5% for the 2E-1hot-Con dataset and 2.5% for the 2E-Gen-Con dataset, respectively. For 4E-1hot-Con and 4E-Gen-Con datasets, CCC can achieve the distribution errors below 5%.

4.5 Conclusion

In this chapter, we extend CC to CCC, which is a conditional neural architecture that has an advantage of being more interpretable and explainable. In addition,

we demonstrate the effectiveness of CCC in learning conditional distribution and encodings of synthetic data. We acknowledge this study has a few limitations. First, due to its exploratory nature, this study only focuses on applying CCC to synthetic data. It does not apply CCC to real world data to compare its performance with other frequently used conditional models. Second, the number of neural units in current CCC grows exponentially with the number of configurations in the input space. Despite those limitations, the results of this study indicate the potential of CCC as a conditional neural model with rare and valuable merit of interpretability and transparency. In future work, we would like to apply CCC to real world data to demonstrate its broader applicability. We would also suggest that future work be conducted to investigate methods that would collapse leaf nodes of CCC based on conditional distribution.

CHAPTER V

CONCLUSION

5.1 Main Contributions of the Dissertation

A list of main contributions of the dissertation are summarized below.

- In Chapter 2, we propose the Stable Neighbor Match (SNM) training to approximate the Wasserstein distance in the context of generative modeling. Like Generative Adversarial Network (GAN), the proposed SNM training does not require expensive calculations of joint distributions in high dimensional space.

- To investigate the stability of generators trained with SNM, we conduct four experiments to compare its performance with other related generative models. The experimental results indicate that the SNM training not only avoids expensive computation of high dimensional distribution, but also exhibits valuable training stability.

- In Chapter 3, we propose an interpretable neural architecture called Choice Cell (CC). Like gated units in Long Short-Term Memory, CC controls and manipulates information flowing through it. It has an additional advantage of being able to reduce its hidden representations to more explainable interpretation of probability distribution.

- In Chapter 3, we combine other generators with CC to build the Choice Generator (CG). Results from both synthetic and real world data demonstrate that CG

is not only more explainable and transparent than standard neural networks, but also maintains comparable performance with other models.

- We also conduct experiments that use the SNM training with semantic representation of input data, and the results indicate further improvement of performance in learning input data.

- In Chapter 3, we apply all-to-all and one-to-other regularization during the training of CG. The experimental results indicate that using regularization helps subgenerators in CG learn more homogeneous representations, further improving the interpretability of these subgenerators.

- In Chapter 4, we present an extension of CC called Conditional Choice Cell (CCC). CCC is a conditional model with an advantage of being more interpretable and transparent.

5.2 Future Research Directions

Some of the future research directions are listed below.

- It is suggested that future studies be conducted to improve the running time of SNM so that the SNM training could be more applicable to higher dimensional space. Faster SNM could also afford matching between two batches with larger sizes, which would result in a more accurate approximation for optimal Wasserstein distance.

- Another research direction is to explore the combination of the SNM training and the adversarial training. This might lead to a new discovery of more efficient SNM training and more robust adversarial training at the same time.

- We also suggest that future work investigate more regularization techniques to help each subnetwork of CG focus on a specific class of object. Then both

subnetworks of CG and CG as a whole might become more transparent.

- Experiments that apply CCC to real world datasets could be conducted to further study its strengths and weaknesses.

- It would also be beneficial to investigate the possibility of collapsing subbranches of CCC so that the number of leaf nodes do not grow exponentially with the number of conditions.

REFERENCES

[1] "The size of the world wide web (the internet)," https://www.worldwidewebsize.com, accessed: 2020-09-29.

[2] "Facebook's top open data problems," https://research.fb.com/blog/2014/10/facebook-s-top-open-data-problems/, accessed: 2020-09-29.

[3] "Hours of video uploaded to youtube every minute as of may 2019," https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/, accessed: 2020-09-29.

[4] "Twitter usage statistics," https://www.internetlivestats.com/twitter-statistics/#sources, accessed: 2020-09-29.

[5] T. M. Mitchell, *Machine Learning*, 1997.

[6] K. P. Murphy, *Machine learning : a probabilistic perspective*. MIT Press, 2013.

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, p. 436–444, 2015.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. Cambridge, MA, USA: The MIT Press, 2016.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, 2012.

[10] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, 2012.

[11] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems 27*, 2014.

[12] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc., 2014, pp. 2672–2680.

[13] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *CoRR*, vol. abs/1511.06434, 2016.

[14] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, "Photo-realistic single image super-resolution using a generative adversarial network," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[15] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[16] I. Deshpande, Z. Zhang, and A. G. Schwing, "Generative modeling using the sliced wasserstein distance," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[17] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein generative adversarial networks," ser. Proceedings of Machine Learning Research, vol. 70. International Convention Centre, Sydney, Australia: PMLR, 06–11 Aug 2017, pp. 214–223.

[18] Q. Zhang and S. Zhu, "Visual interpretability for deep learning: a survey," *CoRR*, vol. abs/1802.00614, 2018.

[19] Q. Zhang, Y. N. Wu, and S. Zhu, "Interpretable convolutional neural networks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 8827–8836.

[20] T. Wu, X. Li, X. Song, W. Sun, L. Dong, and B. Li, "Interpretable R-CNN," *CoRR*, vol. abs/1711.05226, 2017.

[21] D. Gunning, "Darpa's explainable artificial intelligence (xai) program," in *Proceedings of the 24th International Conference on Intelligent User Interfaces*. New York, NY, USA: Association for Computing Machinery, 2019.

[22] Z. C. Lipton, "A critical review of recurrent neural networks for sequence learning," *CoRR*, vol. abs/1506.00019, 2015.

[23] I. Goodfellow, "NIPS 2016 tutorial: Generative adversarial networks," *CoRR*, vol. abs/1701.00160, 2017.

[24] X. Mao, Q. Li, H. Xie, R. Y. Lau, Z. Wang, and S. Paul Smolley, "Least squares generative adversarial networks," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[25] C. Tao, L. Chen, R. Henao, J. Feng, and L. C. Duke, "Chi-square generative adversarial network," ser. Proceedings of Machine Learning Research, vol. 80. Stockholmsmässan, Stockholm Sweden: PMLR, 10–15 Jul 2018, pp. 4887–4896.

[26] L. Metz, B. Poole, D. Pfau, and J. Sohl-Dickstein, "Unrolled generative adversarial networks," *CoRR*, vol. abs/1611.02163, 2016.

[27] K. Li and J. Malik, "On the implicit assumptions of gans," *ArXiv*, 2018.

[28] M. Arjovsky and L. Bottou, "Towards principled methods for training generative adversarial networks," *ArXiv*, 2017.

[29] S. Arora, R. Ge, Y. Liang, T. Ma, and Y. Zhang, "Generalization and equilibrium in generative adversarial nets (gans) (invited talk)," 2018.

[30] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[31] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *CoRR*, vol. abs/1708.07747, 2017.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, cite arxiv:1412.6980.

[33] T. Tieleman and G. Hinton, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude," COURSERA: Neural Networks for Machine Learning, 2012.

[34] T. Adel, Z. Ghahramani, and A. Weller, "Discovering interpretable representations for both deep generative and discriminative models," ser. Proceedings of Machine Learning Research, Stockholmsmässan, Stockholm Sweden, 2018, pp. 50–59.

[35] A. Nguyen, J. Yosinski, and J. Clune, "Understanding neural networks via feature visualization: A survey," *CoRR*, vol. abs/1904.08939, 2019.

[36] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *In Computer Vision–ECCV 2014*. Springer, 2014, pp. 818–833.

[37] A. Mahendran and A. Vedaldi, "Understanding deep image representations by inverting them," *CoRR*, vol. abs/1412.0035, 2014.

[38] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps." *CoRR*, vol. abs/1312.6034, 2013.

[39] A. Dosovitskiy and T. Brox, "Generating images with perceptual similarity metrics based on deep networks," *CoRR*, vol. abs/1602.02644, 2016.

[40] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–1780, 1997.

[41] F. A. Gers, J. A. Schmidhuber, and F. A. Cummins, "Learning to forget: Continual prediction with lstm," *Neural Comput.*, vol. 12, no. 10, p. 2451–2471, 2000.

[42] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, "Show, attend and tell: Neural image caption generation with visual attention," ser. Proceedings of Machine Learning Research, vol. 37. Lille, France: PMLR, 2015, pp. 2048–2057.

[43] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014.

[44] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *CoRR*, vol. abs/1312.6114, 2014.

[45] A. Graves, N. Jaitly, and A. rahman Mohamed, "Hybrid speech recognition with deep bidirectional lstm," 2013.

[46] M. Schuster, K. K. Paliwal, and A. General, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, 1997.

[47] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *CoRR*, 2014.

[48] A. Odena, C. Olah, and J. Shlens, "Conditional image synthesis with auxiliary classifier gans," *ArXiv*, vol. abs/1610.09585, 2017.

[49] D. G. McVitie and L. B. Wilson, "The stable marriage problem," *Commun. ACM*, vol. 14, no. 7, p. 486–490, 1971.

[50] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

APPENDICES

settings	2E-1hot	2E-Gen	4E-1hot	4E-Gen
true encoding	[1.0, 0.0], [0.0, 1.0]	[0.8, 0.1], [0.3, 0.7]	[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]	[0.8, 0.1, 0.2, 0.4], [0.3, 0.7, 0.3, 0.3], [0.5, 0.1, 0.9, 0.4], [0.3, 0.6, 0.3, 1.0]
true distribution	[0.8, 0.2]	[0.8, 0.2]	[0.1, 0.5, 0.1, 0.3]	[0.1, 0.5, 0.1, 0.3]
number of samples	6400	6400	6400	6400

Table A.1: Datasets used in the experiment *CC:LD:SD*

settings	2E-1hot-Con	2E-Gen-Con	4E-1hot-Con	4E-Gen-Con
encodings	[1.0, 0.0], [0.0, 1.0]	[0.8, 0.1], [0.3, 0.7]	[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]	[0.8, 0.1, 0.2, 0.4], [0.3, 0.7, 0.3, 0.3], [0.5, 0.1, 0.9, 0.4], [0.3, 0.6, 0.3, 1.0]
$P(\boldsymbol{out}_0)$	[0.8, 0.2]	[0.8, 0.2]	[0.1, 0.5, 0.1, 0.3]	[0.1, 0.5, 0.1, 0.3]
$P(\boldsymbol{out}_t \mid \boldsymbol{out}_{t-1})$	0: [0.9, 0.1], 1: [0.1, 0.9]	0: [0.9, 0.1], 1: [0.1, 0.9]	0: [0.7, 0.1, 0.1, 0.1], 1: [0.1, 0.7, 0.1, 0.1], 2: [0.1, 0.1, 0.7, 0.1], 3: [0.1, 0.1, 0.1, 0.7]	0: [0.7, 0.1, 0.1, 0.1], 1: [0.1, 0.7, 0.1, 0.1], 2: [0.1, 0.1, 0.7, 0.1], 3: [0.1, 0.1, 0.1, 0.7]
n	6400	6400	6400	6400

Table A.2: Datasets used in the experiment *CCC:LD:SD*

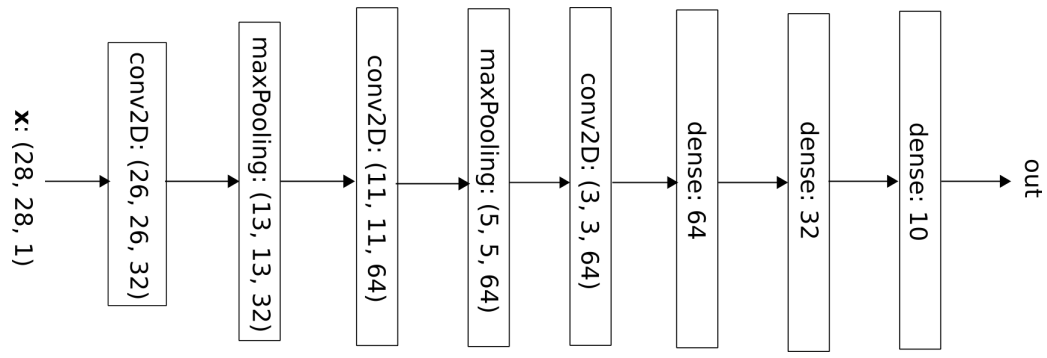
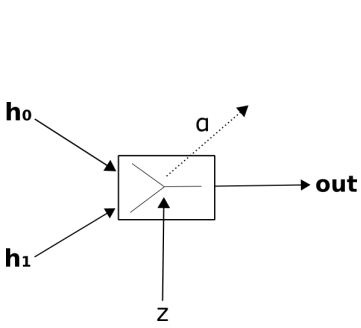
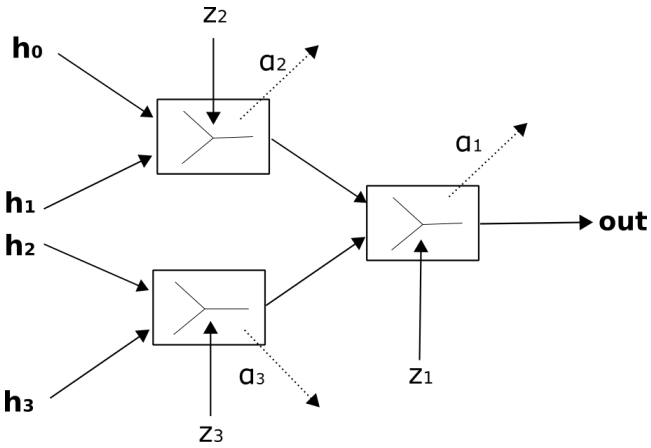


Figure A.1: Architecture of pre-trained classifier on MNIST



(a) BCC in experiment of learning distribution and encodings of synthetic data



(b) 4-nary CC in experiment of learning distribution and encodings of synthetic data

Figure A.2: Network architectures in the experiment *CC:LDE:SD*

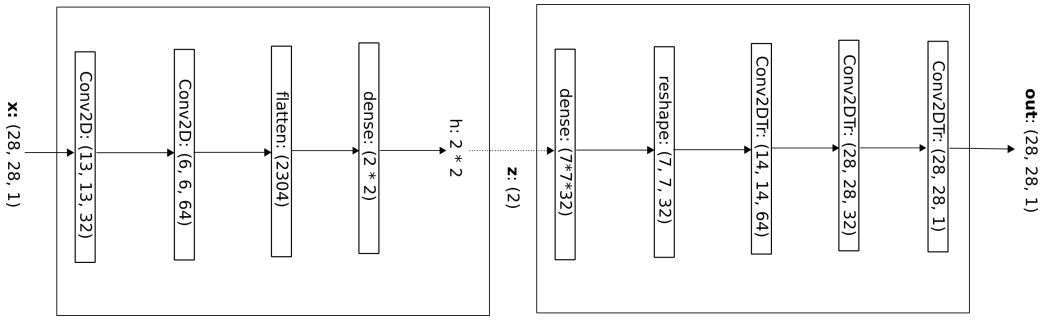


Figure A.3: Encoder-decoder structure of the VAE for training CC with pre-trained VAEs on MNIST

VITA

Zhuxi Yang

Candidate for the Degree of

Doctor of Philosophy

Dissertation: CHOICE CELL ARCHITECTURE AND STABLE NEIGHBOR MATCH TRAINING TO INCREASE INTERPRETABILITY AND STABILITY OF DEEP GENERATIVE MODELING

Major Field: Computer Science

Biographical:

Education:

Completed the requirements for the Doctor of Philosophy in Computer Science at Oklahoma State University, Stillwater, Oklahoma in December, 2020.

Completed the requirements for the Master of Science in International Economics and Finance at Valparaiso University, Valparaiso, Indiana in 2012.

Completed the requirements for the Bachelor of Engineering in Civil Engineering at Central South University, Changsha, China in 2009.

Experience:

Data Scientist, Microsoft Corporation, Redmond, WA

Sr. Statistical Analyst, Walmart, Inc., Bentonville, AR

Data Analyst Intern, Walmart, Inc., Bentonville, AR

Software Engineering Intern, Ansys, Inc., Canonsburg, PA

Graduate Teaching Assistant, Oklahoma State University, Stillwater, OK