Generating Graphs with Deep Learning and Graph Theory

By

Yuliang Ji
Doctor of Philosophy

Mathematics

Generating Graphs with Deep Learning and Graph Theory

By

Yuliang Ji
B.S., University of Science and Technology of China, 2017

Advisor: Yuanzhe Xi, Ph.D.

An abstract of
A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Mathematics
2022

Abstract

Generating Graphs with Deep Learning and Graph Theory
By Yuliang Ji

Deep generative models attract lots of attention in recent years. With deep neural networks and specific designs, deep generative models can generate high-quality realistic data. In this thesis, I focus on combining the deep generative models with the traditional graph theory algorithms to reduce the dependence on the volume of the training data and also improve the quality of the generated graphs. In particular, I first propose a deep learning method to improve the Havel-Hakimi graph realization algorithm in order to generate doppelganger graphs from a single graph. Second, I present a few new architectures of normalizing flow models with improved performance and theoretical guarantees. Finally, I develop a permutation invariant method via leveraging graph theory and denoising diffusion models for generating molecular graphs.

Generating Graphs with Deep Learning and Graph Theory

By

Yuliang Ji
B.S., University of Science and Technology of China, 2017

Advisor: Yuanzhe Xi, Ph.D.

Acknowledgments

First and most importantly, I want to thank my advisor, Prof. Yuanzhe Xi, for the continuous support, and patience during my PhD study. Prof. Xi encouraged me all the time of my academic research and daily life. I also want to thank Prof. Lars Ruthotto and Prof. Hao Huang for their support and serving as my committee members. I was inspired by their knowledge and their encouragements during these years. Furthermore, I would like to thank the chair of our math department, Prof. James Nagy for his support in the last few years.

I also would like to thank Dr. Jie Chen, Prof. Qiang Ye, Prof. Shanshuang Yang, Prof. Jie Ma, Dr. Difeng Cai and Dr. Mingbo Ma for their instructions. Thanks for their academic support during my research.

Finaly, I would like to thank all my collaborators and friends - Ru Huang, Chang Meng, Shifan Zhao, Jian Wu, Xingjian Li, Derek Onken, Zihao Li for memorable time spent together in these years.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 History of Deep Neural Networks

In 1943, Warren McCulloch and Walter Pitts [79] described how neurons might work, and they modeled a simple neural network. In 1975, Werbo [110] proposed the back propagation algorithm, which made the training multi-layer neural networks possible. However, due to the lack of computation resources and data, these methods were not popular before 2000.

In 2009, Stanford published ImageNet [17] dataset, which is a dataset that contains more than 14 million labeled images. Also, as the speed of GPUs increases significantly during these years, it became computational feasible to train large multi-layer neural networks by the back propagation algorithm to achieve good results. For example, AlexNet [68], won several international image classification competitions during 2011 and 2012. After that, multi-layer neural networks, which are also called deep neural networks, attract people from all over the world.

1.2 Basic Structure of Deep Neural Networks

The basic component in the deep neural networks is the fully connected layer.

Assume the input data of the fully connected layer is $x \in \mathbb{R}^{d_{in}}$, the output is $z \in \mathbb{R}^{d_{out}}$. Under this assumption, the fully connected layer contains a matrix $W \in \mathbb{R}^{d_{in} \times d_{out}}$, a vector $b \in \mathbb{R}^{d_{out}}$ and a non-linear function $\sigma : \mathbb{R} \to \mathbb{R}$. The relationship between z and x is given by the following formula:

$$z = \sigma(W^T x + b) \tag{1.1}$$

where the non-linear function σ is applied element-wise.

Equation (1.1) describes the operations of neural network performed at one single layer. Suppose the neural network has n fully connected layers, the output of the ith layer is vector $h^{(i)}$ for $i = 1, 2, \ldots, n-1$, and $W^{(i)}, b^{(i)}, \sigma^{(i)}$ are the components in the ith layer for $i = 1, 2, \ldots, n$. Then the output z of the neural network can be calculated by (1.2) from the given input x:

$$\begin{aligned} h^{(1)} &= \sigma^{(1)}(W^{(1)T} x + b^{(1)}) \\ h^{(i+1)} &= \sigma^{(i+1)}(W^{(i+1)T} h^{(i)} + b^{(i+1)}) \\ z &= \sigma^{(n)}(W^{(n)T} h^{(n-1)} + b^{(n)}) \end{aligned} \tag{1.2}$$

where $i = 1, 2, \ldots, n-2$. Here, vector $h^{(i)}$ are called "hidden units".

There are two other popular layers: convolution neural network layer [29] and recurrent neural network layer [96]. Usually, convolution neural network layer is used for image tasks [71], and recurrent neural network layer is used for sequential data, for example, text and audio.

Consider the simple convolution neural network layer defined with a matrix $K \in \mathbb{R}^{M \times N}$ and a number b. When the input x is a matrix of size $d_1 \times d_2$, the output z

of the layer will be a matrix whose (i, j)th element is computed by the convolution operation defined in (1.3) [34]:

$$z(i,j) = \sum_{m=1}^{M}\sum_{n=1}^{N} x(i+m, j+n)K(m,n) + b. \tag{1.3}$$

There are lots of work focusing on modifying the basic convolution neural network layer to improve the performance on image tasks during these years. For example, AlexNet [68] uses multi-channel convolution layer, ResNet [44] tries the residual blocks, and CoAtNet [14] combines convolution layers with some newest neural network layers to further improve the prediction accuracy.

The basic recurrent neural network layer contains a function f. Suppose the input data is $x = \{x^{(1)}, \ldots, x^{(T)}\}$, then the output $h^{(T)}$ will be computed recursively by Equation (1.4) [34]:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}) \tag{1.4}$$

with $h^{(0)} = 0$ and $t = 1, \ldots, T$.

Since 1990, researchers who focus on sequential data have designed different recurrent layers to improve the performance in certain speech applications. Hochreiter and Schmidhuber designed LSTM [46] in 1997, and a model [28] constructed by LSTM layer outperformed traditional models in specific speech tasks in 2007. Kyunghyun Cho et al. [13] introduced Gated Recurrent Units (GRU) layer in 2014, which has much fewer parameters than traditional LSTM.

1.3 Training Process of Deep Neural Networks

When researchers have already designed a deep neural network architecture, they need to use computers to calculate the value of parameters (e.g. each entry of the matrix W in fully connected layer) to make their models achieve desirable performance. Usually, this process is called the training process.

To evaluate the performance of the designed neural network model, researchers need to define specific functions of their model, which are called loss functions.

Mathematically, in supervised learning, if the given data is x and the given target is $y(x)$, assume the neural network model is a function f_θ, where θ are the parameters. Researchers need to define a loss function $L(x, y(x), f_\theta)$ to evaluate the model f_θ.

The common loss function is Mean Squared Error:

$$L(x, y(x), f_\theta) = \sum_x ||f_\theta(x) - y(x)||^2 \tag{1.5}$$

If the loss function has already been designed, researchers could calculate the value of each parameter in their models by optimization techniques. A traditional optimization technique is called Gradient Descent, which has the formula:

$$\theta_i = \theta_i - \alpha \frac{\partial L(x, y(x), f_\theta)}{\partial \theta_i} \tag{1.6}$$

where α is a number called learning rate.

More formally, Algorithm 1 shows the process of training deep neural networks.

Algorithm 1: Training process of Deep Neural Networks

```
  Input  : training data $x$
  Output: Model $f$ with parameters $\theta$
1 Design a neural network $f$.
2 Initialize the parameters $\theta$.
3 Define the loss function $L(x, f_\theta)$.
4 repeat
5     Compute $f(x)$.
6     Compute loss function $L(x, f_\theta)$.
7     Use optimization techniques and backpropagation to update $\theta$.
8 until loss is small enough or iteration number reaches max iteration;
```

In recent years, researchers have developed many stochastic optimization algorithms for the training process of neural networks, such as Stochastic Gradient Descent (SGD) [94], Adam [58], Adagrad [23]. They all estimate the gradient based on a batch of data instead of the whole dataset. These optimization techniques make the training process faster and more stable.

1.4 Deep Generative Model

In artificial intelligence, one of the most important fields is to find specific techniques to make computers have the ability to generate data based on the given data. Mathematically, researchers assume that the given data follow a particular distribution, and they wish to design models which could represent that distribution and generate new samples from the represented distribution. In recent years, researchers find that some deep neural network models perform well in this task, and people named those models as *Deep Generative Model* [85] [97].

There are several popular types of Deep Generative Model: Variational AutoEncoder [59], Generative Adversarial Network [33], Normalizing Flow [92] [20], and Denoising Diffusion Model [55] [102].

1.4.1 Variational AutoEncoder

Diederik P.Kingma and Max Welling introduced Variational AntoEncode (VAE) [59] in 2014. VAE can learn smooth latent state representations of the input data, and usually the dimension of the latent state is much smaller than the dimension of the input data.

VAE contains two parts: *Encoder* and *Decoder*. To get the output of the VAE model from input x, first, the Encoder maps the input x to two vectors μ_x, σ_x, which represent the mean and the variance, respectively. Then, a vector $z = \mu_x + \sigma_x \odot \epsilon$ is constructed, where $\odot$ denotes the element-wise product and ϵ samples from $\mathcal{N}(0, I)$. This step is called "reparameterization trick" [59]. Finally, the model passes z to the Decoder and gets the output $\hat{x}$. This process is illustrated in Figure 1.1.

The loss function of the VAE model is often defined based on Equation (1.7):

$$Recon(x, \hat{x}) + KL(\mathcal{N}(\mu_x, \sigma_x), \mathcal{N}(0, I)) \tag{1.7}$$

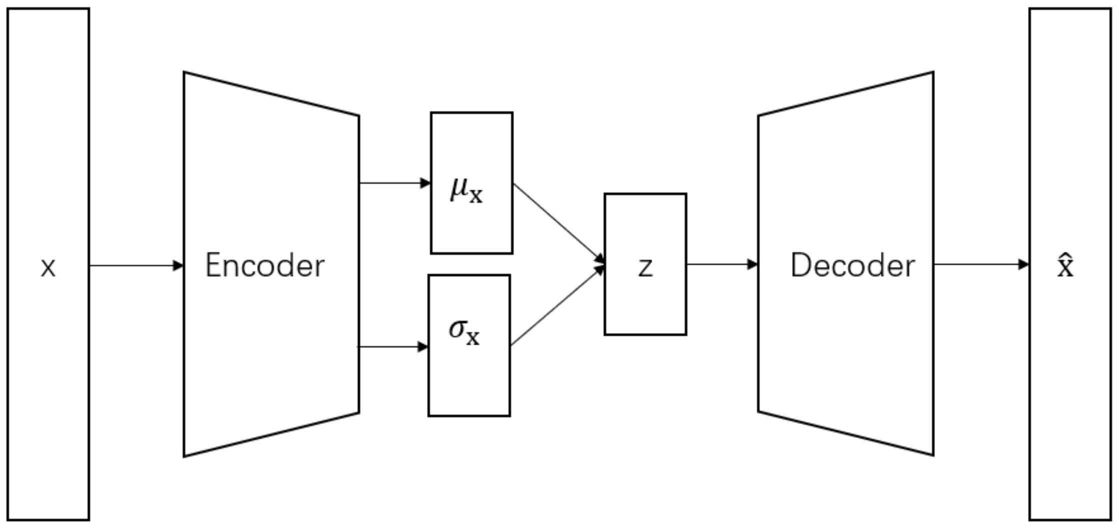


Figure 1.1: The architecture of VAE model where $\hat{x}$ is the output and x is the input.

where *Recon* is the loss of the reconstruction error, which is usually chosen as Mean Square Error or Cross-Entropy, and KL denotes the Kullback–Leibler divergence [69].

Many efforts have been made to develop variations of VAE to generate different types of data. For example, VQ-VAE-2 [91] tries to generate high-fidelity images and GraphVAE [100] uses VAE to generate graphs.

1.4.2 Generative Adversarial Network

Generative Adversarial Network was introduced by Goodfellow et al. [33] in 2014. In the Generative Adversarial Network model, there is a rivalry between two neural networks: *Discriminator* and *Generator*. The Generator network tries to generate data which comes from a similar distribution as the distribution of the input data, while the Discriminator network tries to distinguish the input data and the generated data.

Suppose the input data has a distribution $p_{data}(x)$, and G represents the generator neural network, D represents the discriminator neural network. Mathematically, the generator maps z, which comes from a simple distribution $p_Z(z)$, to $G(z)$, which has

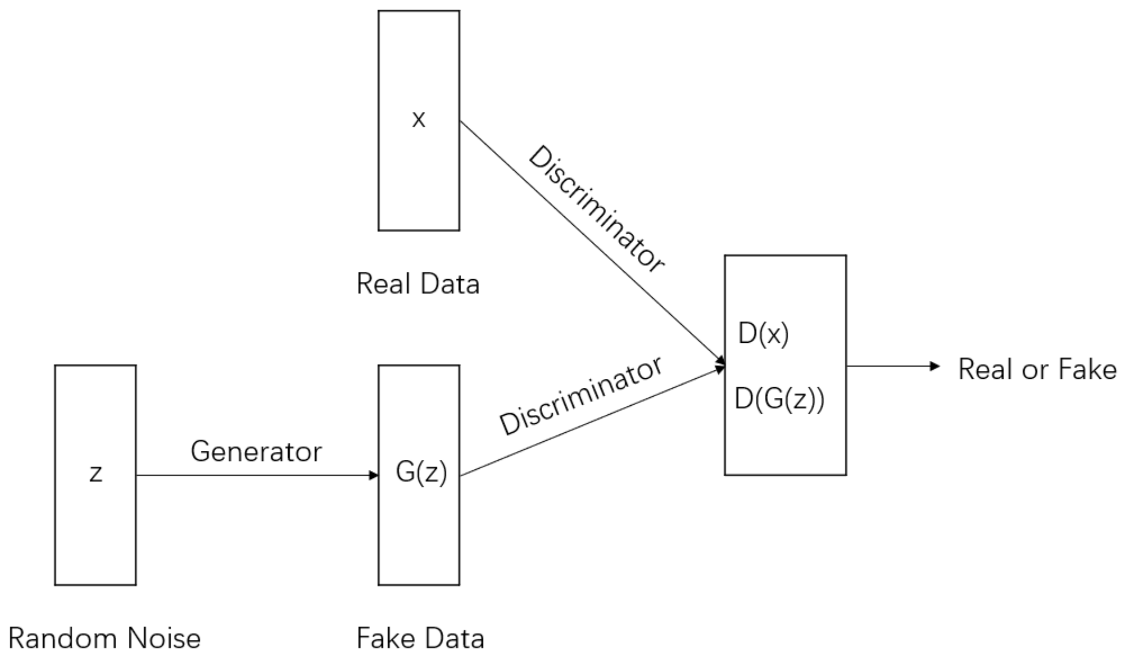


Figure 1.2: A demonstration of Generative Adversarial Network.

a similar distribution as $p_{data}(x)$, and the discriminator tries to assign $D(x)$ with the real data label and $D(G(z))$ with the generated (or called 'fake') data label. See Figure 1.2 for a pictorial demonstration.

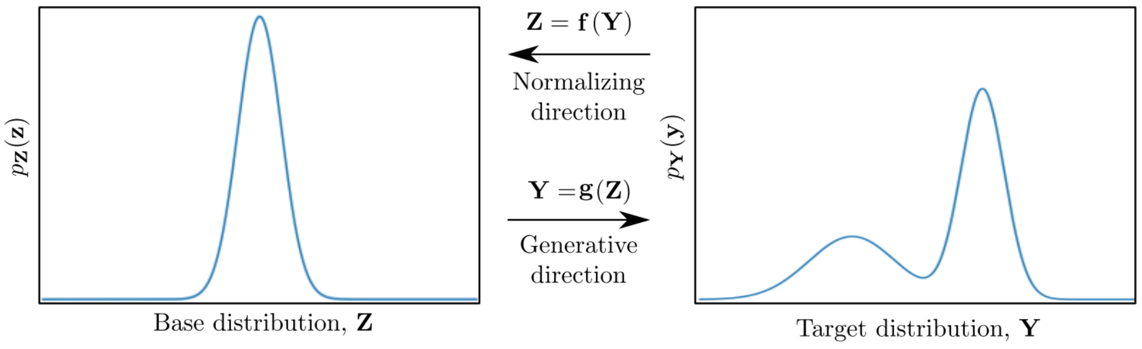
The generator network G and the discriminator network D are trained simultaneously using the following minimax loss function (1.8) [33]:

$$\underset{\mathbf{G}}{min}\,\underset{\mathbf{D}}{max}\,\mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_Z(z)}[\log(1 - D(G(z))] \tag{1.8}$$

Ideally, after the training process, the discriminator can no longer distinguish real data from the data generated by the generator.

Generative Adversarial Networks have been applied in many different applications. For example, DCGAN [90] and CycleGAN [116] use Generative Adversarial Network to generate images, MuseGAN [22] can generate music, and NetGAN [8] can generate similar graphs via Generative Adversarial Network.

Although Generative Adversarial Network can generate high-quality data, the



$\mathbf{Z} = \mathbf{f}(\mathbf{Y})$
Normalizing direction
$\mathbf{Y} = \mathbf{g}(\mathbf{Z})$
Generative direction
$p_{\mathbf{Z}}(\mathbf{z})$
$p_{\mathbf{Y}}(\mathbf{y})$
Base distribution, **Z**
Target distribution, **Y**

Figure 1.3: The idea of normalizing flow [65].

generated samples from Generative Adversarial Network sometimes can be very similar or even identical. This issue is called *mode collapse*. Recent work [4] [80] try to alleviate this issue but haven't completely resolved it.

1.4.3 Normalizing Flow

Normalizing Flow was first introduced by Rezende and Mohamed [92] and Dinh et al. [20]. Normalizing flow constructs a transformation between the target distribution and a simple probability distribution, which is usually called a base distribution, by a sequence of invertible and differentiable mappings. One can generate a sample z from the base distribution, and then apply the inverse of the normalizing flow model on z to get a generated data. This process is illustrated in Figure 1.3.

The loss function of normalizing flow models is the negative log-likelihood loss function $-\log(p_Y(y))$, where Y is the target distribution. Equation (1.9) shows how to calculate the negative log-likelihood loss function, which is obtained by using the change of variables formula from statistics:

$$\log(p_Y(y)) = \log(p_Z(z)) + \log(|\det(\frac{\partial z}{\partial y})|) \tag{1.9}$$

The $\det(\frac{\partial z}{\partial y})$ in Equation (1.9) denotes the determinant of the Jacobian matrix J, where J is defined as $J_{ij} = \frac{\partial z_i}{\partial y_j}$ for $i, j \in \{1, 2, 3, ..., n\}$ if $z, y \in \mathbb{R}^n$.

If the mapping $f : Y \to Z$ has the form $f = f_1 \circ f_2 \circ \cdots \circ f_N$, the determinant of the Jacobian could be written as

$$|\det(\frac{\partial f(y)}{\partial y})| = \prod_{i=1}^{N} |\det(\frac{\partial f_i(z_i)}{\partial z_i})| \tag{1.10}$$

where $z_i = f_{i+1} \circ f_{i+2} \circ \cdots \circ f_N(y)$ and $z_N = y$.

Hence, Equation (1.9) could be written as

$$\log(p_Y(y)) = \log(p_Z(z)) + \sum_{i=1}^{N} \log(|\det(\frac{\partial f_i(z_i)}{\partial z_i})|) \tag{1.11}$$

for $z_i = f_{i+1} \circ f_{i+2} \circ \cdots \circ f_N(y)$ and $z_N = y$.

One can see from Equation (1.11) that, an efficient and practical normalizing flow model should satisfy the following three principles [65]:

- invertible
- sufficiently expressive to map the target distribution to base distribution
- computationally efficient for computing f, f^{-1} and determinant of Jacobian

In recent years, researchers have developed several different types of normalizing flow layers following these principles. The three most representative ones are coupling layer [20], autoregressive layer [61] and continuous normalizing flow layer [11].

Coupling Layer

Dinh et al. [20] introduced a method called coupling layer. Suppose the input of the layer is $y \in \mathbb{R}^D$ and a disjoint partition of y into $(y_1, y_2) \in \mathbb{R}^d \times \mathbb{R}^{D-d}$, define an invertible function $g(\cdot; \theta) : \mathbb{R}^d \to \mathbb{R}^d$, parameterized by θ. Then, the output $z = (z_1, z_2) \in \mathbb{R}^d \times \mathbb{R}^{D-d}$ of the layer can be calculated as:

$$\begin{aligned} z_1 &= y_1 \\ z_2 &= g(y_2, \theta(y_1)) \end{aligned} \tag{1.12}$$

In 2017, Ding et al. [21] proposed a specific function g shown in Equation (1.12) which is widely used nowadays:

$$\begin{aligned} z_1 &= y_1 \\ z_2 &= y_2 \odot \exp(s(y_1)) + t(y_1) \end{aligned} \tag{1.13}$$

where $\odot$ denotes the element-wise product, $s(\cdot)$, $t(\cdot)$ are functions from $\mathbb{R}^d$ to $\mathbb{R}^{D-d}$, usually parameterized by neural networks.

The inverse of Equation (1.13) is given by

$$\begin{aligned} y_1 &= z_1 \\ y_2 &= (z_2 - t(z_1)) \odot \exp(-s(z_1)) \end{aligned} \tag{1.14}$$

The determinant of the Jacobian matrix of Equation (1.13) is simply $\prod_j \exp(s(y_1)_j)$.

Many recent popular Normalizing flow models are based on the coupling layers, such as Glow [60], Flow++ [45].

Autoregressive Layer

Kingma et al. [61] introduced the autoregressive normalizing flow layer. Suppose the input of the layer is $y \in \mathbb{R}^D$, the output $z \in \mathbb{R}^D$ of this layer can be calculated as:

$$z_t = h(y_t; \theta_t(y_{1:t-1})) \tag{1.15}$$

where $h(\cdot; \theta) : \mathbb{R} \to \mathbb{R}$ is an invertible function parameterized by θ, $y_{1:t-1} = (y_1, y_2, ..., y_{t-1})$ for $t = 1, 2, ..., D$.

The inverse of the autoregressive layer can be calculated as:

$$y_t = h^{-1}(z_t; \theta_t(y_{1:t-1})) \tag{1.16}$$

for $t = 2, 3, ..., D$ and $y_1 = h^{-1}(z_1)$.

The Jacobian matrix of the autoregressive layer is triangular. This is because z_t only depends on $y_1, y_2, ..., y_t$. As a result, the determinant of the Jacobian matrix is $\prod_{t=1}^{D} \frac{\partial z_t}{\partial y_t}$.

Some recent work try to improve the performance of the standard autoregressive layer. For example, MAF [86] adopts the mask techniques and BNAF [16] uses the block matrix structure to make the determinant computation more efficient.

Continuous Normalizing Flow Layer

Chen et al. [11] proposed the continuous normalizing flow layer

$$\frac{dz}{dt} = f(z(t), t),$$

which is an ordinary differential equation describing a continuous transformation of $z(t)$. The input of continuous normalizing flow layers is $z(0) = y$, which is the value of $z(t)$ at the time $t = 0$ and the output of this layer is $z(1)$.

One needs to apply an ordinary differential equation solver to calculate $z(1)$ based on $z(0)$, and also $z(1)$ based on $z(0)$.

The change in log probability formula, which is a revised vision of Equation (1.11), is:

$$\frac{\partial \log(p(z(t)))}{\partial t} = -tr(\frac{df}{dz(t)}) \tag{1.17}$$

Recent studies focus on generalizing the form of continuous normalizing flow, such as FFJORD [35], or trying to improve the performance of continuous normalizing flow,

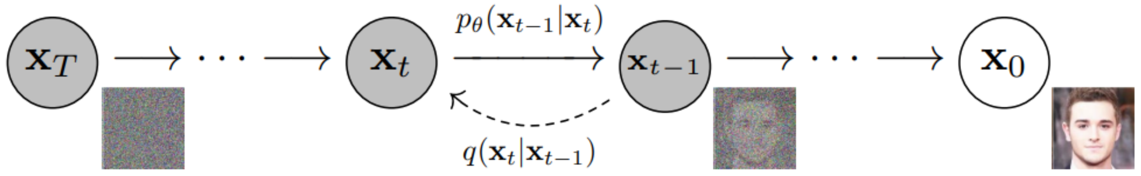

$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$
$\mathbf{x}_T$
$\mathbf{x}_t$
$\mathbf{x}_{t-1}$
$\mathbf{x}_0$
$q(\mathbf{x}_t|\mathbf{x}_{t-1})$

Figure 1.4: The directed graphical model considered in DDPM [55].

such as OT-Flow [83].

1.4.4 Denoising Diffusion Model

Diffusion-based generative models are the latest deep generative models, which was first proposed by J. Sohl-Dickstein et. al. [101] in 2015 based on statistical thermodynamics.

The denoising diffusion model has two processes. One process is called "forward (diffusion) process", which aims to transform the target distribution to normal distribution. In the forward process, the diffusion model sequentially adds noise to the given data. Another process is called "reverse (denoising) process", a model is trained in this process to reverse the diffusion process. Figure 1.4 shows the graphical model of denoising diffusion probabilistic models (DDPM) [55].

In Figure 1.4, from right to left, DDPM adds noise to the given data x_0 by function q, which is the "forward process". From left to right, DDPM tries to find p_θ to reverse the forward process to get the image from x_T, which is the "reverse process".

Mathematically, given a data point $x_0 \sim q(x_0)$, DDPM defines the forward diffusion process as a Markov chain, which gradually adds Gaussian noise to the data over T steps, producing a sequence of data $x_1, x_2, ..., x_T$. The formula of getting x_t is:

$$\begin{aligned} q(x_t|x_{t-1}) &= \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I) \\ q(x_{1:T}|x_0) &= \prod_{t=1}^{T} q(x_t|x_{t-1}) \end{aligned} \tag{1.18}$$

The reverse process in DDPM is also defined as a Markov chain with the assumption $p(x_T) = \mathcal{N}(x_t; 0, I)$. To get x_0 from x_T, DDPM trains two functions $\mu_\theta(x_t, t)$ and $\Sigma_\theta(x_t, t)$. Define the joint distribution $p_\theta(x_{0:T})$ as:

$$\begin{aligned} p_\theta(x_{0:T}) &= p(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t) \\ p_\theta(x_{t-1}|x_t) &= \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t)) \end{aligned} \tag{1.19}$$

The loss function in the training process of DDPM is the variational bound on negative log likelihood:

$$\begin{aligned} \mathbb{E}[-\log p_\theta(x_0)] &\leq \mathbb{E}_q[-\log \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)}] \\ &= \mathbb{E}_q[-\log p(x_T) - \sum_{t\geq 1} \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_t|x_{t-1})}] \end{aligned} \tag{1.20}$$

Denoising diffusion probabilistic models (DDPM) [55] achieved the state-of-the-art performance on image sampling task in 2021, which was introduced by Prafulla Dhariwal and Alex Nichol [18]. After that, DDPM attracts reserachers from all over the world. There are many other diffusion models, such as sore-based model [103] which uses the gradients of $\log p(x)$ in the forward process.

Moreover, diffusion models can be used to generate different types of data. For example, DiffWave [66] generates wave by using diffusion models, Austin et al. [5] work on designing diffusion models for discrete data.

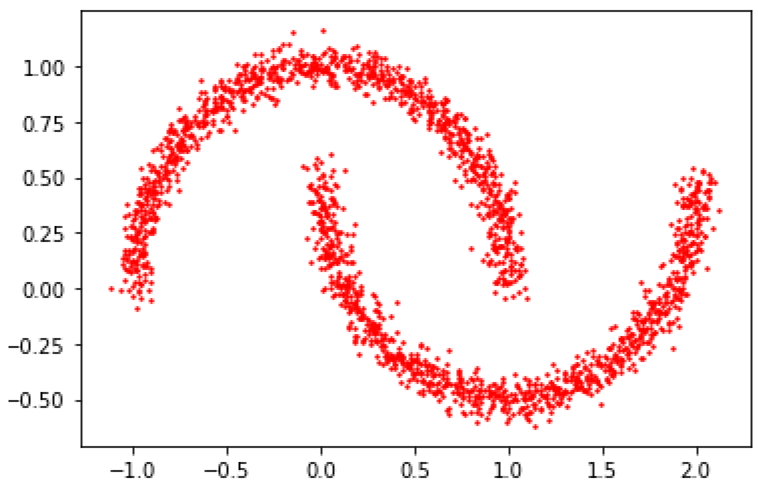


Figure 1.5: 2000 sampled data in moon dataset with 0.05 noise.

1.4.5 Comparison of Generative Models on a Simple 2D Dataset

In this section, I compare the following four generative models discussed in the previous section on a simple 2D dataset: two moon dataset.

- Variational AutoEncoder [59]
- Generative Adversarial Network [33]
- Normalizing Flow: RealNVP [21]
- Denoising Diffusion Model: DDPM [55]

Dataset configuration

The two moon dataset is got from “sklearn” Python package by using command

“*sklearn.datasets.make_moons*”

Figure 1.5 shows the data in moon dataset.

Model configuration

I try to control the number of parameters roughly the same in all four generative models. However, Variational AutoEncoder requires more parameters in order to get a reasonable result in this task. In each epoch in all models, I sample 2,000 data points. The batch size is set as 128 and the optimizer used in this experiment is Adam [58].

For Variational AutoEncoder, I set the dimension of latent space as 128. In Encoder model, I set one hidden layer with dimension 128, and in Decoder model, I also set one hidden layer with dimension 128. Learning rate is 1e-4. The number of parameters in this model is 83,202.

For Generative Adversarial Network, the dimension of the noise in Generator is set as 32. Both the Generator and the Discriminator have only one hidden layer with dimension 128. Learning rate is 1e-4. The number of parameters in this model is 4,482 (Generator) + 513 (Discriminator).

For Normalizing Flow: I choose 8 coupling layers in RealNVP [21] and use a deep neural network with only one hidden layer with dimension 128 to parameterize both s and t at each layer. Learning rate is set as 1e-4. The number of parameters in this model is 10,272.

For Denoising Diffusion Model: I choose DDPM model with a Conditional Linear Model of two hidden layers with dimension 128. Learning rate is 1e-3. The number of parameters in this model is 11,874.

Results

Figure 1.6 shows the samples generated by VAE after 200, 400, 600, 800, 1,000 epochs. The total time used for training 1000 epochs is 306.89 seconds.

Figure 1.7 shows the samples generated by GAN after 2,000, 4,000, 6,000, 8,000, 10,000 epochs. The total time used for training 10000 epochs is 467.50 seconds.

Figure 1.8 shows the samples generated by Normalizing Flow after 200, 400, 600, 800, 1,000 epochs. The total time used for the training 1000 epochs is 87.10 seconds.

Finally, Figure 1.9 shows the samples generated by DDPM after 200, 400, 600,

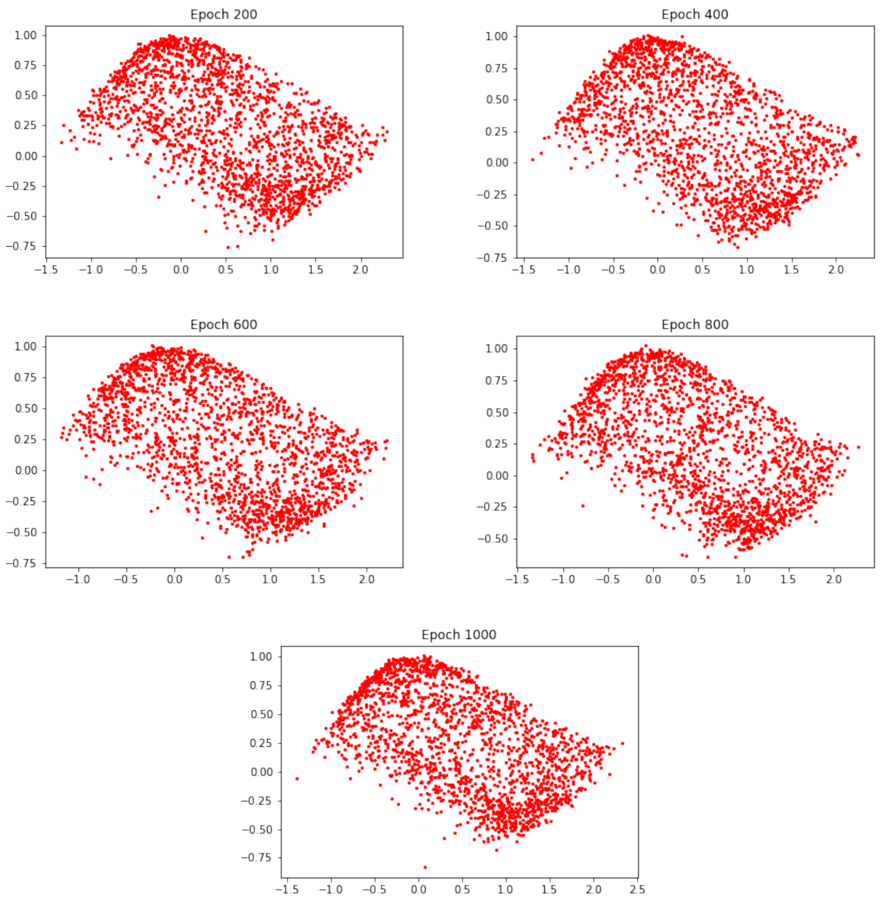


Figure 1.6: Samples generated by VAE after different training epochs.

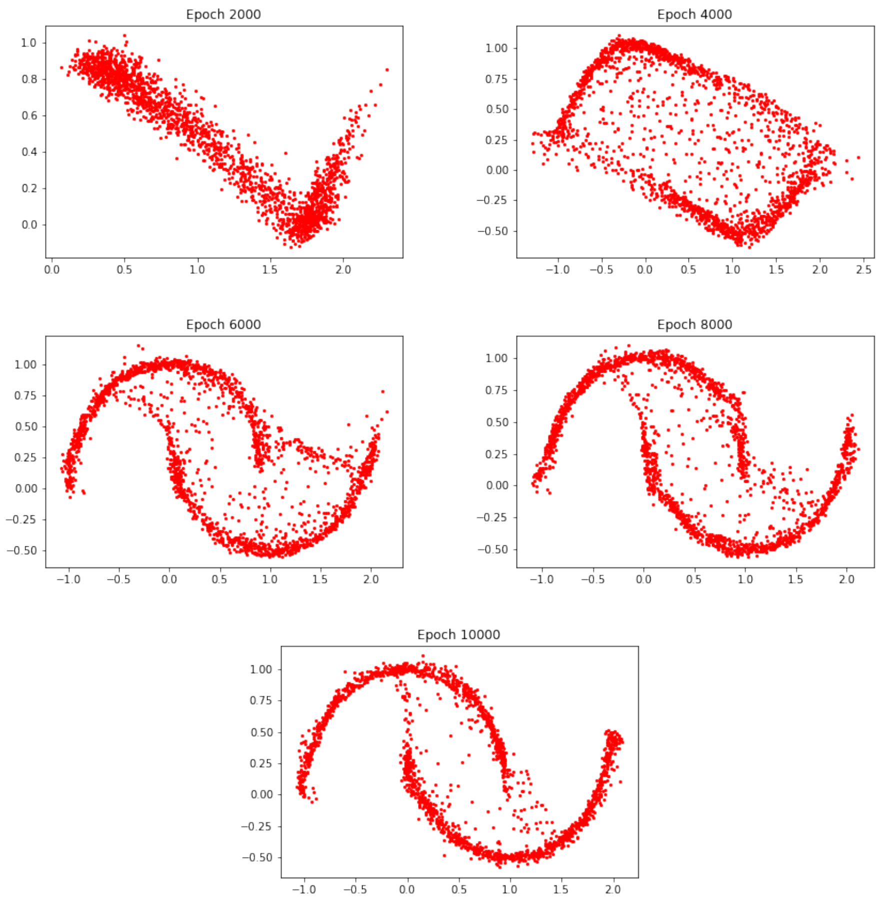


Figure 1.7: Samples generated by GAN after different training epochs.

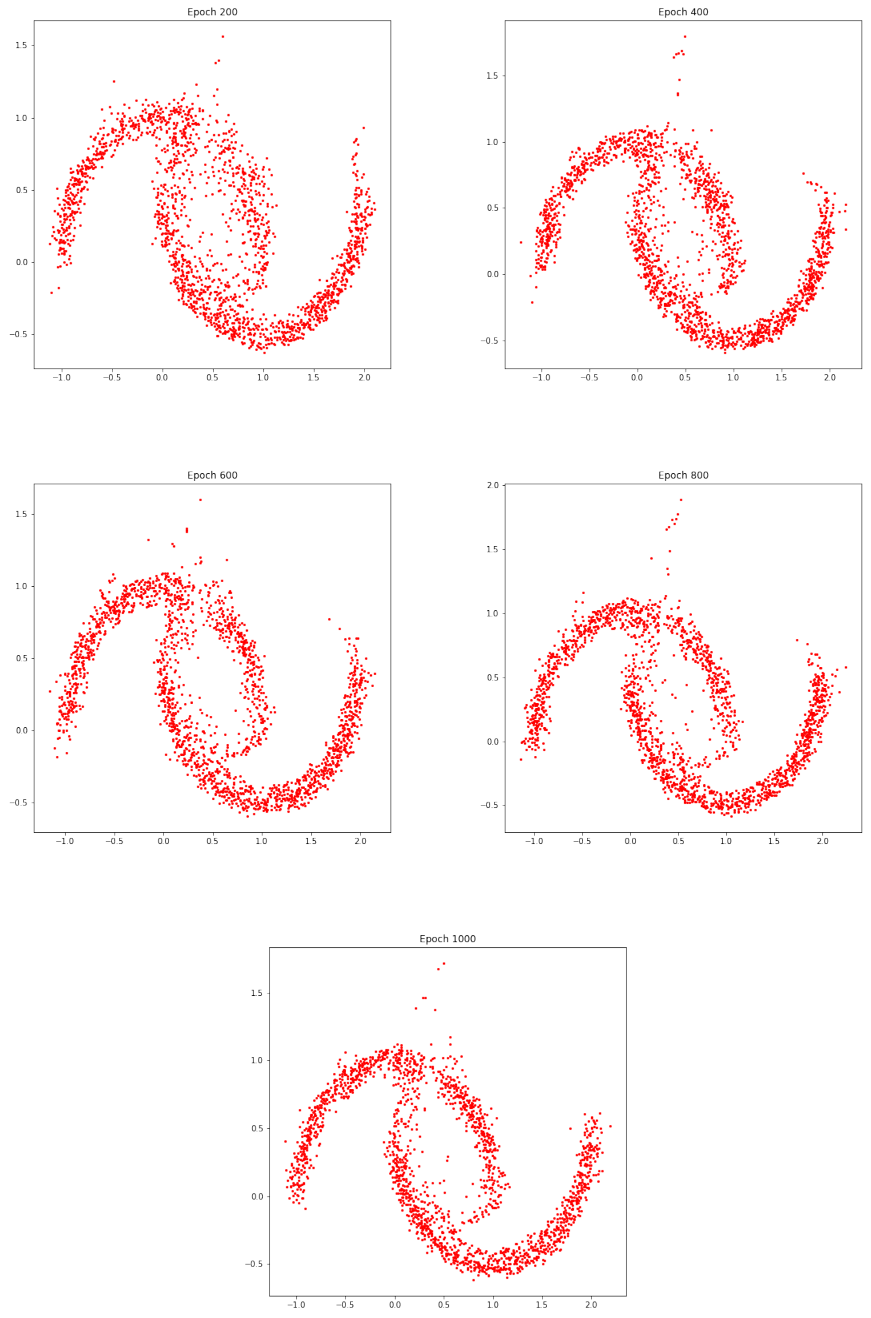


Figure 1.8: Samples generated by Normalizing Flow after different training epochs.

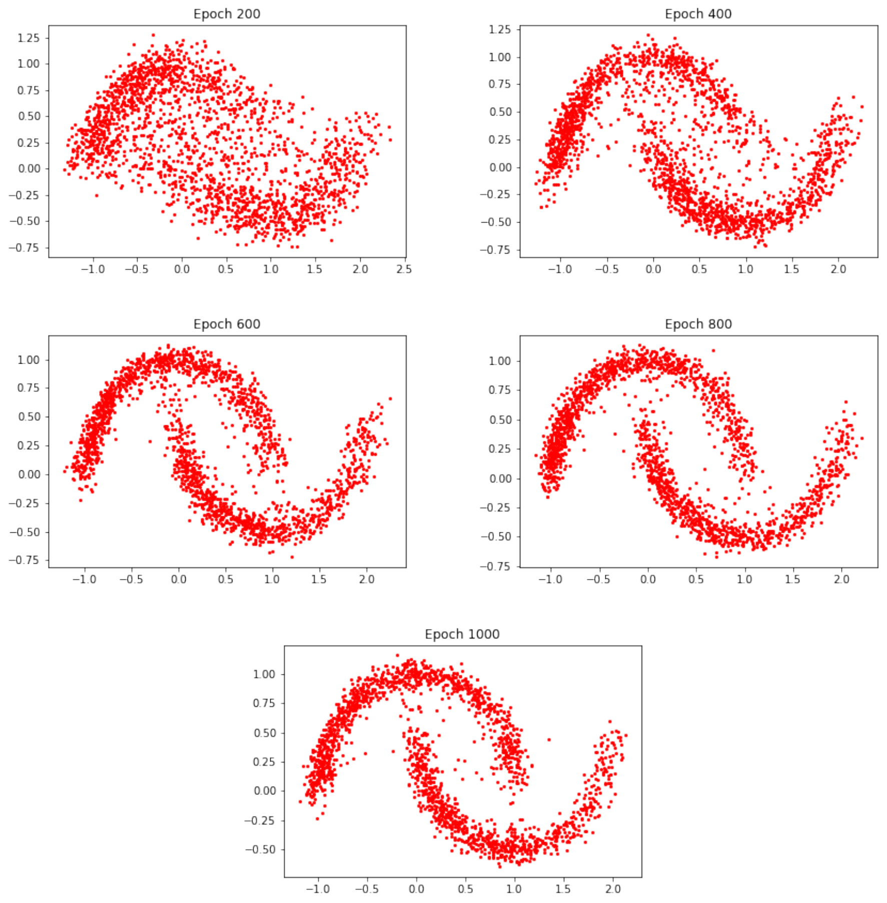


Figure 1.9: Samples generated by DDPM after different training epochs.

800, 1000 epochs. The total time used for training 1000 epochs is 151.85 seconds.

I initially fixed the number of training epochs as 1000 in all four models. However, for GAN model, 1000 epochs are not enough for getting a reasonable result. Thus, I increased the number of training epochs to 10,000.

We can see that, to get reasonable results in this task, Normalizing Flow is the fastest among the four models, and GAN is the lowest. Besides, the performance of VAE is the worst.

However, in different tasks, the results of the four generative models are different. For example, in image tasks, DDPM models have the lowest speed among these models, both for training and sampling but usually outperform other generative models in term of the quality of the generated samples.

1.5 Graph and Graph Neural Network

Since this thesis focuses on combining graph theory with deep generative models to improve the generation quality and efficiency, I will review some classical definitions from graph theory as well as the related graph neural networks in this section.

1.5.1 Graph

The word "graph" was first used by J. J. Sylvester in 1878 [111]. A graph (or called "simple graph") G contains a set $V(G)$, which is the set of elements called vertices (or nodes), and a set $E(G)$, which is the set of distinct unordered pairs of distinct elements of $V(G)$ [9]. The elements in set $E(G)$ are called edges, and two vertices (or nodes) are called connected or adjacent if there is an edge (i, j) in set $E(G)$. Graph G can be written as $G(V, E)$.

The adjacency matrix is a matrix used to represent a simple graph. Suppose the matrix is A. If the element A_{ij} is equal to 1, it indicates that the ith node and the jth node are adjacent. On the other hand, if the element A_{ij} is equal to 0, it indicates that the ith node and the jth node are not adjacent.

1.5.2 Graph Neural Network

There are three general types of tasks on graphs: node-level tasks, edge-level tasks, and graph-level tasks [115]. Node-level tasks include node classification, node regression, node clustering, etc. Edge-level tasks include edge classification, link prediction, edge generation, and Graph-level tasks include graph classification, graph regression, graph matching, etc. [115]

In order to get high performance on these tasks, Graph Neural Network (GNN) has been introduced. Graph Neural Network is the family of Neural Networks designed for operating on the graph structure.

Kipf and Welling proposed Graph Convolutional Network (GCN) [64] in 2017, which is a widely used Graph Neural Network layer. Graph Convolutional Network defines the layers based on Equation (1.21):

$$Y = \sigma(\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}XW) \tag{1.21}$$

where $\widetilde{A} = A + I_{N \times N}$. Here, A is the adjacency matrix of the graph G, N is the number of nodes in the graph G, $I_{N \times N}$ is the identity matrix, $\widetilde{D}_{ii} = \sum_j \widetilde{A}_{ij}$, X is the input, and W is a trainable weight matrix. Here $\sigma(\cdot)$ denotes a non-linear function.

Another widely used Graph Neural Network layer is called GraphSAGE [41]. For each node in the graph, GraphSAGE aggregates the node information from its neighbors in each GraphSAGE layer. More formally, GraphSAGE defines each layer based on Equation (1.22):

$$\begin{aligned} y_{\mathbf{N}(v)} &= AGGREGATE(\{x_u, \forall u \in \mathbf{N}(v)\}) \\ y_v &= \sigma(W \cdot concat(x_v, y_{\mathbf{N}(v)})) \end{aligned} \tag{1.22}$$

where x_v is the input feature and y_v is the output feature, W is a trainable weight matrix, σ is a non-linear function. AGGREGATE denotes a differentiable aggregator function, and $\mathbf{N}(v)$ represents the neighbors of node v.

Also, recent studies provide improvements on different types of Graph Neural Networks, such as FastGCN [10], Graph Attention Network [105], Graph LSTM [87].

1.6 Summary of Contributions

My research mainly focuses on combining graph theory and deep generative model to improve the training efficiency as well as the quality of the generated samples.

Most of the AI breakthroughs during the last several years relied on enormous data in the datasets, such as CIFAR10 datasets with 60000 images. However, in some cases, people only have a small dataset. Without using domain knowledge, it is hard to achieve desirable performance if one blindly applies existing deep neural network models.

In Chapter 2, I propose a machine learning algorithm to generate doppelganger graphs of a given graph. The method is able to produce a different graph surrogate through sampling a node set each time, which is the first work on this topic. The method is an innovative combination of graph representation learning, generative adversarial networks and graph realization algorithms in combinatorics. The graphs generated by the proposed method preserve global and local properties. Such an outcome is because the graph realization algorithm guarantees that the generated graphs preserve the degree sequence, while the generative adversarial network limits the number and the sizes of the cliques. To demonstrate the application of the doppelgangers, I perform a downstream task: node classification, on these generated graphs and show that they achieve a similar classification performance to the original one.

In Chapter 3, I propose AUTM flow, which provides a new set of normalizing flow layers. In this work, a new integral-based monotone triangular flow with proven universal approximation property for monotonic flows is developed. The proposed integral-based transformation is strictly increasing with small constraint, and the inverse formula is explicitly given and compatible with fast root-finding methods for numerical inversion. Also, the proposed model is universal in the sense that any monotonic normalizing flow is a limit of the proposed flows.

In Chapter 4, I propose a deep learning method to learn the distribution of a given set of small graphs and generate similar ones based on diffusion models. This method combines Graph Auto-Encoder and the Denoising Diffusion Models to generate graphs with guaranteed permutation invariant property. Besides, to our knowledge, it is the first method that combines the Denoising Diffusion Models and the node generation.

In Chapter 5, I summarize my work and also discuss some research topics that can be studied based on the methods proposed in this thesis.

Chapter 2

Generating a doppelganger graph from a single graph

2.1 Background

The majority of deep generative models for graphs [53, 76, 99, 38, 113, 72, 73, 12] follows the paradigm of "learning from a distribution of graphs and generating new ones". Another problem, which is equal significance in practice, is to learn from a single graph and generate new ones. This problem is largely under explored. The problem meets graphs that are generally large and unique, which contains rich structural information but rarely admits replicas. Examples of such graphs are financial networks, power networks, and domain-specific knowledge graphs. Often, these graphs come from proprietary domains are hardly released to the public for open science.

One live example that necessitates the creation of surrogate graphs is in the financial industry. A financial entity (e.g., banks) maintains a sea of transaction data, from which traces of fraud are identified. Graph neural network techniques have been gaining momentum for such investigations [107, 108], where a crucial challenge is the lack of realistic transaction graphs that no existing graphs resemble and that synthetic generators fail to replicate. On the other hand, many barriers exist for the financial entities to distribute their data, even to its own departments or business partners, because of levels of security and privacy concerns.

2.2 Problem

In this project, we consider the problem of generating a new (simple) graph G based on one given (simple) graph G_0. The generated graph G and the original graph G_0 should have low edge overlap under permutation of nodes but share similar metrics (e.g. number of triangles).

2.3 Related Work

There are two different approaches to solve this problem: Graph Theory approach and Deep Generative Model approach.

Early generative models for graphs focus on characterizing certain properties of interest, which are based on graph theory. The Erdös-Rényi model [26, 31], dating back to the 1950s, is one of the most extensively studied random models. The Watts-Strogatz model [106] intends to capture the small-world phenomenon seen in social networks; the Barabási-Albert model [6] mimics the scale-free (powerlaw) property; and the stochastic block model [32, 57] models community structures. These models, though entail rich theoretical results, are insufficient to capture all aspects of real-life graphs.

The complex landscape of graphs in applications urges the use of higher-capacity models, predominantly neural networks, to capture all properties beyond what random graph models cover. Hence, researchers try to use Deep Generative Models in these years. Three most widely used frameworks are VAEs [59], GANs [33], and normalizing flows [92].

The VAE framework encodes a graph into a latent representation and decodes it for maximal reconstruction. Approaches such as GraphVAE [99] and Graphite [38] reconstruct the graph adjacency matrix (and labels if any). The Junction Tree VAE approach [53] leverages knowledge of molecular substructures and uses the tree skeleton rather than the original graph for encoding and decoding.

The GAN framework uses a generator to produce graphs (typically in the form of adjacency matrices) and a discriminator to tell produced results from training examples. The MolGAN approach [15] incorporates a reinforcement learning term into the discriminator loss to promote molecule validity. The LGGAN approach [27] additionally generates graph-level labels. Besides decoding a graph adjacency matrix, one may treat the graph generation process as sequential decision making, producing nodes and edges one by one. Representative examples of RNN-type of graph decoders are DeepGMG [114], GraphRNN [113], and GRAN [72].

The normalizing flow framework uses an invertible transformation to map between the training data distribution and a "simple distribution", such as standard normal. Methods of this framework generally adapt different flow architectures to graphs. For example, GNF [73] is based on coupling flows, GraphAF [12] is based on autoregressive flows, and MolGrow [70] is based on hierarchical flows.

All the above methods learn from a collection of graphs. More relevant to this chapter is the problem that the training set consists of one single graph. NetGAN [7] is one of the rare studies under this problem. This approach reformulates the problem as learning from a collection of random walks, so that the graph is recovered through assembling sampled random walks. VGAE [62] also learns from a single graph; but its aim is to reconstruct the graph rather than producing new ones.

2.4 Generating a Doppelganger Graph: Resembling but Distinct

We propose a new approach to learn the structure of a given graph and construct new ones that carry similar properties. This approach samples another set of nodes with novel embeddings and improves the Havel-Hakimi graph realization algorithm to determine a new connectivity structure, based on the embeddings. We call the generated graphs *doppelgangers*, which (i) match the input graph in global and local characteristics and (ii) are not limited to a single surrogate.

It is worthwhile to distinguish our setting from that of NetGAN [7], a related and representative approach to learning from a single graph. NetGAN solves the problem by learning the distribution of random walks on the given graph; then, it assembles a graph from new random walks sampled from the learned model. With more and more training examples, the generated graph is closer and closer to the original one, which goes in the opposite direction of aim (ii) above. In essence, NetGAN learns a model that memorizes the transition structure of the input graph and does not aim to produce a graph with an entirely different structure. This approach capitalizes on a fixed set of nodes and determines only edges among them, as most of the prior work under the same setting does. We intend to generate new nodes and new topologies on the contrary.

In the following subsections, I will discuss about the details of our work "Generating a Doppelganger Graph: Resembling but Distinct" [52].

2.4.1 Graph Realization Algorithm: Havel–Hakimi

A sequence S of integers $(d_1, d_2, \ldots, d_n)$ is called *graphic* if the d_i's are nonnegative and they are the degrees of a graph with n nodes. Havel-Hakimi [43, 40] is an algorithm that determines whether a given sequence is graphic. The algorithm is constructive in that it will construct a graph whose degrees coincide with S, if graphic.

At the heart of HH is the following theorem: Let S be nonincreasing. Then, S is graphic if and only if $S' = (d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n)$ is graphic. Note that S' is not necessarily sorted.

This theorem gives a procedure to attempt to construct a graph whose degree sequence coincides with S, as follows. Label each node with the desired degree. Beginning with the node v of highest degree d_1, connect it with d_1 nodes of the next highest degrees. Set the remaining degree of v to be zero and reduce those of the v's neighbors identified in this round by one. Then, repeat: select the node with the highest remaining degree and connect it to new neighbors of the next highest remaining degrees. The procedure terminates when (i) one of the nodes to be connected to has a remaining degree zero or (ii) the remaining degrees of all nodes are zero. If (i) happens, then S is not graphic. If (ii) happens, then S is graphic and the degree sequence of the constructed graph is S. The pseudocode is given in Algorithm 2. Figure 2.1 illustrates an example.

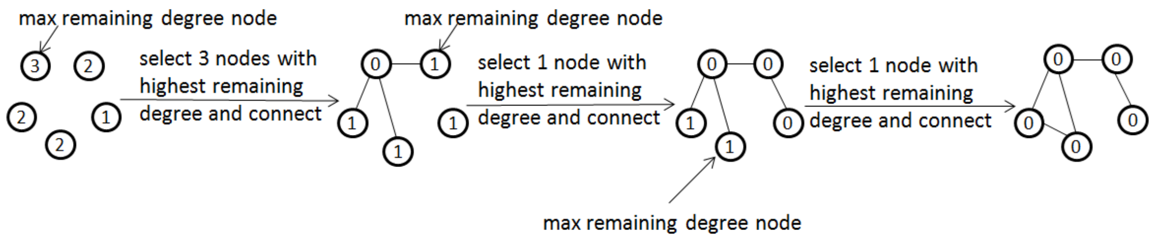

max remaining degree node
select 3 nodes with highest remaining degree and connect
max remaining degree node
select 1 node with highest remaining degree and connect
select 1 node with highest remaining degree and connect
max remaining degree node

Figure 2.1: Examples of generating a graph by using the original HH algorithm

Algorithm 2: Havel–Hakimi Algorithm

```
Input  : Graphic sequence ${d_i}$
Output: Adjancency list $adj\_list(i)$ for each node $i$
1  Initialize $adj\_list(i) = \{\}$ for all nodes $i$
2  repeat
3      Initialize $rd_i = d_i - len(adj\_list(i))$ for all $i$
4      Set $done = true$
5      Select node $k = \arg\max_i(rd_i)$
6      repeat
7          Select node $t = \arg\max_{i \neq k,\ i \notin adj\_list(k)}(rd_i)$
8          if $rd_t > 0$ then
9              Add $t$ to $adj\_list(k)$ and add $k$ to $adj\_list(t)$
10             Set $done = false$
11         end
12     until $rd_t = 0$ or $d_k = len(adj\_list(k))$;
13 until $done$ is true or $len(adj\_list(i)) = d_i$ for all $i$;
```

The most attractive feature of HH is that the constructed graph reproduces the degree sequence of the original graph, thus sharing the same degree-based properties, as the following Theorem 2.4.1 formalizes.

Theorem 2.4.1. Let $P(\{d_1, \ldots, d_n\})$ be a graph property that depends on only the node degrees d_1, ..., d_n. A graph G realized by using the Havel-Hakimi algorithm based on the degree sequence of G_0 has the same value of P as does G_0. Examples of P include wedge count, powerlaw exponent, entropy of the degree distribution, and the Gini coefficient.

Proof. Since the degree sequence of the original graph is graphic, the Havel–Hakimi algorithm is guaranteed to generate a new graph. By construction, the new graph has the same degree sequence.

Let V be the node set and $d(v)$ be the degree of a node v. The wedge count is defined as $\sum_{v \in V} \binom{d(v)}{2}$. The powerlaw exponent is $1+n(\sum_{v\in V} \log \frac{d(v)}{d_{\min}})^{-1}$, where $d_{\min}$ denotes the smallest degree. The entropy of the degree distribution is $\frac{1}{|V|}\sum_{v\in V} -\frac{d(v)}{|E|}\log\frac{d(v)}{|E|}$, where E denotes the edge set. The Gini coefficient is $\frac{2\sum_{i=1}^{|V|} i\widehat{d}_i}{|V|\sum_{i=1}^{|V|}\widehat{d}_i} - \frac{|V|+1}{|V|}$, where $\widehat{d}$ is the list of sorted degrees. All these quantities are dependent solely on the degree sequence. $\square$

On the other hand, a notable consequence of HH is that the constructed graph tends to generate large cliques.

Theorem 2.4.2. Suppose the degree sequence of a given graph is $d_1 \geq d_2 \geq ... \geq d_n$. If there exists an integer k such that $d_k - d_{k+1} \geq k-1$, then the graph realized by the Havel–Hakimi algorithm has a clique with at least k nodes.

Proof. Consider the k such that $d_1 \geq d_k \geq d_{k+1} + k - 1$. Because $d_{k+1} + k - 1 \geq k - 1$, the first node must have a degree $\geq k-1$. Hence, the Havel–Hakimi algorithm will connect this node to the 2nd, 3rd, ..., kth nodes. Then, the remaining degrees become $d_2 - 1, d_3 - 1, ..., d_k - 1, d'_{k+1}, \ldots, d'_n)$, where $d'_i = d_i$ or $d_i - 1$.

For the second node with $d_2 - 1$ remaining degrees, because $d_2 - 1 \geq k - 2$, it must be connected to the 3rd, 4th, ..., kth nodes. Then, the remaining degrees become $(d_3 - 2, ..., d_k - 2, d''_{k+1}, ..., d''_n)$, where $d''_i = d'_i$ or $d'_i - 1$.

After this process repeats $k-1$ times, we can see that the first k nodes have been connected to each other. As a result, the graph generated by the Havel–Hakimi algorithm must form a clique with at least k nodes. $\square$

Theorem 2.4.2 indicates a potential drawback of HH. When connecting with neighbors, HH always prefers high-degree nodes. Thus, iteratively, these nodes form a large clique (complete subgraph), which many real-life graphs lack. Furthermore, since any induced subgraph (say, with m nodes) of a k-clique is also a clique, the number of

m-cliques grows exponentially with k. Then, the number of small motifs (e.g., triangles and squares) easily explodes, departing substantially from the characteristics of a real-life graph. In other words, graphs generated by HH possibly fail to preserve certain local structures of the original graph.

2.4.2 Improved Havel–Hakimi

Since the original HH algorithm tends to produce graphs with large cliques, several properties of the original graph can hardly be preserved. To mitigate this drawback, a good link prediction model can redefine what nodes are linked in order, eliminating the tendency of linking nodes always ordered by their degrees. In this section, we propose two improvements over HH to achieve this goal. First, rather than selecting neighbors in each round by following a nonincreasing order of the remaining degrees, we select neighbors according to link probabilities. Second, such probabilities are computed by using a link prediction model together with new nodes sampled from the node distribution of the original graph G_0. This way, the new graph G carries the node distribution and the degree sequence information of G_0 but has a completely different node set.

Specifically, in each iteration, a node k with maximal remaining degree is first selected. This node is then connected to a node t with the highest link probability p_{kt} among all nodes t not being connected but with a nonzero remaining degree. Connect as many such t as possible to fill the degree requirement of k. If k cannot find enough neighbors (no more t exists with nonzero remaining degree), the algorithm will skip node k. This process is repeated until no more nodes can add new neighbors. The pseudocode is given in Algorithm 3. Figure 2.2 illustrates an example.

Algorithm 3: Improved Havel-Hakimi (our method)

```
    Input  : Graphic sequence $\{d_i\}$, $link\_predictor(\cdot,\cdot)$
    Output: Adjancency list $adj\_list(i)$ for each node $i$
 1  Initialize $adj\_list(i) = \{\}$ for all nodes $i$
 2  repeat
 3      Initialize $rd_i = d_i - len(adj\_list(i))$ for all $i$
 4      Set $done = true$
 5      Select node $k = \arg\max_i(rd_i)$
 6      repeat
 7          Select node $t = \arg\max_{i \notin adj\_list(k),\, rd_i > 0}(p_{ki})$, where
             $p_{ki} = link\_predictor(k, i)$
 8          if no such t exists then
 9              Exit Repeat
10          end
11          Add $t$ to $adj\_list(k)$ and add $k$ to $adj\_list(t)$
12          Set $done = false$
13      until $len(adj\_list(k)) = d_k$;
14  until done is true or $len(adj\_list(i)) = d_i$ for all $i$;
```

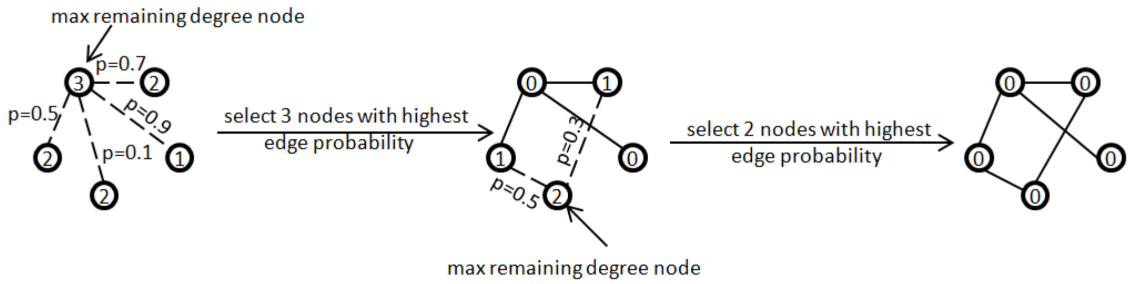

max remaining degree node
p=0.7
p=0.5
p=0.9
p=0.1
3
2
2
1
2
select 3 nodes with highest edge probability
0
1
1
0
2
p=0.3
p=0.5
max remaining degree node
select 2 nodes with highest edge probability
0
0
0
0
0

Figure 2.2: Examples of generating a graph by using the improved HH algorithm (ours)

2.4.3 Link Predictor for Improved HH

In this subsection, we elaborate how the probabilities for neighbor selection in Algorithm 3 are computed.

Link prediction model

Probabilities naturally come from a link prediction model, which dictates how likely an edge is formed. We opt to use a graph neural network (GNN) trained with a link prediction objective to obtain the link predictor. The GNN produces, as a byproduct, node embeddings that will be used subsequently for sampling new nodes. There exist many non-neural network approaches [88, 104, 37] for producing node embeddings, but they are not trained with a link predictor simultaneously.

We use GraphSAGE [42], which offers satisfactory link prediction performance, but enhance the standard link prediction model $p = \text{sigmoid}(z_v^T z_u)$ for two nodes v and u by a more complex one:

$$p = \text{sigmoid}(W_2 \cdot \text{LeakyReLU}(W_1(z_v \circ z_u) + b_1) + b_2) \tag{2.1}$$

where W_1, b_1, W_2, b_2 are parameters trained together with GraphSAGE. This link prediction model is slightly parameterized and we find that it works better in practice.

Sampling new nodes

After GraphSAGE is trained, we obtain as byproduct node embeddings of the input graph. These embeddings form a distribution, from which one may sample new embeddings. They form the node set of the doppelganger graph. Sampling may be straightforwardly achieved by using a deep generative model; we choose Wasserstein GAN [3]. We train WGAN by using the gradient penalty approach proposed by [39]. We also generate node input features alongside embeddings, if they are needed by

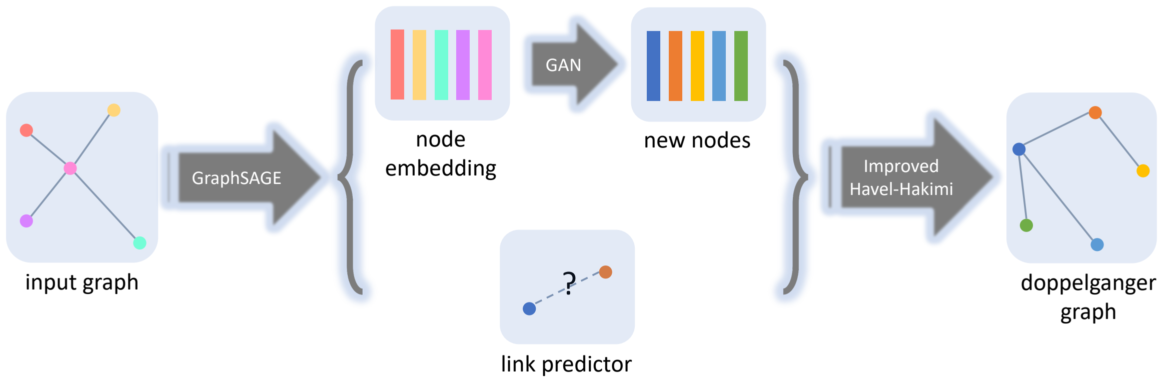


Figure 2.3: Pipeline to generate a doppelganger graph.

a downstream task, by concatenating the vector of input features and the vector of embeddings when forming the training set of WGAN.

Node labeling with degree sequence

A remaining detail is the labeling of the new nodes with the desired degree. We apply the link predictor on the new nodes to form an initial graph. With this graph, nodes are ordered by their degrees. We then relabel the ordered nodes by using the degree sequence of the input graph. This way, the improved HH algorithm will run with the correct degree sequence.

2.4.4 Summary

Given an input graph G_0, our method first trains a GraphSAGE model on G_0 with a link prediction objective, yielding node embeddings and a link predictor. We then train a Wasserstein GAN on the node embeddings of G_0 and sample new embeddings from it. Additionally, input node features can be generated similarly if they are needed in a downstream task. Afterward, we run the improved HH algorithm, wherein needed link probabilities are calculated by using the link predictor on the new embeddings generated by the GAN model. The resulting graph is a doppelganger of G_0. Figure 2.3 shows the framework of our method.

To generate k doppelgangers, we sample nk node embeddings from the GAN model, where n is the number of nodes in the original graph. Then, we run the improved HH algorithm on every n embeddings to obtain k different graphs.

2.4.5 Theoretical Analysis

In this subsection, we analyze the improved Havel–Hakimi algorithm in support of the superior quality of the graphs it generates, with respect to the preservation of graph properties. We first state in Lemma 2.4.3 that a property can be written as the value of a function of node embeddings under mild assumptions.

Lemma 2.4.3. Assume each node i of a graph has an embedding emb_i and there exists an exact link prediction model LP which satisfies that $LP(emb_i, emb_j) = 1$ if there is an edge between v_i and v_j and $LP(emb_i, emb_j) = 0$ otherwise. Under these assumptions, any graph property (e.g., triangle count and Gini coefficient) can be written as a function $F_{LP}(emb_1, emb_2, ..., emb_n)$, where n is the number of nodes in the graph. Furthermore, the property is in the form $F_{LP}(emb_1, emb_2, ..., emb_n) = \sum_{k=1}^{K_F} f_{LP}(emb_{k_1}, ..., emb_{k_t})$ for K_F sub-functions f_{LP}, where each $k_i \in \{1, 2, 3, ..., n\}$.

Proof. By the assumption, $LP(emb_i, emb_j)$ denotes whether there is an edge between nodes i and j. Hence, the edge set is simply $E = \{(i, j)|LP(emb_i, emb_j) = 1\}$. As a result, the value of a graph property can be written as

$$F(emb_1, emb_2, ...emb_n, \{(i, j)|LP(emb_i, emb_j) = 1\}) = F_{LP}(emb_1, emb_2, ..., emb_n).$$

For the second part of the lemma, if the property is global, then $K_F = 1$ and $f_{LP} = F_{LP}$. On the other hand, for local properties, the function F_{LP} is the sum of sub-functions for local structures. Therefore, K_F is the number of local structures and each sub-function f_{LP} is defined for one local structure.

We will give several examples of the form of function F_{LP} of the graph properties.

Consider the number of triangles in a graph. This property can be written as $\sum_{1\leq i<j<k\leq n} LP(emb_i, emb_j)LP(emb_i, emb_k)LP(emb_j, emb_k)$. In this case, we have

$$\begin{aligned}F_{LP}(emb_1, emb_2, ..., emb_n) &= \sum_{1\leq a<b<c\leq n} LP(emb_a, emb_b)LP(emb_a, emb_c)LP(emb_b, emb_c)\\ &= \sum_{1\leq a<b<c\leq n} f_{LP}(emb_a, emb_b, emb_c)\\ &= \sum_{k=1}^{K_F} f_{LP}(emb_{k_1}, emb_{k_2}, emb_{k_3})\end{aligned}$$

where $f_{LP}(emb_a, emb_b, emb_c) = LP(emb_a, emb_b)LP(emb_a, emb_c)LP(emb_b, emb_c)$. Here, K_F is the number of distinct tuples (a, b, c) and (k_1, k_2, k_3) takes values from all (a, b, c).

Consider the number of wedges in a graph. This property can be written as $\sum_{1\leq i<k\leq n, 1\leq j\leq n} LP(emb_i, emb_j)LP(emb_j, emb_k)$. In this case, we have

$$\begin{aligned}F_{LP}(emb_1, emb_2, ..., emb_n) &= \sum_{1\leq a<c\leq n, 1\leq b\leq n} LP(emb_a, emb_b)LP(emb_b, emb_c)\\ &= \sum_{1\leq a<c\leq n, 1\leq b\leq n} f_{LP}(emb_a, emb_b, emb_c)\\ &= \sum_{k=1}^{K_F} f_{LP}(emb_{k_1}, emb_{k_2}, emb_{k_3})\end{aligned}$$

where $f_{LP}(emb_a, emb_b, emb_c) = LP(emb_a, emb_b)LP(emb_b, emb_c)$. Here, K_F is the number of distinct tuples (a, b, c) satisfy $1 \leq a < c \leq n, 1 \leq b \leq n$ and (k_1, k_2, k_3) takes values from all (a, b, c).

Consider the number of global clustering coefficient in a graph. This property can be written as $[\sum_{1\leq i,j,k\leq n} LP(emb_i, emb_j)LP(emb_i, emb_k)LP(emb_j, emb_k)]/[\sum_{1\leq i\leq n} k_i(k_i-1)]$, where $k_i = \sum_j LP(emb_i, emb_j)$. In this case, we have

$$F_{LP}(emb_1, emb_2, ..., emb_n) = \frac{\sum_{1\leq i,j,k\leq n} LP(emb_i, emb_j)LP(emb_i, emb_k)LP(emb_j, emb_k)}{\sum_{1\leq i\leq n}\sum_j LP(emb_i, emb_j)(\sum_j LP(emb_i, emb_j) - 1)}$$

Here, K_F is just 1 and f_{LP} has the same form as F_{LP}. □

Based on Lemma 2.4.3, we can show that the expectation of the property of the graphs generated through sampling node embeddings is equal to the property of the original graph.

Theorem 2.4.4. Assume for a graph G_0 there exist an embedding emb_i for each node i, an exact link prediction model LP, and a property function F_{LP} in the form $F_{LP}(emb_1, emb_2, ..., emb_n) = \sum_{k=1}^{K_F} f_{LP}(emb_{k_1}, ..., emb_{k_t})$. Denote the property value for G_0 as C. When the embedding distribution learned by a trained generative model satisfies $\mathbb{E}[f_{LP}(x_1, ..., x_t)] = \frac{C}{K_F}$, where each x_i is an i.i.d. sample from the distribution, the expectation of $F_{LP}(emb_1, emb_2, ..., emb_n)$ is equal to C.

Proof. By linearity of expectation,

$$\begin{aligned}\mathbb{E}[F_{LP}(x_1, x_2, ..., x_n)] &= \mathbb{E}\left[\sum_{k=1}^{K_F} f_{LP}(x_{k_1}, ..., x_{k_t})\right] \\ &= \sum_{k=1}^{K_F} \mathbb{E}[f_{LP}(x_{k_1}, ..., x_{k_t})] \\ &= \sum_{k=1}^{K_F} \frac{C}{K_F} \\ &= C\end{aligned} \tag{2.2}$$

□

Theorem 2.4.4 only implies that if we construct graphs based on node embeddings generated by GANs, in expectation the property of these graphs is equal to that of the original graph. When we generate a single graph, the property value for this graph can have a large variance. In the next theorem, we show that a proper post processing technique applied on the link prediction model can bound the difference of property values of two graphs with high probability. For example, since the improved

Havel–Hakimi algorithm preserves the node degrees of the original graph, it can be considered as one post processing technique. For a post processing technique T, let us denote the property function resulting from applying T on a link prediction model LP as $F_{T(LP)}$, and similarly for the sub-functions $f_{T(LP)}$. We obtain have the theorem 2.4.5.

Theorem 2.4.5. Assume for a graph G_0 there exist an embedding emb_i for each node i, an exact link prediction model LP, and a property function F_{LP} in the form $F_{LP}(emb_1, emb_2, ..., emb_n) = \sum_{k=1}^{K_F} f_{LP}(emb_{k_1}, ..., emb_{k_t})$. Denote the property value for G_0 as C and the embedding distribution learned by the generative model as p_g. Suppose the post processing T applied on LP satisfies $\Pr(|f_{T(LP)}(x_1, ..., x_t) - \frac{C}{K_F}| \geq \frac{\epsilon}{K_F}) < \frac{\delta}{K_F}$ for any i.i.d. samples $x_1, ..., x_t$ from p_g. Then, we have $\Pr(|F_{T(LP)}(x_1, x_2, ..., x_n) - C| < \epsilon) > 1 - \delta$.

Proof. From the equation $F_{T(LP)}(x_1, ..., x_n) = \sum_{k=1}^{K_F} f_{T(LP)}(x_{k_1}, ..., x_{k_t})$, we see that any $(x_1, x_2, .., x_n)$ in the set $\cap_{k=1}^{K_F}\{|f_{T(LP)}(x_{k_1}, ..., x_{k_t}) - \frac{C}{K_F}| < \frac{\epsilon}{K_F}\}$ is also in the set $\{|F_{T(LP)}(x_1, x_2, ..., x_n) - C| < \epsilon\}$. As a result, we have

$$\Pr(|F_{T(LP)}(x_1, x_2, ..., x_n) - C| < \epsilon) \geq \Pr\left(\bigcap_{k=1}^{K_F}\left\{\left|f_{T(LP)}(x_{k_1}, ..., x_{k_t}) - \frac{C}{K_F}\right| < \frac{\epsilon}{K_F}\right\}\right). \tag{2.3}$$

Then, the union bound gives

$$\begin{aligned}&\Pr\left(\bigcap_{k=1}^{K_F}\left\{\left|f_{T(LP)}(x_{k_1}, ..., x_{k_t}) - \frac{C}{K_F}\right| < \frac{\epsilon}{K_F}\right\}\right)\\ &= 1 - \Pr\left(\bigcup_{k=1}^{K_F}\left\{\left|f_{T(LP)}(x_{k_1}, ..., x_{k_t}) - \frac{C}{K_F}\right| \geq \frac{\epsilon}{K_F}\right\}\right)\\ &\geq 1 - \sum_{k=1}^{K_F}\Pr\left(\left|f_{T(LP)}(x_{k_1}, ..., x_{k_t}) - \frac{C}{K_F}\right| \geq \frac{\epsilon}{K_F}\right).\end{aligned} \tag{2.4}$$

Therefore, combining Equation (2.3) and Equation (2.4) and the condition $\Pr(|f_{T(LP)}(x_1, ..., x_t) -$

$\frac{C}{K_F}| \geq \frac{\epsilon}{K_F}) < \frac{\delta}{K_F}$ for all i.i.d. samples, we have

$$\begin{aligned}
\Pr(|F_{T(LP)}(x_1, x_2, ..., x_n) - C| < \epsilon) &\geq 1 - \sum_{k=1}^{K_F} \Pr\left(\left|f_{T(LP)}(x_{k_1}, ..., x_{k_t}) - \frac{C}{K_F}\right| \geq \frac{\epsilon}{K_F}\right) \\
&> 1 - K_F \frac{\delta}{K_F} \\
&= 1 - \delta
\end{aligned} \tag{2.5}$$

□

2.5 Numerical Experiments

In this section, we perform comprehensive experiments to show that the proposed method generates graphs closely matching an input one in a wide range of properties. We also perform a downstream task: node classification, on the generated graphs and demonstrate that they can be used as surrogates of the input graph with similar classification performance.

We experiment with three commonly used data sets for graph-based learning: Cora-ML [78][1], CiteSeer [98][2], and Gene [95][3]. As is the usual practice, we experiment with the largest connected component of each graph. Their information is summarized in Table 2.1.

Table 2.1: Data sets (largest connected component).

Name	# Nodes	# Edges	# Classes	# Features
Cora-ML	2810	6783	7	1433
CiteSeer	2120	3679	6	3703
Gene	814	1436	2	N/A

2.5.1 Experiment Setting

Baseline models

We compare with a number of graph generative models, either deep learning or non-deep learning based. For non-deep learning based models, we consider the configuration model (CONF) [81], the Chung-Lu model [2], the degree-correlated stochastic block model (DC-SBM) [57], and the exponential random graph model (ERGM) [47]. For deep learning based models, we consider the variational graph autoencoder (VGAE) [62] and NetGAN [7]. We use either off-the-shelf software packages

[1]`https://github.com/danielzuegner/netgan`
[2]`https://linqs.soe.ucsc.edu/data`
[3]`http://networkrepository.com/gene.php`

or the codes provided by the authors to run these models.

It is important to note that the deep learning models VGAE and NetGAN desire a high edge overlap by the nature of their training algorithms; and the input features are either recovered or fixed, in straight contrast to our method. Additionally, for CONF, the way it works is that one specifies a desired edge overlap, retains a portion of edges to satisfy the overlap, and randomly rewires the rest while preserving the node degrees. The random rewiring succeeds with a certain probability that increases as overlap increases. Hence, to ensure that CONF successfully returns a graph in a reasonable time limit, the overlap cannot be too small, also at odds with our purpose.

Graph properties

We use a number of graph properties to evaluate the closeness of the generated graph to the original graph. Nine of them are global properties: clustering coefficient, characteristic path length, triangle count, square count, size of the largest connected component, powerlaw exponent, wedge count, relative edge distribution entropy, and Gini coefficient. These properties are also used by Bojchevski et al. [7] for evaluation.

Additionally, we measure three local properties: local clustering coefficients, degree distribution, and local square clustering coefficients. They are all distributions over nodes and hence we compare the maximum mean discrepancy (MMD) of the distributions between the generated graph and the original graph.

Table 2.2 shows the detail of each graph property that we use to evaluate generative models.

To evaluate the difference in topology between the generated graphs and the original graph, for the baseline models, we calculate the overlap of 1s between the respective adjacency matrices. The reason is that baseline models do not modify the node set. On the other hand, for our model, we sample new nodes, which do not correspond to the original node set. Hence, to evaluate the overlap of 1s between the

Table 2.2: Graph properties.

Property	Description
Clustering coefficient	Number of closed triplets divided by number of all triplets.
Characteristic path length	The median of the means of the shortest path lengths connecting each node to all other nodes.
Triangle count	Number of triangles in the graph.
Square count	Number of squares in the graph.
LCC	Size of the largest connected component.
Power law exponent	Exponent of the power law distribution.
Wedge count	Number of wedges (i.e., 2-stars; two-hop paths).
Relative edge distribution entropy	Entropy of degree distribution.
Gini coefficient	Common measure for inequality in a distribution.
Local clustering coefficients	The coefficients form a distribution. We compute the MMD between the distribution of the original graph and that of the generated graph.
Degree distribution	The degrees form a distribution. We compute the MMD between the distribution of the original graph and that of the generated graph.
Local square clustering coefficients	The coefficients form a distribution. We compute the MMD between the distribution of the original graph and that of the generated graph.

respective adjacency matrices, a naive approach is to enumerate all $n!$ possible node permutations and find the largest overlap, which is prohibitively expensive. Hence, instead, we use GOT [89], an optimal transport framework for graph matching, to estimate a reasonable overlap.

Model configuration

A high quality link predictor is key for the improved HH algorithm to select the right neighbors. A typical problem for link prediction is that negative links (unconnected node pairs) dominate and they scale quadratically with the number of nodes. A common remedy is to perform negative sampling during training, so that the training examples do not explode in size and remain relatively balanced. However, model evaluation is generally not properly conducted, because not every negative link is tested. As more negative links are evaluated, false positive rate tends to increase.

Thus, with the slight modify of the GraphSAGE equation (See Equation (2.1)), we also enhance GraphSAGE training with a cycling approach (C cycles of T rounds). In each cycle, we start the first round by using the same number of positive and negative links. All negative links (hereby and subsequent) are randomly sampled. We run E_0 epochs for the first round. Afterward, we insert K unused negative links into the training set and continue training with E_1 epochs, as the second round. In all subsequent rounds we similarly insert further K unused negative links into the training set and train for E_1 epochs. With T rounds we complete one cycle. Afterward, we warm start the next cycle with the trained model but a new training set, which contains the same number of positive and random negative links. Such a warm-start cycling avoids exploding the number of negative links used for training.

Details of the choice of C, T, E_0, E_1, and K are given in Table 2.3. In this table, we also summarize the architectures and hyperparameters for the WGAN for all graphs using in our experiments.

Table 2.3: Architectures and hyperparameters.

Dataset	Architecture
CORA-ML	GraphSAGE training: $C = 1$, $T = 20$, $E_0 = 5000$, $E_1 = 5000$, $K = 2000$.
	GAN generator: $FC(16, 32) \to FC(32, 64) \to FC(64, 100) \to FC(100, 128)$.
	GAN discriminator: $FC(128, 100) \to FC(100, 64) \to FC(64, 32) \to FC(32, 1)$.
CITESEER	GraphSAGE training: $C = 2$, $T = 20$, $E_0 = 2500$, $E_1 = 4000$, $K = 1000$.
	GAN generator: $FC(16, 32) \to FC(32, 64) \to FC(64, 100) \to FC(100, 128)$.
	GAN discriminator: $FC(128, 100) \to FC(100, 64) \to FC(64, 32) \to FC(32, 1)$.
GENE	GraphSAGE training: $C = 1$, $T = 20$, $E_0 = 2500$, $E_1 = 4000$, $K = 1000$.
	GAN generator: $FC(16, 32) \to FC(32, 64) \to FC(64, 100) \to FC(100, 128 + 2)$.
	GAN discriminator: $FC(128 + 2, 100) \to FC(100, 64) \to FC(64, 32) \to FC(32, 1)$.

2.5.2 Comparison of Baseline Models and the Proposed Method

In this section, we compare the graphs generated by our improved HH algorithm with those by other generative models introduced in the preceding subsection. Table 2.4 summarizes the results on CORA-ML in numeric format. Results of CITESEER and GENE, can be found in Tables 2.5 and 2.6.

First, for most of the global properties considered, our model produces graphs closest to the original graph, with a few properties exactly matched. The exact matching (powerlaw exponent, wedege count, relative edge distribution entropy, and Gini coefficient) is not surprising in light of Theorem 2.4.1, because these properties are degree-based. Furthermore, for other non-degree based properties (clustering coefficients, characteristic path length, triangle count, square count, and largest connected component), a close match indicates that our model learns other aspects of the graph characteristics reasonably well. Note that CONF also matches the degree-based properties, by design. However, CONF results are difficult to obtain, because the random wiring hardly successfully produces a graph matching the degree sequence when the

Table 2.4: Properties of Cora-ML and the graphs generated by different models (averaged over five random repetitions). The numbers in the bracket are standard deviations. Boldfaced numbers are closest to those for the original graph.

	Cluster. coeff. $\times 10^{-3}$	Charcs. path length $\times 10^{+0}$	Triangle count $\times 10^{+3}$	Square count $\times 10^{+2}$	LCC $\times 10^{+3}$	Powerlaw exponent $\times 10^{+0}$	Edge Overlap
Cora-ML	2.73	5.61	2.81	5.17	2.81	1.86	
CONF	0.475(0.023)	4.47(0.02)	0.490(0.024)	0.094(0.049)	**2.79(0.00)**	**1.86(0.00)**	42.4%
DC-SBM	2.26(0.13)	4.59(0.02)	1.35(0.05)	**0.923(0.015)**	2.48(0.03)	5.35(0.06)	4.88%
Chung-Lu	0.548(0.0583)	4.08(0.01)	0.550(0.032)	0.324(0.106)	2.47(0.01)	1.79(0.01)	1.29%
ERGM	1.53(0.08)	4.86(0.01)	0.0586(0.01)	0.00(0.00)	**2.79(0.00)**	1.65(0.00)	4.43%
VGAE	6.30(0.24)	5.16(0.09)	13.4(0.3)	2320(120)	1.98(0.04)	1.82(0.00)	53.1%
NetGAN	3.75(0.94)	4.30(0.30)	12.5(0.5)	184(19)	1.96(0.07)	1.77(0.01)	49.9%
HH	5.25(0.08)	5.50(0.07)	5.42(0.09)	169(2)	2.58(0.01)	**1.86(0.00)**	2.64%
Improved HH	**2.97(0.20)**	**5.67(0.11)**	**3.06(0.20)**	22.2(5.6)	2.52(0.02)	**1.86(0.00)**	2.33%

	Wedge count $\times 10^{+5}$	Rel. edge distr. entr. $\times 10^{-1}$	Gini coefficient $\times 10^{-1}$	Local cluster. $\times 10^{-2}$	Degree distr. $\times 10^{-2}$	Local sq. cluster. $\times 10^{-3}$	
Cora-ML	1.02	9.41	4.82	0	0	0	
CONF	**1.02(0.00)**	**9.41(0.00)**	**4.82(0.00)**	4.76(0.04)	**0**	3.08(0.07)	
DC-SBM	0.923(0.015)	9.30(0.02)	5.35(0.06)	4.40(0.09)	1.55(0.08)	2.89(0.11)	
Chung-Lu	1.08(0.03)	9.26(0.01)	5.46(0.02)	5.53(0.07)	1.66(0.18)	3.29(0.03)	
ERGM	0.426(0.007)	9.84(0.00)	2.67(0.02)	6.09(0.03)	9.67(0.08)	3.38(0.00)	
VGAE	1.79(0.02)	8.73(0.00)	7.03(0.01)	4.44(0.27)	10.25(0.28)	9.65(0.69)	
NetGAN	2.08(0.18)	8.70(0.06)	7.04(0.15)	4.01(0.63)	13.13(1.77)	5.52(3.18)	
HH	**1.02(0.00)**	**9.41(0.00)**	**4.82(0.00)**	5.35(0.04)	**0**	3.15(0.02)	
Improved HH	**1.02(0.00)**	**9.41(0.00)**	**4.82(0.00)**	**3.91(0.10)**	**0**	**2.08(0.07)**	

prescribed overlap is small.

Second, for local properties (local clustering coefficients, degree distribution, and local square clustering coefficients), graphs from our model generally are closest to the original graphs, than are those from other models. For our model, the MMD of the degree distributions is zero according to Theorem 2.4.1. On the other hand, that the MMD of the distributions of the clustering coefficients is the smallest manifests that local characteristics of the input graph are well learned.

As elucidated earlier (see Theorem 2.4.2), a potential drawback of HH is that it tends to generate large cliques, which lead to an exploding number of small motifs that mismatch what real-life graphs entail. The improved HH, on the other hand, uses link probabilities to mitigate the tendency of forming large cliques. Table 2.4 verifies that improved HH indeed produces triangle (3-clique) counts and square (4-clique) counts much closer to those of the original graph. One also observes the exploding square count on CiteSeer and Gene by using the original HH algorithm (see Tables 2.5 and 2.6), corroborating the tendency of large cliques in large graphs, which will lead to poor node classification results as shown next.

Table 2.5: Properties of CiteSeer and the graphs generated by different models (averaged over five random repetitions). EO means edge overlap and numbers in the bracket are standard deviations. Boldfaced numbers are closest to those for the original graph.

	Cluster. coeff. $\times 10^{-2}$	Charcs. path length $\times 10^{+0}$	Triangle count $\times 10^{+3}$	Square count $\times 10^{+2}$	LCC $\times 10^{+3}$	Powerlaw exponent $\times 10^{+0}$	Edge Overlap
CiteSeer	1.30	9.33	1.08	2.49	2.12	2.07	
CONF	0.160(0.019)	5.39(0.01)	0.133(0.016)	0.02(0.02)	**2.09(0.01)**	**2.07(0.00)**	42.4%
DC-SBM	0.755(0.065)	5.58(0.09)	0.500(0.025)	0.794(0.134)	1.75(0.02)	1.95(0.01)	5.35%
Chung-Lu	0.0786(0.0088)	4.73(0.05)	0.076(0.005)	0.006(0.008)	1.80(0.01)	1.94(0.02)	0.86%
ERGM	0.18(0.036)	6.15(0.04)	0.00940(0.00196)	0.00(0.00)	2.04(0.00)	1.87(0.00)	8.30%
VGAE	2.92(0.14)	**8.66(0.87)**	6.94(0.14)	133(11)	0.90(0.129)	1.86(0.01)	45.8%
NetGAN	2.42(0.70)	6.75(0.63)	4.53(0.18)	45.8(5.0)	1.36(0.04)	1.81(0.01)	55.9%
HH	1.85(0.03)	6.55(0.03)	**1.12(0.02)**	18.1(0.55)	1.86(0.01)	**2.07(0.00)**	2.71%
Improved HH	**0.839(0.057)**	7.64(0.29)	0.700(0.048)	**2.10(0.72)**	1.67(0.02)	**2.07(0.00)**	2.56%

	Wedge count $\times 10^{+4}$	Rel. edge distr. entr. $\times 10^{-1}$	Gini coefficient $\times 10^{-1}$	Local cluster. $\times 10^{-2}$	Degree distr. $\times 10^{-2}$	Local sq. cluster. $\times 10^{-3}$	
CiteSeer	2.60	9.54	4.28	0	0	0	
CONF	**2.60(0.00)**	**9.54(0.00)**	**4.28(0.00)**	3.20(0.07)	**0**	7.26(0.26)	
DC-SBM	2.73(0.04)	9.34(0.01)	5.16(0.04)	**2.82(0.10)**	3.15(0.12)	6.72(0.21)	
Chung-Lu	3.00(0.12)	9.33(0.01)	5.18(0.04)	3.65(0.03)	3.15(0.19)	7.54(0.07)	
ERGM	1.31(0.02)	9.79(0.00)	2.98(0.02)	3.83(0.02)	5.90(0.25)	7.66(0.03)	
VGAE	5.38(0.08)	8.60(0.00)	7.15(0.01)	3.60(0.06)	19.41(0.62)	33.59(2.24)	
NetGAN	4.21(0.30)	8.96(0.03)	6.40(0.10)	5.10(0.28)	16.26(0.58)	15.47(2.15)	
HH	**2.60(0.00)**	**9.54(0.00)**	**4.28(0.00)**	3.47(0.02)	**0**	7.45(0.01)	
Improved HH	**2.60(0.00)**	**9.54(0.00)**	**4.28(0.00)**	3.20(0.04)	**0**	**4.71(0.19)**	

Table 2.6: Properties of Gene and the graphs generated by different models (averaged over five random repetitions). EO means edge overlap and numbers in the bracket are standard deviations. Boldfaced numbers are closest to those for the original graph.

	Cluster. coeff. $\times 10^{-2}$	Charcs. path length $\times 10^{+0}$	Triangle count $\times 10^{+2}$	Square count $\times 10^{+2}$	LCC $\times 10^{+2}$	Powerlaw exponent $\times 10^{+0}$	Edge Overlap
Gene	10.41	7.01	8.09	9.68	8.14	2.05	
CONF	1.21(0.06)	5.01(0.05)	0.94(0.04)	0.11(0.04)	**7.98(0.06)**	**2.05(0.00)**	42.7%
DC-SBM	4.36(0.38)	5.01(0.06)	3.46(0.36)	3.09(0.78)	6.81(0.14)	1.92(0.03)	11.8%
Chung-Lu	0.382(0.058)	4.33(0.03)	0.42(0.07)	0.002(0.004)	6.90(0.06)	1.91(0.01)	1.52%
ERGM	0.374(0.146)	5.24(0.06)	0.08(0.03)	0.00(0.00)	7.92(0.02)	1.84(0.02)	0.60%
VGAE	13.8(0.05)	**6.99(0.19)**	17.2(0.40)	19.81(1.07)	5.44(0.22)	1.86(0.01)	64.9%
NetGAN	7.54(1.18)	4.50(0.31)	22.53(1.99)	28.78(5.95)	4.60(0.28)	1.72(0.03)	51.6%
HH	4.52(0.08)	5.80(0.06)	3.51(0.06)	3.51(0.07)	6.91(0.04)	**2.05(0.00)**	10.52%
Improved HH	**8.41(0.25)**	10.72(0.87)	**6.54(0.19)**	**4.18(0.73)**	6.32(0.25)	**2.05(0.00)**	11.67%

	Wedge count $\times 10^{+3}$	Rel. edge distr. entr. $\times 10^{-1}$	Gini coefficient $\times 10^{-1}$	Local cluster. $\times 10^{-2}$	Degree distr. $\times 10^{-2}$	Local sq. cluster. $\times 10^{-3}$	
Gene	7.79	9.53	4.26	0	0	0	
CONF	**7.79(0.00)**	**9.53(0.00)**	**4.26(0.00)**	5.39(0.19)	**0**	14.24(0.28)	
DC-SBM	8.20(0.33)	9.35(0.03)	4.95(0.01)	4.10(0.06)	2.71(0.55)	10.72(0.20)	
Chung-Lu	9.49(0.39)	9.30(0.01)	5.09(0.04)	6.12(0.14)	2.49(0.25)	14.54(0.25)	
ERGM	5.51(0.18)	9.78(0.01)	2.86(0.04)	6.41(0.06)	7.55(0.83)	14.79(0.12)	
VGAE	10.48(1.58)	9.09(0.01)	5.79(0.05)	2.27(0.13)	8.34(0.21)	14.47(1.68)	
NetGAN	15.40(1.20)	8.62(0.10)	6.91(0.21)	**1.62(0.27)**	23.5(3.21)	6.89(1.73)	
HH	**7.79(0.00)**	**9.53(0.00)**	**4.26(0.00)**	5.79(0.07)	**0**	14.5(0.1)	
Improved HH	**7.79(0.00)**	**9.53(0.00)**	**4.26(0.00)**	3.45(0.15)	**0**	**2.02(0.54)**	

2.5.3 Visualization of Generated Graphs

For qualitative evaluation, I visualize the original graph and two generated graphs: one by NetGAN and the other by our model. The Cora-ML graphs are given in Figure 2.4, the CiteSeer graphs are given in Figure 2.5 and Gene graphs are given in Figure 2.6. Owing to the objective, NetGAN generates a graph considerably overlapping with the original one, with the delta edges scattering like noise. On the other hand, our model generates a graph that minimally overlaps with the original one. Hence, the drawing shows a quite different look. We highlight, however, that despite the dissimilar appearance, the many global and local properties remain close as can be seen from Table 2.4, 2.5 and 2.6.

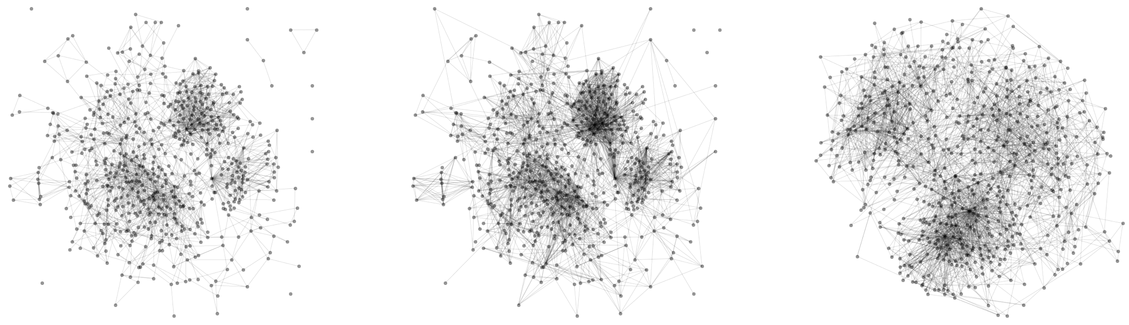


Figure 2.4: Cora-ML. Left: original graph; middle: generated by NetGAN; right: generated by improved HH. Drawn are the subgraphs induced by the first 600 nodes ordered by degrees.

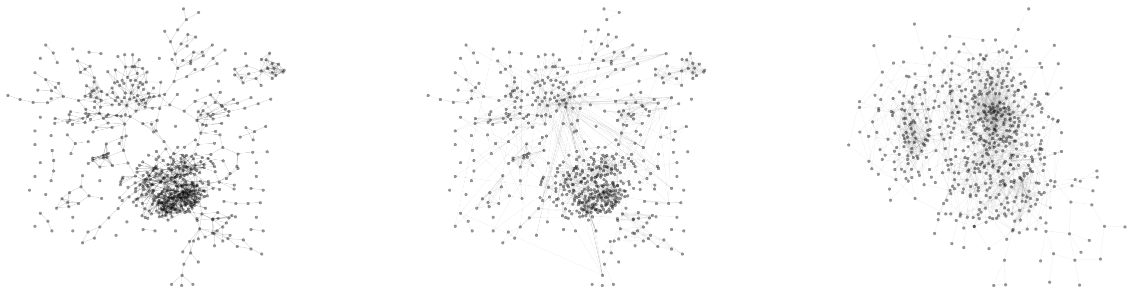


Figure 2.5: CiteSeer. Left: original graph; middle: generated by NetGAN; right: generated by improved HH. Drawn are the subgraphs induced by the first 600 nodes ordered by degrees.

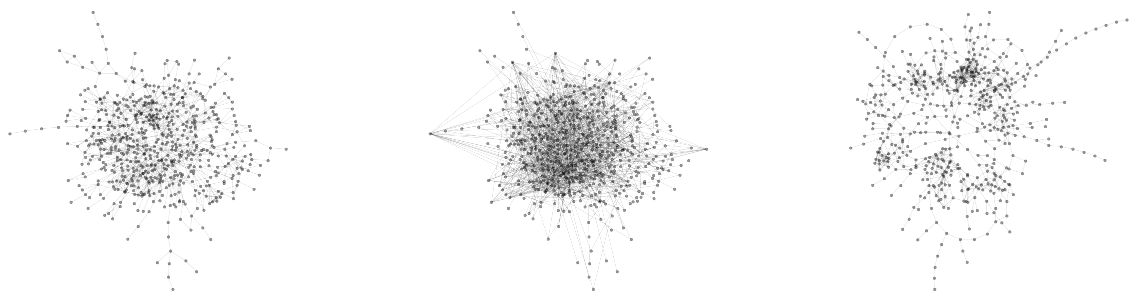


Figure 2.6: Gene. Left: original graph; middle: generated by NetGAN; right: generated by improved HH. Drawn are the subgraphs induced by the first 600 nodes ordered by degrees.

2.5.4 Link Prediction and GAN Training Results

In this section, I discuss about the training of the link predictor. The result of the cycling approach described in the preceding section is illustrated in Figure 2.7, Figure 2.8 and Figure 2.9. The metrics are AP and AUC scores evaluated on all node pairs. For Cora-ML, we use two cycles. One sees that the second cycle improves over the score at the end of the first cycle, although the warm start begins at a lower position. For CiteSeer and Gene, we use only one cycle, because the AUC score is already close to full at the end of the cycle. For both data sets, the purpose of incrementally inserting more negative edges in each round is fulfilled: scores progressively increase. Had we not inserted additional negative edges over time, the curves would plateau after the first round. Overall, the convergence history indicates that the enhanced GraphSAGE training strategy is effective.

We also show the training process of GAN. GAN results are known to be notoriously hard to robustly evaluate. Here, we show the MMD; see the middle panel of Figure 2.7, 2.8, 2.9. It decreases generally well, with small bumps throughout. The challenge is that it is hard to tell from the value whether training is sufficient. Therefore, we also investigate the empirical cdf of the pairwise data distance. As shown in the bottom of Figure 2.7, 2.8, 2.9, the cdf for the training data and that for the generated data are visually close. Such a result indicates that GAN has been trained reasonably well.

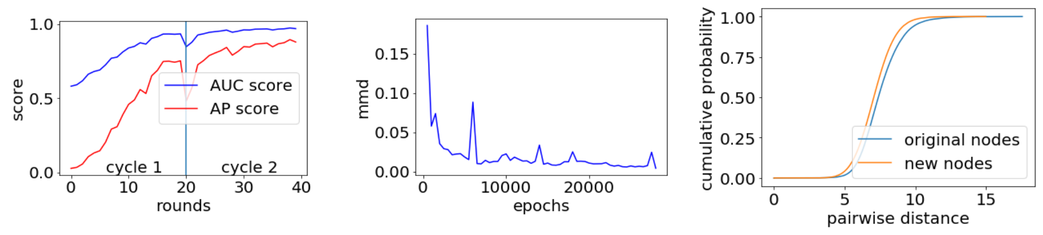


Figure 2.7: Results of link prediction and GAN for CORA-ML dataset. Left: GraphSAGE training progress (AP, AUC); Middle: GAN training progress (MMD); Right: GAN result (pairwise distance distribution).

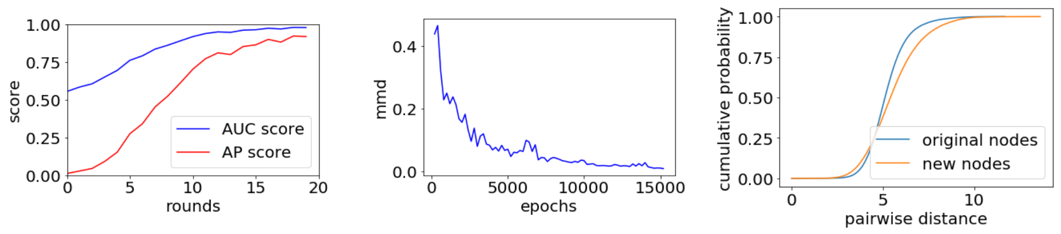


Figure 2.8: Results of link prediction and GAN for CITESEER dataset. Left: GraphSAGE training progress (AP, AUC); Middle: GAN training progress (MMD); Right: GAN result (pairwise distance distribution).

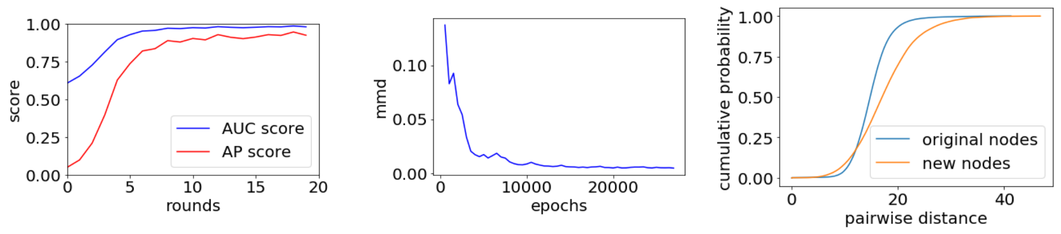


Figure 2.9: Results of link prediction and GAN for GENE dataset. Left: GraphSAGE training progress (AP, AUC); Middle: GAN training progress (MMD); Right: GAN result (pairwise distance distribution).

2.6 Downstream Task Example: Node Classification

2.6.1 Background

In practice, a generated graph is used as a surrogate of the original one in downstream tasks. For model developers, they desire that the model performs similarly on both graphs, so that both the graph owner and the developer are confident of the use of the model on the original graph. As a result, we use node classification task to show our generated graph achieve this goal.

2.6.2 Experiment Setting

We experiment with the node classification task and test with two commonly used model: GCN [63] and GraphSAGE [42]. The codes are downloaded from links given by the authors and we use the default hyperparameters without tuning.

We use the same three datasets:Cora-ML ,CiteSeer , and Gene. Cora-ML and CiteSeer come with node features while Gene not. For Gene, following convention we use the identity matrix as the feature matrix. For all compared methods, the node set is fixed and hence the node features are straightforwardly used when running GCN and GraphSAGE; whereas for our method, because nodes are new, we use their embedding generated by GAN as the input node features. To obtain node labels, we augment GAN to learn and generate them. Specifically, we concatenate the node embeddings with a one-hot vector of the class label as input/output of GAN. It is senseless to use a fixed train/test split, which is applicable to only the original node set. Hence, we conduct ten-fold cross validation to obtain classification performance.

Note that each generative model can generate an arbitrary number of graphs. In practice, one selects one or a small number of them to use. In Table 2.7, we select the best one based on cross validation performance after 20 random trials.

2.6.3 Results

Table 2.7 shows that the graphs generated by our method indeed satisfy the desire, producing competitively similar classification accuracies; whereas those generated by other methods fall short, with accuracies departing from the original graph.

Table 2.7: Node classification results (10-fold cross validation) on original/generated graphs. Top: GCN method; Bottom: GraphSAGE method. Boldfaced numbers are closest to those for the original graph.

	Cora-ML	CiteSeer	Gene
Origin. graph	76.37(0.07)	84.11(0.07)	81.46(0.12)
CONF	35.61(0.12)	51.99(0.06)	64.37(0.69)
DC-SBM	44.29(0.18)	50.06(0.20)	64.25(1.82)
Chung-Lu	35.61(0.12)	36.10(0.08)	58.99(1.24)
ERGM	38.16(0.05)	40.08(0.04)	55.90(0.22)
VGAE	55.05(0.20)	72.44(0.28)	76.21(1.85)
NetGAN	49.34(0.22)	63.62(0.32)	66.96(5.77)
HH	40.75(0.34)	40.12(0.15)	89.31(1.71)
Improved HH	**74.15(0.13)**	**80.44(0.20)**	**80.84(0.17)**

	Cora-ML	CiteSeer	Gene
Origin. graph	79.10(0.06)	86.52(0.04)	85.14(0.16)
CONF	35.61(0.12)	51.99(0.06)	64.37(0.69)
DC-SBM	56.72(0.10)	60.63(0.05)	69.54(0.18)
Chung-Lu	49.07(0.11)	47.88(0.06)	62.02(0.08)
ERGM	52.59(0.05)	55.07(0.06)	59.41(0.31)
VGAE	50.75(0.18)	74.04(0.07)	76.42(0.32)
NetGAN	44.10(0.19)	67.97(0.04)	68.30(0.15)
HH	53.71(0.19)	53.66(0.13)	68.74(0.19)
Improved HH	**78.07(0.10)**	**83.30(0.16)**	**80.89(0.12)**

Chapter 3

AUTM Normalizing Flows

3.1 Background

Normalizing flows are a class of generative models that aim to find an invertible function f maps the distribution formed by the training data to a simple distribution, such as Gaussian distribution.

Once the mapping is found, generating new data points boils down to simply sampling from the base distribution and applying the forward transformation f to the samples.

This makes normalizing flows a popular choice in density estimation, variational inference, data generation, etc.

Let $Y \in \mathbb{R}^D$ be a random variable with a possibly complicated probability density function $p_Y(y)$ and $X \in \mathbb{R}^D$ be a random variable with a well-studied probability density function $p_X(x)$. Assume that there is an invertible (vector-valued) function $f : \mathbb{R}^D \to \mathbb{R}^D$ that transforms the "base" variable X to Y, i.e. $Y = f(X)$. Then according to the change of variables formula, the probability density functions p_Y and p_X satisfy

$$p_Y(y) = p_X(x)\,|\det J_{f^{-1}}(y)| = p_X(x)\,|\det J_f(x)|^{-1} \tag{3.1}$$

where $J_f(x)$ denotes the Jacobian of f evaluated at x.

People usually calculate the log of the density functions. As a result, Equation (3.1) will become

$$\begin{aligned} \log p_Y(y) &= \log(p_X(x)) + \log(|\det J_{f^{-1}}(y)|) \\ &= \log(p_X(x)) + \log(|\det J_f(x)|^{-1}) \\ &= \log(p_X(x)) - \log(|\det J_f(x)|) \end{aligned} \tag{3.2}$$

In order to design practical normalizing flow model, the model should satisfy the following three principles [65]:

- invertible
- sufficiently expressive to map the target distribution to base distribution
- computationally efficient for computing f, f^{-1} and determinant of Jacobian

A popular architectural design to address those points is to employ an invertible triangular transformation, whose Jacobian is triangular and inversion can be computed in an entrywise fashion. Two representative triangular normalizing flows are autoregressive flows and coupling flows. Besides, another widely used type of normalizing flow is called continuous flows, in which the calculation of the Jacobian is different from the calculation in autoregressive flows and in coupling flows.

We use $x_{1:k}$ to denote the vector $(x_1, \ldots, x_k)$ in the following sections.

3.1.1 Coupling Flows

Coupling flows [21] partition the input vector $x = (x_1, \ldots, x_D)$ into two parts $x_{1:d}$ and $x_{d+1:D}$ and then apply the following transformation:

$$\begin{aligned} y_{1:d} &= x_{1:d} \\ y_{d+1:D} &= q(x_{d+1:D}; \theta(x_{1:d})) \end{aligned} \tag{3.3}$$

where the parameter $\theta(x_{1:d})$ is an arbitrary function of $x_{1:d}$ and the scalar *coupling function* q is applied entrywisely, i.e., $q(x_{d+1:D}) = (q(x_{d+1}), \ldots, q(x_D))$.

The transformation in Equation (3.3) from x to y is called a *coupling layer*. In normalizing flows, multiple coupling layers are composed to obtain a more complex transformation with the role of the two mappings in Equation (3.3) swapped in alternating layers. The Jacobian of Equation (3.3) is a lower triangular matrix with a 2-by-2 block structure corresponding to the partition of x. In [21], the coupling function q is chosen as an affine function: $q(z) = z \cdot \exp(s(x_{1:d})) + t(x_{1:d})$ where s and t are neural networks, and the resulting flow is termed *affine coupling flow*.

3.1.2 Autoregressive Flows

Autoregressive flows [61] choose $f : \mathbb{R}^D \to \mathbb{R}^D$ to be a mapping with autoregressive structure:

$$\begin{aligned} f(x; \theta) = (&q_1(x_1; \theta_1), \\ &q_2(x_2; \theta_2(x_1)), \\ &\ldots, \\ &q_k(x_k; \theta_k(x_{1:k-1})), \\ &\ldots, \\ &q_D(x_D; \theta_D(x_{1:D-1}))) \end{aligned} \tag{3.4}$$

where $x = (x_1, \ldots, x_D)$ and each q_k is a bijection mapping.

The Jacobian of f is a lower triangular matrix. To guarantee the invertibility of f, q_k is chosen to be a monotone function of x_k. Since q_k is usually a nonlinear neural network, an analytic inverse is not available and the inverse is computed by root-finding algorithms.

3.1.3 Continuous Flows

Continuous normalizing flows [11] let the mapping $f : \mathbb{R}^D \to \mathbb{R}^D$ has the form of ordinary differential equation: $\frac{df}{dt} = h(f(t), t)$. The input of continuous normalizing flow layers is $f(0)$ and the output is $f(1)$.

Instead of calculating the determinant of the Jacobian, in continuous flows, people calculate the change in log probability directly. The change in log probability formula, which is:

$$\frac{\partial \log(p(f(t)))}{\partial t} = -tr(\frac{dh}{df(t)}) \tag{3.5}$$

3.1.4 Applications

There are many applications of normalizing flows, for example, image generation [60, 45], noise modelling [1], and reinforcement learning [77].

3.2 Problem

For the three types of normalizing flows, they have their own advantages and drawbacks.

The advantages of **coupling flows** are:

- computationally efficient for computing f, f^{-1}

- computationally efficient for computing the determinant of Jacobian, due to the triangular structure of the Jacobian
- fast training speed

But the drawbacks are:

- insufficiently expressive to map the target distribution to base distribution (compared with continuous flows)
- performing poorly in specific 2D problems, in my experiments

The advantages of **autoregressive flows** are:

- computationally efficient for computing f, f^{-1}
- computationally efficient for computing the determinant of Jacobian, due to the triangular structure of the Jacobian

The drawback is:

- the data MUST be autoregressive structure

The advantages of **continuous flows** are:

- sufficiently expressive to map the target distribution to base distribution (compared with coupling flows)
- performing the best among these three types of normalizing flows on the same tasks

But the drawbacks are:

- requiring ODE solvers to computing f, f^{-1}

- high cost of calculating f, f^{-1}
- high cost of calculating the change in log probability formula (3.5)
- slow training speed

As a result, researchers focus on designing normalizing flow models which are both computationally efficient and sufficiently expressive.

3.3 Related Work

Different architectures have been proposed to improve existing normalizing flow structures. To simplify the Jacobian computation, most flows employ monotone triangular mappings so that the Jacobian is triangular. Examples of monotone mappings used in those flows include

Special function classes

- Affine functions used in NICE [20], RealNVP [21], GLOW [60].
- Rational functions used in Latent normalizing flow [117].
- Logistic mixture used in Flow++ [45].
- Splines used in Cubic-spline flows [24], Neural Spline Flows [25].

Neural networks

- Neural autoregressive flows [48] replaced the affine functions used in previous autoregressive normalizing flow models with a neural network.
- Block Neural Autoregressive Flow [16] used a block matrix to improve Neural Autoregressive Flow.

Integral of positive functions

- SOS flow [51] used a sum of square of several functions.
- Unconstrained Monotonic Neural Networks(UMNN) [109] defined an integral of specific functions to form the invertible transform.

To ensure monotonicity, methods using "special function classes" and neural networks have to impose constraints on model parameters, which often impede the expressive power of the transformation as well as the efficiency of training. Integral-based methods rely on the simple fact that the function is always increasing as long as the integrand g is globally positive. For example, g is modeled as positive polynomials in sum-of-squares (SOS) polynomial flow by [51] and as positive neural networks in unconstrained monotonic neural networks (UMNN) by [109]. Due to the flexibility offered by an integral form, those methods allow unrestricted model parameters as compared to other methods. It was shown in [109] that the method requires fewer parameters than straightforward neural network-based methods in [48, 16] and can scale to high dimensional datasets. However, those integral-based monotone mappings do not possess an explicit inverse formula and one has to find root-finding algorithms to compute the inverse transformation. This leads to increased computational cost because in each iteration of root-finding, one has to compute an integral of a complicated function. As discussed in [109], a judicious choice of quadrature rule is needed.

Also, there are some studies trying to improve the continuous flow. Neural ODE [11] and Free-form Jacobian of Reversible Dynamics (FFJORD) [35] pioneered the way of modeling the transformation as a dynamical system. The inverse can be easily computed by reversing the dynamics in time, but Jacobian determinant is hard to compute and the use of neural network to model the dynamics often leads to high computational cost. OT-flow [83] phrases the continuous flows as optimal transport (OT) problem by adding a transport cost, which will reduce the training time of continuous flows. CP-flow [49] tries a combination of optimal transport and convex optimization to make the training process more efficienct.

3.4 Atomic Unrestricted Time Machine (AUTM) flows

Lots of efforts have been made in recent years to construct a coupling function $q(x)$ which is strictly monotone (thus invertible) as well as expressive enough. As shown in Table 3.1, sophisticated machinery is used to improve the expressive power and ensure the monotonicity (invertibility) of the function q, which usually requires restricting the form of q or the model parameters, for example, Latent normalizing flows [117], Flow++ [45], Cubic-spline flows [24], Neural Spline Flows [25]. Moreover, since q is a complicated nonlinear function, an analytic format of q^{-1} is generally not available and thus numerical root-finding algorithms are often used to compute the inverse transformation. It is natural to find a family of universal monotone functions with analytic inverses and unrestricted model parameters or representations.

Hence, we propose the AUTM flows: Atomic Unrestricted Time Machine (AUTM) flows.

3.4.1 AUTM Flow Layer

We propose a new approach to construct a monotone $q(x)$ based on integration with respect to a free latent variable. The introduction of the latent variable enables the use of unconstrained transformations and renders exceptional flexibility for manipulating the transformation and its inverse. The resulting coupling flows and autoregressive flows can be inverted easily using the inverse formula and the Jacobian is triangular.

Table 3.1: Comparison of different normalizing flow architectures. `E.I.` stands for explicit inverse, `U.P.` for unrestricted parameters, `U.F.` for unrestricted function representations.

Method	monotone map	`E.I`	`U.P.`	`U.F.`
Real NVP [21]	affine function	yes	yes	no
Glow [60]	affine function	yes	yes	no
Flow++ [45]	$\alpha\sigma^{-1}\left(\sum_{i=1}^{r} c_i\sigma\left(\frac{x-a_i}{b_i}\right)\right)+\beta$	no	no	no
NSF [25]	rational-quadratic spline	yes	no	no
SOS [51]	$\int_0^x \sum_{i=1}^{L} p_i(x)^2 dx + c$	no	yes	no
UMNN [109]	$\int_0^x f(x)dx + \beta \;\; (f>0)$	no	yes	no
AUTM (new)	$x + \int_0^1 g(v(t),t)dt$	yes	yes	yes

We define $q : \mathbb{R} \to \mathbb{R}$ through a latent function $v(t)$ by

$$q : x \to y = v(1), \quad v(t) = x + \int_0^t g(v(t),t)dt \tag{3.6}$$

where $g(v,t)$ is uniformly Lipschitz continuous in v and continuous in t $(0 \le t \le 1)$. Equivalently, $v(t)$ satisfies $v'(t) = g(v(t),t)$ and $v(0) = x$. So the transformation from x to y can be viewed as an evolution of the latent dynamic $v(t)$. Note that the integral in (3.6) is with respect to t instead of x and the integrand does *not* have to be positive. Moreover, we can easily find the inverse transform as

$$q^{-1} : y \to x = v(0),\; v(t) = y + \int_1^t g(v(t),t)dt \tag{3.7}$$

An explicit inverse formula brings significant computational speedups as compared to existing integral-based methods that rely on numerical root-finding algorithms. Compared with other coupling functions/transformers, there is no assumption on g other than Lipschitz continuity. We will show later in Section 3.4.2 that the transformation in (3.6) is strictly increasing and is general enough to approximate *any* given continuously increasing map.

We term the mapping q in (3.6) as an "Atomic Unrestricted Time Machine (AUTM)". "Atomic" means that

- function q is always univariate and scalar-valued
- $g(v, t)$ can be as simple as an affine function in v and does NOT have to be a deep neural network to achieve good performance
- the computation of the transformation as well as Lipschitz constant is lightweight
- the model can be easily incorporated into existing normalizing flow architectures

In fact, we will see later in Section 3.4.2 that *any* monotonic normalizing flow is a limitation of AUTM flows.

"Unrestricted" means that there is no constraint on parameters or function forms in the model.

"Time Machine" refers to the fact that the model is automatically invertible and the computation of inverse is essentially a reverse of integral limits.

By the "atomic" property, AUTM can be easily incorporated into triangular flow architectures such as coupling flows and autoregressive flows.

AUTM coupling flows

Given a D dimensional input $x = (x_1, \ldots, x_D)$ and $d < D$, the AUTM coupling layer $f : \mathbb{R}^D \to \mathbb{R}^D$ is defined as follows.

$$\begin{aligned} y_{1:d} &= x_{1:d} \\ y_{d+1:D} &= q(x_{d+1:D}; \theta(x_{1:d})) \end{aligned} \tag{3.8}$$

where q is the AUTM map defined in Equation (3.6).

Let f_* denote the AUTM coupling layer with q applied to $x_{1:d}$ instead of $x_{d+1:D}$,

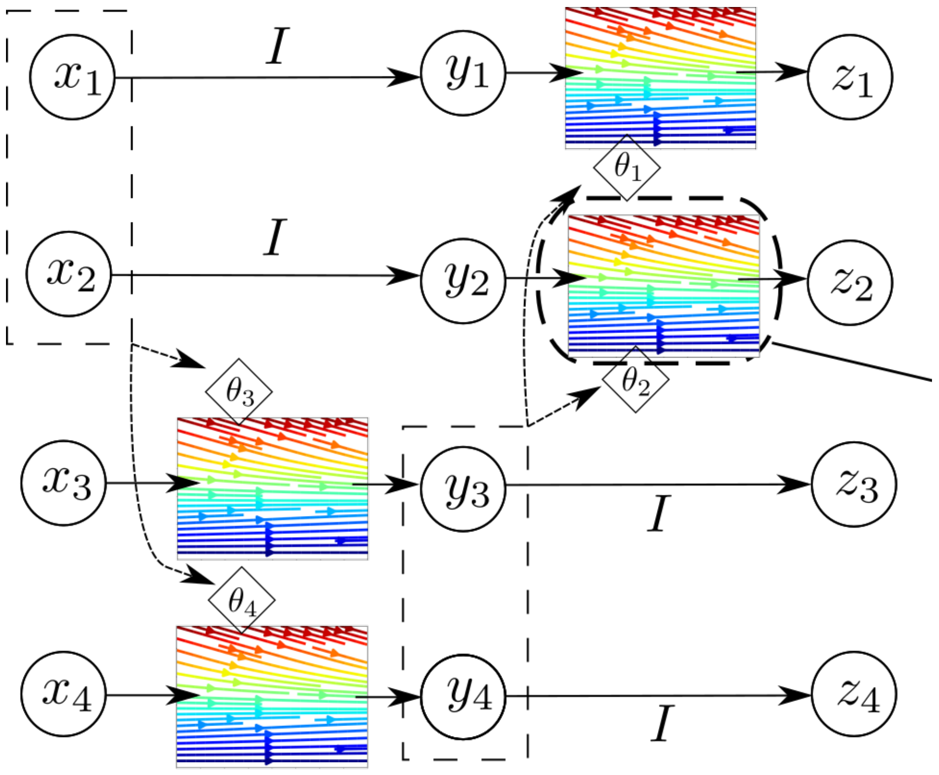


Figure 3.1: Structure of AUTM coupling flow.

$$\begin{aligned} y_{1:d} &= q(x_{1:d}; \theta(x_{d+1:D})) \\ y_{d+1:D} &= x_{d+1:D} \end{aligned} \tag{3.9}$$

The AUTM coupling flow is defined by stacking f and f_* in a multi-layer model. Figure 3.1 illustrates the process of AUTM coupling flow when the dimension D of the input data is 4 and set $d = 2$. Since the inverse q^{-1} is given in Equation (3.6), the inverse transformation f^{-1} or f_*^{-1} is readily available.

AUTM autoregressive flows

An autoregressive flow is composed of autoregressive mappings, similar to coupling layers in coupling flows.

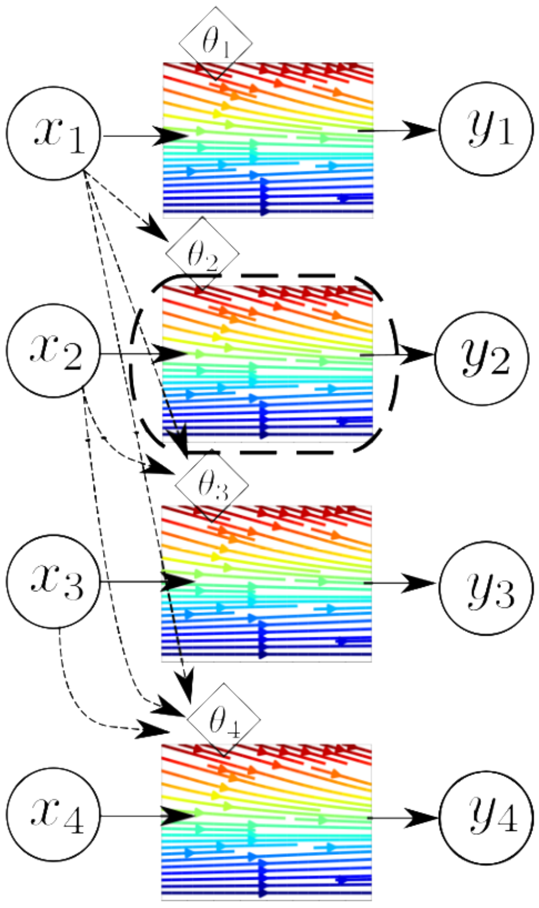


Figure 3.2: Structure of AUTM autoregressive flow.

The AUTM autoregressive mapping on $\mathbb{R}^D$ is defined in Equation (3.10)

$$f(x;\theta) = (q_1(x_1;\theta_1), \ldots, q_D(x_D;\theta_D(x_{1:D-1}))) \tag{3.10}$$

where each $q_k(x_k;\theta_k(x_{1:k-1}))$ is an AUTM mapping with unrestricted conditioner $\theta_k(x_{1:k-1})$.

Figure 3.2 illustrates the process of AUTM autoregressive flow when the dimension D of the input data is 4.

The inverse mapping f^{-1} can be computed rapidly by first computing q_1^{-1} (which gives x_1) and then $q_2^{-1}, q_3^{-1}, \ldots, q_D^{-1}$, where each q_k^{-1} is explicitly given by (3.7). The Jacobian of f in Equation (3.10) is lower triangular, as in traditional autoregressive flow.

Jacobian determinant and log-density The Jacobian of AUTM flow is lower triangular. It will be shown in Theorem 3.4.1 that the derivative of the mapping $q(x)$ is given by $q'(x) = \exp\left(\int_0^1 \frac{\partial g}{\partial v}(v(t), t)dt\right)$. Thus one can immediately derive the Jacobian determinant of AUTM flow. If coupling layer is used, the Jacobian determinant is

$$\exp\left(\int_0^1 \sum_{k=d+1}^{D} \frac{\partial g}{\partial v}(v_k(t), t; \theta(x_{1:d}))dt\right) \tag{3.11}$$

If autoregressive layer is used, the Jacobian determinant is

$$\exp\left(\int_0^1 \sum_{k=1}^{D} \frac{\partial g}{\partial v}(v_k(t), t; \theta(x_{1:k-1}))dt\right). \tag{3.12}$$

From the above formulas and Equation (3.2), the change of log-density of an AUTM flow follows immediately.

If coupling layer is used, the change of log-density formula will be

$$\log p_Y(y) = \log p_X(x) - \int_0^1 \sum_{k=d+1}^{D} \frac{\partial g}{\partial v}(v_k(t), t; \theta(x_{1:d}))dt. \tag{3.13}$$

If autoregressive layer is used, the change of log-density formula will be

$$\log p_Y(y) = \log p_X(x) - \int_0^1 \sum_{k=1}^{D} \frac{\partial g}{\partial v}(v_k(t), t; \theta(x_{1:k-1}))dt. \tag{3.14}$$

3.4.2 Theoretical Analysis

In this section, we present several theoretical analysis of AUTM flows, including monotonicity and universality. The derivative of $q(x)$ in Equation (3.6) is explicitly available.

Theorem 3.4.1 (Derivative)**.** Let $q(x)$ be defined in Equation (3.6). Then $q'(x) =$

$\exp\left(\int_0^1 \frac{\partial g}{\partial v}(v(t),t)dt\right)$.

It is easy to see that $q' > 0$, so q is strictly increasing.

Proof. Let $v = v(t,x)$ as a function of t and x. In fact,

$$v(t,x) = x + \int_0^t g(v(t,x),t)dt.$$

Let $u(t) = \frac{\partial v}{\partial x}$. It follows from the formula of v that

$$u(t) = \frac{\partial v}{\partial x} = 1 + \int_0^t \frac{\partial g}{\partial v}\frac{\partial v}{\partial x}dt \tag{3.15}$$

Then we have that

$$\frac{du}{dt} = \frac{\partial g}{\partial v}\frac{\partial v}{\partial x} = \frac{\partial g}{\partial v}u \tag{3.16}$$

This implies

$$u(t) = C\exp\left(\int_0^t \frac{\partial g}{\partial v}dt\right) \tag{3.17}$$

Recall Equation ch3:eq:AUTM, we have $u(0) = 1$, as a result $C = 1$.

Since $q(x) = v(1,x)$, we now conclude that

$$q'(x) = \frac{\partial v}{\partial x}(1,x) = u(1) = \exp\left(\int_0^1 \frac{\partial g}{\partial v}dt\right) \tag{3.18}$$

□

Theorem 3.4.2 (Monotonicity)**.** The mapping $q(x)$ defined in Equation (3.6) is invertible and strictly increasing.

Proof. Obviously, monotonicity implies invertibility, so it suffices to show that q is strictly increasing. There are several ways to prove this. The simplest way is to use Theorem 3.4.1 to see that $q'(x) > 0$, so q must be increasing. Below we present a different proof without using the analytic expression of $q'(x)$.

Let $v_x(t)$ denote the function in Equation (3.6) with $v(0) = x$, where $g(v,t)$ is continuous on t and uniformly Lipschitz continuous on v. We need to show that for any $x < x'$, there holds $q(x) < q(x')$. We prove this by contradiction. Assume that there exist $x < x'$ such that $q(x) \geq q(x')$. There are two cases to consider: $q(x) = q(x')$ and $q(x) > q(x')$.

Case 1: $q(x) = q(x') = C$ for some constant C.

In this case, $v_x(1) = v_{x'}(1) = C$. Define $w_a(t) = v_a(1-t)$ for any $a \in \mathbb{R}$ and $t \in [0,1]$. It is easy to see that $w_x(t)$, $w_{x'}(t)$ are both solutions to the ODE

$$\frac{dw}{dt} = -g(w(t), 1-t), \quad w(0) = C, \quad t \in [0,1]. \tag{3.19}$$

Note that $w_x(t)$ and $w_{x'}(t)$ are two different solutions of Equation (3.19) because $w_x(1) = x < x' = w_{x'}(1)$. This contradicts the uniqueness of solution to the ODE (which is well-posed since g is Lipschitz) and we conclude that the assumption $q(x) = q(x')$ can NOT hold.

Case 2: $q(x) > q(x')$.

In this case, we have $v_x(1) > v_{x'}(1)$ and $v_x(0) < v_{x'}(0)$. Applying intermediate value theorem to $v_x(t) - v_{x'}(t)$ yields that there exists $\tau \in (0,1)$ such that

$$v_x(\tau) = v_{x'}(\tau) = C$$

for some constant C.

Similar to Case 1, if we define $w_a(t) = v_a(\tau - t)$ for $t \in [0, \tau]$, then we can deduce that the ODE

$$\frac{dw}{dt} = -g(w(t), 1-t), \quad w(0) = C \tag{3.20}$$

has two different solutions $w_x(t)$ and $w_{x'}(t)$ as $w_x(\tau) = x < x' = w_{x'}(\tau)$, which contradicts the well-posedness of Equation (3.20).

We can now conclude that the inequality $q(x) \geq q(x')$ can NOT hold. Consequently, $q(x)$ must be strictly increasing and the proof is complete. $\square$

The expressive power of AUTM is summarized in the following three theorems, which state that one can approximate ANY monotone continuous transformation with a family of AUTM layers.

Theorem 3.4.3 (AUTM as a universal monotone mapping)**.** Let $\mathcal{C}$ be the space of continuous functions on $\mathbb{R}$ with compact-open topology and let $\mathcal{M} \subset \mathcal{C}$ be the cone of (strictly) increasing continuous functions. Then the set of AUTM bijections

$$\mathcal{Q} = \{q(x) \text{ in } \textit{Equation } (3.6) : v(0) = x \in \mathbb{R}\}$$

is dense in $\mathcal{M}$.

Proof. Since the set of increasing Lipschitz continuous functions is dense in $\mathcal{M}$, it suffices to consider Lipschitz functions in $\mathcal{M}$.

We need to show that, given an arbitrary increasing Lipschitz continuous function $\phi(x)$, there exists a family of AUTM bijections $\{q_s(x)\}_{s>0} \subset \mathcal{Q}$ that converge compactly to $\phi(x)$ as $s \to 0$, i.e. $q_s|_K \to \phi|_K$ uniformly on any compact set $K \subset \mathbb{R}$ as $s \to 0$.

We construct q_s as follows. First we define

$$g_*(v) = \phi(v) - v \quad \text{and} \quad g_s(v,t) = C_s e^{-\frac{t^2}{s}} g_*(v) \quad (s > 0)$$

where $C_s = \left(\int_0^1 e^{-\frac{t^2}{s}} dt\right)^{-1}$ is a normalizing constant. Since g_* is Lipschitz continuous, we see that $g_s(v,t)$ is uniformly Lipschitz continuous in v and continuous in t. Then we define

$$q_s(x) = x + \int_0^1 C_s e^{-\frac{t^2}{s}} [\phi(v_s(t)) - v_s(t)] dt \tag{3.21}$$

which has the form

$$q_s(x) = x + \int_0^1 g_s(v_s(t), t) dt \quad \text{with} \quad v_s(t) := x + \int_0^t g_s(v_s(t), t) dt \tag{3.22}$$

Note that $v_s(0) = x$.

We prove next that q_s converges compactly to ϕ as $s \to 0$. That is, for any compact set $K \subset \mathbb{R}$,

$$\lim_{s\to 0} \max_{x\in K} |q_s(x) - \phi(x)| = 0$$

Notice that

$$\phi(x) = x + g_*(x) = x + g_*(v_s(0))$$

We can deduce that

$$\begin{aligned} |q_s(x) - \phi(x)| &= \left| \int_0^1 g_s(v_s(t), t) dt - g_*(v_s(0)) \right| \\ &= \left| \int_0^1 C_s e^{-\frac{t^2}{s}} g_*(v_s(t)) dt - g_*(v_s(0)) \right| \\ &= \left| \int_0^1 C_s e^{-\frac{t^2}{s}} [g_*(v_s(t)) - g_*(v_s(0))] dt \right| \le I_{\rho,s} + J_{\rho,s} \end{aligned} \tag{3.23}$$

where $\rho \in (0,1)$ and

$$I_{\rho,s} := \left| \int_0^\rho C_s e^{-\frac{t^2}{s}} [g_*(v_s(t)) - g_*(v_s(0))] dt \right|, \ J_{\rho,s} := \left| \int_\rho^1 C_s e^{-\frac{t^2}{s}} [g_*(v_s(t)) - g_*(v_s(0))] dt \right|.$$

For an arbitrary $\epsilon > 0$, we choose $\rho \in (0,1)$ such that $|g_*(v_s(t)) - g_*(v_s(0))| \le \epsilon$

for all $v_s(0) = x \in K$ and $t \in [0, \rho]$. It then follows that

$$I_{\rho,s} \leq \epsilon \int_0^{\rho} C_s e^{-\frac{t^2}{s}} dt \leq \epsilon \int_0^1 C_s e^{-\frac{t^2}{s}} dt = \epsilon. \tag{3.24}$$

Next we estimate $J_{\rho,s}$. Since $K \times [0, 1]$ is compact, $|g_*(v_s(t))|$ is uniformly bounded by a constant M as $s \to 0$. Thus we deduce that

$$\lim_{s\to 0} J_{\rho,s} \leq 2M \lim_{s\to 0} \int_{\rho}^1 C_s e^{-\frac{t^2}{s}} dt = 0. \tag{3.25}$$

Combine Equations (3.23), (3.24) and (3.25), we see that

$$\lim_{s\to 0} \max_{x\in K} |q_s(x) - \phi(x)| \leq \epsilon.$$

Since $\epsilon > 0$ is arbitrary, we conclude that

$$\lim_{s\to 0} \max_{x\in K} |q_s(x) - \phi(x)| = 0,$$

which implies that $\mathcal{Q}$ is dense in $\mathcal{M}$. □

Theorem 3.4.4 (AUTM as a universal flow)**.** For any coupling or autoregressive flow $F = F_1 \circ F_2 \circ \cdots \circ F_p$ from $\mathbb{R}^D$ to $\mathbb{R}^D$, where each F_k is a triangular monotone transformation, there exists a family of AUTM flows $\{T_s\}_{s>0} = \{T_{s,1} \circ T_{s,2} \circ \cdots \circ T_{s,p}\}_{s>0}$ such that T_s converges to F pointwisely and compactly as $s \to 0$.

Proof. This is a result of Theorem 3.4.3. Since compact convergence implies pointwise convergence, it suffices to show the compact convergence. According to Theorem 3.4.3, for each F_k (where each entry is a monotone continuous function), we can construct a family of triangular AUTM transformations $T_{s,k}$ (parametrized by $s > 0$) that converge compactly to F_k as $s \to 0$. Then it follows immediately that $T_s := T_{s,1} \circ T_{s,2} \circ \cdots \circ T_{s,p}$ converges compactly to $F = F_1 \circ F_2 \circ \cdots \circ F_p$ as $s \to 0$. □

Theorem 3.4.3 and Theorem 3.4.4 imply that all coupling flows and autoregressive flows can be approximated arbitrarily well by AUTM flows. In the following, we present an explicit construction of such a family of AUTM flows that converge to an arbitrarily given monotonic normalizing flow. The convergence result provides a link between the proposed AUTM flows and existing monotonic normalizing flows and illustrates the representation power of AUTM. Notice that every AUTM flow layer has explicit inverse, so the universality result in this section shows that we can approximate any monotonic flow by a flow with explicit inverse.

Universal AUTM flows

Let $\phi(x)$ be an arbitrary increasing continuous function on $\mathbb{R}$. Define a family of AUTM bijections parametrized by $s > 0$ as Equation (3.21):

$$q_s(x) = x + \int_0^1 C_s e^{-\frac{t^2}{s}}[\phi(v_s(t)) - v_s(t)]dt, \tag{3.26}$$

where $v_s(t) = x + \int_0^t C_s e^{-\frac{t^2}{s}}[\phi(v_s(t)) - v_s(t)]dt$ and $C_s = \left(\int_0^1 e^{-\frac{t^2}{s}}dt\right)^{-1}$ is a normalization constant. Then it can be shown that (see proof of Theorem 3.4.3): $q_s|_K$ converges to $\phi|_K$ uniformly for any compact set $K \subset \mathbb{R}$. Based on q_s, one can construct a family of AUTM coupling flows or autoregressive flows that converge to the given flow based on ϕ.

In fact, the family of AUTM flows in Equation (3.26) is just one particular family of universal AUTM flows. The function $C_s e^{-\frac{t^2}{s}}$ in Equation (3.26) can be replaced with a much larger class of functions $\kappa_s(t)$ and the resulting AUTM flows are still universal. This is formalized in the theorem below regarding universal AUTM flows that generalize Equation (3.26).

Theorem 3.4.5. For $s > 0$, let $\kappa_s \in C([0,1])$ be a positive function such that $\int_0^1 \kappa_s(t)dt = 1$ and that for any $\rho \in (0,1)$, $\lim_{s\to 0}\int_\rho^1 \kappa_s(t)dt = 0$. Given any increasing continuous function $\phi(x)$, we define q_s $(s > 0)$ as follows $q_s(x) = x + \int_0^1 \kappa_s(t)[\phi(v_s(t)) - v_s(t)]dt$, with $v_s(t) = x + \int_0^t \kappa_s(t)[\phi(v_s(t)) - v_s(t)]dt$. Then as $s \to 0$, $q_s|_K$ converges to $\phi|_K$ uniformly for any compact set $K \subset \mathbb{R}$.

Proof. The proof follows the same argument as the proof of Theorem 3.4.3, where the only difference is that the function $C_s e^{-\frac{t^2}{s}}$ in the proof of Theorem 3.4.3 is replaced with a general function $\kappa_s(t)$. In this general case, the argument in the proof of Theorem 3 still holds because of the two properties of $\kappa_s(t)$:

- $\int_0^1 \kappa_s(t)dt = 1$
- for any $\rho \in (0, 1)$, $\lim\limits_{s \to 0} \int_\rho^1 \kappa_s(t)dt = 0$

where the normalization property (i) is used in (3.23) and (3.24), while the limit property (ii) is used in (3.25). Therefore, we conclude that $q_s|_K \to \phi|_K$ uniformly as $s \to 0$. □

3.5 Numerical Experiments

In this section, we present the results of some experiments to evaluate AUTM flows. In Section 3.5.1, we perform density estimation on five tabular datasets and compare with other methods. In Section 3.5.2, we train our model on the CIFAR10 and ImageNet32 datasets for image generation.

For image datasets, we model $g(v, t)$ in Equation (3.6) as a quadratic polynomial in v. For density estimation, we consider three different choices of $g(v, t)$ in Equation (3.6):

- $g(v, t) = av + b + cv^2$
- $g(v, t) = av + b + cv^3$
- $g(v, t) = av + b + c\sigma(v)$

where σ denotes the sigmoid function:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{3.27}$$

Note that this is different from many existing methods that rely on deep neural networks to model the core function in the model, such as UMNN [109] for the positive integrand, NAF [48] and BNAF [16] for the autoregressive mapping, neural ODEs [11] and FFJORD [35] for the entire dynamical system. We show in the following that, compared to the state-of-the-art models, our proposed AUTM model achieves excellent performance with simple choices of g. More importantly, for high-dimensional image datasets like ImageNet32, AUTM model requires less model parameters compared to other models.

3.5.1 Density Estimation

Data sets and baselines

We first evaluate our method on four datasets from the UCI machine-learning repository [19]: POWER, GAS. HEPMASS, MINIBOONE, and also the BSDS300 dataset, which are all preprocessed by [86]. We compare our method to several existing normalizing flow models, including Real NVP [21], Glow [60], RQ-NSF [25]), CP-FLOW [49], FFJORD [35], UMNN [109] and autoregressive models such as MAF [86], MADE [30] and BNAF [16].

Model configuration and training

We use 10 (or 5) masked AUTM layers and set the hidden dimensions 40 times (or 10 times) the dimension of the input. We apply a random permutation of the elements of each output vector, as the masked linear coupling layer, so that a different set of elements is considered at each layer, which is a widely used technique [16], [21], [86].

We use Adam as the optimizer and select hyperparameters after an extensive grid search.

The hyperparameters used are shown in Table 3.2. Hyperparameters are obtained after extensive grid search. For the number of layers, we tried 5,10,20. For the hidden layer dimensions, we tried $10d, 20d, 40d$, where d is the dimension of the vector in the dataset. We trained our model by using Adam. We stop the training process when there is no improvement on validation set in several epochs.

Table 3.2: Hyperparameters for Power, GAS, Hepmass, Miniboone, BSDS300 datasets, d is the dimension of the vector in the dataset.

	POWER	GAS	Hepmass	Miniboone	BSDS300
layers	10	10	10	5	10
hidden layer dimensions	40d	40d	40d	10d	40d
epochs	450	1000	500	1000	1000
batch size	256	256	256	256	128
optimizer	adam	adam	adam	adam	adam
learning rate	0.01	0.01	0.01	0.01	0.01
lr decay rate	0.5	0.5	0.5	0.5	0.5

Results

We report average negative log-likelihood metric, which is a widely used metric in this task, on the test sets in Table 3.3.

We test three different $g(v,t)$ functions in our AUTM flows: $g_1(v,t) = av+b+cv^2$, $g_2(v,t) = av + b + cv^3$, $g_3(v,t) = av + b + c\sigma(v)$.

It can be observed that AUTM consistently outperforms Real NVP, Glow, MADE, MAF, CP-Flow. On MINIBONDE dataset, our models perform better than all other models except BNAF. On POWER, HEPMASS, BSDS300 dataset, one of our model performs best among all baseline models. On GAS datase, our results are competitive at either top 2 or top 3 spot with a tiny gap from the best.

Table 3.3: Average test negative log-likelihood (in nats) of tabular datasets (lower is better). Numbers in the bracket are standard deviations.Average/std are computed by 3 runs.

Model	POWER	GAS	HEPMASS	MINIBOONE	BSDS300
Real NVP [21]	-0.17(0.01)	-8.33(0.14)	18.71(0.02)	13.55(0.49)	-153.28(1.78)
Glow [60]	-0.17(0.01)	-8.15(0.40)	18.92(0.08)	11.35(0.07)	-155.07(0.03)
FFJORD [35]	-0.46(0.01)	-8.59(0.12)	14.92(0.08)	10.43(0.04)	-157.40(0.19)
UMNN [109]	-0.63(0.01)	-10.89(0.70)	13.99(0.21)	9.67(0.13)	-157.98(0.01)
MADE [30]	3.08(0.03)	-3.56(0.04)	20.98(0.02)	15.59(0.50)	-148.85(0.28)
MAF [86]	-0.24(0.01)	-10.08(0.02)	17.70(0.02)	11.75(0.44)	-155.69(0.28)
CP-Flow [49]	-0.52(0.01)	-10.36(0.03)	16.93(0.08)	10.58(0.07)	-154.99(0.08)
BNAF [16]	-0.61(0.01)	-12.06(0.09)	14.71(0.38)	8.95(0.07)	-157.36(0.03)
RQ-NSF (C) [25]	-0.64(0.01)	-13.09(0.02)	14.75(0.03)	9.67(0.47)	-157.54(0.28)
AUTM: $g_1(v,t)$	-0.63(0.03)	-12.24(0.04)	14.62(0.30)	9.16(0.18)	-157.45(0.05)
AUTM: $g_1(v,t)$	-0.64(0.01)	-12.37(0.06)	14.76(0.25)	9.33(0.10)	-157.54(0.10)
AUTM: $g_1(v,t)$	-0.61(0.02)	-12.03(0.06)	14.94(0.33)	9.29(0.20)	-157.28(0.14)

3.5.2 Experiments on Image Dataset

Data sets and baselines

We then evaluate our method on the CIFAR10 [67] and ImageNet32 [84] datasets. Unlike density estimation tasks, image datasets are large-scale and high-dimensional. As a result, there are only a limited number of normalizing flow models available for image tasks. We calculate bits per dim and compare with other normalizing flow models including Real NVP [21], Glow [60], Flow++ [45], NSF [25]. The results of bits per dim for each model are given in Table 3.4. We also show sampled images by using AUTM in Figure 3.3 and Figure 3.4.

Model configuration and training

In this experiment, most hyperparameters come from Deep Residual Learning [56]. We use 14 AUTM coupling layers with 8 residual blocks for each layer in our model.

Like GLOW [60], before each coupling layers, there is an actnorm layer and a conv 1×1 layer. Each residual block has three convolution layers with 128 channels. Our method is trained for 100 epochs with batch size 64. We trained our model by using Adamax with Polyak.

Results

From Table 3.4, we can find our method outperforms all baselines with the exception of Flow++ [45] on CIFAR10 dataset, which uses a variational dequantization technique. In addition, Table 3.4 shows AUTM is not sensitive to the size of the dataset when we transfer from CIFAR10 to ImageNet32. Comparing the number of parameters used for each method, we find that AUTM yields the best performance with much fewer parameters compared to other models for ImageNet32. In particular, the number of model parameters of Flow++ [45] increases dramatically as we move from CIFAR10 to ImageNet32 while coupling normalizing flow models like RealNVP [21] and Glow [60] only have a moderate increase in the number of parameters. This is reasonable since Flow++ is the only nonlinear model other than AUTM. It demonstrates that AUTM, as a nonlinear model, yields better efficiency and robustness in terms of parameter use.

Table 3.4: Results of BPD (bits per dim) on CIFAR10 and ImageNet32 datasets. Results in brackets indicate the model use variational dequantization.

Model	CIFAR10 BPD	CIFAR10 parameters	ImageNet32 BPD	ImageNet32 parameters
Real NVP	3.49	44.0M	4.28	66.1M
Glow	3.35	44.7M	4.09	67.1M
Flow++	(3.08)	31.4M	(3.86)	169.0M
RQ-NSF (C)	3.38	11.8M	-	-
Our Method	3.29	35.5M	3.80	35.5M

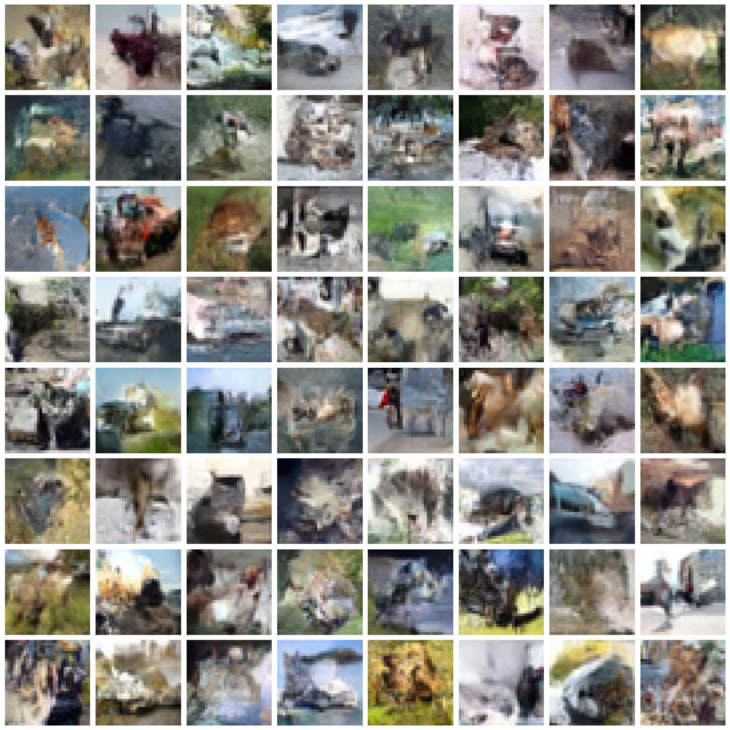


Figure 3.3: Samples generated by using a pretrained model on CIFAR10 dataset.

3.5.3 Numerical Inversion of AUTM

Existing normalizing flow models with no explicit inverse usually employ bisection to compute the inverse transformation. For example, the neural spline flow [25].

For AUTM, more options are available to compute the inverse mapping, such as fixed point iteration, which offers faster convergence than bisection. We compare the performance of bisection and fixed point iteration for AUTM by considering a toy example where function g in Equation (3.7) is chosen as a specific quadratic polynomial in v and the input variable x is randomly chosen from the unit interval. We use the discretized version (five-point trapezoidal rule) of the inverse formula in Equation (3.7) as the initial guess for fixed point iteration.

Table 3.5 shows that this leads to significantly fewer (around 50%) iteration steps than bisection to achieve the same solution accuracy.

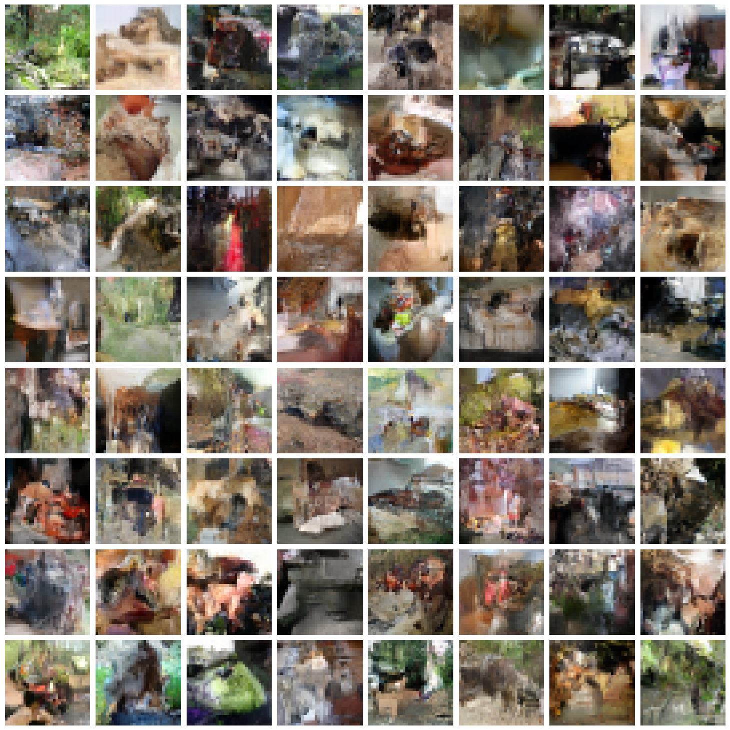


Figure 3.4: Samples generated by using a pretrained model on ImageNet32 dataset.

Table 3.5: Number of steps (averaged over 1000 random input) of root-finding method to reach a certain error tolerance.

Error tolerance	1e-3	1e-4	1e-5	1e-6
Iteration Method	4.565	6.642	8.658	10.831
Binary Search	8.967	12.398	15.668	19.073

3.5.4 Reconstruction on Image Dataset

We examine the reserve step of the AUTM layer by showing the reconstruction of images. The used model is the same as the model in Section 3.5.2 and use CIFAR10 and ImageNet32 dataset in this experiment. The computation of the inverse of our layer is by using iterative method with the reverse of integral as the initial guess. As Figure 3.5 shows, the average L1 reconstruction error converges in 15 steps. Also, Figure 3.6 shows that the reconstructed images look the same as original images.

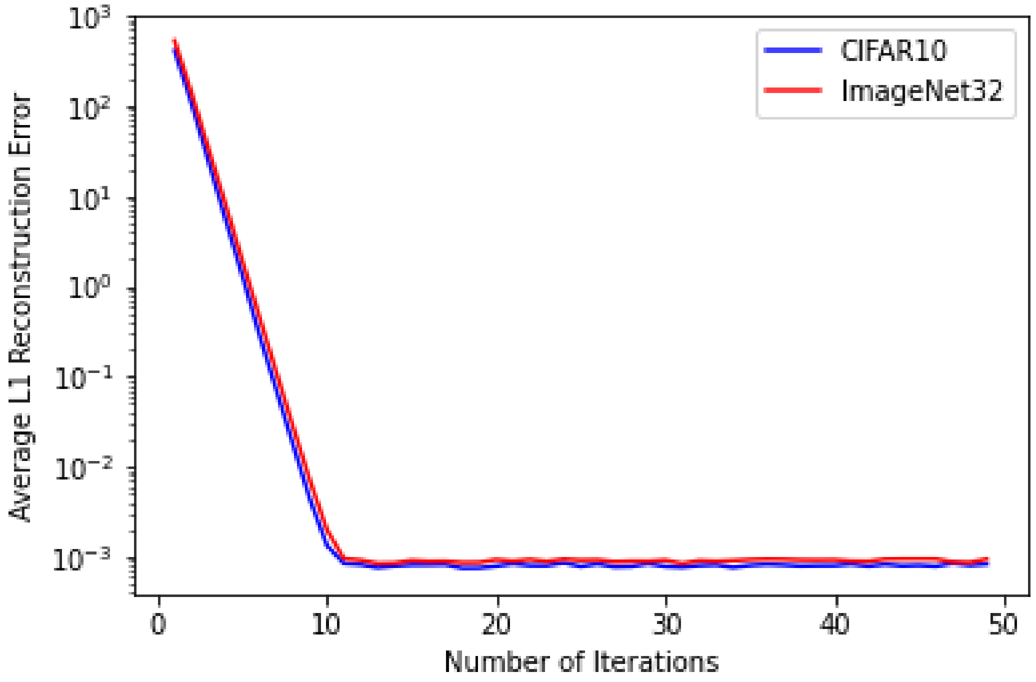


Figure 3.5: The average value of the L1 of reconstruction error for 64 images.

3.5.5 Comparison of FFJORD and AUTM

In this subsection, we test the running time of FFJORD and AUTM on four datasets from the UCI machine-learning repository POWER, GAS. HEPMASS, MINIBOONE, all preprocessed by [86]. For AUTM, we choose $g(v, t) = av + b + cv^3$. We define the target negative log-likelihood (target NLL) as the NLL achieved by FFJORD after training for 12 hours. The time for each method to reach the target NLL is reported in Table 3.6.

It demonstrates that AUTM is significantly more efficient than FFJORD. This is attributed to the structural advantages of AUTM. Firstly, AUTM transforms the input vector $x \in \mathbb{R}^D$ in an entrywise fashion where the ith entry is a univariate function of x_i. In FFJORD, the transformation of x is characterized by a neural network where each output dimension is a nonlinear multivariate function of $x = (x_1, \ldots, x_D)$. Secondly, due to the structural differences, AUTM has a triangular

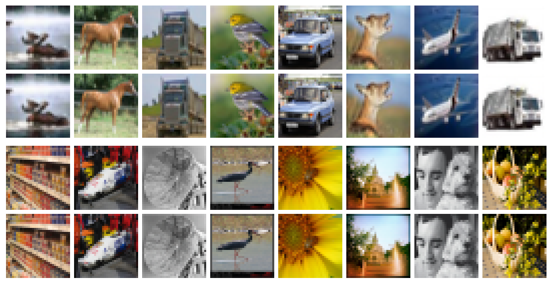


Figure 3.6: The reconstruction of selected images in CIFAR10 and ImageNet32 dataset. The 1st, 3rd rows are the original images, and the 2nd, 4th rows are the reconstructions.

Jacobian while FFJORD has a dense Jacobian that induces difficulty in computing the log-determinant accurately. Thirdly, the integrand g in AUTM can be chosen as a simple function, for example, a quadratic function in v. In FFJORD, the integrand needs to be a neural network with D input variables $x_1, \ldots, x_D$. To evaluate the integral of such a complicated integrand accurately, a large number of quadrature nodes are needed, which will increase the cost in both forward and backward transformations. Additionally, AUTM enables the use of user-defined integrand g, which will be beneficial if prior information of the transformation to be learned is available.

Table 3.6: Runtime for FFJORD and AUTM to reach the target negative log-likelihood for each dataset.

Dataset	Target NLL	FFJORD	AUTM
POWER	0.23	12hr	6.92min
GAS	-5.24	12hr	3.67min
HEPMASS	21.85	12hr	7.40min
MINIBOONE	11.29	12hr	1.75min

3.6 For Graph Problems

Recall Section 2.4.4, the framework of "Generating a Doppelganger Graph: Resembling but Distinct" is that, given an input graph G_0, first train a GraphSAGE model on G_0 with a link prediction objective, yielding node embeddings and a link predictor.

Then train a Wasserstein GAN on the node embeddings of G_0 and sample new embeddings from it. Additionally, input node features can be generated similarly if they are needed in a downstream task.

Finally, run the improved HH algorithm, wherein needed link probabilities are calculated by using the link predictor on the new embeddings generated by the GAN model. The resulting graph is a doppelganger of G_0.

In this framework, we could use normalizing flow model to replace the Wasserstein GAN in the second step in order to avoid the mode collapse. More specifically, we can use the newly proposed AUTM flow model to generate node embeddings.

Chapter 4

Denoising Diffusion Models for Generating Graphs

4.1 Background

In this project, we consider generating graphs based on a set of small graphs instead of one single graph.

The main motivation of this project is to generate molecules. The generation of molecules is an important task in chemistry, as it can help researchers find new molecules, especially new drugs for specific disease. Although molecules are 3D objects, researchers usually use 2D graphs to describe them. The atoms in molecules are described by nodes in graphs, and the chemical bonds between two atoms are described by edges in graphs. As a result, most researchers in the fields of computer science and mathematics focus on generating molecular graphs instead of considering 3D molecules.

Traditional methods for generating small graphs are designed for specific purpose: The Erdös-Rényi model [26, 31] is designed for generating edges with same probability, Barabási-Albert model [6] mimics the scale-free (powerlaw) property, and the stochastic block models [32, 57] try to keep community structures. However, those traditional methods only focus on specific metrics and often perform poorly on certain metrics.

As deep learning techniques can capture the complex information from a set of examples, many researchers have proposed several novel deep learning techniques for addressing this problem in recent years.

4.2 Problem

Given a set of small graphs $\{G_i\}$, generate a set of new graphs $\{G_i^{new}\}$, which has similar metrics (e.g. Maximum Mean Discrepancy of degrees between the given set and generated graphs) with $\{G_i\}$,.

4.3 Related Work

Classical Graph Theory methods discussed in Section 2.3 can be applied to solve this problem and more recent work involves deep generative models discussed in Section 1.4. In particular, many work combine deep generative models and graph neural networks.

For example, GraphVAE [99] and Graphite [38] try to reconstruct the graph adjacency matrix. The Junction Tree VAE [53] constructs molecular graphs by trees, and this method can be extended to construct molecular graphs by other substructures [54]. CGVAE [74] applies the gated graph neural network technique to the VAE framework, and CCGVAE [93] improves CGVAE for molecule design. MolGAN [15] combines GAN framework and reinforcement learning techniques to generate molecular graphs. GraphRNN [113] and GRAN [72] are sequential molecular graph generators based on Graph Neural Network and GAN. Graph Normalizing Flow [73] generates molecular graphs based on coupling flows, GraphAF [12] is based on autoregressive flows, and MolGrow [70] is based on hierarchical flows. GraphDF [75] uses discrete flow to generate molecular graphs. Niu et al. [82] applies score-based diffusion model to generate molecular graphs, and Xu et al. [112] uses Denoising Diffusion Probabilistic model generate 3D molecular graphs directly. Also, diffusion models with discrete state spaces have been discussed in [50].

4.4 Graph DDPM

In this section, we propose *Graph DDPM*, which is a graph generative method based on DDPM for generating similar graphs based on a set of graphs.

The proposed method contains two parts: (i) One *Graph Auto-Encoder model*, which maps the space of the features of nodes to small latent space, and (ii) DDPM model, which generates new node features. This method first trains a Graph Auto-Encoder to get node embeddings and a link prediction model, then uses DDPM to sample sets of new embeddings and get the edges by the link prediction model. Unlike the method proposed in Chapter 2, we do not need the post processing technique such as H-H algorithm.

Our method can generate permutation invariant graphs with good performance. Besides, to our knowledge, it is the first method combining the DDPM model and the node generation.

Next, I will provide the details of the proposed method “Graph DDPM” for generating simple graphs.

4.4.1 Graph Auto-Encoder

The *graph auto-encoder model* used in our work is inspired by VGAE [62] and Graph Normalizing Flow [73] which has two separate components: encoder and decoder.

The input of the encoder is a graph G with node features $X \in \mathbb{R}^{N\times d}$ and adjancency matrix $A \in \mathbb{R}^{N \times N}$, where N is the number of nodes in graph G and d is the dimension of the node feature. The output of the encoder is node embeddings $Emb \in \mathbb{R}^{N \times k}$, where k is the dimension of the node embeddings.

The encoder is designed as a 2-layer Graph Convolutional Neural Network [64]. More specifically, the decoder takes the output Emb as the input, and the output of the decoder is the edge probabilities matrix $\hat{A} \in \mathbb{R}^{N \times N}$.

Unlike the decoder $\hat{A} = EmbEmb^T$ used in VGAE [62], in our work, we use $\hat{A}_{ij} = \text{sigmoid}(5 - 5\|Emb_i - Emb_j\|_2^2)$ as our fixed decoder:

$$\hat{A}_{ij} = \frac{1}{1 + \exp(5\|Emb_i - Emb_j\|_2^2 - 5)} \tag{4.1}$$

We design this decoder based on the assumption that the connected nodes have similar embeddings.

The loss function used in the graph auto-encoder is the binary cross entropy loss function:

$$L(\theta) = -\sum (A_{ij} \log(\hat{A}_{ij}) + (1 - A_{ij}) \log(1 - \hat{A}_{ij})) \tag{4.2}$$

where θ is the parameters in the encoder.

4.4.2 DDPM

After training the graph auto-encoder as discussed in the previous section, we obtain node embeddings for each graph in the dataset. Assume all these node embeddings form a distribution. We choose DDPM [55] to learn this distribution and then sample new embeddings.

4.4.3 Summary

The input data of our method is a set of graphs. First, our method trains a graph auto-encoder model with a fixed decoder to get node embeddings. After that, we

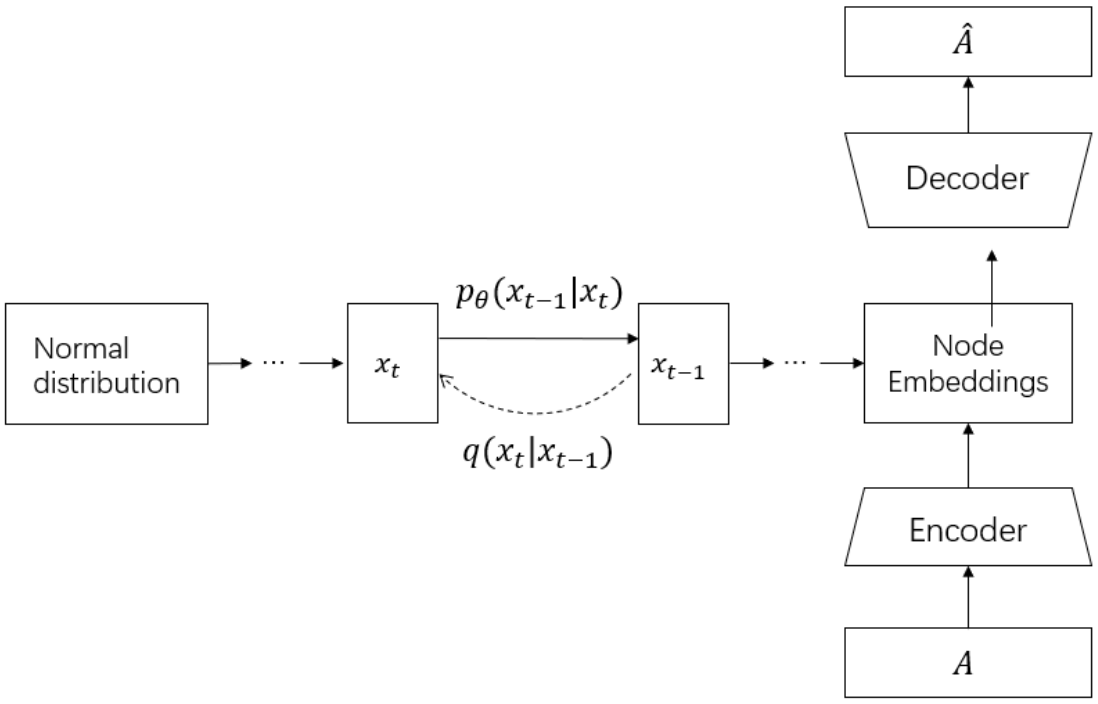


Figure 4.1: Framework of Graph DDPM.

train a DDPM on the node embeddings returned from the graph auto-encoder.

To generate new graphs, we sample new node embeddings from DDPM and use the fixed decoder in the graph auto-encoder model to generate edges. Figure 4.1 shows the framework of Graph DDPM.

4.4.4 Theoretical Analysis

In this subsection, we analyze the Graph DDPM method. We prove that our method is a permutation invariant method in Theorem 4.4.1.

Theorem 4.4.1. Assume the embeddings of the generated nodes are emb_i, and we have a link prediction model LP with $LP(a, b) = LP(b, a)$, and the graph generated by emb_i and LP is $G(V = \{emb_i\}, E = \{LP(emb_i, emb_j)\})$. This graph generation method is permutation invariant.

Proof. The permutation invariant property of graph generation method is that, for any reordering sequence $(k_1, k_2, ..., k_n)$ of sequence $(1, 2, ..., n)$, the graph $G_k(V = \{emb_{k_i}\}, E = \{LP(emb_{k_i}, emb_{k_j})\})$ should be isomorphism to the graph $G(V = \{emb_i\}, E = \{LP(emb_i, emb_j)\})$.

For the reordering sequence $(k_1, k_2, ..., k_n)$, we define a function $f(i) = k_i$, and we can see that nodes i and j of G are adjacent if and only if $f(i)$ and $f(j)$ are adjacent in G_k. As a result, the graph $G_k(V = \{emb_{k_i}\}, E = \{LP(emb_{k_i}, emb_{k_j})\})$ is isomorphism to the graph $G(V = \{emb_i\}, E = \{LP(emb_i, emb_j)\})$, which means that our method is permutation invariant. $\square$

4.5 Numerical Experiments

In this section, we provide several experiments to demonstrate that the proposed method can generate graphs which closely match the distribution of the input graphs.

We run our experiments with the dataset: EGO-SMALL. The dataset was introduced by GraphRNN [113]. EGO-SMALL is the set of 200 graphs. The number of nodes in each graph in EGO-SMALL is larger than 4 and smaller than 18. The EGO-SMALL dataset is derived from the Citeseer dataset [98].

For all experiments described in this section, 80% of the graphs are used for training and the other 20% graphs are used for testing.

4.5.1 Experiment Setting

Baseline models

We compare our method with the four existing methods: VGAE [62], DEEPGMG [114], GraphRNN [113], and GraphNF [73]. In our experiments, the data of the baseline models comes from [73].

Evaluation of generated graphs

The evaluation of our method contains two parts:

1. We provide visual graphs generated by our method and the original graphs.

2. Follow the method from GraphRNN [113], we calculate the Maximum Mean Discrepancy (MMD) [36] score between the graphs generated by our method and the graphs in the test set. We consider the MMD scores on three widely-used metrics: degrees, clustering coefficients, and orbit counts.

Table 4.1 shows the detail of each graph property that we use to evaluate generative models.

Table 4.1: Graph properties.

Property	Description
Clustering coefficient	Number of closed triplets divided by number of all triplets.
Degree	The degree of a node is the number of edges connecting it. The degree of the graph is the set of the node degrees.
Orbit count	A (vertex) automorphism in a graph $G = (V, E)$ is a permutation σ of the vertices that preserves adjacency. Mathematically, $(\sigma(u), \sigma(v)) \in E$ if and only if $(u, v) \in E$. The automorphisms of a graph induce a partition of the vertices into orbits, where two vertices belong to the same orbit if and only if there exists an automorphism that takes one to the other. The number of the different orbits is the orbit count.

Since the test set is small in our experiments, in order to avoid potential variance, we calculate the MMD scores between the test set and the specific graphs selected from the generated graphs. The process of the selection is that, first compute the number of nodes in each graph in the test set, and then find a set of graphs with the same distribution of node numbers from the generated graphs. This technique is proposed in [113].

Model configuration

Since the given dataset does not provide the node features, we generate node features $X_i \sim N(0, \sigma^2 I)$ for each node i, where $\sigma^2 = 0.1$. And, in different epoch, we generate different node features from the normal distribution. This method forces the graph auto-encoder to generate node embeddings only from the adjacency matrix.

The encoder in the graph auto-encoder is a 2-layer Graph Convolutional Neural Network [64], with hidden dimension 128. The dimension of the node embeddings is set as 20. We train the graph auto-encoder for 100 epochs with learning rate equal to 0.01. We also set a dropout layer in GCN with parameter 0.1. We use Adam as our optimizer.

The model used in DDPM contains 3 hidden layers, the dimension of each hidden layers are 128. The other settings are the same with the DDPM released code. We use some codes from DDPM [55].

4.5.2 Comparison of Baseline Models and Our Method

In this section, we compare the graphs generated by our method with the four baseline models. Table 4.2 shows the results of MMD of degree, cluster, orbit on EGO-SMALL dataset. All the results of baseline models come from [73].

We can see that our method performs best on the MMD of cluster and the MMD of orbit. Also, our method performs competitive on the MMD of degree.

4.5.3 Visualization of Generated Graphs

In this section, we provide the visualization of the graphs generated by our method and the graphs from the test set of EGO-SMALL. Figure 4.2 shows 10 selected graphs from the test set of EGO-SMALL. Figure 4.3 shows 10 selected graphs generated by our method.

Table 4.2: The results of MMD (lower is better) of three graph statistics between the test set of EGO-SMALL and graphs generated by our method. The results of baseline models come from Graph Normalizing Flow [73]. We train models 3 times and generate 3 sets of graphs, then calculate the mean and variance over these 9 trials.

Model	Degree	Cluster	Orbit
GraphVAE	0.13	0.17	0.05
DEEPGMG	0.04	0.10	0.02
GraphRNN	0.09 ± 0.10	0.22 ± 0.16	0.003 ± 0.004
GraphNF	$\mathbf{0.03 \pm 0.03}$	0.10 ± 0.05	0.001 ± 0.0009
Our Method	0.09 ± 0.01	$\mathbf{0.07 \pm 0.03}$	$\mathbf{0 \pm 0}$

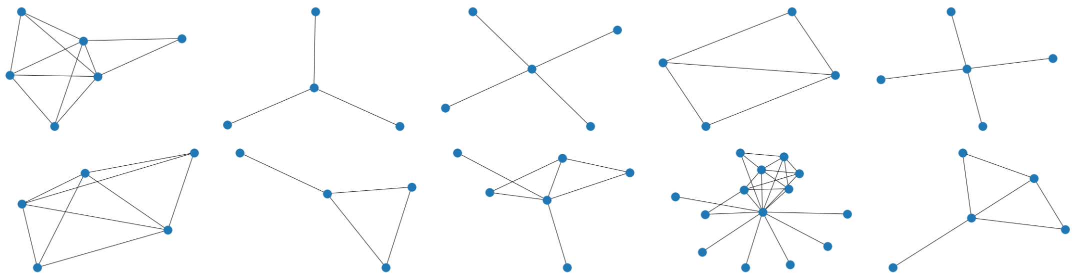


Figure 4.2: Selected graphs from the test set of EGO-SMALL.

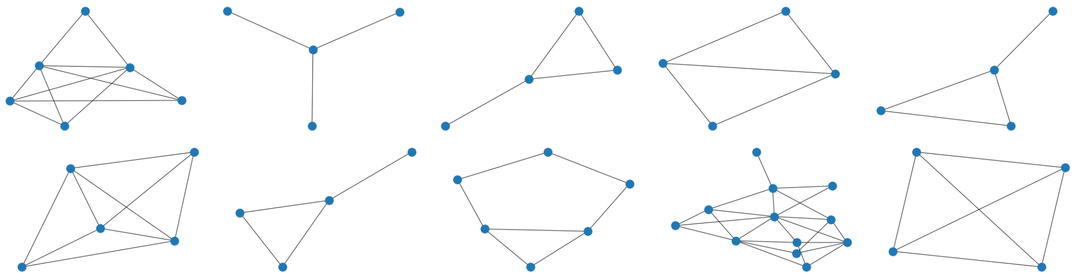


Figure 4.3: Selected graphs generated by our method.

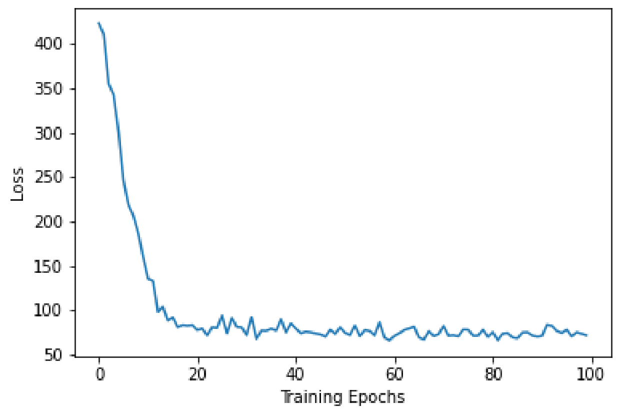


Figure 4.4: Training loss of graph auto encoder.

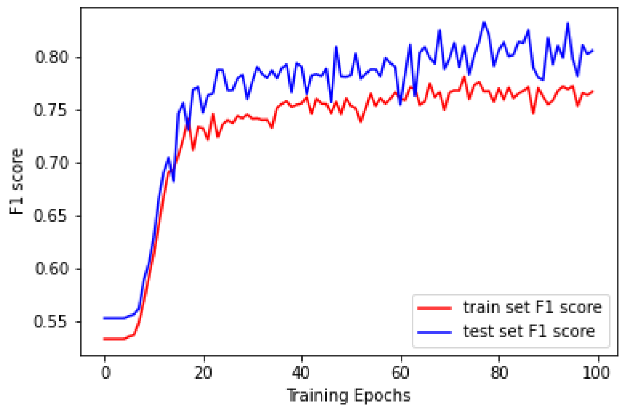


Figure 4.5: F1 score of graph auto encoder.

4.5.4 Graph Auto-Encoder Training Results

In this section, we discuss about the training process of the graph auto-encoder in our method. Figure 4.4 shows the training loss in each epoch during the training process. The definition of the loss could be found in Section 4.4.1.

We use the F1 score, which is defined as the harmonic mean of precision and recall, to evaluate the performance of our trained model. Figure 4.5 shows the F1 score for both the train set and the test set in each epoch. From the two figures, it is obvious that our graph auto-encoder is effective.

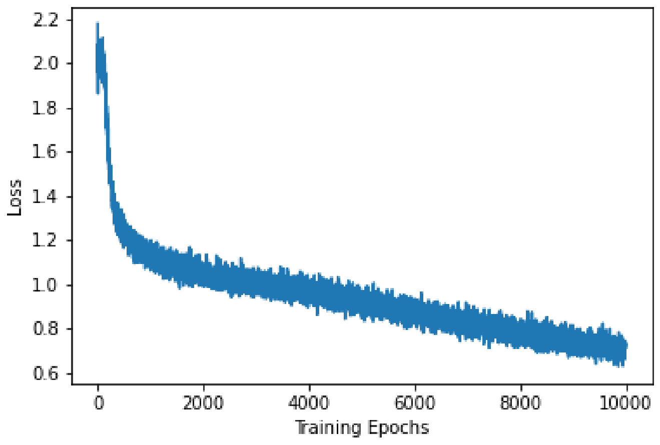


Figure 4.6: Training loss of DDPM.

4.5.5 DDPM Training Results

In this section, we discuss about the training process of the DDPM in our method. Figure 4.6 shows the training loss in each epoch during the training process. We train this model for 10000 epochs. Although it does not converge in 10000 epochs, the generated distribution of the first dimension and the second dimension is almost the same as the distribution of the two dimensions in the given training data when we plot the 2D figure. Also, the performance of the generated graphs is acceptable. As a result, we stop the training process at 10000 epochs.

Chapter 5

Summary and Future Work

In this dissertation, I proposed several machine learning enhanced graph generation methods.

In Chapter 1, I provided the mathematical background of graph generation problems. I introduced the general deep neural networks with emphasis on deep generative models as well as several classical results from graph theory. Also, I compared four popular deep generative models on a 2D toy problem.

In Chapter 2, I introduced our work "Generating a Doppelganger Graph: Resembling but Distinct". In this work, we try to combine Generative Adversarial Network with Havel-Hakimi algorithm, which is a traditional graph theory algorithm, to improve the performance of graph generation problem through introducing new node embeddings and using their link probabilities to guide the execution of HH. Our work shows that for some specific targets, it is better to consider graph theory rather than only use deep learning methods. We empirically validate the method on three benchmark data sets, demonstrating appealing match of the graph properties and similar performance in node classification.

In Chapter 3, I introduced our work "AUTM Normalizing flows". It is an improvement of the structure of normalizing flow layers. This work applies the continuous flow idea to coupling flows and autoregressive flows. The numerical experiments show that our new normalizing flow layers perform well and reduce the training cost significantly. Compared to FFJORD, AUTM demonstrates much better computational efficiency because of the triangular Jacobian structure. Compared to other monotonic flows, AUTM has unrestricted parameters and more convenient computation of the inverse transformation. Theoretically, we have proved that AUTM is a universal approximator for any monotonic normalizing flow. The performance is demonstrated by comparison to the state-of-the-art models in density estimation and image generation.

For Chapter 4, I introduced our work “graph DDPM”. This work aims to apply the DDPM to design a permutation invariant method for generating graphs based on a given set of graphs. This method performs competitively with existing state-of-the-art methods.

This dissertation tries to bridge the gap between the classical graph theory and deep neural networks on graph generation problems. The deep learning methods usually do not consider the graph theory or the domain knowledge behind the problem and the metrics. However, the results from this thesis confirm that the combination of graph theory and deep learning can reduce the dependence on the volume of the training data and also improve the performance of deep learning methods on several graph generation problems. More creative combination of classical graph theory algorithms and deep learning methods can be studied to solve other challenging graph related problems in the future.

Bibliography

[1] Michael S. Brown Abdelrahman Abdelhamed, Marcus A. Brubaker. Noise flow: Noise modeling with conditional normalizing flows. In *IEEE International Conference on Computer Vision*, 2019.

[2] W. Aiello, F. Chung, and L. Lu. A random graph model for power law graphs. *Experiment. Math.*, 10:53–66, 2001.

[3] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *International Conference on Machine Learning (ICML)*, 2017.

[4] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 2017.

[5] Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[6] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[7] Aleksandar Bojchevski, Oleksandr Shchur, Daniel Zügner, and Stephan

Günnemann. NetGAN: Generating graphs via random walks. In *International Conference on Machine Learning (ICML)*, 2018.

[8] Aleksandar Bojchevski, Oleksandr Shchur, Daniel Zügner, and Stephan Günnemann. NetGAN: Generating graphs via random walks. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 610–619, 2018.

[9] Bela Bollobas. *Modern Graph Theory*. Springer-Verlag, New York, USA, 1998.

[10] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations (ICLR)*, 2018.

[11] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[12] Shi Chence, Xu Minkai, Zhu Zhaocheng, Zhang Weinan, Zhang Ming, and Tang Jian. Graphaf: a flow-based autoregressive model for molecular graph generation. In *International Conference on Learning Representations (ICLR)*, 2020.

[13] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, pages 1724–1734. ACL, 2014.

[14] Zihang Dai, Hanxiao Liu, Quoc V. Le, and Mingxing Tan. Coatnet: Marrying convolution and attention for all data sizes. *CoRR*, abs/2106.04803, 2021. URL `https://arxiv.org/abs/2106.04803`.

[15] Nicola De Cao and Thomas Kipf. MolGAN: An implicit generative model for small molecular graphs. arXiv preprint arXiv:arXiv:1805.11973, 2018.

[16] Nicola De Cao, Wilker Aziz, and Ivan Titov. Block neural autoregressive flow. In *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*, 2020.

[17] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *IEEE conference on computer vision and pattern recognition (CVPR)*, 2009.

[18] Prafulla Dhariwal and Alexander Quinn Nichol. Diffusion models beat gans on image synthesis. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[19] D. Dheeru and E. Karra Taniskidou. Uci machine learning repository. In *URL http://archive.ics.uci.edu/ml.*, 2017.

[20] Laurent Dinh, David Krueger, and Yoshua Bengio. NICE: non-linear independent components estimation. In *International Conference on Learning Representations Workshop*, 2015.

[21] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp. In *International Conference on Learning Representations (ICLR)*, 2017.

[22] Hao-Wen Dong, Wen-Yi Hsiao, Li-Chia Yang, and Yi-Hsuan Yang. Musegan: Multi-track sequential generative adversarial networks for symbolic music generation and accompaniment. In *AAAI*, 2018.

[23] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods

for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[24] Conor Durkan, Artur Bekasov, Iain Murray, and George Papamakarios. Cubic-spline flows. *arXiv preprint arXiv:1906.02145*, 2019.

[25] Conor Durkan, Artur Bekasov, Iain Murray, and George Papamakarios. Neural spline flows. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[26] P. Erdös and A Rényi. On random graphs I. *Publicationes Mathematicae*, 6: 290–297, 1959.

[27] S. Fan and B. Huang. Labeled graph generative adversarial networks. arXiv preprint arXiv:1906.03220, 2019.

[28] Santiago Fernández, Alex Graves, and Jürgen Schmidhuber. An application of recurrent neural networks to discriminative keyword spotting. In *Artificial Neural Networks - ICANN, 17th International Conference*, volume 4669, pages 220–229. Springer, 2007.

[29] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202, 1980.

[30] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. Made: Masked autoencoder for distribution estimation. In *International Conference on Machine Learning (ICML)*, 2015.

[31] E.N. Gilbert. Random graphs. *Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.

[32] Anna Goldenberg, Alice X Zheng, Stephen E Fienberg, and Edoardo M Airoldi. A survey of statistical network models. *Foundations and Trends in Machine Learning*, 2(2), 2010.

[33] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2014.

[34] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016.

[35] Will Grathwohl, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. FFJORD: free-form continuous dynamics for scalable reversible generative models. In *International Conference on Learning Representations (ICLR)*, 2019.

[36] Arthur Gretton, Karsten M. Borgwardt, Malte J. Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *J. Mach. Learn. Res.*, 13: 723–773, mar 2012.

[37] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2016.

[38] Aditya Grover, Aaron Zweig, and Stefano Ermon. Graphite: Iterative generative modeling of graphs. In *International Conference on Machine Learning (ICML)*, 2019.

[39] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of Wasserstein GANs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[40] S. L. Hakimi. On the realizability of a set of integers as degrees of the vertices of a simple graph. *J. SIAM Appl. Math.*, 10:496–506, 1962.

[41] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[42] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[43] V. Havel. A remark on the existence of finite graphs. *Časopis pro pěstování matematiky*, 80:477–480, 1955.

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[45] Jonathan Ho, Xi Chen, Aravind Srinivas, Yan Duan, and Pieter Abbeel. Flow++: Improving flow-based generative models with variational dequantization and architecture design. In *International Conference on Machine Learning (ICML)*, 2019.

[46] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[47] P. W. Holland and S. Leinhardt. An exponential family of probability distributions for directed graphs. *Journal of the American Statistical Association*, 76 (373):33–50, 1981.

[48] Chin-Wei Huang, David Krueger, Alexandre Lacoste, and Aaron Courville.

Neural autoregressive flows. In *International Conference on Machine Learning (ICML)*, 2018.

[49] Chin-Wei Huang, Ricky T. Q. Chen, Christos Tsirigotis, and Aaron Courville. Convex potential flows: Universal probability distributions with optimal transport and convex optimization. In *International Conference on Learning Representations (ICLR)*, 2021.

[50] Austin Jacob, D. Johnson Daniel, Ho Jonathan, Tarlow Daniel, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[51] Priyank Jaini, Kira A. Selby, and Yaoliang Yu. Sum-of-squares polynomial flow. In *PInternational Conference on Machine Learning (ICML)*, 2019.

[52] Yuliang Ji, Ru Huang, Jie Chen, and Yuanzhe Xi. Generating a doppelganger graph: Resembling but distinct. *CoRR*, abs/2101.09593, 2021. URL `https://arxiv.org/abs/2101.09593`.

[53] Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Junction tree variational autoencoder for molecular graph generation. In *International Conference on Machine Learning (ICML)*, 2018.

[54] Wengong Jin, Regina Barzilay, and Tommi S. Jaakkola. Hierarchical generation of molecular graphs using structural motifs. 2020. URL `https://arxiv.org/abs/2002.03230`.

[55] Ho Jonathan, Jain Ajay, and Abbeel Pieter. Denoising diffusion probabilistic models. In *Neural Information Processing Systems (NeurIPS)*, 2020.

[56] He Kaiming, Zhang Xiangyu, Ren Shaoqing, and Sun Jian. Deep residual

learning for image recognition. In *IEEE conference on computer vision and pattern recognition (CVPR)*, 2016.

[57] B. Karrer and M. E. Newman. Stochastic blockmodels and community structure in networks. *Physical Review E*, 83(1):016107, 2011.

[58] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[59] Diederik P Kingma and Max Welling. Auto-encoding variational Bayes. In *International Conference on Learning Representations (ICLR)*, 2014.

[60] Durk P Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[61] Durk P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

[62] Thomas N Kipf and Max Welling. Variational graph auto-encoders. arXiv preprint arXiv:1611.07308, 2016.

[63] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[64] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[65] Ivan Kobyzev, Simon J.D. Prince, and Marcus A. Brubaker. Normalizing flows:

An introduction and review of current methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 2020.

[66] Zhifeng Kong, Wei Ping, Jiaji Huang, Kexin Zhao, and Bryan Catanzaro. Diffwave: A versatile diffusion model for audio synthesis. In *International Conference on Learning Representations (ICLR)*, 2021.

[67] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[68] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

[69] S. Kullback and R. A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[70] Maksim Kuznetsov and Daniil Polykovskiy. MolGrow: A graph normalizing flow for hierarchical molecular generation. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2021.

[71] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio. Object recognition with gradient-based learning. In *Shape, Contour and Grouping in Computer Vision*, pages 319–345. Springer, Berlin, Heidelberg, 1999.

[72] Renjie Liao, Yujia Li, Yang Song, Shenlong Wang, Charlie Nash, William L. Hamilton, David Duvenaud, Raquel Urtasun, and Richard Zemel. Efficient graph generation with graph recurrent attention networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[73] Jenny Liu, Aviral Kumar, Jimmy Ba, Jamie Kiros, and Kevin Swersky. Graph

normalizing flows. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[74] Qi Liu, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. Constrained graph variational autoencoders for molecule design. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[75] Youzhi Luo, Keqiang Yan, and Shuiwang Ji. Graphdf: A discrete flow model for molecular graph generation. In *International Conference on Machine Learning (ICML)*, 2021.

[76] Tengfei Ma, Jie Chen, and Cao Xiao. Constrained generation of semantically valid graphs via regularizing variational autoencoders. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[77] Bogdan Mazoure, Thang Doan, Audrey Durand, R Devon Hjelm, and Joelle Pineau. Leveraging exploration in off-policy algorithms via normalizing flows. In *3rd Conference on Robot Learning*, 2019.

[78] A. K. McCallum, K. Nigam, J. Rennie, and K. Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3(2):127–163, 2000.

[79] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[80] Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled generative adversarial networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[81] M. Molloy and B. Reed. A critical point for random graphs with a given degree sequence. *Random structures & algorithms*, 6(2-3):161–180, 1995.

[82] Chenhao Niu, Yang Song, Jiaming Song, Shengjia Zhao, Aditya Grover, and Stefano Ermon. Permutation invariant graph generation via score-based generative modeling. In *International Conference on ArtificialIntelligence and Statistics (AISTATS)*, 2020.

[83] Derek Onken, Samy Wu Fung, Xingjian Li, and Lars Ruthotto. OT-Flow: Fast and accurate continuous normalizing flows via optimal transport. In *AAAI Conference on Artificial Intelligence*, 2021.

[84] Aaron Van Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. In *International Conference on Machine Learning (ICML)*, 2016.

[85] OpenAI. Generative models, 2016. URL `https://openai.com/blog/generative-models/`.

[86] George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[87] Nanyun Peng, Hoifung Poon, Chris Quirk, Kristina Toutanova, and Wen-tau Yih. Cross-Sentence N-ary Relation Extraction with Graph LSTMs. *Transactions of the Association for Computational Linguistics*, 5:101–115, 04 2017.

[88] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: Online learning of social representations. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2014.

[89] Hermina Petric Maretic, Mireille El Gheche, Giovanni Chierchia, and Pascal Frossard. Got: An optimal transport framework for graph comparison. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[90] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2015. URL `http://arxiv.org/abs/1511.06434`.

[91] Ali Razavi, Aaron van den Oord, and Oriol Vinyals. Generating diverse high-fidelity images with vq-vae-2. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[92] Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International Conference on Machine Learning (ICML)*, 2015.

[93] Davide Rigoni, Nicolò Navarin, and Alessandro Sperduti. Conditional constrained graph variational autoencoders for molecule design. 2020. URL `https://arxiv.org/abs/2009.00725`.

[94] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400 – 407, 1951.

[95] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL `http://networkrepository.com`.

[96] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[97] Lars Ruthotto and Eldad Haber. An introduction to deep generative modeling. *CoRR*, abs/2103.05180, 2021. URL `https://arxiv.org/abs/2103.05180`.

[98] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93, 2008.

[99] Martin Simonovsky and Nikos Komodakis. GraphVAE: Towards generation of small graphs using variational autoencoders. In *ICANN*, 2018.

[100] Martin Simonovsky and Nikos Komodakis. Graphvae: Towards generation of small graphs using variational autoencoders. 2018. URL `http://arxiv.org/abs/1802.03480`.

[101] Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International Conference on Machine Learning (ICML)*, 2015.

[102] Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International Conference on Machine Learning (ICML)*, 2015.

[103] Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. In *International Conference on Learning Representations (ICLR)*, 2021.

[104] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. LINE: Large-scale information network embedding. In *International Conference on World Wide Web*, 2015.

[105] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[106] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[107] Mark Weber, Jie Chen, Toyotaro Suzumura, Aldo Pareja, Tengfei Ma, Hiroki Kanezashi, Tim Kaler, Charles E. Leiserson, and Tao B. Schardl. Scalable graph learning for anti-money laundering: A first look. In *Advances in Neural Information Processing Systems (NeurIPS) workshop*, 2018.

[108] Mark Weber, Giacomo Domeniconi, Jie Chen, Daniel Karl I. Weidele, Claudio Bellei, Tom Robinson, and Charles E. Leiserson. Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining workshop*, 2019.

[109] Antoine Wehenkel and Gilles Louppe. Unconstrained monotonic neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[110] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[111] Wikipedia. Graph. URL `https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)`.

[112] Minkai Xu, Wujie Wang, Shitong Luo, Chence Shi, Yoshua Bengio, Rafael Gomez-Bombarelli, and Jian Tang. An end-to-end framework for molecular conformation generation via bilevel programming. In *International Conference on Learning Representations (ICLR)*, 2022.

[113] J. You, R. Ying, X. Ren, W. Hamilton, and J. Leskovec. GraphRNN: Generating realistic graphs with deep auto-regressive models. In *International Conference on Machine Learning (ICML)*, 2018.

[114] Li Yujia, Vinyals Oriol, Dyer Chris, Pascanu Razvan, and Battaglia Peter. Learning deep generative models of graphs. In *International Conference on Machine Learning (ICML)*, 2018.

[115] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018. URL `http://arxiv.org/abs/1812.08434`.

[116] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *ICCV*, 2017.

[117] Zachary Ziegler and Alexander Rush. Latent normalizing flows for discrete sequences. In *International Conference on Machine Learning (ICML)*, 2019.