An Unpooling Layer for Graph Generation

Yinglong Guo
School of Mathematics
University of Minnesota
Minneapolis, MN 55455

Dongmian Zou
Division of Natural and Applied Sciences
Duke Kunshan University
Jiangsu, China 215316

Gilad Lerman
School of Mathematics
University of Minnesota
Minneapolis, MN 55455

Abstract

We propose a novel and trainable graph unpooling layer for effective graph generation. The unpooling layer receives an input graph with features and outputs an enlarged graph with desired structure and features. We prove that the output graph of the unpooling layer remains connected and for any connected graph there exists a series of unpooling layers that can produce it from a 3-node graph. We apply the unpooling layer within the generator of a generative adversarial network as well as the decoder of a variational autoencoder. We give extensive experimental evidence demonstrating the competitive performance of our proposed method on synthetic and real data.

1 INTRODUCTION

Graph data appear in many application areas, such as chemistry (Duvenaud et al., 2015), biology (Maere et al., 2005) and social recommendation (Fan et al., 2019). Common tasks that arise with graph data include regression and classification of either graph nodes or whole graphs and graph generation, which is useful for molecule generation and drug discovery. Graph neural networks (GNNs) have successfully generalized standard methods and architectures of neural networks to graph data and have achieved great success in many common tasks.

The task of graph generation is challenging due to its vast search space and the complexity of the graph structure. Furthermore, as we clarify next, it is hard to generalize basic procedures of deep generative networks in image generation to graph generation. For image data, a generative neural network, which may take the form of the decoder of a variational auto-encoder (VAE) or the generator of a generative adversarial network (GAN), usually first converts the input to a small intermediate image and then applies convolutional-transpose layers (Zeiler et al., 2010; Radford et al., 2016) or unpooling layers (Pu et al., 2016) to upsample and refine the image. In the graph domain, it is hard to form a similar convolution-transpose or unpooling layer in order to upsample graphs. Indeed, convolution and message passing do not change the structure of the underlying graph and there is no natural way of building structure for an unpooled graph. We are unaware of any graph generation work that follows the same idea of image generation and produces intermediate graphs and upsamples them to obtain the desired graph.

Inspired by image generation, we propose a novel unpooling layer for graph data that is similar to the unpooling operator for images. By incorporating this layer, one can build a deep graph generative network that utilizes intermediate graph structures.

1.1 Related Work

Graph neural networks. There is already a vast amount of recent work on GNNs. Many of those works focus on regression or classification tasks of nodes or whole graphs. A common building block of a GNN is the message-passing neural network (MPNN) layer, which was first proposed for predicting molecular properties (Scarselli et al., 2008; Duvenaud et al., 2015) and was immediately extended to other applications. More recently, different variants and extensions of the MPNN layer have been proposed within GNNs. For example, graph convolutional networks (GCNs) (Kipf and Welling, 2017) use a first-order approximation of the spectral convolution to derive a simple propagation rule for node classification, graph attention networks (Veličković et al., 2018) use self-attention to assign different weights to different nodes in a neighborhood, and graph isomorphism networks (Xu et al., 2019) adopt multilayer perceptrons (MLPs) after message passing to enhance expressivity. These networks are simple to implement and also follow a message-passing scheme.

Graph pooling and unpooling. The common idea of pooling layers in 2D convolutional neural networks has been generalized to graph-based data in order to produce smaller graphs. A graph pooling layer was first proposed by Bruna et al. (2014) and later extended in various works (Defferrard et al., 2016; Ying et al., 2018; Ma et al., 2019; Lee et al.,

Proceedings of the 26th International Conference on Artificial Intelligence and Statistics (AISTATS) 2023, Valencia, Spain. PMLR: Volume 206.

2019). Pooling layers are widely used in classification and regression on graphs since they downsample the graph while summarizing the aggregated presence of encoded features.

Unlike the convolution-transpose or unpooling operation for images, there is no obvious way to define a trainable unpooling procedure for graphs. Some works (Jin et al., 2018, 2019; Bongini et al., 2021) sequentially expand graphs by adding one node at a time. While their operations can be considered as unpooling, they cannot be regarded as the inverse operation of common pooling, since graph pooling is not generally done by removing one node at a time. Among all works related to graph pooling, Gao and Ji (2019) is the only one that seeks to define graph unpooling. They proposed Graph-UNet to combine pooling and unpooling processes in an encoder-decoder architecture. A Graph-UNet first pools an input graph into a smaller graph, encodes its global features and then applies the exact inverse process to perform the unpooling procedure. Since this operation deterministically depends on the pooling layers in the encoder, it is not suitable to be used for graph generation.

Graph generation. Faez et al. (2021) nicely survey graph generation models and categorized them as follows: auto-regressive (AR) (You et al., 2018b; Bongini et al., 2021; Ahn et al., 2021), VAE (Simonovsky and Komodakis, 2018; Jin et al., 2018; Samanta et al., 2020; Guo et al., 2021), GAN (De Cao and Kipf, 2018), reinforcement learning (RL) (You et al., 2018a), and normalized flow (Madhawa et al., 2019; Shi et al., 2020; Zang and Wang, 2020; Luo et al., 2021). Another recent category is diffusion models (Jo et al., 2022). There are two types of graph generation strategies: one-shot and sequential, where the former generates the output graph at once and the latter generates it in a node-by-node and edge-by-edge fashion. Most existing methods that leverage the one-shot strategy, such as De Cao and Kipf (2018); Simonovsky and Komodakis (2018); Zang and Wang (2020), produce a vectorized adjacency matrix, which does not utilize any graph structure during generation. On the other hand, methods that sequentially generate graphs typically use the graph structure. For example, You et al. (2018b); Shi et al. (2020); Luo et al. (2021) predict the next node or edge based on the features extracted from the existing graph.

Molecule generation is the most common application in this area as molecules can be naturally represented as graphs with features. However, molecules need to be chemically valid and this validity issue does not arise in general graph generation. Many methods (De Cao and Kipf, 2018; Simonovsky and Komodakis, 2018; Samanta et al., 2020; Zang and Wang, 2020) generate molecules at one-shot by producing an adjacency matrix that captures the molecular graph structure. Mahmood et al. (2021) propose the masked graph model (MGM) that generates graphs at one-shot by sampling masked sub-graphs of the respective complete graph. This method suggests a new graph generation category. On the other hand, it is also possible to sequentially generate molecules. For example, Jin et al. (2018, 2019) first generate a junction-tree as the scaffolding and then complete the graph. The junction-tree is generated recursively from the root, one node at a time. Some other recent models (Shi et al., 2020; Ahn et al., 2021; Luo et al., 2021) apply node-by-node and edge-by-edge sequential generation. Bongini et al. (2021) break the graph generation into three subproblems: node classification (which leads to node expansion), edge classification, and edge addition, where three separate GNNs are trained for each subproblem.

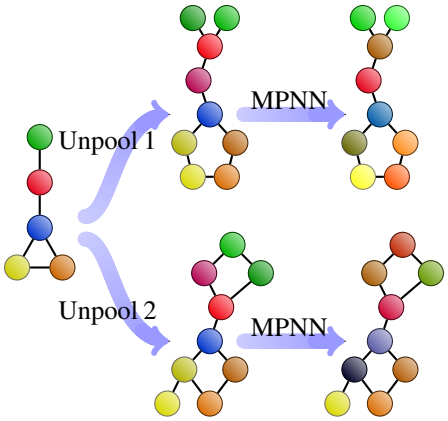

Figure 1: Demonstration of possible outputs of the proposed unpooling layer. Left: input graph, middle: two potential outputs of the unpooling layer, right: further application of an MPNN layer. The colors of nodes represent their features.

1.2 This Work

We propose a novel unpooling layer that effectively leverages the features and structure of a given graph to form an enlarged graph with learned features and structure. One may apply additional layers, such as MPNN layers, to further refine the features of nodes and edges in the graph. Figure 1 demonstrates two possible outputs of the unpooling layer with a followup MPNN layer for a given input graph. By incorporating unpooling layers in deep graph generative networks, we can generate a graph at one shot. In the experiments, we incorporate unpooling layers within both the generator of GAN and the decoder of VAE. We demonstrate in a synthetic setting how the unpooling layers reveal useful intermediate graph structures. We believe that the incorporation of such structures results in the competitive performance which is evident in all numerical experiments.

Our proposed unpooling layer (UL) leads to one-shot generation that utilizes the graph structure during generation. Among all existing methods, only the masked graph model (MGM) of Mahmood et al. (2021) applies one-shot generation that utilizes the graph structure during generation. Nevertheless, the implementation of our method is very different from Mahmood et al. (2021). In particular, the proposed unpooling layer enlarges the graph at each intermediate step, whereas Mahmood et al. (2021) sample the graph in masked subgraphs. As mentioned earlier, the graph generation category of MGM is rather different from the common categories. Furthermore, MGM was implemented and applied for molecule generation

and not general graph generation. Therefore, in terms of methodology our proposed strategy is distinguished from the many previous graph generation methods.

We emphasize the following contributions of our work:

- We propose a novel unpooling layer that produces an enlarged graph with learnable structure. It can be inserted into GANs and VAEs. The resulting generation framework is distinguished in its ability to both generate graphs at one shot and utilize the graph structures for generation.
- We show that the unpooling layer is valid and expressive. That is, the unpooled graph remains connected and any connected graph can be generated by a series of unpooling layers from a 3-nodes graph.
- We test the unpooling layer within both GANs and VAEs on a random graph dataset, a protein dataset and two molecule datasets and demonstrate competitive performance.

1.3 Structure of the Rest of the Paper

Section 2 details our proposed methodology; §3 provides theoretical guarantees of connectivity and expressivity; §4 reports numerical results on synthetic and real data of protein and molecule generation; and §5 concludes this work and discusses its limitations.

2 METHODOLOGY

Section 2.1 clarifies the construction of the unpooling layer and §2.2 explains how to use such layers for graph generation and how we update the parameters for the unpooling layers.

2.1 Unpooling Layer

Given an input graph (with features), the unpooling layer first unpools some of its nodes by replacing them with two "children" nodes and then learns a graph structure for the new set of nodes (in the output graph) and further learns new features.

Notation. We use the following notation for the input graph and its features. Its node set is $V = [N] := \{1, \cdots, N\}$; its edge set is E, where its members are of the form $\{i, j\}$ for some i, $j \in V$; its number of edges is $M = |E|$; its node feature matrix is $\boldsymbol{X} \in \mathbb{R}^{N\times d}$, where its i-th row, $\boldsymbol{x}_i$, is the d-dimensional feature of node i; and its edge feature matrix is $\boldsymbol{W} \in \mathbb{R}^{M\times e}$, where for edge $\{i, j\} \in E$, its corresponding row of $\boldsymbol{W}$ is the e-dimensional feature of that edge, which we denote by $\boldsymbol{w}_{i,j}$. Similarly, we use the following notation for the output graph and its features: V^o is its node set, E^o is its edge set and $\boldsymbol{Y}$ and $\boldsymbol{U}$ are the feature matrices for the output nodes and edges, respectively. We remark that the size of V^o lies in $[|V| + 1, 2|V|]$ and depends on the hyperparameters that determine which nodes should be unpooled.

The input and output graphs of the unpooling layers with their features are respectively denoted by

$$\mathcal{G} = (V, E, \boldsymbol{X}, \boldsymbol{W}) \text{ and } \mathcal{G}^o = (V^o, E^o, \boldsymbol{Y}, \boldsymbol{U}).$$

We will refer to them as featured graphs and to (V, E) and (V^o, E^o) as graphs.

An overview of the unpooling layer. The unpooling layer determines the output graph in a stochastic manner. Ideally, it should produce probabilities of every possible output graph based on the input graph by using trainable parameters, $\boldsymbol{\theta}_P$. That is, it would output a probability mass function, $\boldsymbol{p}$, on the sample space $\mathbb{G}^o = \{(V^o, E^o) : \text{unpooled from } \mathcal{G}\}$, where $\boldsymbol{p}(\mathcal{G}; \boldsymbol{\theta}_P) \in [0, 1]^{|\mathbb{G}^o|}$. In this ideal case, the unpooling layer then randomly draws an output graph, (V^o, E^o), according to this probability mass function. The drawn probability of this sample point, which we denote by

$$\mathbb{P}(V^o, E^o|\mathcal{G}; \boldsymbol{\theta}_P), \tag{1}$$

can then be used to update the parameters $\boldsymbol{\theta}_P$ during training. Therefore, the unpooling layer can ideally refine the probability distribution $\boldsymbol{p}(\mathcal{G}; \boldsymbol{\theta}_P)$ in order to obtain graphs that minimize the training loss function.

Since the sample space $\mathbb{G}^o$ is huge, we cannot explicitly produce the probabilities of all possible output graphs. In practice, we use several multi-layered perceptrons (MLPs) to produce probabilities to determine if nodes should be unpooled and if edges between unpooling children nodes should be included in the output graph. The product of all these probabilities for nodes and edges gives the probability of the entire output graph in (1) (assuming these events are independent), which is used during training.

After forming the output graph, the unpooling layer also produces the node and edge features using two MLPs that exploit the input featured graph and possibly the output graph, that is,

$$\boldsymbol{Y} = \text{MLP-}Y(\mathcal{G}; \theta_Y), \boldsymbol{U} = \text{MLP-}W(V^o, E^o, \mathcal{G}; \theta_W).$$

Detailed mechanism of the unpooling layer. The unpooling layer contains seven MLPs that serve different purposes, which we introduce below and in the supplementary materials §A. For the formation of the unpooling layer We use the following three node sets that partition the input node set: (1) the set of nodes I'_s that are unchanged in the unpooling layer; (2) the set of nodes I'_u that are determined to be unpooled in the unpooling layer; and (3) the set of nodes I'_r that requires a probabilistic decision whether to unpool or not.

With the specified node sets, we describe the procedure of generating the output graph in the unpooling layer according to 3 steps, which we demonstrate in Figure 2.

Step 1. Generating the output nodes and node features ((a) in Figure 2) We first probabilistically determine which nodes in I'_r will be unpooled. For each node $i \in I'_r$, we determine the probability of unpooling it, $p_r(\boldsymbol{x}_i)$, by an MLP as follows: $p_r(\boldsymbol{x}_i) = \text{MLP-}R(\boldsymbol{x}_i)$. Then we draw uniform random variables $U_i \sim U[0, 1]$ and form the following sets I_u and I_s of unpooled nodes and unchanged (or stable) nodes: $I_u := I'_u \cup \{i \in I'_r : U_i < p_r(\boldsymbol{x}_i)\}$ and $I_s := I'_s \cup \{i \in I'_r : U_i \geq p_r(\boldsymbol{x}_i)\}$.

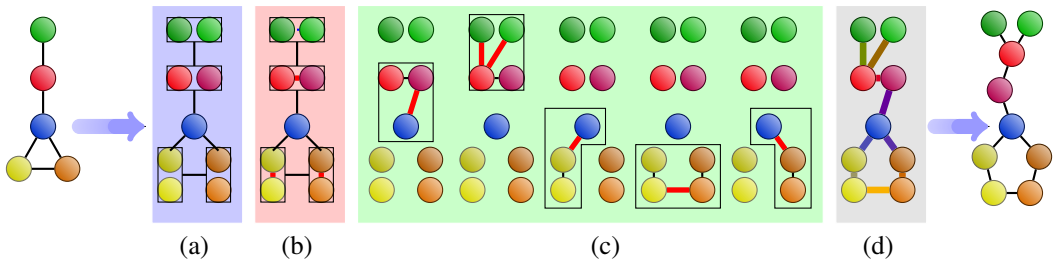

Figure 2: Demonstration of the steps of the unpooling layer: (a) unpool and generate children nodes; (b) build intra-links within pairs of children nodes; (c) build inter-links involving children nodes; (d) build features for the edges. Right: The output graph.

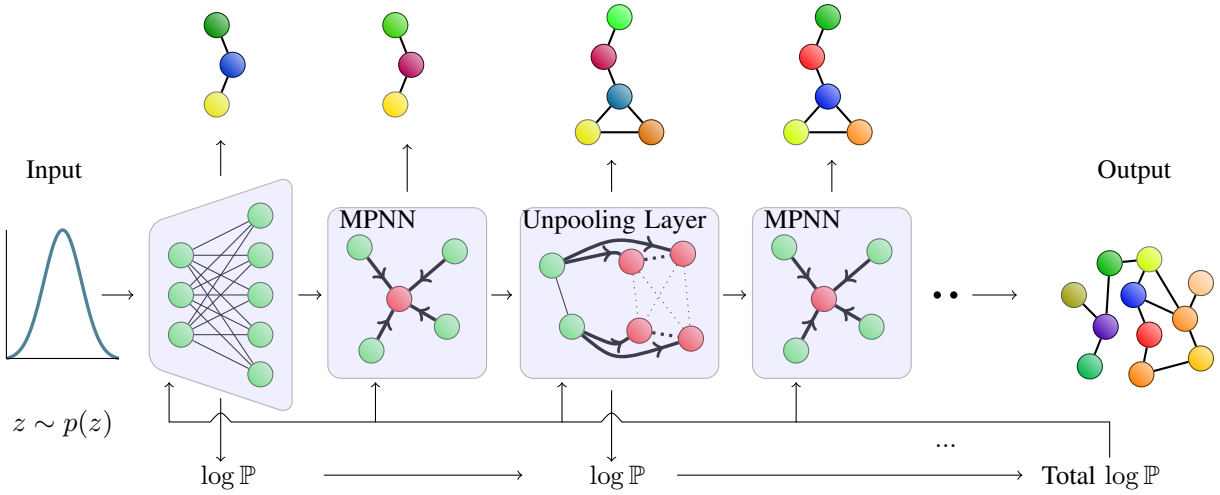

Figure 3: Demonstration of a generative neural network with unpooling and MPNN layers. The central row shows the input latent vector, an initial layer that creates a 3-node graph, a series of MPNN and unpooling layers, and the output featured graph (the features are represented by colors). The top row shows the intermediate hidden featured graphs outputted at every layer. The initial layer and the unpooling layers generate log probabilities. The generative GNN accumulates them to obtain the total log probability (shown in the bottom row); it uses it in the REINFORCE algorithm to train the unpooling layers (see (3)).

We remark that if we want to generate the output graph with a fixed number of nodes, we could choose $I'_r = \emptyset$ for the unpooling layer.

The set of nodes of the output graph V^o is the union of the nodes in I_s (with different indices) and a set of nodes of size $2|I_u|$ representing the unpooled nodes from I_u. For each node $i \in I_s$, we denote by $f(i)$ the index of this node in the output graph. For each node $i \in I_u$, we denote by $f_1(i)$ and $f_2(i)$ the indices of the two unpooled nodes in the output graph. The output node features are generated by an MLP and two projection operators P_{S_1} and P_{S_2}, which are defined in detail in §C.2

$$\boldsymbol{y}_{f(i)} = \text{MLP-}y(P_{S_1}\boldsymbol{x}_i), \quad \text{for } i \in I_s.$$
$$\boldsymbol{y}_{f_j(i)} = \text{MLP-}y(P_{S_j}\boldsymbol{x}_i), \quad \text{for } i \in I_u,\ j = 1, 2.$$

Step 2. Building output edges. We sequentially generate the set E^o of edges in the output graph following the next two substeps. We initiate this set by $E^o := \emptyset$.

Step 2.1. Building intra-links ((b) in Figure 2). For each node $i \in I_u$, we determine whether to generate an edge that connects the children nodes $f_1(i)$ and $f_2(i)$ based on a probability, which we denote by $p_c(\boldsymbol{x}_i)$. This probability is produced using an MLP as follows: $p_c(\boldsymbol{x}_i) \equiv \text{MLP-IA}(\boldsymbol{x}_i)$. Then we draw uniform random variable $U_i \sim U[0, 1]$ and if $U_i < p_c(\boldsymbol{x}_i)$, we add this edge to the output graph, that is, $E^o = E^o \cup \big\{\{f_1(i), f_2(i)\}\big\}$.

Step 2.2. Building inter-links ((c) in Figure 2). For each edge $\{i, j\} \in E$ in the input graph, we determine the edges for the corresponding nodes in the output graph according to the following three different cases: (1) $i, j \in I_s$: we include the edge $\{f(i), f(j)\}$ in the output graph; (2) $i \in I_s$ and $j \in I_u$: we probabilistically determine what are the edges between $f(i)$ and $\{f_1(j), f_2(j)\}$; and (3) $i, j \in I_u$: we probabilistically determine what are the edges between $\{f_1(i), f_2(i)\}$ and $\{f_1(j), f_2(j)\}$.

For each edge $\{i, j\} \in E$, we introduce node sets $N_{\{i,j\},i}$ and $N_{\{i,j\},j}$ in order to uniformly handle the above cases. For $\{i, j\} \in E$ and $i \in V$, we form $N_{\{i,j\},i}$

as follows: If $i \in I_s$, then $N_{\{i,j\},i} = \{f(i)\}$ and if $i \in I_u$, then $N_{\{i,j\},i}$ is a nonempty subset of the children nodes of i, which we probabilistically determine as follows. We use an MLP to calculate two probabilities: $(p_1, p_2) = \text{MLP-IE}(\boldsymbol{y}_{f_1(i)}, \boldsymbol{y}_{f_2(i)}, \boldsymbol{w}_{i,j}, \boldsymbol{x}_j)$. We then draw a uniform random variable $U \sim U[0,1]$ and determine $N_{\{i,j\},i}$ as follows:

$$N_{\{i,j\},i} = \begin{cases} \{f_1(i)\}, & \text{if } U < p_1; \\ \{f_1(i), f_2(i)\}, & \text{if } U \geq p_1 + p_2; \\ \{f_2(i)\}, & \text{otherwise.} \end{cases}$$

For $\{i,j\} \in E$ and $j \in V$ we similarly define $N_{\{i,j\},j}$, while swapping i and j. The edges in the output graph are updated as follows:

$$E^o = E^o \cup \{\{k,l\} : k \in N_{\{i,j\},i},\, l \in N_{\{i,j\},j}\}. \quad (2)$$

We need to take extra care to ensure connectivity and expressivity of the output graph. We first form two additional MLPs, MLP-C and MLP-IE-A to calculate probabilities. We then probabilistically form some additional edges. These details are a bit technical and can be fully understood after getting familiar with our theory for connectivity and expressivity (see §3 and the proofs in the supplementary materials §D). Therefore we leave these details to §A of the supplementary materials (see Step 2b and Step 2d in §A).

Step 3. Constructing the edge features ((d) in Figure 2). For each edge $\{k,l\} \in E^o$, we construct the edge feature $\boldsymbol{u}_{k,l}$ by an MLP as follows:

$$\boldsymbol{u}_{k,l} = \text{MLP-}u\big(\text{LeakyReLU}(\boldsymbol{y}_k + \boldsymbol{y}_l)\big).$$

Summary. We described a probabilistic construction of $\mathcal{G}^o = (V^o, E^o, \boldsymbol{Y}, \boldsymbol{U})$. It contains seven MLPs to produce various probabilities and features for the nodes and edges of $\mathcal{G}^o$. The parameters in those seven MLPs form the training parameters of the unpooling layer. The overall probability $\mathbb{P}(V^o, E^o | \mathcal{G}; \boldsymbol{\theta}_P)$ is the product of all the probabilities in the first two steps and is used to update the training parameters in the unpooling layer, while using the REINFORCE algorithm introduced below.

2.2 Graph Generation and Training

We use the unpooling layer within a generative GNN, which can be either a generator of a GAN or a decoder of a VAE. We describe here its basic mechanism and demonstrate it in Figure 3. Complete details of implementation are in both §4 and the supplementary materials. The generative GNN first maps a given latent vector into a featured 3-nodes graph, whose structure is probabilistically determined by an MLP and its edge features are determined by another MLP (details are in the supplementary materials). Next, it applies a GCN (such as MPNN) to update the node features for this initial featured graph. It then sequentially applies unpooling layers, where each of them is followed by a GCN (such as MPNN), which further updates the node features. The final output is the generated graph.

The parameters used for generating features in the unpooling layer can be updated during training following the common framework of GAN or VAE. The major challenge in using the unpooling layers for graph generation is that the graph generation process is not differentiable. To overcome this, we follow REINFORCE with baseline (Weaver and Tao, 2001; Sutton and Barto, 2018) to update all the graph generation parameters in the unpooling layer.

In order to explain this procedure with more details, we need the following notation. We denote by G a generative GNN (a generator of a graph GAN or the decoder of a graph VAE) with several unpooling layers that takes a latent vector and produces a generated graph. We denote by m the number of unpooling layers of G, by $U_1, U_2, \cdots, U_m$ the unpooling layers and by $(V_1^o, E_1^o), \cdots, (V_m^o, E_m^o)$ the generated intermediate graphs. Let $\boldsymbol{\theta}$ denote all the parameters of G, which include the parameters of the MLPs in the unpooling layers. In view of (1), the total log probability of the unpooling layer U_k is $\log \mathbb{P}(V_k^o, E_k^o)$. We define the total log probability of the generator G as $\log \mathbb{P} := \sum_k \log \mathbb{P}(V_k^o, E_k^o)$ and note that $\log \mathbb{P}$ depends on $\boldsymbol{\theta}$. We denote the learning rate by α and the reward for the generated graph by r. Note that this reward depends on the specific generation task, e.g., it can be the likelihood predicted by the discriminator or the chemical property which one aims to optimize.

We update $\boldsymbol{\theta}$ as follows

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha(\nabla_{\boldsymbol{\theta}} \log \mathbb{P}|_{\boldsymbol{\theta}_k})(r - \mathbb{E}r), \quad (3)$$

where we approximate $\mathbb{E}r$ by the sample mean. In our experiments we incorporate the unpooling layers within a GAN and a VAE. For a GAN, we set the reward r to be $D(G(z;\boldsymbol{\theta}))$, where D is the GAN's discriminator, in order to compete with the discriminator. For a VAE, we choose the reward r to be the negative of the reconstruction error in order to minimize the reconstruction error.

3 THEORETICAL GUARANTEES

We establish the connectivity and expressivity of the unpooling layer. All proofs are in the supplementary materials.

3.1 Guarantee of Connectivity of the Output Graph

The following proposition implies that if the input graph is connected, then the unpooling layer will produce a connected graph. This is an important property in molecule generation since otherwise the output molecule will be invalid. Adjacency matrix-based generators cannot ensure connectivity.

Proposition 3.1. *Given an unpooling layer and any connected input graph $\mathcal{G}$, the output graph, $\mathcal{G}^o$, of this unpooling layer is connected.*

3.2 Guarantee of Expressivity for the Unpooling Layer

It is important to know whether a series of unpooling layers can produce any connected graph. For instance, in molecule generation, a good generative model should contain all valid molecules in the set of possible output. Some previous work (e.g., Jin et al. (2018)) cannot produce some valid molecular

structures and is thus not fully expressive. Fortunately, we are able to produce any connected graph by applying certain unpooling layers to an input graph with three nodes (our implementation of the graph generative model starts with a 3-nodes graph). We first formulate a theorem on the expressivity of a single unpooling layer and then formulate the desired corollary when starting with a 3-nodes graph and using a series of unpooling layers.

Theorem 3.2. *Given a connected graph $\mathcal{G}^o$ with N nodes and an integer $K \in [\lceil N/2 \rceil, N-1]$, there exist an unpooling layer and an input graph $\mathcal{G}$ with K nodes so that $\mathcal{G}^o$ is the corresponding output.*

Corollary 3.3. *Given a connected graph $\mathcal{G}^o$ with N nodes, there exist a 3-nodes graph $\mathcal{G}$ and $\lceil \log_2(N/3) \rceil$ unpooling layers, so that $\mathcal{G}^o$ is the output of this series of unpooling layers acting on $\mathcal{G}$.*

We remark that the proof of Theorem 3.2 naturally provides a "pooling" procedure on the graph structure which can be regarded as the inverse operation of our unpooling layer. This validates the name "unpooling".

4 EXPERIMENTS

We demonstrate the effectiveness of the unpooling layer for molecule generation. We describe the two datasets in §4.1, the evaluation metrics in §4.2 and the details of the implemented methods in §4.3. We then report the results in §4.4, while comparing with benchmark methods. All implemented codes are provided in the supplementary materials.

4.1 Datasets

Waxman random graph dataset. The dataset contains randomly generated Waxman graphs (Waxman, 1988). More precisely, we first created 20,000 graphs with 12 nodes and node features uniformly drawn from $[0,1]^2$. For each graph, we connected nodes i and $j \in [12]$ with probability $qe^{-sd_{ij}}$, where $q = 0.65$, $s = 0.3$ and d_{ij} is the Euclidean distance between their features. We do not assign edge features. Next, for each graph we compute the largest connected subgraph as long as it has at least 5 nodes. The final set contains these subgraphs with at least 5 nodes (the ones with less nodes are discarded). Thus the number of nodes ranges from 5 to 12 and the node features are the x and y coordinates. On average, each graph contains 9.2 nodes and 10.3 edges. There are 18,910 graphs in this dataset.

Protein dataset. We use the protein dataset introduced in Guo et al. (2021), which is a benchmark for graph generation (Du et al., 2021). All the graphs contain 8 nodes and their node features are their 3D coordinate vectors. There are no edge features. On average, each graph contains 8 nodes and 19.3 edges. There are 76,000 graphs in the datasets, where we use 38,000 for training and 38,000 for testing.

Molecule datasets. We use two common datasets for molecule generation: QM9 (Ramakrishnan et al., 2014) and ZINC (Sterling and Irwin, 2015). QM9 contains 130k molecules and each molecule consists of up to 9 heavy atoms among carbon (C), oxygen (O), nitrogen (N) and fluorine (F). In §4.3 we explain how we choose the hyperparameters of the unpooling layers so that the generator will generate molecules with the number of atoms ranging between 6 and 9. On average, each graph contains 8.8 nodes and 9.4 edges.

ZINC contains about 250k molecules, where each molecule consists of 9 to 38 heavy atoms among carbon (C), oxygen (O), nitrogen (N), sulfur (S), fluorine (F), chlorine (Cl), bromine (Br), iodine (I) and phosphorus (P). For simplicity, we only take molecules with 11 - 36 heavy atoms (99.8% of ZINC). On average, each graph contains 23.2 nodes and 24.9 edges.

For both QM9 and ZINC, we include the following node features: atom type, chiral specification of an atom (unspecified, clockwise or counter-clockwise) and the formal charge of an atom (0, +1 or -1). We use bond type (single, double or triple) as edge features. The node and edge features are represented as one-hot vectors.

4.2 Evaluation Metrics

In the numerical experiments of the Waxman random graph and protein datasets, we evaluate the similarity of the generated data and source data by comparing the distributions of some graph properties in the generated graphs and source data. We use the following graph properties for comparison: average node connectivity, average clustering coefficient, edge density and node features. For both datasets, we report the Kullback–Leibler divergence and Wasserstein distance between the distributions of the source and generated data. For the Waxman random graph dataset we further demonstrate the distributions of the four properties for the source and generated data, while considering several methods for graph generation.

In the numerical experiments of molecule generation, we compare the different generators by generating 10,000 molecules and applying the following metrics: validity (the ratio between the number of generated valid molecules and all generated graphs); uniqueness (the ratio between the number of unique valid molecules and generated valid molecules); and novelty (the ratio between the number of unique valid molecules which are different from all molecules in the dataset and the total number of generated unique valid molecules). We also report the geometric mean of the above three metrics (G-mean).

4.3 Implementation Details

We implemented a GAN with the unpooling layers (UL GAN), a GAN with an adjacency matrix-based generator (Adj GAN) and a VAE with the unpooling layers (UL VAE). For simplicity, we just describe here the implementation for molecule generation using the QM9 dataset. Indeed, the implementations of Adj GAN, UL GAN and UL VAE for the other applications are similar. Additional details are in

the supplementary materials.

Discriminator for Adj GAN and UL GAN. It takes an input graph and uses two MPNN layers with 128 units to generate a graph $\mathcal{G} = (V, E, \boldsymbol{X}, \boldsymbol{W})$. It then aggregates the node features (the rows $\{\boldsymbol{x}_j\}_{j\in V}$ of $\boldsymbol{X}$) to produce the following single feature vector for the graph:

$$\boldsymbol{h}(\mathcal{G}) = \sum_{j\in V} \sigma(\mathrm{lin}_1(\boldsymbol{x}_j)) \odot \tanh(\mathrm{lin}_2(\boldsymbol{x}_j));$$

where σ is logistic sigmoid, $\odot$ is element-wise multiplication and lin_1 and lin_2 are two layers with 128 units. Then it applies a layer lin_3 with 256 units. A final layer with a single unit then produces the output, where its activation function is $\tanh$. A batch normalization and leaky ReLU activation function are used after the two MPNNs and $\mathrm{lin}_1, \mathrm{lin}_2, \mathrm{lin}_3$ layers.

Encoder for UL VAE. It has the same architecture as the above discriminator, except that the final layer maps into a 256-dimensional vector with a linear activation function. This vector further splits to two 128-dimensional vectors: $\boldsymbol{z}_\mu$ and $\boldsymbol{z}_\sigma$. It then generates the following latent vector: $\boldsymbol{z} = \boldsymbol{z}_\mu + \exp(\frac{1}{2}\boldsymbol{z}_\sigma) \odot \boldsymbol{x}$, where $\boldsymbol{x} \sim N(0, 1)$.

Generator for Adj GAN. It contains four linear layers with $128, 256, 256, 512$ units with batch normalization and leaky ReLU activation function. The last layer generates a 9×11 tensor (matrix) for the node features and $9 \times 9 \times 4$ tensor for the edge features. We use a hard Gumbel softmax to produce the one-hot feature vectors for the nodes and edges.

Generator for UL GAN and decoder for UL VAE. It takes a 128-dimensional vector and outputs a graph. The generator contains the following layers: an initial MLP layer that takes the random noise vector and creates a 3-nodes graph with 256-dimensional node features; an MPNN layer with 128 units; an unpooling layer that maps the 3-nodes graph to a 4-or-5-nodes graph; an MPNN layer with 128 units; an unpooling layer that maps the 4-or-5-nodes graph to a 6-to-9-nodes graph; an MPNN layer with 64 units; a linear layer with 64 hidden units and two final layers that produces node and edge features in $\mathbb{R}^{10}$ and $\mathbb{R}^3$, respectively. A skip connection, which takes the input vector, is added to the node features after each unpooling layer. Finally, a hard Gumbel softmax generates the desired one-hot features. The dimension of the edge features in all intermediate graphs is 32. The log probabilities from this sampling process are added to obtain the total log probability when updating the parameters according to (3).

Training process. For UL GAN and Adj GAN, the loss function corresponds to Wasserstein GAN with gradient penalty Gulrajani et al. (2017). For UL VAE, the loss function is the sum of the reconstruction errors of node features and edge features and the Kullback-Leibler divergence between the latent vector and a standard Gaussian. We use the Adam optimizer with a learning rate 2×10^{-4} for the generator and a learning rate 10^{-4} for the discriminator with a training batch size of 64. During the training process, we evaluate the model every 500 iterations and we report the result with optimal validity before a mode collapse occurs. In each training step, we alternatively minimize the loss function and update the parameters according to the policy gradient procedure in (3).

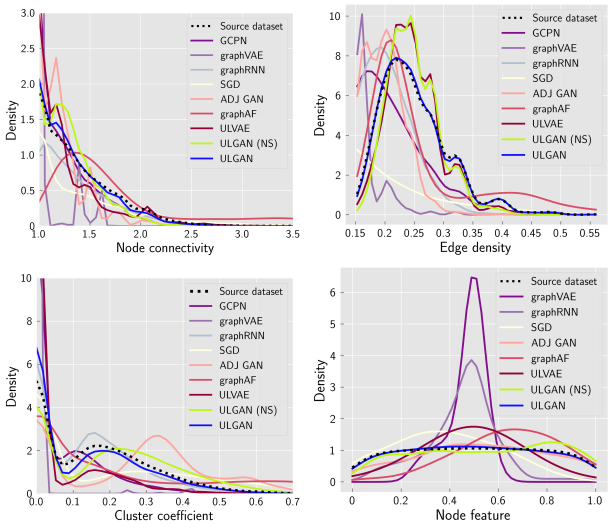

Figure 4: Distributions of the four graph properties for the generated and source data using the Waxman random graph dataset. The graph properties include average node connectivity (top left), average clustering coefficient (bottom left), edge density (top right), and node features (bottom right).

4.4 Results

For all datasets, we generate 10,000 samples for evaluation. For the Waxman random graph and protein datasets, we compared Adj GAN, UL GAN and UL VAE with our implementations of the following baseline methods: GraphVAE (Simonovsky and Komodakis, 2018), GraphRNN You et al. (2018b), GCPN (You et al., 2018a), GraphAF (Shi et al., 2020) and SGD-VAE (Guo et al., 2021). Since GraphAF, GraphVAE and GCPN were designed for molecule generation, we adapted them to general graph generation. We remark that GCPN is not able to produce numeric node features and we thus do not report the node feature metrics for it. In order to test the effect of skips connections, we implemented for the random graph dataset a version of UL GAN that has no skip connections from the input vector and we refer to it as UL GAN (NS).

Figure 4 demonstrates the distributions of the four graph properties of both the source data and the different generating methods for the Waxman random graph dataset. For all four properties, the distribution obtained by UL GAN seems to be the closest to the source data. Table 1 reports the 8 evaluation metrics for the Waxman random graph dataset. We note that UL GAN outperforms the other methods in most of the metrics, except for Wasserstein distance between average clustering coefficients, where GraphRNN achieves the smallest metric. Nevertheless, ULGAN achieves the smallest KL divergence and its distribution

Table 1: Results for the Waxman random graph dataset. We report the KL divergence and Wasserstein distance with respect to the following quantities: edge density (kl edge dense and wd edge dense), average node connectivity (kl conn and wd conn), average clustering coefficient (kl clust and wd clust) and node features (kl node feat and wd node feat).

Methods	kl edge dense	kl clust	kl conn	kl node feat	wd edge dense	wd clust	wd conn	wd node feat
GraphVAE	4.009	6.127	2.363	5.088	0.118	0.140	0.315	0.183
GraphRNN	0.592	0.034	0.373	0.717	0.052	**0.019**	0.217	0.135
GCPN	0.402	0.292	0.096	N/A	0.043	0.086	0.116	N/A
GraphAF	0.507	0.249	0.460	0.382	0.032	0.079	0.503	0.163
SGD-VAE	1.493	0.321	0.448	0.378	0.122	0.070	0.294	0.131
ADJ GAN	1.128	1.081	0.247	0.196	0.065	0.097	0.178	0.088
UL VAE	0.092	0.408	0.108	0.635	0.010	0.099	0.105	0.132
UL GAN (NS)	0.100	0.098	0.088	0.147	0.011	0.076	0.046	0.056
UL GAN	**0.001**	**0.023**	**0.034**	**0.145**	**0.004**	0.027	**0.011**	**0.042**

Table 2: Results for the protein dataset. We report the same quantities summarized in the caption of Table 1.

Methods	kl edge dense	kl clust	kl conn	kl node feat	wd edge dense	wd clust	wd conn	wd node feat
GraphVAE	2.497	0.343	3.228	9.777	0.065	0.020	0.758	10.590
GraphRNN	0.082	0.337	0.163	2.818	0.013	0.071	0.165	4.473
GCPN	1.567	4.943	7.358	N/A	0.483	0.508	2.292	N/A
GraphAF	1.942	1.659	1.987	2.603	0.043	0.163	1.543	17.217
SGD-VAE	1.035	1.228	0.975	10.195	0.169	0.311	1.549	10.698
ADJ GAN	6.176	4.600	7.145	0.730	0.086	0.050	0.815	6.705
UL VAE	0.492	0.889	0.791	0.181	0.041	0.072	0.373	4.344
UL GAN	**0.074**	**0.224**	**0.101**	**0.095**	**0.011**	**0.009**	**0.127**	**3.142**

seems to be closer to the source data according to Figure 4. The better performance of UL GAN over Adj GAN and of UL VAE over GraphVAE indicates the clear advantage of using the unpooling layer over a standard adjacency-based method. We further note that the improvement of UL GAN over UL GAN (NS) is not significant. This indicates that even without the skip connection our model performs really well and that its main advantage is due to the unpooling layer and not the skip connection.

Table 2 reports the evaluation metrics for the protein dataset. We note that UL GAN outperforms the other baseline methods in all the metrics. Also, the adjacency matrix-based methods (GraphVAE and Adj GAN) perform poorly in this dataset while their unpooling-layer-based counterparts (UL VAE and UL GAN) perform much better.

For QM9, we compare UL VAE, UL GAN, Adj GAN, MolGAN (De Cao and Kipf, 2018), CharacterVAE (Gómez-Bombarelli et al., 2018), GrammarVAE (Kusner et al., 2017), Graph VAE (Simonovsky and Komodakis, 2018), GraphAF (Shi et al., 2020), GraphDF (Luo et al., 2021), MoFlow (Zang and Wang, 2020), Spanning tree (Ahn et al., 2021), and MGM (Mahmood et al., 2021). For ZINC, we compare UL GAN, Adj GAN, CharacterRNN (Segler et al., 2018), LatentGAN (Prykhodko et al., 2019), junction tree VAE (JT VAE) (Jin et al., 2018), GraphAF (Shi et al., 2020), GraphDF (Luo et al., 2021), MoFlow (Zang and Wang, 2020), and Spanning tree (Ahn et al., 2021). We do not report results of UL VAE for ZINC because its training was slow and we do not have results for its counterpart, GraphVAE. For Adj GAN, UL GAN and UL VAE, we report the means based on 100 runs of generating 10k samples. The results of the other baseline methods are copied from their original papers. Standard deviations for our implementations are included in the supplementary materials.

Table 3 and Table 4 report validity, uniqueness, novelty and their geometric mean for QM9 and ZINC, respectively. For QM9, UL GAN improves significantly from Adj GAN. Its performance is overall competitive when compared to other state-of-the-art approaches. In particular, UL GAN achieves the third-highest geometric mean. UL VAE significantly outperforms its adjacency-matrix-based counterpart, GraphVAE. For ZINC, UL GAN achieves perfect uniqueness and novelty scores. In terms of validity, it outperforms Adj GAN, whose validity and uniqueness scores are poor. We thus note that our unpooling layer is able to generate graphs of moderate sizes, while adjacency-matrix generators are only suitable for small graphs. Although some other methods achieve better validity, the overall performance of UL GAN is comparable with state-of-the-art methods.

Figure 5 studies the latent space structure of UL GAN for QM9. It picks two latent vectors $\boldsymbol{z}_0$ and $\boldsymbol{z}_1$ corresponding to a string-like molecule and a molecule with a ring, respectively. Then, it forms a series of latent vectors $\boldsymbol{z}_t = t\boldsymbol{z}_1 + (1-t)\boldsymbol{z}_0$ and shows their corresponding molecules. We note that as t increases the molecular structures are changing from string-like to ring-like ones.

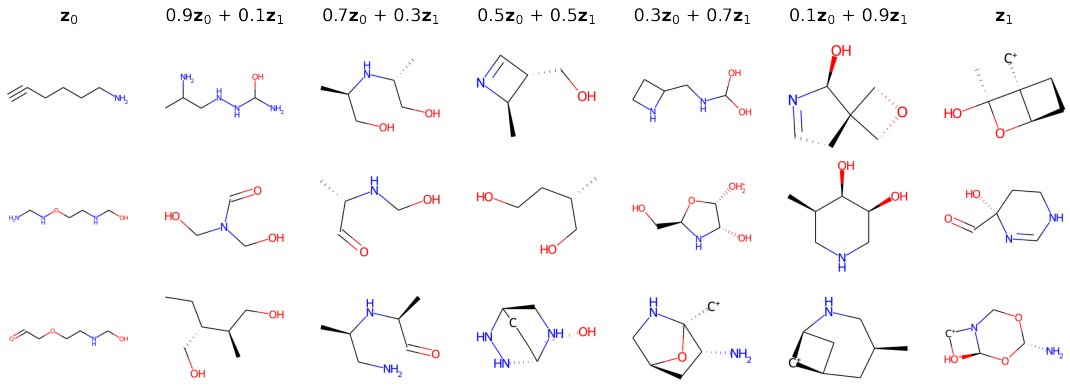

Figure 5: Demonstration of gradual change between two different molecular properties when applying UL GAN for QM9. Left: 3 string-like molecules (with latent vector z_0), Right: 3 molecules with rings (with latent vector z_1). We gradually change the latent vector (on top) and notice the gradual change of the 3 molecules from having string-like structure to having ring-like structure.

Table 3: Validity, uniqueness, novelty and their geometric mean (G-mean) for molecule generation using QM9. Scores for the competing methods (listed above the middle line) were copied from their original papers.

Method	Valid	Unique	Novel	G-mean
CharacterVAE	0.103	0.675	0.900	0.397
GrammarVAE	0.602	0.093	0.809	0.356
GraphVAE	0.557	0.760	0.616	0.639
MolGAN	0.981	0.104	0.942	0.458
GraphAF	0.670	0.945	0.888	0.825
GraphDF	0.827	0.976	0.981	0.925
MoFlow	0.962	**0.992**	**0.980**	**0.978**
Spanning tree	**1.00**	0.968	0.727	0.889
MGM	0.886	0.978	0.518	0.766
UL VAE	0.735	0.940	0.949	0.869
Adj GAN	0.941	0.139	0.886	0.488
UL GAN	0.907	0.826	0.949	0.893

Table 4: Validity, uniqueness, novelty and their geometric mean (G-mean) for molecule generation using ZINC. Scores for the first three methods were copied from Polykovskiy et al. (2020) and scores for other competing methods (listed above the middle line) were copied from their original papers.

Method	Valid	Unique	Novel	G-mean
CharRNN	0.975	**1.00**	0.842	0.936
LatentGAN	0.897	0.997	0.950	0.947
JT VAE	**1.00**	**1.00**	0.914	0.970
GraphAF	0.680	0.991	**1.00**	0.877
GraphDF	0.890	0.992	**1.00**	0.959
MoFlow	0.818	**1.00**	**1.00**	0.935
Spanning tree	0.995	**1.00**	0.999	**0.998**
Adj GAN	0.109	0.196	**1.00**	0.277
UL GAN	0.871	**1.00**	**1.00**	0.955

5 CONCLUSION AND LIMITATIONS

We introduced a novel unpooling layer that can enlarge a given graph. We have proved that this unpooling layer generates connected graphs and its range covers all possible connected graphs. We utilized such layers within GANs and VAEs and tested their performance for graph, protein and molecule generation. Our unpooling-based generation outperforms other methods for graph and protein generation and is competitive for molecule generation. In particular, a significant improvement was noticed in comparison to other methods that are based on adjacency matrices, such as Adj GAN, MolGAN and Graph VAE.

The unpooling layer can be used for other purposes. In future work, we plan to apply it to generic graph reconstruction, while considering the particular applications of recommender systems and graph anomaly detection. We also plan to apply it to conditional graph generation, that is, generating graphs with some desired properties.

The unpooling layer has some limitations. First, its training requires relatively large computational resources, since it relies on various probabilities in order to determine the graph structure. Second, it becomes difficult to optimize several unpooling layers within the generative model because the log probabilities are added and backpropagated in all the unpooling layers.

Our work has practical applications of societal impact, such as drug discovery. However, our main focus has been on general graph generation and in order to have a stronger impact on drug discovery we need to carefully specialize the method for this purpose. For example, we need to improve the druglikeness, or other desired chemical properties, of the generated molecules.

Acknowledgements

This work was partially supported by NSF award DMS 2124913 and the Kunshan Municipal Government research funding.

References

Ahn, S., Chen, B., Wang, T., and Song, L. (2021). Spanning tree-based graph generation for molecules. In *International Conference on Learning Representations*.

Bongini, P., Bianchini, M., and Scarselli, F. (2021). Molecular generative graph neural networks for drug discovery. *Neurocomputing*, 450:242–252.

Brock, A., Donahue, J., and Simonyan, K. (2019). Large scale GAN training for high fidelity natural image synthesis. *ArXiv*, abs/1809.11096.

Bruna, J., Zaremba, W., Szlam, A., and Lecun, Y. (2014). Spectral networks and locally connected networks on graphs. In *International Conference on Learning Representations (ICLR2014), CBLS, April 2014*.

De Cao, N. and Kipf, T. (2018). MolGAN: An implicit generative model for small molecular graphs. *ICML 2018 workshop on Theoretical Foundations and Applications of Deep Generative Models*.

Defferrard, M., Bresson, X., and Vandergheynst, P. (2016). Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*.

Du, Y., Wang, S., Guo, X., Cao, H., Hu, S., Jiang, J., Varala, A., Angirekula, A., and Zhao, L. (2021). Graphgt: Machine learning datasets for graph generation and transformation. In *NeurIPS 2021*.

Duvenaud, D. K., Maclaurin, D., Iparraguirre, J., Bombarell, R., Hirzel, T., Aspuru-Guzik, A., and Adams, R. P. (2015). Convolutional networks on graphs for learning molecular fingerprints. In Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc.

Ertl, P. and Schuffenhauer, A. (2009). Estimation of synthetic accessibility score of drug-like molecules based on molecular complexity and fragment contributions. *Journal of cheminformatics*, 1(1):1–11.

Faez, F., Ommi, Y., Baghshah, M. S., and Rabiee, H. R. (2021). Deep graph generators: A survey. *IEEE Access*, 9:106675–106702.

Fan, W., Ma, Y., Li, Q., He, Y., Zhao, E., Tang, J., and Yin, D. (2019). Graph neural networks for social recommendation. In *The World Wide Web Conference*, pages 417–426.

Gao, H. and Ji, S. (2019). Graph U-Nets. In *International Conference on Machine Learning (ICML 2019)*, pages 2083–2092. PMLR.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. (2017). Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR.

Gómez-Bombarelli, R., Wei, J. N., Duvenaud, D., Hernández-Lobato, J. M., Sánchez-Lengeling, B., Sheberla, D., Aguilera-Iparraguirre, J., Hirzel, T. D., Adams, R. P., and Aspuru-Guzik, A. (2018). Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 4(2):268–276.

Guimaraes, G. L., Sanchez-Lengeling, B., Outeiral, C., Farias, P. L. C., and Aspuru-Guzik, A. (2017). Objective-reinforced generative adversarial networks (ORGAN) for sequence generation models. *arXiv preprint arXiv:1705.10843*.

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., and Courville, A. (2017). Improved training of Wasserstein GANs. In *Advances in Neural Information Processing Systems*, pages 5769—-5779.

Guo, X., Du, Y., and Zhao, L. (2021). Deep generative models for spatial networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 505–515.

Jin, W., Barzilay, R., and Jaakkola, T. (2018). Junction tree variational autoencoder for molecular graph generation. In *International Conference on Machine Learning*, pages 2323–2332. PMLR.

Jin, W., Yang, K., Barzilay, R., and Jaakkola, T. (2019). Learning multimodal graph-to-graph translation for molecule optimization. In *International Conference on Learning Representations*.

Jo, J., Lee, S., and Hwang, S. J. (2022). Score-based generative modeling of graphs via the system of stochastic differential equations. In *ICML*.

Kipf, T. N. and Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR2017)*.

Kusner, M. J., Paige, B., and Hernández-Lobato, J. M. (2017). Grammar variational autoencoder. In *International Conference on Machine Learning*, pages 1945–1954. PMLR.

Lee, J., Lee, I., and Kang, J. (2019). Self-attention graph pooling. In *International Conference on Machine Learning*, pages 3734–3743. PMLR.

Luo, Y., Yan, K., and Ji, S. (2021). Graphdf: A discrete flow model for molecular graph generation. In *International Conference on Machine Learning*, pages 7192–7203. PMLR.

Ma, Y., Aggarwal, C., Wang, S., and Tang, J. (2019). Graph convolutional networks with eigenpooling. In *KDD 2019 - Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 723–731.

Madhawa, K., Ishiguro, K., Nakago, K., and Abe, M. (2019). Graphnvp: An invertible flow model for generating molecular graphs. *arXiv preprint arXiv:1905.11600*.

Maere, S., Heymans, K., and Kuiper, M. (2005). Bingo: a cytoscape plugin to assess overrepresentation of gene ontology categories in biological networks. *Bioinformatics*, 21(16):3448–3449.

Mahmood, O., Mansimov, E., Bonneau, R., and Cho, K. (2021). Masked graph modeling for molecule generation. *Nature Communications*, 12(1):1–12.

Polykovskiy, D., Zhebrak, A., Sanchez-Lengeling, B., Golovanov, S., Tatanov, O., Belyaev, S., Kurbanov, R., Artamonov, A., Aladinskiy, V., Veselov, M., et al. (2020). Molecular sets (MOSES): a benchmarking platform for molecular generation models. *Frontiers in pharmacology*, 11:1931.

Prykhodko, O., Johansson, S. V., Kotsias, P.-C., Arús-Pous, J., Bjerrum, E. J., Engkvist, O., and Chen, H. (2019). A de novo molecular generation method using latent vector based generative adversarial network. *Journal of Cheminformatics*, 11(1):1–13.

Pu, Y., Gan, Z., Henao, R., Yuan, X., Li, C., Stevens, A., and Carin, L. (2016). Variational autoencoder for deep learning of images, labels and captions. *Advances in neural information processing systems*, 29:2352–2360.

Radford, A., Metz, L., and Chintala, S. (2016). Unsupervised representation learning with deep convolutional generative adversarial networks. In *International Conference on Learning Representations (ICLR2016)*.

Ramakrishnan, R., Dral, P. O., Rupp, M., and Von Lilienfeld, O. A. (2014). Quantum chemistry structures and properties of 134 kilo molecules. *Scientific data*, 1(1):1–7.

Samanta, B., De, A., Jana, G., Gómez, V., Chattaraj, P. K., Ganguly, N., and Gomez-Rodriguez, M. (2020). NeVAE: A deep generative model for molecular graphs. *Journal of machine learning research. 2020 Apr; 21 (114): 1-33*.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2008). The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80.

Segler, M. H., Kogej, T., Tyrchan, C., and Waller, M. P. (2018). Generating focused molecule libraries for drug discovery with recurrent neural networks. *ACS central science*, 4(1):120–131.

Shi, C., Xu, M., Zhu, Z., Zhang, W., Zhang, M., and Tang, J. (2020). Graphaf: a flow-based autoregressive model for molecular graph generation. In *International Conference on Learning Representations*.

Simonovsky, M. and Komodakis, N. (2017). Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3693–3702.

Simonovsky, M. and Komodakis, N. (2018). GraphVAE: Towards generation of small graphs using variational autoencoders. In *International conference on artificial neural networks*, pages 412–422. Springer.

Sterling, T. and Irwin, J. J. (2015). Zinc 15–ligand discovery for everyone. *Journal of chemical information and modeling*, 55(11):2324–2337.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. (2018). Graph attention networks. In *International Conference on Learning Representations (ICLR2018)*.

Waxman, B. (1988). Routing of multipoint connections. *IEEE Journal on Selected Areas in Communications*, 6(9):1617–1622.

Weaver, L. and Tao, N. (2001). The optimal reward baseline for gradient-based reinforcement learning. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, UAI'01, page 538–545, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. (2019). How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR2019)*.

Ying, R., You, J., Morris, C., Ren, X., Hamilton, W. L., and Leskovec, J. (2018). Hierarchical graph representation learning with differentiable pooling. *arXiv preprint arXiv:1806.08804*.

You, J., Liu, B., Ying, Z., Pande, V., and Leskovec, J. (2018a). Graph convolutional policy network for goal-directed molecular graph generation. *Advances in neural information processing systems*, 31.

You, J., Ying, R., Ren, X., Hamilton, W., and Leskovec, J. (2018b). Graphrnn: Generating realistic graphs with deep auto-regressive models. In *International conference on machine learning*, pages 5708–5717. PMLR.

Zang, C. and Wang, F. (2020). Moflow: an invertible flow model for generating molecular graphs. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 617–626.

Zeiler, M. D., Krishnan, D., Taylor, G. W., and Fergus, R. (2010). Deconvolutional networks. In *2010 IEEE Computer Society Conference on computer vision and pattern recognition*, pages 2528–2535. IEEE.

SUPPLEMENTARY MATERIALS

We supplement the main text as follows: §A includes more details on the unpooling layer; §B discusses the tested codes; §C describes implementation details of both the unpooling layer and the generative neural network; §D carefully validates the theory stated in §3; §E reports molecule generation results in a particular setting that aims to optimize specific chemical properties; and §F reports some additional numerical results that supplement the ones in §4.

A ADDITIONAL DETAILS ABOUT THE UNPOOLING LAYER

We describe in more details the unpooling layer mechanism. For completeness, we repeat details explained in §2.1.

Notations. For an unpooling layer U, the input featured graph is denoted by $\mathcal{G} = (V, E, \boldsymbol{X}, \boldsymbol{W})$, which is an undirected graph (V, E) with feature matrices$\boldsymbol{X}$ and $\boldsymbol{W}$ for nodes and edges respectively. More specifically, $V = [N] := \{1, 2, \cdots N\}$ are node indices, E is the edge set whose members are of the form $\{i, j\}$ where $i, j \in V$, and $M = |E|$ is the number of edges in the graph. $\boldsymbol{X} \in \mathbb{R}^{N\times d}$ is the node features matrix whose i-th row $\boldsymbol{x}_i$ is a d-dimensional vector for node i. $\boldsymbol{W} \in \mathbb{R}^{M\times e}$ is the edge features matrix whose rows are $e-$dimensional vectors for edge features and we refer to the feature vector on edge $\{i, j\} \in E$ as $\boldsymbol{w}_{i,j}$.

Detailed Steps of the Unpooling Layer. We provide more details of the unpooling layer according to the following steps. Recall that the unpooling layer enlarges an input featured graph $\mathcal{G} = (V, E, \boldsymbol{X}, \boldsymbol{W})$ to a graph $\mathcal{G}^o = (V^o, E^o, \boldsymbol{Y}, \boldsymbol{U})$, while using the set of unchanged (or stable) nodes, I'_s, the set of unpooled nodes, I'_u, and the set of nodes that may be unpooled, I'_r.

Step 1. Generating nodes and node features. We first probabilistically determine the eventual set of nodes to be unpooled and consequently form the set of nodes in the output graph. We finally construct node features.

Step 1a. Determining nodes to be unpooled. We determine a set of nodes to be unpooled and a set of nodes to stay unchanged based on I'_s, I'_r and I'_u. For each node $i \in I'_r \subset V$, we determine the probability of being unpooled by

$$p_r(\boldsymbol{x}_i) = \text{MLP-}\, R(\boldsymbol{x}_i).$$

A node $i \in I'_r$ is unpooled if a randomly drawn uniform random variable $U_i \sim U[0, 1]$ is smaller than $p_r(\boldsymbol{x}_i)$. We form a set of nodes to be unpooled as

$$I_u = I'_u \cup \{i : i \in I'_r, U_i < p_r(\boldsymbol{x}_i)\}.$$

The rest nodes stay unchanged and form a set $I_s := V \setminus I_u$. The total log probability in this step is

$$\text{logP-R} = \sum_{i\in I'_r} \log\left(p_r(\boldsymbol{x}_i)\mathbb{1}_{U_i<p_r(\boldsymbol{x}_i)} + (1 - p_r(\boldsymbol{x}_i))\mathbb{1}_{U_i\geq p_r(\boldsymbol{x}_i)}\right) \tag{4}$$

We remark that it is possible to have $I'_r = \emptyset$ (e.g. the unpooling layers used for the protein dataset), thus all the unpooling nodes are pre-determined. In this case, we obtain $I_u := I'_u$ and $I_s = I'_s$ without using MLP- R and we do not establish the log probability logP-R.

Step 1b. Unpooling nodes and constructing node features. From the previous step, we obtain I_s, a set of nodes to stay unchanged, and I_u, a set of nodes to be unpooled. We thus clarified the set V^o, where we note that $|V^o| = |I_s| + 2|I_u|$. Given two children nodes in V^o, we refer to the "subdivided" node in V as their parent node. We define a map f that assigns to each node in $i \in I_s$ the corresponding index, $f(i) \in V^o$, and to each node in $i \in I_u$ the set of two indices of its children nodes in V^o; in the latter case we denote $f(i) = \{f_1(i), f_2(i)\}$. The feature vectors of the nodes in V^o are obtained by an MLP for vertices, MLP- y. To produce different features for the children nodes $f_1(i)$ and $f_2(i)$, we apply MLP-y to two different pieces of the node feature vector $\boldsymbol{x}_i$. More specifically, we use the orthogonal projectors, P_{S_1} and P_{S_2}, onto two fixed subspaces, S_1 and S_2 in $\mathbb{R}^d$ (which are later specified in §C), as follows:

$$\begin{aligned} \boldsymbol{y}_{f(i)} &= \text{MLP-}y(P_{S_1}\boldsymbol{x}_i), && \text{for } i \in I_s; \\ \boldsymbol{y}_{f_j(i)} &= \text{MLP-}y(P_{S_j}\boldsymbol{x}_i), && \text{for } j = 1, 2,\ i \in I_u. \end{aligned}$$

One can also use three different MLP-y's for the static nodes and the two different children nodes.

Step 2. Building edges. Next, the unpooling layer takes the generated children nodes in V^o and their node features and builds edges for them. It first builds intra-links, which are edges within the pairs of children nodes; it then builds inter-links, which are edges between pairs of children nodes and images (according to the function f) of neighbors of their parent node (in I_u). At last, a single step ensures the connectivity of the output graph.

We also aggregate all log-probabilities in all three steps and use that in the policy gradient algorithm to train the layer and tune those probabilities for edge construction.

We start the edge generation process with $E^o = \emptyset$.

Step 2a. Building intra-links. We aim to probabilistically sample a set V_c of nodes in $I_u \subset V$ whose two children (in V^0) will be connected to each other by intra-links. We first initialize $V_c = \emptyset$. For each node $j \in I_u$ with feature vector $\boldsymbol{x}_j$ we add node j into V_c with a probability outputted by an MLP for intra-links, $p_c(j) = \text{MLP-IA}(\boldsymbol{x}_j)$. That is, we draw a uniform random variable, $U \sim U[0,1]$, and if $U < p_c(j)$, then $V_c = V_c \cup \{j\}$ and $\{f_1(j), f_2(j)\}$ is an intra-link, which is added to E^o; otherwise, V_c is unchanged and no intra-link is added.

We add all the intra-link to edge set in output graph

$$E^o = E^o \cup \big\{\{f_1(j), f_2(j)\}, \forall j \in V_c\big\}.$$

We track the log probabilities for later use as follows:

$$\text{logP-IA} := \sum_{j \in V_c} \ln(p_c(j)) + \sum_{j \notin V_c} \ln(1 - p_c(j)). \tag{5}$$

Step 2b. Finding a shared node for disconnected children. For disconnected children pairs, we designate a node that will connect to these children. In the next step these nodes and possibly additional ones will be connected to a subset of the children pairs. For each node $j \in I_u \setminus V_c$, we denote the set of all edges in E that are connected to j by $E_j = \big\{\{i_1, j\}, ...\{i_{m_j}, j\}\big\}$, and we calculate probabilities of selecting those edges as

$$\begin{aligned}\big(p_b(j,i_1), p_b(j,i_2), ...p_b(j,i_{m_j})\big) \propto \big(&\text{MLP-C}(\boldsymbol{y}_{f_1(j)}, \boldsymbol{y}_{f_2(j)}, \boldsymbol{w}_{\{j,i_1\}}, \boldsymbol{x}_{i_1}),\\ &\text{MLP-C}(\boldsymbol{y}_{f_1(j)}, \boldsymbol{y}_{f_2(j)}, \boldsymbol{w}_{\{j,i_2\}}, \boldsymbol{x}_{i_2}), ..., \text{MLP-C}(\boldsymbol{y}_{f_1(j)}, \boldsymbol{y}_{f_2(j)}, \boldsymbol{w}_{\{j,i_{m_j}\}}, \boldsymbol{x}_{i_{m_j}})\big).\end{aligned}$$

We draw from uniform distribution $U \sim U[0,1]$ and let

$$b_j = \begin{cases} i_1, & \text{if } U < p_b(j,i_1);\\ i_k, & \text{if } U \in \Big[\sum_{l=1}^{k-1} p_b(j,i_l), \sum_{l=1}^{k} p_b(j,i_l)\Big);\\ i_{m_j}, & \text{otherwise.}\end{cases}$$

Then, we define $N_{\{b_j,j\}}(j) := \{f_1(j), f_2(j)\}$.

In the next step, each edge $\{i,j\}$ in E gives rise to adding edges between $N_{\{i,j\}}(i)$ and $N_{\{i,j\}}(j)$ in $\mathcal{G}^o$. Therefore, our approach ensures that $f_1(j)$ and $f_2(j)$ both connect to $N_{\{b_j,j\}}(b_j)$ so $\mathcal{G}^o$ is connected.

We track the log-probabilities from this step that ensures connectivity as follows:

$$\text{logP-C} := \sum_{j \in I_u, j \notin V_c} \ln(p_b(j, b_j)) \tag{6}$$

Step 2c. Building inter-links. For each edge $\{i,j\} \in E$ and node j, we first generate a set of nodes $N_{\{i,j\}}(j) \subset V^o$ as follows. For $j \in I_u \setminus V_c$ and $i = b_j$, we have already defined $N_{\{i,j\}}(j)$ in the above step. If $j \in I_s$, we let $N_{\{i,j\}}(j) = \{f(j)\}$. If $j \in I_u$, we first calculate probabilities from an MLP of intra-links as follows: $(p_1(i,j), p_2(i,j)) = \text{MLP-IE}(\boldsymbol{y}_{f_1(j)}, \boldsymbol{y}_{f_2(j)}, \boldsymbol{w}_{\{i,j\}}, \boldsymbol{x}_i)$. Then, we draw a random variable $U \sim U(0,1)$ and let

$$N_{\{i,j\}}(j) = \begin{cases} \{f_1(j)\}, & \text{if } U < p_1(i,j);\\ \{f_1(j), f_2(j)\}, & \text{if } U \geq p_1(i,j) + p_2(i,j);\\ \{f_2(j)\}, & \text{otherwise.}\end{cases}$$

We similarly form $N_{\{i,j\}}(i)$ by swapping the j and i nodes.

The log probability for each edge-node pair $(\{i,j\}, j)$ is

$$\begin{aligned}\text{logP-IE}(\{i,j\}, j) := \Big(&\mathbb{1}_{N_{\{i,j\}}(j) = \{f_1(j)\}} \ln(p_1(i,j)) + \mathbb{1}_{N_{\{i,j\}}(j) = \{f_2(j)\}} \ln(p_2(i,j))\\ &+ \mathbb{1}_{N_{\{i,j\}}(j) = \{f_1(j), f_2(j)\}} \ln(1 - p_1(i,j) - p_2(i,j))\Big).\end{aligned} \tag{7}$$

Lastly, we add to E^o all inter-links, i.e., all possible edges that connect nodes in $N_{\{i,j\}}(i)$ and $N_{\{i,j\}}(j)$:

$$E^o = E^o \cup \big\{\{k,l\},\ \forall k \in N_{\{i,j\}}(i),\ l \in N_{\{i,j\}}(j)\big\}. \tag{8}$$

Let $A_j := \{j \in V_c\}$ and $A_{i,j} := \{j \in I_u,\ j \notin V_c,\ i \neq b_j\}$. The cumulative log-probability for inter-links outputted by MLP-IE is

$$\text{logP-IE} = \sum_{\{i,j\} \in E} \mathbb{1}_{A_j \cup A_{i,j}} \text{logP-IE}(\{i,j\}, j) + \mathbb{1}_{A_i \cup A_{j,i}} \text{logP-IE}(\{i,j\}, i). \tag{9}$$

Step 2d. Building additional edges between children node pairs. We insert additional edges between children nodes in order to assure the expressivity of the output graph (this will be clarified in the proof of Theorem 2). Let $E_u = \{\{i,j\} \in E : i \in I_u, j \in I_u\} \subset E$ denote the collection of edges whose two ends are both unpooling nodes. We initialize $E_a := \emptyset$. For each edge $\{i,j\} \in E_u$, we generate a probability of adding one more additional edge using an MLP as follows: $p_a(\{i,j\}) = \text{MLP-IE-A}(\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{w}_{\{i,j\}})$. We draw a random variable $U_1 \sim U[0,1]$ and if $U_1 < p_a(\{i,j\})$, we let $E_a :=$

$E_a \cup \{\{i,j\}\}$ and $E^o := E^o \cup E_a^o$, where E_a^o defined in the following three cases: (1) if $|N_{\{i,j\}}(i)| = |N_{\{i,j\}}(j)| = 1$, then $E_a^o(\{i,j\}) := \{\{k,l\}, k \in f(i), l \in f(j),$ and $k \notin N_{\{i,j\}}(i), l \notin N_{\{i,j\}}(j)\}$; (2) if $|N_{\{i,j\}}(i)| + |N_{\{i,j\}}(j)| = 3$, without loss of generality, assume $|N_{\{i,j\}}(i)| = 1$ and $|N_{\{i,j\}}(j)| = 2$, then draw $U_2 \sim U[0,1]$. If $U_2 < \frac{p_1(i,j)}{p_1(i,j)+p_2(i,j)}$ ($p_1(i,j)$ and $p_2(i,j)$ were obtained in Step 3c), we set $r_{ij}(j) := 1$, otherwise $r_{ij}(j) := 2$. We then define $E_a^o(\{i,j\}) := \{\{k, f_{r_{ij}(j)}(j)\}, k \in f(i),$ and $k \notin N_{\{i,j\}}(i)\}$; (3) if $|N_{\{i,j\}}(i)| + |N_{\{i,j\}}(j)| = 4$, then $E_a^o(\{i,j\}) := \emptyset$. We thus updated the edge set of the output graph as follows

$$E^o := E^o \bigcup \bigcup_{\{i,j\} \in E_a} E_a^o(\{i,j\}).$$

We also record the total log probability of this step:

$$\begin{aligned} \text{logP-A} = &\sum_{\{i,j\} \in E_a} \ln p_a(\{i,j\}) + \sum_{\{i,j\} \notin E_a, \{i,j\} \in E_c} \ln(1 - p_a(\{i,j\})) \\ &+ \sum_{\substack{\{i,j\} \in E_a, \\ |N_{\{i,j\}}(i)|=1, \\ |N_{\{i,j\}}(j)|=2}} \ln \frac{p_{r_{ij}(j)}(i,j)}{p_1(i,j) + p_2(i,j)}. \end{aligned} \tag{10}$$

Summary of step 2. The edges of the output graph include all intra-links and inter-links as follows:

$$\begin{aligned} E^o = &\Big\{\{f_1(j), f_2(j)\} : j \in I_u, j \in V_c\Big\} \\ &\bigcup \Big\{\{k,l\} : \{i,j\} \in E, k \in N_{\{i,j\}}(i), l \in N_{\{i,j\}}(j)\Big\} \\ &\bigcup \bigcup_{\{i,j\} \in E_a} E_a^o(\{i,j\}). \end{aligned}$$

Step 3. Constructing edge features. Using the node features and edges from obtained from the previous steps, for each edge $\{k,l\} \in E^o$, we build the corresponding edge feature $\boldsymbol{u}_{k,l} = \text{MLP-}u(\boldsymbol{y}_k, \boldsymbol{y}_l)$.

The overall probability for updating the training parameters: The final probability, P, is the product of all probabilities from step 1a and step 2a-2d. We obtain its logarithm, logP, by combining (4) - (7) and (9) as follows:

$$\text{logP} = \text{logP-R} + \text{logP-IA} + \text{logP-C} + \text{logP-IE} + \text{logP-A}.$$

This probability is used to update the training parameters in the unpooling layer while using the REINFORCE algorithm.

B COMMENTS ON THE CODES

We included a zipped folder of codes that implement the proposed method and some baseline methods. It also contains notebooks that implement numerical experiments for the Waxman random graph, protein, QM9, and ZINC datasets. All the implemented codes can be found in `https://github.com/guo00413/graph_unpooling`.

We implemented five baseline methods in the numerical experiments for the Waxman random graph and the protein dataset: GraphVAE (Simonovsky and Komodakis, 2018), GraphRNN (You et al., 2018b), GCPN (You et al., 2018a), GraphAF (Shi et al., 2020) and SGD-VAE (Guo et al., 2021). We implement GraphVAE and GraphRNN based on codes from `https://github.com/JiaxuanYou/graph-generation` (licensed by MIT), we implement GCPN based on codes from `https://github.com/bowenliu16/rl_graph_generation` (licensed by BSD 3-Clause). We implement SGD-VAE and GraphAF from scratch, because we do not find codes with the appropriate license to use and modify. We select model hyperparameters based on their original papers.

We remark that GCPN can not produce numeric node features due to its generation strategy. As a result, we do not report the metric related to node features for GCPN. To clarify, GCPN (You et al., 2018a) enlarges a single node at each time based on the existing graph as follows: it first adds 9 new nodes, corresponding to 9 types of heavy atoms in ZINC dataset; it constructs one edge connecting one of the new nodes to one of the nodes in the existing graph; it further sequentially determines to stop this step or to construct more edges. This approach can naturally handle graphs with categorical node features, but it fails to generate numeric node features. Therefore in the numerical experiments, we only generate a non-featured graph from GCPN and do not report metrics related to node features.

C IMPLEMENTATION DETAILS

Section C.1 introduces layers used in the experiments other than the unpooling layer; §C.2 provides details of the components in the unpooling layer; §C.3 describes the architectures of the discriminators in UL GAN and of the encoders in UL VAE;

§C.4 details the architectures of the generative networks, including generators in UL GAN and decoders in UL VAE; and §C.5 explains how we calculate the reconstruction error for UL VAE.

We remark that all numerical experiments are run in a machine with Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz and NVIDIA TITAN RTX (576 tensor cores, and 24 GB GDDR6 memory).

C.1 Details of Other Neural Layers

Aggregation. We introduce the following aggregation function $\mathrm{agg}(\boldsymbol{x}_1, \boldsymbol{x}_2) = \mathrm{LeakyReLU}(\boldsymbol{x}_1 + \boldsymbol{x}_2)$. It is used when the inputs are order invariant, for example, to produce edge feature based on features of two end nodes.

MLP. We denote a multilayer perceptron by $\mathrm{MLP}[k_0, k_1, k_2, ..k_m]$. It contains $m-1$ hidden blocks, where the i-th block, $i = 1, \ldots, m-1$, cascades a linear layer with input dimension k_{i-1} and output dimension k_i, a batch normalization and a LeakyReLU activation with negative slope 0.05 (the same slope is used for all LeakyReLU activations in our experiments); and an output block which is a linear layer with output dimension k_m.

Initial layer. An initial layer of a generator takes a latent vector $\boldsymbol{z}$ and produces a 3-nodes graph. To define an initial layer, we need to specify the dimension of the input vector, d_{in}, the dimension of the output node feature, d_x, the dimension of the output edge feature, d_w.

The initial layer generates a 3-nodes graph as follows: First, a multilayer perceptron $\text{MLP-INI-V} = \mathrm{MLP}[d_{in}, 6d_x, 3d_x]$ takes a d_{in}-dimensional input $\boldsymbol{z}$ and generates the initial node features for the 3-nodes graph. It further reshapes it to a matrix in $\mathbb{R}^{3\times d_x}$, where d_x is the dimension of the output node features. Second, it calculates the probabilities from $(p_1, p_2, p_3, p_4) = \text{MLP-INI-E}(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3) = \mathrm{softmax}(\mathrm{MLP}[3d, 16, 4])(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3)$, draws a uniform random variable $U \sim U[0,1]$, and determines the initial edge set to be

$$E = \begin{cases} \{\{1,2\},\{1,3\}\}, & \text{if } U < p_1, \\ \{\{1,2\},\{2,3\}\}, & \text{if } p_1 \le U < p_1 + p_2 \\ \{\{1,3\},\{2,3\}\}, & \text{if } p_1 + p_2 \le U < p_1 + p_2 + p_3 \\ \{\{1,2\},\{2,3\},\{1,3\}\}, & \text{otherwise} \end{cases}$$

The log probability from the initial layer is $\ln(p_s)$, where $s \in \{1,2,3,4\}$ corresponding to the determined initial edge set. Third, it constructs edge feature vectors using a multilayer perceptron as follows: $\text{MLP-INI-W}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \mathrm{LeakyReLU}(\mathrm{BN}(\mathrm{MLP}[d, d_w, d_w](\mathrm{agg}(\boldsymbol{x}_i, \boldsymbol{x}_j))))$, for each $\{i,j\} \in E$, where BN is a batch normalization.

Skip connection. The generator of UL GAN adopts a skip connection procedure (Brock et al., 2019) to avoid mode collapse. A skip connection can be specified by the input dimension, d_z, the node feature dimension, d_y, the multiplier for hidden dimension, N_z, and the maximal number of nodes in the output graph, N. The skip connection generates additional node features by

$$\mathrm{LeakyReLU}(\mathrm{BN}(\mathrm{MLP}[d_z, N_z d_y, N d_y](\boldsymbol{z}))).$$

MPNN. Our implemented models use an edge-conditional MPNN (Gilmer et al., 2017; Simonovsky and Komodakis, 2017), which can be characterized by input node feature dimension d_x, input edge feature dimension d_w and output node feature dimension d_y. For a featured graph $(V, E, \boldsymbol{X}, \boldsymbol{W})$ with d_x-dimensional node features and d_w-dimensional edge features, the rows of the output features matrix $\boldsymbol{Y}$ are formed by

$$\boldsymbol{y}_j = \mathrm{LeakyReLU}\left(\mathrm{BN}\left(\boldsymbol{x}_j \boldsymbol{\Theta}_s + \sum_{i \in N_j} \boldsymbol{x}_i H_n(\boldsymbol{w}_{i,j})\right)\right),$$

where N_j is a set of neighbors of j, H_n is a linear layer, which maps an edge feature vector to a matrix with the same dimensions as $\boldsymbol{\Theta}_s$.

Final layer for edge features. The final layer outputs feature vectors for each edge in the output graph, while leveraging the node features of the connecting nodes and the edge features from the last intermediate graph. The parameters of this layer include the dimension of input edge features, d_w, the dimension of input node features, d_x, and the dimension of the output edge features, d_u. It produces the final edge features by

$$\text{MLP-}uf(\boldsymbol{w}_{i,j}, \boldsymbol{x}_i, \boldsymbol{x}_j) = \mathrm{MLP}[d_x + 2d_w, d_u](\boldsymbol{w}_{i,j}, \mathrm{MLP}[d_x, d_w, d_w](\mathrm{agg}(\boldsymbol{x}_i, \boldsymbol{x}_j)), \mathrm{agg}(\boldsymbol{x}_i, \boldsymbol{x}_j)).$$

C.2 Details of the Unpooling Layer

Hyperparameters. Recall from §A that an unpooling layer takes a featured input graph $\mathcal{G} = (V, E, \boldsymbol{X}, \boldsymbol{W})$, with d_x-dimensional node features and d_w-dimensional edge features. This unpooling layer is thus specified using the following hyperparameters:

- I'_s: a set of nodes fixed as static (i.e., they will not be unpooled).

- I_r: a set of nodes that are determined to be static based on the probabilities $p_j^s = \text{MLP-S}(\boldsymbol{x}_j)$ for $j \in I_r$, where MLP-S $= \text{sigmoid}(\text{MLP}[d_x, \lfloor d_x/2 \rfloor, 1])$. The overall static nodes for the unpooling layer are
$$I_s = I_s' \cup \{j : j \in I_r,\, U_j < \text{MLP-S}(\boldsymbol{x}_j)\},$$
where U_j are i.i.d. random variables drawn from a uniform distribution on $U[0, 1]$.
- k_v, d_y: hidden dimension and output dimension of node features. We let $d_x' = \lfloor d_x/2 \rfloor + \lfloor d_x/4 \rfloor$ and
$$\text{MLP-}y = \text{MLP}[d_x', k_v, d_y],$$
so that the child node feature vector is given by $\boldsymbol{y}_{f_j(i)} = \text{MLP-}y(P_{S_j}(\boldsymbol{x}_i))$. The latter projection, P_{S_j}, is defined for $j = 1$, 2, and each feature vector $\boldsymbol{x} = (x_1, ...x_d)$ of a node in I_u as
$$P_{S_1}(\boldsymbol{x}) = (x_1, ..., x_{d_s}, x_{d_s+1}, ...x_{d_s+D})^T,$$
$$P_{S_2}(\boldsymbol{x}) = (x_1, ..., x_{d_s}, x_{d_s+D+1}, ...x_{d_s+2D})^T,$$
where $d_s = \lfloor d_x/2 \rfloor$ and $D = \lfloor \frac{d_x}{4} \rfloor$.
- k_{ia} and k_{ie}: hidden dimensions in MLP-IA and MLP-IE, respectively. The probabilities used for generating intra- and inter-links are given by

$$\text{MLP-IA}(\boldsymbol{x}) = \text{sigmoid}(\text{MLP}[d_y, k_{ia}, 1](\boldsymbol{x})), \tag{11}$$
$$\text{MLP-IE}(\boldsymbol{y}_1, \boldsymbol{y}_2, \boldsymbol{w}, \boldsymbol{x}) = \text{softmax}\Big(\text{MLP-IE-1}(\boldsymbol{y}_1, \boldsymbol{w}, \boldsymbol{x}), \text{MLP-IE-1}(\boldsymbol{y}_2, \boldsymbol{w}, \boldsymbol{x}), \text{MLP-IE-2}(\boldsymbol{y}_1, \boldsymbol{y}_2, \boldsymbol{w}, \boldsymbol{x})\Big), \tag{12}$$

where the first two MLPs used in MLP-IE are the same networks, defined as
$$\text{MLP-IE-1} := \text{MLP}[d_y + d_w + d_x, k_{ie}, 1],$$
and the third term is defined as
$$\text{MLP-IE-2}(\boldsymbol{y}_1, \boldsymbol{y}_2, \boldsymbol{w}, \boldsymbol{x}) := \text{MLP}[d_y + d_w + d_x, k_{ie}, 1](\text{agg}(\boldsymbol{y}_1, \boldsymbol{y}_2), \boldsymbol{w}, \boldsymbol{x}).$$
In practice MLP-C in Step 3b in §A is based on MLP-IE as follows. It uses MLP-IE-2 and calculates
$$h_C(\boldsymbol{y}_1, \boldsymbol{y}_2, \boldsymbol{w}, \boldsymbol{x}) := \text{MLP-IE-2}(\boldsymbol{y}_1, \boldsymbol{y}_2, \boldsymbol{w}, \boldsymbol{x}).$$
The probabilities used in Step 3b in §A are then calculated as

$$\big(p_b(j, i_1), p_b(j, i_2), ...p_b(j, i_{m_j})\big) = \text{softmax}\big(h_C(\boldsymbol{y}_{f_1(j)}, \boldsymbol{y}_{f_2(j)}, \boldsymbol{w}_{\{j,i_1\}}, \boldsymbol{x}_{i_1}), h_C(\boldsymbol{y}_{f_1(j)}, \boldsymbol{y}_{f_2(j)}, \boldsymbol{w}_{\{j,i_2\}}, \boldsymbol{x}_{i_2}), ..., h_C(\boldsymbol{y}_{f_1(j)}, \boldsymbol{y}_{f_2(j)}, \boldsymbol{w}_{\{j,i_{m_j}\}}, \boldsymbol{x}_{i_{m_j}})\big).$$

- k_w, d_u: hidden dimension and output dimension of the edge features. The output edge feature vectors are built as follows:
$$\boldsymbol{u}_{i,j} = \text{MLP-}u(\boldsymbol{y}_i, \boldsymbol{y}_j) = \text{LeakyReLU}(\text{BN}(\text{MLP}[d_y, k_w, d_u](\text{agg}(\boldsymbol{y}_i, \boldsymbol{y}_j)))).$$

Preference score. We further introduce preference score and modify the direct way of generating the probabilities obtained in (12), in order to avoid all the edges being inherited by one single node in a pair of children nodes and thus to ensure that the unpooling layer can produce all possible output graphs with probabilities that are not too small. Denote
$$h_s(\boldsymbol{y}, \boldsymbol{w}, \boldsymbol{x}) := \text{MLP-IE-1}(\boldsymbol{y}, \boldsymbol{w}, \boldsymbol{x}) \text{ and } h_b(\boldsymbol{y}_1, \boldsymbol{y}_2, \boldsymbol{w}, \boldsymbol{x})) := \text{MLP-IE-2}(\boldsymbol{y}, \boldsymbol{w}, \boldsymbol{x}),$$
where the forms of MLP-IE-1 and MLP-IE-2 are defined below (12). For each child node $f_1(j)$ with feature $\boldsymbol{y}$, which is generated from an input node j, we rescale the h_s's to obtain the preference score $h_s^{(p)}$ when $N_{\{k,j\}}(j) = \{f_1(j)\}$ as follows:
$$\Big(\{h_s^{(p)}(\boldsymbol{y}, k) : k \in N_j\}, h_{s,n}^{(p)}(\boldsymbol{y})\Big) := \text{softmax}\Big(\{h_s(\boldsymbol{y}, \boldsymbol{w}_{j,k}, \boldsymbol{x}_k) : k \in N_j\}, h_s^{(0)}(\boldsymbol{y})\Big),$$
where N_j is set of neighbors of node j, and $h_s^{(0)}(\boldsymbol{y})$ is capturing zero-preference (i.e., preference of not connecting any inter-link) and is calculated by a multilayer perceptron MLP$[d_y, 2d_y, 1]$.

Similarly, we also rescale the h_b's to obtain the preference score when $N_{\{k,j\}}(j) = \{f_1(j), f_2(j)\}$. Let $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ be feature vectors of children nodes generated from j. Let $\boldsymbol{x}$ be the feature vector of node j in the input graph. With an added zero preference $h_b^{(0)}(\boldsymbol{x}) = \text{MLP}[d_x, 2d_x, 1](\boldsymbol{x})$, the preference score is calculated as
$$\Big(\{h_b^{(p)}(\boldsymbol{y}_1, \boldsymbol{y}_2, k) : k \in N_j\}, h_{b,n}^{(p)}(\boldsymbol{y}_1, \boldsymbol{y}_2)\Big) := \text{softmax}\left(\{h_b(\boldsymbol{y}_1, \boldsymbol{y}_2, \boldsymbol{w}_{j,k}, \boldsymbol{x}_k) : k \in N_j\}, h_b^{(0)}(\boldsymbol{x})\right).$$

We use the preference scores $h_s^{(p)}$ and $h_b^{(p)}$ instead of MLP-IE to obtain probabilities for building $N_{\{k,j\}}(j)$ for the inter-link in Step 2c in §A as
$$(p_1, p_2) = \left(\frac{(h_s^{(p)}(\boldsymbol{y}_1, k)}{Z}, \frac{h_s^{(p)}(\boldsymbol{y}_1, k)}{Z}\right), \text{ where } Z = \Big(h_s^{(p)}(\boldsymbol{y}_1, k) + h_s^{(p)}(\boldsymbol{y}_2, k) + h_b^{(p)}(\boldsymbol{y}_1, \boldsymbol{y}_2, k)\Big).$$

C.3 Architectures of Discriminators and Encoders

Discriminators of UL GAN. The details of the discriminator in UL GAN and the encoder in UL VAE for QM9 have been described in §4.3. In this part, we describe the details for other datasets.

The discriminator in the Waxman random graph data takes an input graph and uses two MPNN layers with 64 and 128 units to generate a graph $\mathcal{G} = (V, E, \boldsymbol{X}, \boldsymbol{W})$. It then aggregates the node features (the rows $\{\boldsymbol{x}_j\}_{j\in V}$ of $\boldsymbol{X}$) to produce the following single feature vector for the graph:

$$\boldsymbol{h}(\mathcal{G}) = \sum_{j\in V} \sigma(\mathrm{lin}_1(\boldsymbol{x}_j)) \odot \tanh(\mathrm{lin}_2(\boldsymbol{x}_j));$$

where σ is logistic sigmoid, $\odot$ is element-wise multiplication and lin_1 and lin_2 are two layers with 128 units. Then it uses a global sum pooling to extract a signal for each graph. It applies two linear layers with 128 and 256 units. A final layer with a single unit then produces the output, where its activation function is sigmoid. The LeakyReLU activation function (with 0.05 negative slope) is used after the two MPNNs and all linear layers.

The discriminator in the protein dataset is similar to the discriminator in the random graph dataset. Some parameters were changed to simplify the network that fit the protein dataset; we used one MPNN layer with 128 units and one linear layer with 256 units after the global pooling.

The discriminator in ZINC is the same as the discriminator used in QM9 as described in §4.3.

Encoder in UL VAE. It has the same architecture as the corresponding discriminator described above, except that the final layer maps into a 256-dimensional vector with a linear activation function. This vector further splits to two 128-dimensional vectors: $\boldsymbol{z}_\mu$ and $\boldsymbol{z}_\sigma$. It then generates the following latent vector: $\boldsymbol{z} = \boldsymbol{z}_\mu + \exp(\frac{1}{2}\boldsymbol{z}_\sigma) \odot \boldsymbol{x}$, where $\boldsymbol{x} \sim N(0, 1)$.

Additional adjustment when the number of nodes of the output graph is not fixed. For the ZINC dataset, since the number of nodes of the output graph is not fixed by the generator, we added one trivial predictive network to the discriminator. This additional network contains one hidden layer with 256 units and only takes the global sum of node feature vectors and edge feature vectors as input, to encourage a proper distribution of the numbers of atoms and edges. The final output of the discriminator is the sum of the outputs of this network and the original GNN discriminator.

C.4 Architecture of Generators

We describe details of the architecture of the generator in UL GAN and decoder in UL VAE for the different datasets.

Architecture of the generator in UL GAN. We show the generators in UL GAN (same as the decoders in UL VAE, if applicable) for the Waxman random graph, protein, QM9 and ZINC datasets in Figure 6, Figure 7, Figure 8, and Figure 9, respectively. For simplicity, we neglect batch sizes. For example, the input should be $\boldsymbol{z} \in \mathbb{R}^{B\times 128}$, where B is the batch size.

Loss function. In practice, we alternatively train the generator with a GAN's loss function and with REINFORCE. In each training step, we first update the generator using the loss function $-D(G(z))$. We then update the parameter of the generator, θ_G, using

$$\boldsymbol{\theta}_G^{(t+1)} := \boldsymbol{\theta}_G^{(t)} + \alpha\Big(\nabla_{\boldsymbol{\theta}} \mathrm{logP}|_{\boldsymbol{\theta}_G^{(t)}}\Big)(D(G(z)) - \mathbb{E}(D(G(z)))),$$

where the learning rate $\alpha = 5 \times 10^{-2}$.

C.5 Reconstruction Loss for UL VAE

We calculate the reconstruction loss for UL VAE following Simonovsky and Komodakis (2018).

Given two featured graphs with the same number of nodes, we calculate their distance by summing the distances between node features, the distances between adjacency matrices and the distances between edge features of these two graphs. We used the same formula described in Section 3.3 in Simonovsky and Komodakis (2018).

When the number of nodes may vary (e.g., in the QM9 dataset, the corresponding graphs may contain 6–9 nodes) and the input/output graph has less nodes than the maximal number of nodes (e.g., less than 9 nodes in the QM9 experiment), we add artificial nodes to the graph so that it contains the maximal number of nodes (e.g., 9 nodes for QM9). We add an additional binary feature to the node features: it assigns 0 to the real nodes and 1 to the artificial nodes.

For an input graph and an output graph with the same numbers of nodes (if they have different node numbers, we perform the above procedure to make them the same), we permute the node indices of the output graph in order to achieve a minimal distance to the input graph. This minimal distance is regarded as the reconstruction loss between the input and output graph. More details of how to permute the indices appear in Section 3.4 of Simonovsky and Komodakis (2018).

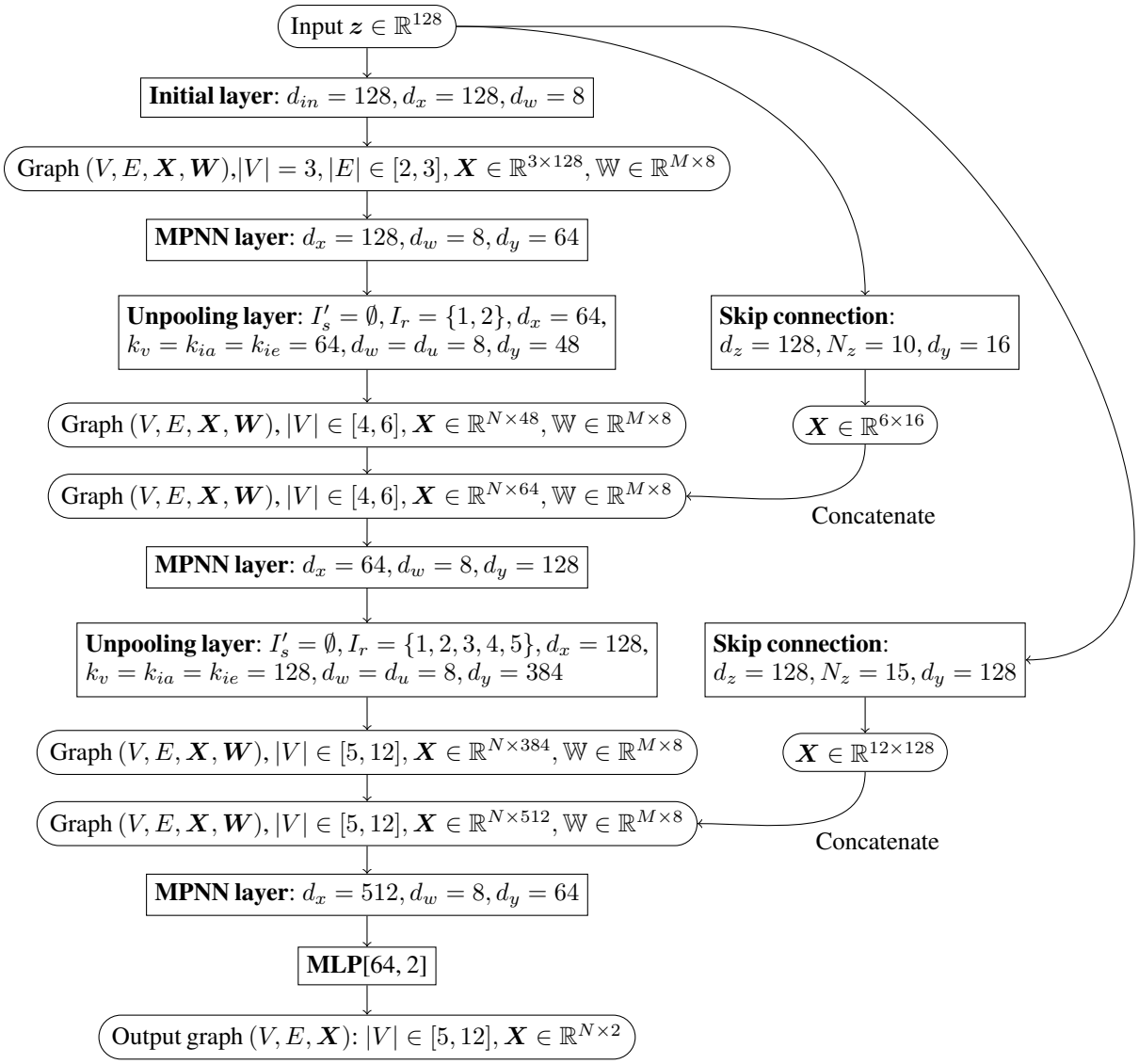

Figure 6: Demonstration of the architecture of the generator in UL GAN and the decoder in UL VAE for Waxman random graph dataset. The rectangles present neural network layers and the rounded rectangles clarify the shape of the hidden data.

D PROOF OF THEOREMS

In this section, we prove claims on connectivity and expressivity for the output graph of our unpooling layer. For the ease of presenting, we denote a graph by $\mathcal{G} = (V, E)$ and omit features in nodes and edges because features are irrelevant with graph connectivity and expressivity.

D.1 Proof of Proposition 1

Let k^o, $l^o \in V^o$ be two arbitrary nodes in the output graph $\mathcal{G}^o = (V^o, E^o)$. In order to prove the connectivity of $\mathcal{G}^o$ we need to find a path in $\mathcal{G}^o$ connecting k^o and l^o. Recall that $\mathcal{G} = (V, E)$ denotes the input graph. Let $k, l \in V$ be the parent nodes of k^o and l^o, respectively. Since the input graph $\mathcal{G}$ is connected, k and l are connected by a path. We denote its length by $n - 1$, where $n \geq 2$, and its edges by $\{i_1, i_2\}$, $\{i_2, i_3\}$, ...$\{i_{n-1}, i_n\}$, where $i_1 = k$ and $i_n = l$. Recall that for $r \in [n]$ $N_{\{i_r, i_{r+1}\}}(i_r)$ is a subset of the children of node i_r connected to $N_{\{i_r, i_{r+1}\}}(i_{r+1})$, which is a subset of the children of node i_{r+1}. For $r \in [n]$ we arbitrarily choose a vertex in $N_{\{i_r, i_{r+1}\}}(i_r)$ and denote it by $v_{\{i_r, i_{r+1}\}}(i_r)$. We thus note that the following edges exist in the output graph:

$$\left\{v_{\{i_1,i_2\}}(i_1), v_{\{i_1,i_2\}}(i_2)\right\}, \left\{v_{\{i_2,i_3\}}(i_2), v_{\{i_2,i_3\}}(i_3)\right\}, \cdots \left\{v_{\{i_{n-1},i_n\}}(i_{n-1}), v_{\{i_{n-1},i_n\}}(i_n)\right\}.$$

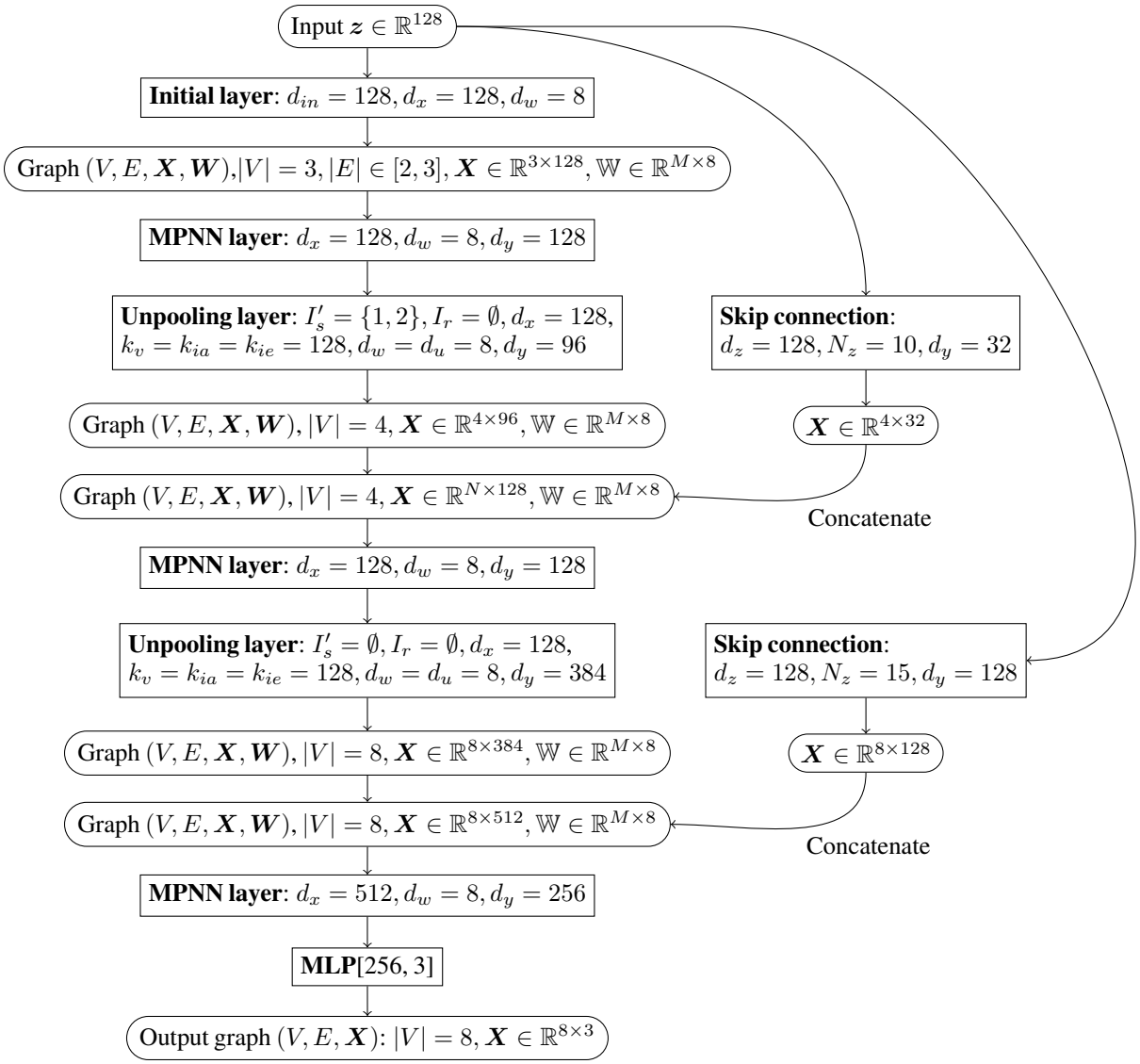

Figure 7: Demonstration of the architecture of the generator in UL GAN and the decoder in UL VAE for protein dataset. The rectangles present neural network layers and the rounded rectangles clarify the shape of the hidden data.

In order to prove that k^o and l^o connect by a path we verify the following properties:

1. For all $r \in [n-2]$, either $v_{\{i_r,i_{r+1}\}}(i_{r+1}) = v_{\{i_{r+1},i_{r+2}\}}(i_{r+1})$ or there is a path connecting $v_{\{i_r,i_{r+1}\}}(i_{r+1})$ and $v_{\{i_{r+1},i_{r+2}\}}(i_{r+1})$
2. Either $k^o = v_{\{i_1,i_2\}}(i_1)$ or there is a path connecting k^o and $v_{\{i_1,i_2\}}(i_1)$
3. Either $l^o = v_{\{i_{n-1},i_n\}}(i_n)$ or there is a path connecting l^o and $v_{\{i_{n-1},i_n\}}(i_n)$

To prove the first property we note that $v_{\{i_r,i_{r+1}\}}(i_{r+1}) \in \{f_1(i_{r+1}), f_2(i_{r+1})\}$ and $v_{\{i_{r+1},i_{r+2}\}}(i_{r+1}) \in \{f_1(i_{r+1}), f_2(i_{r+1})\}$. If $v_{\{i_r,i_{r+1}\}}(i_{r+1}) \neq v_{\{i_{r+2},i_{r+1}\}}(i_{r+1})$, then $\{v_{\{i_r,i_{r+1}\}}(i_{r+1}), v_{\{i_{r+1},i_{r+2}\}}(i_{r+1})\} = \{f_1(i_{r+1}), f_2(i_{r+1})\}$. To show that there is a path connecting $v_{\{i_r,i_{r+1}\}}(i_{r+1})$ and $v_{\{i_{r+1},i_{r+2}\}}(i_{r+1})$, it suffices to show that there is a path connecting $f_1(i_{r+1})$ and $f_2(i_{r+1})$. Assume an arbitrary node $j \in V$. If $j \in V_c$ the construction in Step 3a guarantees the existence of an edge connecting $f_1(j)$ and $f_2(j)$. Otherwise, based on Step 3b, there exists a node b_j such that $N_{\{b_j,j\}}(j) = \{f_1(j), f_2(j)\}$, so that $f_1(j)$ and $f_2(j)$ are connected via $f(b_j)$. Letting $j = i_{r+1}$ we conclude this property.

The above argument also applies to the other two properties. $\square$

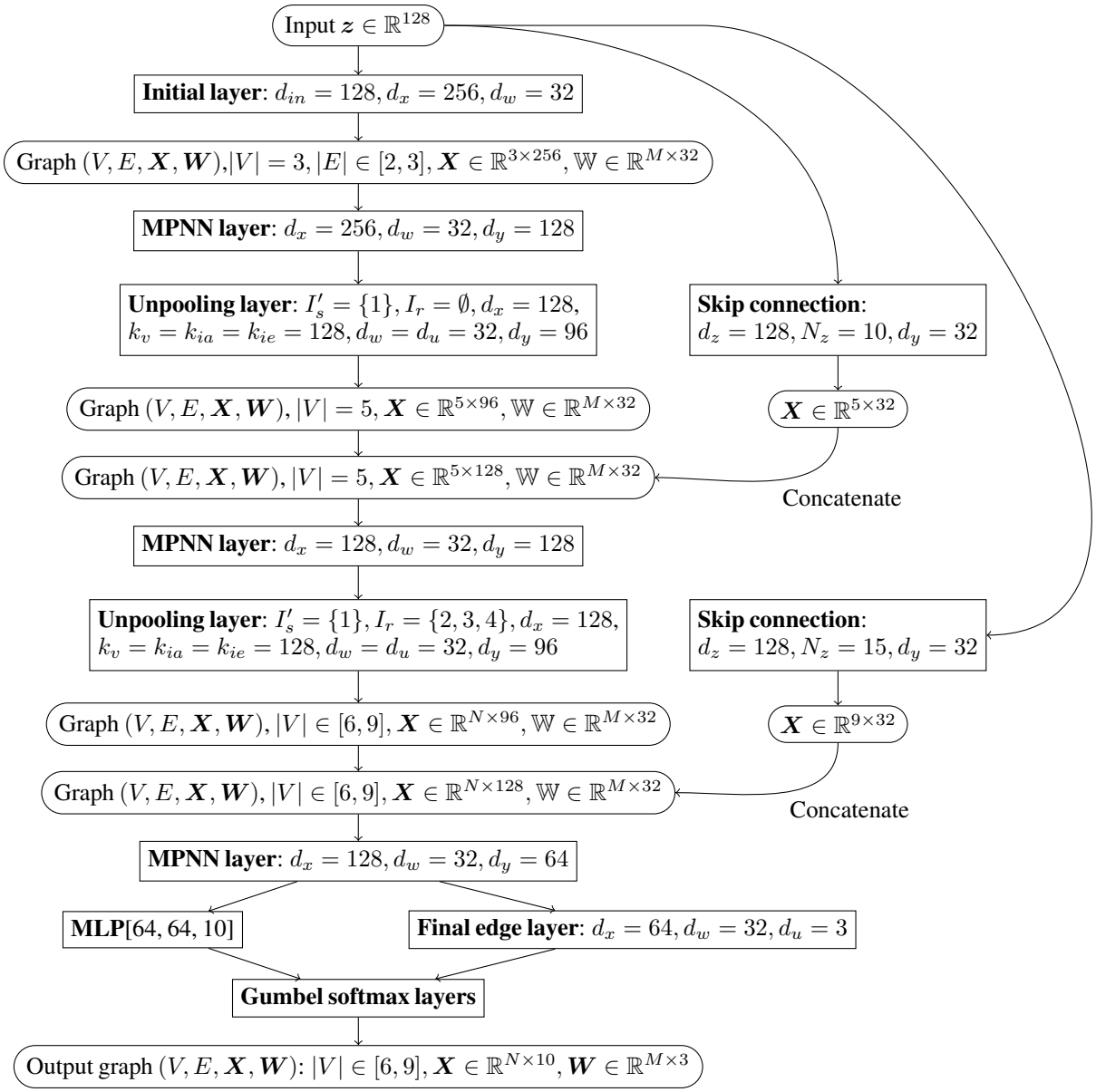

Figure 8: Demonstration of the architecture of the generator in UL GAN and the decoder in UL VAE for QM9. The rectangles present neural network layers and the rounded rectangles clarify the shape of the hidden data.

D.2 Proof of Theorem 2

We will first define a pooling process and show that there exists an unpooling layer that acts as an inverse of this pooling procedure (see Lemma D.1). We then show that any graph with N nodes can be pooled to a graph with $\lceil N/2 \rceil$ nodes (Lemma D.2 clarifies the case where N is even and Lemma D.3 clarifies the cases where N is odd). Finally we use these observations to conclude the proof of the theorem.

We define the pooling process by using an "eligible" set. For a graph $\mathcal{G}^o = (V^o, E^o)$, a set of pairs of nodes in V^o, $S = \{(i_1, j_1), ...(i_n, j_n)\}$, is called *eligible* if all nodes $i_1 \cdots i_n$ and $j_1 \cdots j_n$ are distinct and for any $m \in [n]$, i_m and j_m are connected by a path of length at most 2; that is, there are two cases: either $\{i_m, j_m\} \in E^o$ or there exists $k \in V^o$, such that $\{i_m, k\} \in E^o$ and $\{k, j_m\} \in E^o$. Using an arbitrarily chosen eligible set S, we describe a specific pooling process with respect to S that produces a graph $\mathcal{G}$ from $\mathcal{G}^o$ as follows. We initialize $\mathcal{G}$ with $V = V^o$ and $E = E^o$. For $m = 1, 2, \cdots n$, we follow the next three steps: (1) We remove from V the nodes i_m and j_m. We remove from E all edges in $E_m := \{e \in E : i_m \in e \text{ or } j_m \in e\}$; (2) We add a new node i'_m to V; (3) We add the following set of new edges to E:

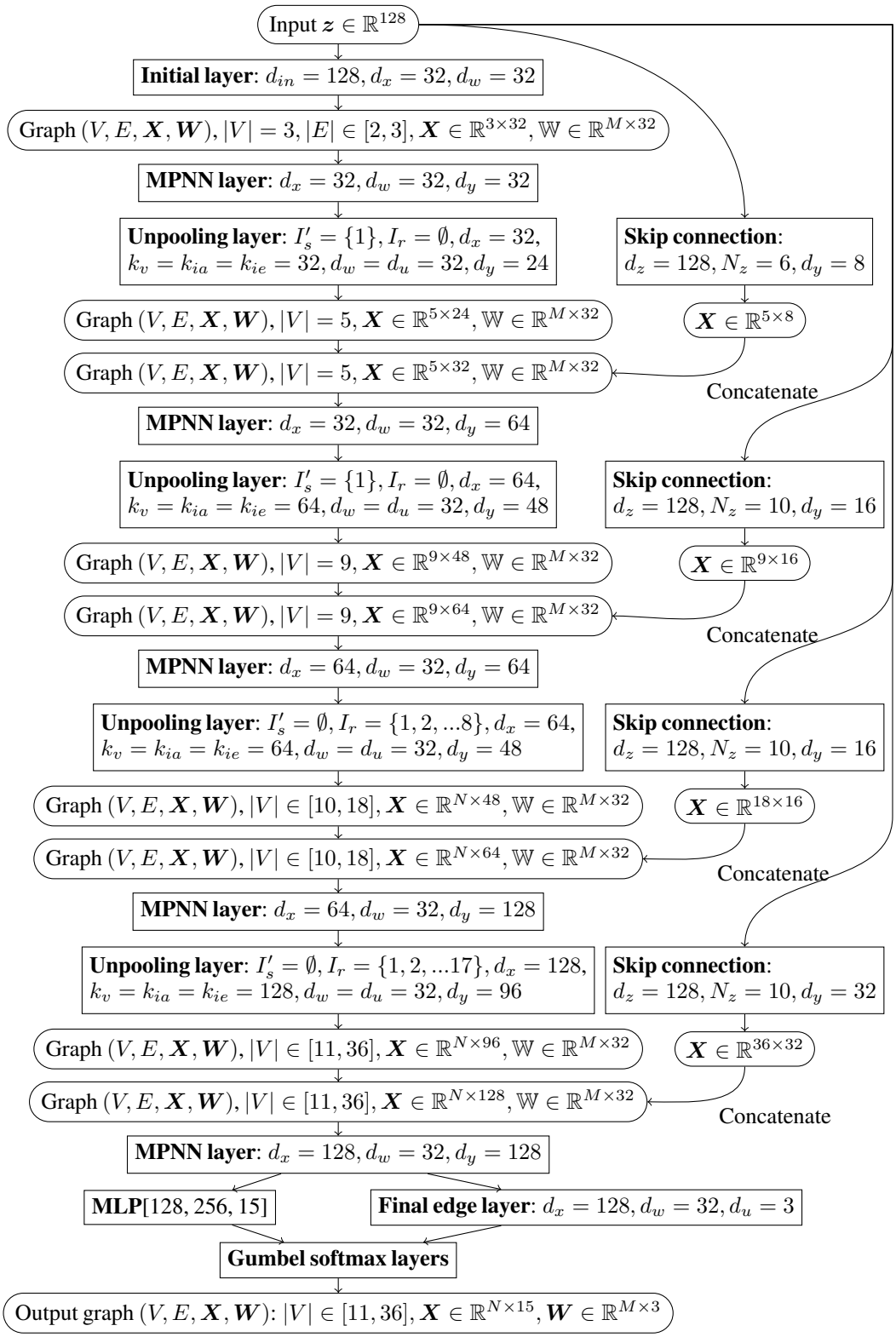

Figure 9: Demonstration of the architecture of the generator in UL GAN for ZINC. The rectangles present neural network layers and the rounded rectangles clarify the shape of the hidden data.

$\{\{i'_m, k\} : k \in V$ and either $\{i_m, k\} \in E_m$ or $\{j_m, k\} \in E_m\}$. It is clear that the resulting $\mathcal{G}$ is connected if $\mathcal{G}^o$ is connected.

We introduce a lemma showing that for a given pooling process which maps $\mathcal{G}^o$ to $\mathcal{G}$, there exists an unpooling layer as the inverse of this pooling process, i.e., it maps $\mathcal{G}$ to $\mathcal{G}^o$.

Lemma D.1. *For any pooling process that maps $\mathcal{G}^o = (V^o, E^o)$ to $\mathcal{G} = (V, E)$, there exists an unpooling layer that maps $\mathcal{G}$ to $\mathcal{G}^o$.*

Proof. Consider the pooling process with respect to the eligible pairs in $S = \{(i_1, j_1), ...(i_n, j_n)\} \subset V^o$. We use the same notation as above for $i'_1, i'_2, ...i'_n \subset V$ that were pooled by the respective eligible pairs.

We construct an unpooling layer whose input is $\mathcal{G}$ and its output is $\hat{\mathcal{G}}^o = (\hat{V}^o, \hat{E}^o)$. The unpooling layer unpools the nodes $i'_1, i'_2, ...i'_n$ and keeps the remaining nodes unchanged. It forms the following children nodes of $i'_1, i'_2, ...i'_n$ in $\hat{V}^o$: $(f_1(i'_1), f_2(i'_1)), ...(f_1(i'_n), f_2(i'_n))$, respectively. For every $r \in [n]$, we identify $(f_1(i'_r), f_2(i'_r))$ with (i_r, j_r) and re-index respectively, so $\hat{V}^o = V^o$.

It remains to show that we can find an unpooling layer such that $\hat{E}^o = E^o$. We first note that the edges in E^o that do not contain nodes from the eligible set remain unchanged in E and E^o since the pooling process is identical on those edges (for clarity, these edges are the ones in $\{\{i, j\} \in E^o : i, j \notin \{i_1, i_2, ...i_n, j_1, j_2, ...j_n\}\}$). Since we restricted above the unpooling layer to only unpool $i'_1, i'_2, ...i'_n$ those edges also remain unchanged in $\hat{E}^o$. We thus only need to show that we can construct the unpooling layer so that the set of edges in $\hat{E}^o$ that contain children nodes is the set of edges that contain nodes from the eligible set in E^o. Each edge in the latter set falls into one of the following three categories, for which we establish the required equality with the corresponding edges in $\hat{E}^o$:

1. Edges whose end nodes form a pair $\{i_r, j_r\} \in S$. For each such edge, we select the intra-link in the unpooling layer (step 3a) so that there is an edge connecting $f_1(i'_r)$ and $f_2(i'_r)$ in $\hat{E}^o$.
2. Edges between an eligible pair (i_r, j_r) and a static node k in V^o. That is, for a fixed $r \in [n]$ and a static node k, there three possibilities for the set of these edges: $\{\{i_r, k\}\}$ or $\{\{j_r, k\}\}$ or $\{\{i_r, k\}, \{j_r, k\}\}$. In view of the pooling process, the edge $\{i'_r, k\}$ is in E. In Step 3c, there are three possibilities for determining $N_{\{i'_r,k\}}(i'_r)$ and we need to select the unpooling layer to match these three possibilities. That is, $N_{\{i'_r,k\}}(i'_r) = \{f_1(i'_r)\}$ in the first case, where the set of the above edges is $\{\{i_r, k\}\}$; $N_{\{i'_r,k\}}(i'_r) = \{f_2(i'_r)\}$ in the second case, where the set of the above edges is $\{\{j_r, k\}\}$ and $N_{\{i'_r,k\}}(i'_r) = \{f_1(i'_r), f_2(i'_r)\}$ in the third case, where the set of the above edges is $\{\{i_r, k\}, \{j_r, k\}\}$. In view of the way $N_{\{i'_r,k\}}(i'_r)$ is used to build inter-links (see (8)), the edges in $\hat{E}^o$ between $(f_1(i'_r), f_2(i'_r))$ and k are the same as the ones in E^o between (i_r, j_r) and k.
3. Edges between two different eligible pairs, (i_r, j_r) and (i_s, j_s), that is $\{\{k, l\} \in E^o : k \in \{i_r, j_r\}, l \in \{i_s, j_s\}\}$. For fixed $s, r \in [n]$ this set of edges in E^o is a nonempty subset of the following set of four edges: $\{\{i_r, i_s\}, \{i_r, j_s\}, \{j_r, i_s\}, \{j_r, j_s\}\}$. Therefore, there are $2^4 - 1 = 15$ such edge sets. In view of the pooling process, the edge $\{i'_r, i'_s\}$ is in E. The unpooling layer unpools i'_r to $(f_1(i'_r), f_2(i'_r))$ and unpools i'_s to $(f_1(i'_s), f_2(i'_s))$. We claim that according to Steps 3c and 3d, the unpooling layer can produce all the 15 possible edge sets between the pair $(f_1(i'_r), f_2(i'_r))$ and the pair $(f_1(i'_s), f_2(i'_s))$. To clarify this claim we specify for all 15 possible edges between (i_r, j_r) and (i_s, j_s) the choice of $N_{\{i'_r,i'_s\}}(i'_r), N_{\{i'_r,i'_s\}}(i'_s)$ in Step 3c and the choice of the additional edge in Step 3d:
   - If the edge set is $\{\{i_r, i_s\}\}$, we set $N_{\{i'_r,i'_s\}}(i'_r) = \{f_1(i'_r)\}, N_{\{i'_r,i'_s\}}(i'_s) = \{f_1(i'_s)\}$ and we do not insert an edge in Step 3d
   - If the edge set is $\{\{i_r, j_s\}\}$, we set $N_{\{i'_r,i'_s\}}(i'_r) = \{f_1(i'_r)\}, N_{\{i'_r,i'_s\}}(i'_s) = \{f_2(i'_s)\}$ and we do not insert an edge in Step 3d
   - If the edge set is $\{\{j_r, i_s\}\}$, we set $N_{\{i'_r,i'_s\}}(i'_r) = \{f_2(i'_r)\}, N_{\{i'_r,i'_s\}}(i'_s) = \{f_1(i'_s)\}$ and we do not insert an edge in Step 3d
   - If the edge set is $\{\{j_r, j_s\}\}$, we set $N_{\{i'_r,i'_s\}}(i'_r) = \{f_2(i'_r)\}, N_{\{i'_r,i'_s\}}(i'_s) = \{f_2(i'_s)\}$ and we do not insert an edge in Step 3d
   - If the edge set is $\{\{i_r, i_s\}, \{i_r, j_s\}\}$, we set $N_{\{i'_r,i'_s\}}(i'_r) = \{f_1(i'_r)\}, N_{\{i'_r,i'_s\}}(i'_s) = \{f_1(i'_s), f_2(i'_s)\}$ and we do not insert an edge in Step 3d
   - If the edge set is $\{\{j_r, i_s\}, \{j_r, j_s\}\}$, we set $N_{\{i'_r,i'_s\}}(i'_r) = \{f_2(i'_r)\}, N_{\{i'_r,i'_s\}}(i'_s) = \{f_1(i'_s), f_2(i'_s)\}$ and we do not insert an edge in Step 3d
   - If the edge set is $\{\{i_r, i_s\}, \{j_r, i_s\}\}$, we set $N_{\{i'_r,i'_s\}}(i'_r) = \{f_1(i'_r), f_2(i'_r)\}, N_{\{i'_r,i'_s\}}(i'_s) = \{f_1(i'_s)\}$ and we do not insert an edge in Step 3d
   - If the edge set is $\{\{i_r, j_s\}, \{j_r, j_s\}\}$, we set $N_{\{i'_r,i'_s\}}(i'_r) = \{f_1(i'_r), f_2(i'_r)\}, N_{\{i'_r,i'_s\}}(i'_s) = \{f_2(i'_s)\}$ and we do not insert an edge in Step 3d
   - If the edge set is $\{\{i_r, i_s\}, \{j_r, j_s\}\}$, we set $N_{\{i'_r,i'_s\}}(i'_r) = \{f_1(i'_r)\}, N_{\{i'_r,i'_s\}}(i'_s) = \{f_1(i'_s)\}$ and in Step 3d we insert the edge $\{f_2(i'_r), f_2(i'_s)\}$

- If the edge set is $\{\{i_r, j_s\}, \{j_r, i_s\}\}$, we set $N_{\{i'_r, i'_s\}}(i'_r) = \{f_1(i'_r)\}, N_{\{i'_r, i'_s\}}(i'_s) = \{f_2(i'_s)\}$ and in Step 3d we insert the edge $\{f_2(i'_r), f_1(i'_s)\}$
- If the edge set is $\{\{i_r, i_s\}, \{i_r, j_s\}, \{j_r, i_s\}\}$, we set $N_{\{i'_r, i'_s\}}(i'_r) = \{f_1(i'_r), f_2(i'_r)\}, N_{\{i'_r, i'_s\}}(i'_s) = \{f_1(i'_s)\}$ and in Step 3d we insert the edge $\{f_1(i'_r), f_2(i'_s)\}$
- If the edge set is $\{\{i_r, i_s\}, \{j_r, i_s\}, \{j_r, j_s\}\}$, we set $N_{\{i'_r, i'_s\}}(i'_r) = \{f_1(i'_r), f_2(i'_r)\}, N_{\{i'_r, i'_s\}}(i'_s) = \{f_1(i'_s)\}$ and in Step 3d we insert the edge $\{f_2(i'_r), f_2(i'_s)\}$
- If the edge set is $\{\{i_r, i_s\}, \{i_r, j_s\}, \{j_r, j_s\}\}$, we set $N_{\{i'_r, i'_s\}}(i'_r) = \{f_1(i'_r), f_2(i'_r)\}, N_{\{i'_r, i'_s\}}(i'_s) = \{f_2(i'_s)\}$ and in Step 3d we insert the edge $\{f_1(i'_r), f_1(i'_s)\}$
- If the edge set is $\{\{i_r, j_s\}, \{j_r, i_s\}, \{j_r, j_s\}\}$, we set $N_{\{i'_r, i'_s\}}(i'_r) = \{f_1(i'_r), f_2(i'_r)\}, N_{\{i'_r, i'_s\}}(i'_s) = \{f_2(i'_s)\}$ and in Step 3d we insert the edge $\{f_2(i'_r), f_1(i'_s)\}$
- If the edge set is $\{\{i_r, i_s\}, \{i_r, j_s\}, \{j_r, i_s\}, \{j_r, j_s\}\}$, we set $N_{\{i'_r, i'_s\}}(i'_r) = \{f_1(i'_r), f_2(i'_r)\}, N_{\{i'_r, i'_s\}}(i'_s) = \{f_1(i'_s), f_2(i'_s)\}$ and we do not insert an edge in Step 3d

The inter-link construction in (8) for the above specified choices of $N_{\{i'_r, i'_s\}}(i'_r)$ and $N_{\{i'_r, i'_s\}}(i'_s)$ together with the above specified choices of inserting an additional edge imply that the output edges between $(f_1(i'_r), f_2(i'_r))$ and $(f_1(i'_s), f_2(i'_s))$ in $\hat{E}^o$ are the same as the edges between (i_r, j_r) and (i_s, j_s).

□

To show that for a graph $\mathcal{G}^o = (V^o, E^o)$ with N node there is an unpooling layer that maps a graph $\mathcal{G}$ with $N - n$ nodes to $\mathcal{G}^o$, we need to show that there exists an eligible set $S = \{(i_1, j_1), ...(i_n, j_n)\} \subset V^o$. Indeed, the pooling process with respect to S maps $\mathcal{G}^o$ to a graph $\mathcal{G}$ with $N - n$ nodes and by Lemma D.1 there exists an unpooling layer that maps $\mathcal{G}$ to $\mathcal{G}^o$.

The next two lemmas conclude the theorem by implying that for a connected graph $\mathcal{G}^o$ there exists an eligible set of maximal size, i.e., $n = \lfloor N/2 \rfloor$. The first lemma considers a graph with an even number of nodes and the second lemma considers a graph with an odd number of nodes.

Lemma D.2. *For any connected graph $\mathcal{G}^o = (V^o, E^o)$ with $2K$ nodes, there exists an eligible set S containing K pairs of nodes in $\mathcal{G}^o$ such that the pooling process with respect to S maps $\mathcal{G}^o$ to a graph $\mathcal{G}$ with K nodes.*

Proof. We prove this lemma by induction using $M = 1, \ldots, K$. When $M = 1$, the lemma is trivial.

Assume the lemma holds for $M = 1, \ldots, K - 1$. Given a connected graph $\mathcal{G}^o = (V^o, E^o)$ with $2K$ nodes, we prove the result, while considering the following two different scenarios:

Case 1: There exists a node in V^o of degree 1. We arbitrarily choose such a node and denote it by j. We consider its only neighbor, which we denote $k \in V^o$, and remove the pair of nodes (j, k) and all the edges connected to them from the graph $\mathcal{G}^o$. The remaining graph has $2K-2$ nodes. If it is also connected, then by the induction assumption there exists an eligible set with $K-1$ pairs of nodes. By adding the pair (j, k) to that eligible set, we obtain an eligible set with K pairs and conclude the proof.

If, on the other hand, the remaining graph is not connected, we partition it into maximally connected subgraphs $\mathcal{G}_1, \mathcal{G}_2, \cdots, \mathcal{G}_m$. That is, each subgraph is connected, but any two subgraphs are not connected to each other. Since $\mathcal{G}_1, \ldots \mathcal{G}_m$ are not connected to each other and to the degree-one node j, they all connect to the node k.

We consider the following four steps that assist in finding an eligible set:

1. We identify the maximally connected subgraphs with even numbers of nodes. Clearly, each of these numbers is not greater than $2K - 2$ and thus by induction all the nodes of each such subgraph can be used to form an eligible set. The union of all these eligible sets forms a larger eligible set that uses all the nodes in these subgraphs. If all maximally connected subgraphs have even numbers of nodes, then we terminate the procedure at this step.
2. We re-index the maximally connected subgraphs with odd numbers of nodes as $\mathcal{G}_1, ...\mathcal{G}_{2s}$, for some $s \in \mathbb{N}$. We note that the total number, $2s$, is indeed even since the total number of the remaining nodes is even and the number of nodes in each subgraph is odd. For $1 \leq i \leq 2s$, let g_i be a node in $\mathcal{G}_i$ that connects to k (we commented above on its existence). We form the following s pairs: $(g_1, g_2), (g_3, g_4), ...(g_{2s-1}, g_{2s})$. We note that they form an eligible set since they all connect to k. This eligible set only uses the nodes $\{g_i\}_{i=1}^{2s}$. We terminate the procedure at this step whenever all maximally connected subgraphs with an odd number of nodes only contain a single node (so that all nodes of these subgraphs are used by this eligible set).
3. If $\mathcal{G}_i \neq (\{g_i\}, \emptyset)$ and the number of nodes of $\mathcal{G}_i$ is odd, then we remove g_i from $\mathcal{G}_i$ and also remove all the edges connected to g_i. The remaining graph contains an even number of nodes. If the remaining graph is connected, then we find an eligible set that uses all nodes in this remaining subgraph (its existence follows from the induction assumption). We terminate the procedure at this step if each of these remaining subgraphs is connected (the union of all such eligible sets then form a larger eligible set that uses all nodes in these subgraphs).

4. If a remaining subgraph from the above step (having g_i and its connected edges removed from $\mathcal{G}_i$) is not connected, we form its maximally connected subgraphs $\mathcal{H}_1^{(i)}, \dots \mathcal{H}_{l_i}^{(i)}$. We note that $\mathcal{H}_1^{(i)}, \dots \mathcal{H}_{l_i}^{(i)}$ are all connected to g_i (since $\mathcal{G}_i$ is connected and $\mathcal{H}_1, \dots \mathcal{H}_{m_i}$ are not connected to each other). We also observe that $\cup_{r=1}^{l_i} \mathcal{H}_r^{(i)}$ contains less than $2K-2$ nodes (indeed the number of nodes of $\mathcal{G}_i$ is strictly less than the number of nodes in $\cup_{r=1}^{m} \mathcal{G}_r$, which is $2K-2$; we remark that since g_i was removed from $\mathcal{G}_i$ the bound $2K-2$ is not tight).

If this procedure terminates in its first three steps, then it finds an eligible set that uses all nodes in the maximally connected subgraphs and thus its size is $2K-2$. Otherwise, we iteratively apply the same four-steps procedure on the maximally connected subgraphs of the last step. At each iteration the total number of nodes in the maximally connected subgraphs reduces (we clarified this at the end of the fourth step). Since the graph is finite, the iteration either terminates or $\cup_{r=1}^{l_i} \mathcal{H}_r^{(i)}$ in step 4 of the procedure is of size 2, that is, there are two subgraphs with single nodes. When inputting these single nodes at the next iteration, the procedure will terminate at step 2.

The final eligible set is the union of all the eligible sets iteratively generated from steps (1)-(3) and the pair (j,k). This eligible set uses all the nodes in V^o and thus it contains K pairs.

We remark that we introduced Case 1 in order to help the reader master the idea of the proof in a simpler case. We actually demonstrated how to consecutively handle the case where a single node is connected to maximally connected subgraphs whose total number of nodes is even. Starting with a node of degree 1 allowed us to proceed with this idea in one direction. Next, we pick up two different points and proceed with this idea in two different directions and, in fact, we could have started the proof with the latter setting right away. This setting has two subcases (2A and 2B). In Case 2A, we still have an even number of nodes in the maximally connected subgraphs, which are connected to a single node, so the ideas of Case 1 immediately apply. In Case 2B, the latter number of nodes is odd, but we can somehow reduce it to Case 2A.

Case 2: There does not exist any node in V^o with degree 1. In this case we randomly select two neighboring nodes $j, k \in V^o$. We consider the remaining graph after removing these two nodes from V^o and also remove all edges connected to them from E^o. If the remaining graph is connected or if all the maximally connected subgraphs of the remaining graph contain an even number of nodes, then the induction assumption concludes the proof. Otherwise, the remaining graph is partitioned into maximally connected subgraphs $\mathcal{G}_1, ..., \mathcal{G}_{n+m}$. We reindex these subgraphs so $\mathcal{G}_1, ...\mathcal{G}_m$ are all connected to node j and not connected to node k in $\mathcal{G}^o$. The other maximally connected subgraphs, $\mathcal{G}_{m+1}, ..\mathcal{G}_{m+n}$, are connected to k in $\mathcal{G}^o$ (they may or may not be connected to j). We prove this case by further considering two different scenarios:

Case 2A: $\cup_{i=1}^{m} \mathcal{G}_i$ contains an even number of nodes. In this case, it is clear that $\cup_{i=m+1}^{n+m} \mathcal{G}_i$ also contains an even number of nodes. We could iteratively perform the above four-steps procedure introduced in Case 1 on $\{\mathcal{G}_i\}_{i=m+1}^{n+m}$ (as they all connect to k) and on $\{\mathcal{G}_i\}_{i=1}^{m}$ (as they all connect to j). Following the same argument at the end of the proof of case 1, we obtain two eligible sets that cover $\{\mathcal{G}_i\}_{i=m+1}^{n+m}$ and $\{\mathcal{G}_i\}_{i=1}^{m}$, respectively. The union of these two sets and the pair (j,k) yields an eligible set that uses all nodes in $\mathcal{G}^o$, which concludes the proof.

Case 2B: $\cup_{i=1}^{m} \mathcal{G}_i$ contains an odd number of nodes. We note that $\cup_{i=m+1}^{n+m} \mathcal{G}_i$ contains an odd number of nodes. We further note that within $\{\mathcal{G}_i\}_{i=1}^{m}$, there is an odd number of subgraphs that contain an odd number of nodes. After reindexing, these subgraphs are $\mathcal{G}_1, \mathcal{G}_2, ...\mathcal{G}_{2r+1}$, where $2r+1 \leq m$. We pick one node from $\mathcal{G}_{2r+1}$ that is connected to j and denote it by l. By definition, l is not connected to $\mathcal{G}_i$, $i \in [2r]$ or $i = 2r+2, \dots m$, and also not connected to node k.Using this special node l, we redefine the "remaining graph" that was described in the beginning of case 2. That is, we remove from $\mathcal{G}^o$ the pair (j,l) (and the associated edges), whereas before the pair (j,k) was removed from it. We can similarly identify maximally connected subgraphs of this new remaining graph. Now the collection of the maximally connected subgraphs that connect to j but not connect to l contains an even number of nodes in total. Indeed, this collection contains all nodes in the former subgraphs, $\mathcal{G}_1, \dots \mathcal{G}_{2r}, \mathcal{G}_{2r+2}, \dots, \mathcal{G}_{m+n}$ and also the node k and there is an odd number of nodes in every $\mathcal{G}_i$, $i = 1, ...2r$, an even number of nodes in every $\mathcal{G}_i$, $i = 2r+2, \dots m$ and an odd number of nodes in $\cup_{i=m+1}^{n+m} \mathcal{G}_i$ and the single node k, which yield an even number of total nodes. Note that we transformed the case to the one in case 2A (with the initially selected pair (j,l)) and the proof is thus concluded. □

Lemma D.3. *For a connected graph $\mathcal{G}^o$ with $2K+1$ nodes, there exists an eligible set S containing K pairs of nodes in $\mathcal{G}^o$ and the pooling process with respect to S maps this graph to a graph with $K+1$ nodes.*

Proof. We note that there exists a node in $\mathcal{G}^o$ so that the remaining graph is still connected after removing this node from V^o and removing all edges connected to this node from E^o. Indeed, it can be selected as a node with degree 1 in the spanning tree of $\mathcal{G}^o$. Considering the remaining graph with $2K$ nodes (after removing this node and the associated edges), Lemma D.2 implies the existence of an eligible set S containing K pairs of nodes so that the pooling process with respect to S will map $\mathcal{G}^o$ to a graph with $K+1$ nodes.

□

Proof of Theorem 2. To conclude the theorem, we just need to show that for $\mathcal{G}^o = (V^o, E^o)$ with $|V^o| = N$, and any number $K \in [\lceil N/2 \rceil, N-1]$, there exists a pooling process with an eligible set S (containing $N-K$ pairs) that maps $\mathcal{G}^o$ to a graph $\mathcal{G}$ with K nodes. It is sufficient to prove the statement for $K = \lceil N/2 \rceil$ with the eligible set S^* that contains $N - \lceil N/2 \rceil$ pairs. Indeed, for any $K > \lceil N/2 \rceil$, we can take a subset S_K of the eligible set S^* (containing $N - \lceil N/2 \rceil$ pairs) such that $|S_K| = |S^*| - (K - \lceil N/2 \rceil)$. The pooling process with respect to S_K produces a graph with K nodes.

Given a graph $\mathcal{G}^o$ with N nodes, Lemma D.2 and Lemma D.3 imply that there exists an eligible set S^* with $\lfloor N/2 \rfloor$ pairs. Therefore, the pooling procedure with respect to S^* maps $\mathcal{G}^o$ to a graph $\mathcal{G}$ with $\lceil N/2 \rceil$ nodes. □

D.3 Proof of Corollary 3

For any connected graph $\mathcal{G}^o$ with N nodes, iterative application of Theorem 2 implies the existence of a series of unpooling layers U_k, $k \geq 1$, and intermediate graphs $\mathcal{G}_k$, $k \geq 1$, with $|\mathcal{G}_k| = \lceil |\mathcal{G}_{k-1}|/2 \rceil$, so that $U_k(\mathcal{G}_k) = \mathcal{G}_{k-1}$, where $\mathcal{G}_0 := \mathcal{G}^o$. We stop the process once $|\mathcal{G}_k| \in [4, 6]$ (we will reach this range because for any $N' > 6$, $\lceil N'/2 \rceil \geq 4$). Another application of Theorem 2 yields the existence of a 3-nodes graph, $\mathcal{G}$, and a last unpooling layer, U_{k+1}, so that $U_{k+1}(\mathcal{G}) = \mathcal{G}_k$.

That is, there exist a 3-nodes graph $\mathcal{G}$ and a series of unpooling layers $U_1, ..U_{k+1}$ so that $U_1 \circ U_2 \circ ... \circ U_{k+1}(\mathcal{G}) = \mathcal{G}^o$. Note that the number of unpooling layers is $k+1 = \lceil \log_2(N/3) \rceil$. □

E ANOTHER GENERATION TASK: OPTIMIZING SPECIFIC CHEMICAL PROPERTIES

In this task, we consider the following three chemical properties (Guimaraes et al., 2017), which we evaluate only on the valid molecules among the 10,000 generated ones: druglikeliness, solubility and synthesizability. We calculate the first property using the Quantitative Estimate of Druglikeness (QED) package in RDKit (`https://www.rdkit.org/`) (licensed by BSD 3-Clause). These scores lie in $[0, 1]$ and their values aim to express the likelihood of being a drug. We calculate solubility by the log octanol-water partition coefficient using the Crippen package in RDKit. We rescale this value to lie in $[0, 1]$, where 1 is the most soluble value. In order to calculate synthesizability, we calculate the synthetic accessibility (Ertl and Schuffenhauer, 2009) and rescale this value to lie in $[0, 1]$ where 1 is the easiest to synthesize. We use codes from `https://github.com/connorcoley/scscore` , licensed by MIT.

For the generative model, we follow with the same Wasserstein GAN architecture as described in §C (with the final activation function to be a sigmoid function), but the discriminator minimizes the error between its output and the objective chemical property; it then outputs a reward score, which we need to maximize when training the generator.

Result of optimizing chemical properties. Using QM9, we generated molecules that aimed to maximize druglikeliness, solubility and synthesizability. Table 5 reports the six evaluation metrics (listed in its columns, whereas the properties we aimed to maximize are in its rows). In terms of generating molecules with the targeted chemical properties, UL GAN is competitive with the other approaches (this is noticed when looking at columns 4, 5, 6 of rows 1, 2, 3, respectively. When considering the other evaluation metrics, UL GAN generally outperforms Adj GAN, except for the objective of druglikeliness and the metric of synthesizability (first row and sixth column). Note that the uniqueness of UL GAN and Adj GAN is very low because our generator does not aim to compete with the discriminator, but to generate molecules with a maximal property of interest. We did not report the good performance of UL GAN when considering this task with ZINC, since the other methods we compared with were not tested on ZINC; furthermore, the superiority of UL GAN over Adj GAN for ZINC is already obvious from Table 4.

F SOME ADDITIONAL NUMERICAL RESULTS

We supplement the numerical results in §4. Section F.1 reports standard deviations for the earlier experiments on QM9 and ZINC; §F.2 includes generation results with only 1,000 samples; and §F.3 demonstrates examples of the generated molecules from UL GAN and UL VAE.

F.1 Standard Deviations for Molecule Generation

Table 6 supplements Table 3 in the main manuscript and reports the means and standard deviations of the evaluation metrics for our methods in QM9, including Adj GAN, UL GAN, and UL VAE. Table 7 supplements Table 4 in the main manuscript and reports the means and standard deviations of the evaluation metrics for our methods in ZINC, including UL GAN and Adj GAN and UL VAE. These means and standard deviations are calculated from 100 runs.

Table 5: The six evaluation metrics (in rows) for generated samples that aim to minimize the three indicated chemical properties (in columns). We remark that QED is the acronym for quantitative estimate of druglikeliness. Scores for competing methods (above the indicated line) were copied from their original papers. NA means that the score is not available in the original papers.

Objective	Method	Valid	Unique	Novel	QED	Solubility	Synthesizability
QED	ORWGAN	0.882	0.694	NA	0.52	0.35	0.32
	Naive RL	0.971	0.540	NA	0.57	0.50	0.53
	MolGAN	1.00	0.022	NA	**0.62**	0.59	0.53
	Adj GAN	0.991	0.005	0.865	0.443	0.288	0.658
	UL GAN	0.9888	0.051	0.978	0.598	0.497	0.485
Solubility	ORWGAN	0.965	0.459	NA	0.50	0.55	0.63
	Naive RL	0.927	1.00	NA	0.49	0.78	0.70
	MolGAN	0.998	0.002	NA	0.44	**0.89**	0.22
	Adj GAN	0.940	0.003	0.958	0.378	0.367	0.007
	UL GAN	0.993	0.010	0.781	0.507	0.700	0.793
Synthesizability	ORWGAN	0.965	0.459	NA	0.51	0.45	0.83
	Naive RL	0.977	0.136	NA	0.52	0.46	0.83
	MolGAN	1.00	0.021	NA	0.53	0.68	0.95
	Adj GAN	0.999	0.003	0.833	0.360	0.331	0.835
	UL GAN	1.00	0.006	0.433	0.468	0.569	**0.953**

Table 6: Validity, uniqueness and novelty with standard deviation for Adj GAN, UL GAN, and UL VAE using QM9.

Method	Valid	Unique	Novel
UL VAE	0.735 ($\pm$ 0.004)	0.940 ($\pm$ 0.003)	0.949 ($\pm$ 0.002)
Adj GAN	0.941 ($\pm$ 0.002)	0.139 ($\pm$ 0.002)	0.886 ($\pm$ 0.006)
UL GAN	0.907 ($\pm$ 0.003)	0.826 ($\pm$ 0.004)	0.949 ($\pm$ 0.002)

Table 7: Validity, Uniqueness and Novelty with standard deviation for Adj GAN and UL GAN using ZINC.

Method	Valid	Unique	Novel
Adj GAN	0.109 ($\pm$ 0.003)	0.196 ($\pm$ 0.011)	1.00 ($\pm$ 0)
UL GAN	0.871 ($\pm$ 0.004)	1.00 ($\pm$ 0)	1.00 ($\pm$ 0)

Table 8: Model performance with 1,000 generated samples on various datasets.

	Waxman random graph								QM9		
Method	kl edge dense	kl clust	kl conn	kl node feat	wd edge dense	wd clust	wd conn	wd node feat	Valid	Unique	Novel
UL VAE	0.115	0.443	0.279	0.451	0.009	0.100	0.106	0.126	0.737	0.991	0.932
UL GAN	0.007	0.030	0.033	0.152	0.002	0.022	0.012	0.039	0.905	0.970	0.927
	Protein dataset								ZINC		
	kl edge dense	kl clust	kl conn	kl node feat	wd edge dense	wd clust	wd conn	wd node feat	Valid	Unique	Novel
UL VAE	0.565	1.235	0.822	0.260	0.034	0.071	0.319	4.400	NA	NA	NA
UL GAN	0.084	0.484	0.136	0.234	0.014	0.011	0.115	3.958	0.870	1.00	1.00

F.2 Evaluation with only 1,000 generated samples.

In the experiments of §4, we generated 10,000 samples and reported statistics based on these samples. To check whether the performance is preserved for a smaller sample, we report here results of UL GAN and UL VAE when generating only 1,000 samples. Table 8 reports such results for the Waxman, protein, QM9 and ZINC datasets. We note that all the reported metrics, but the uniqueness in QM9, are similar to the ones in Table 1, Table 2, Table 3 and Table 4, where 10k samples were generated. The uniqueness in QM9 is significantly higher with 1,000 samples. Polykovskiy et al. (2020) also noticed a higher uniqueness rate with 1,000 samples than with 10,000 samples.

F.3 Synthetic Samples for Molecule Generation

We demonstrate samples generated from the UL GAN for QM9 in Figure 10 and samples generated from the UL VAE in Figure 11. Also, we present samples generated from the UL GAN for ZINC in Figure 12.

We also illustrate some examples of the evolution of intermediate graphs and illustrate how a graph is generated from a generative GNN using unpooling layers, in Figure 13 for QM9 and in Figure 14 for ZINC.

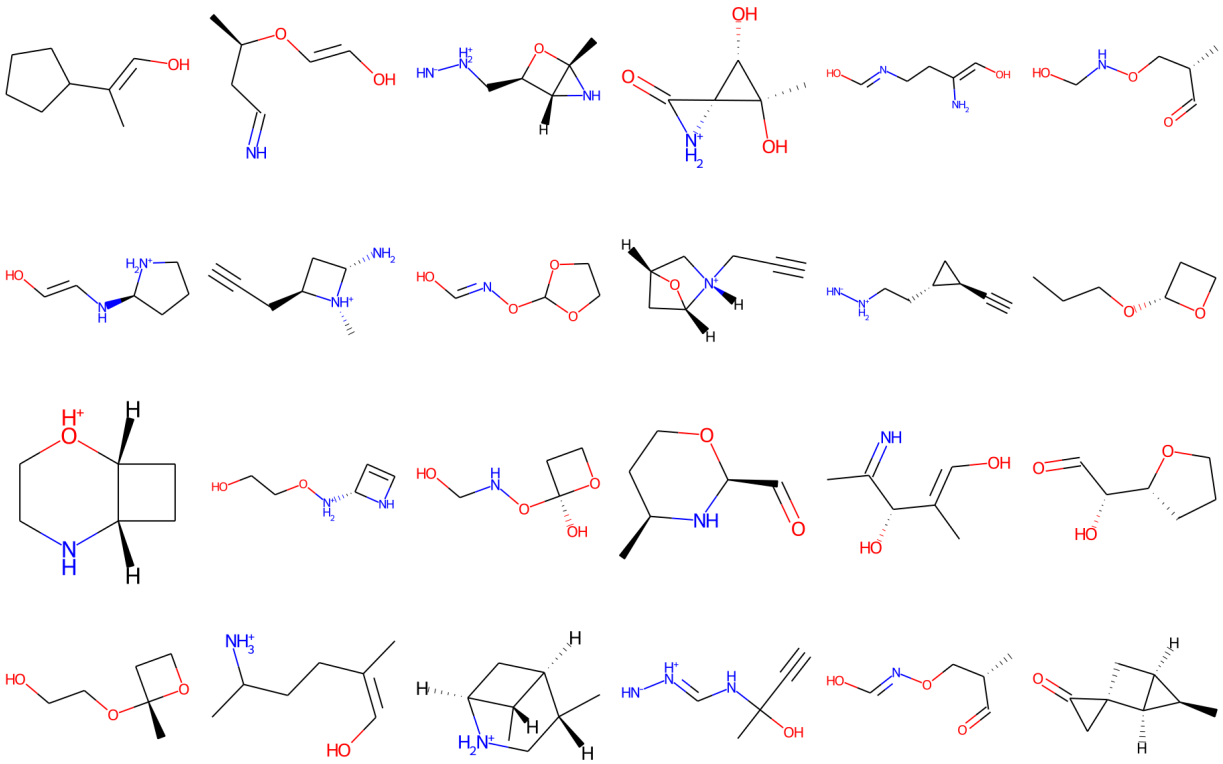

Figure 10: Samples of molecules generated by UL GAN based on QM9 dataset.

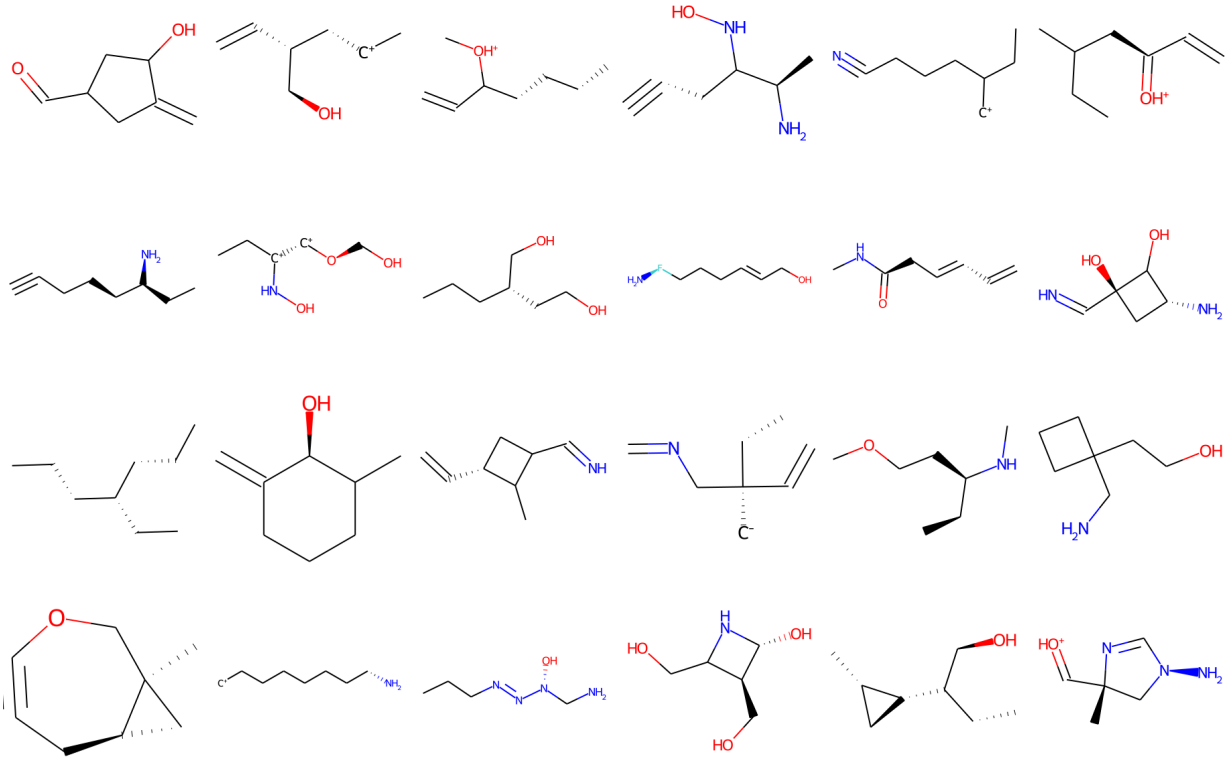

Figure 11: Samples of molecules generated from UL VAE based on QM9 dataset.

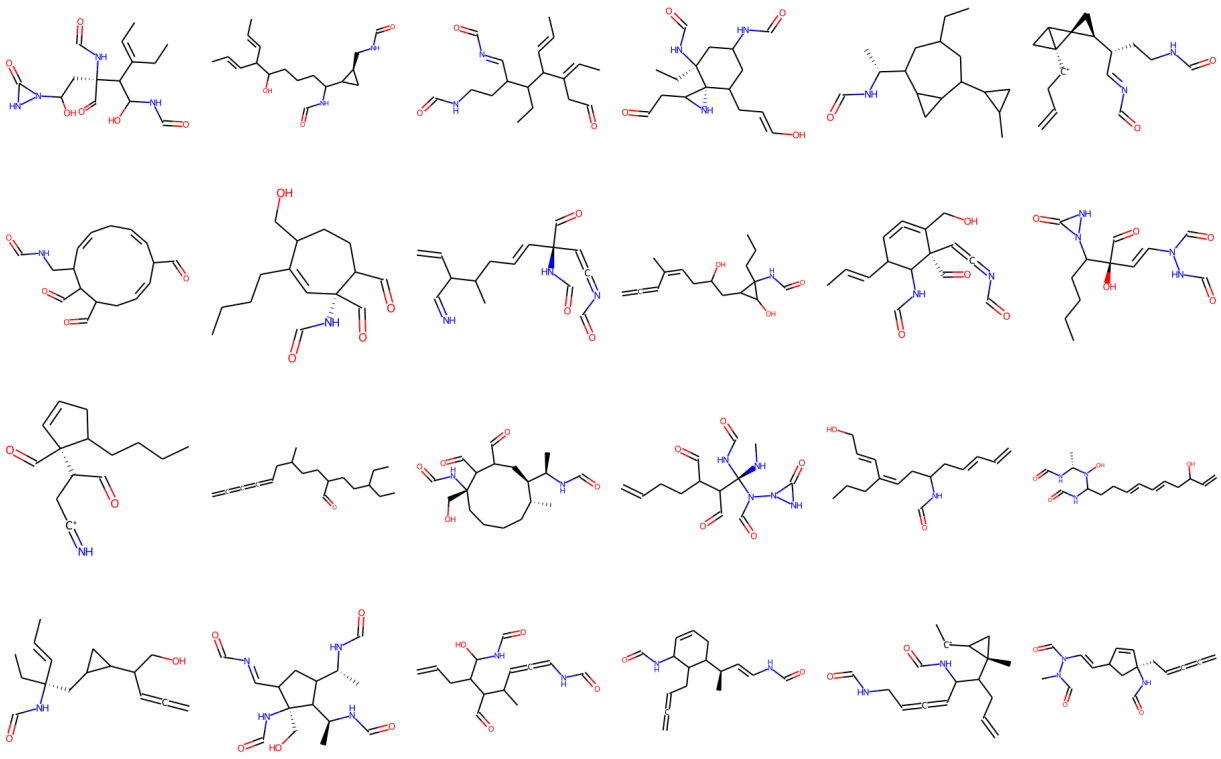

Figure 12: Samples of molecules generated from UL GAN based on ZINC dataset.

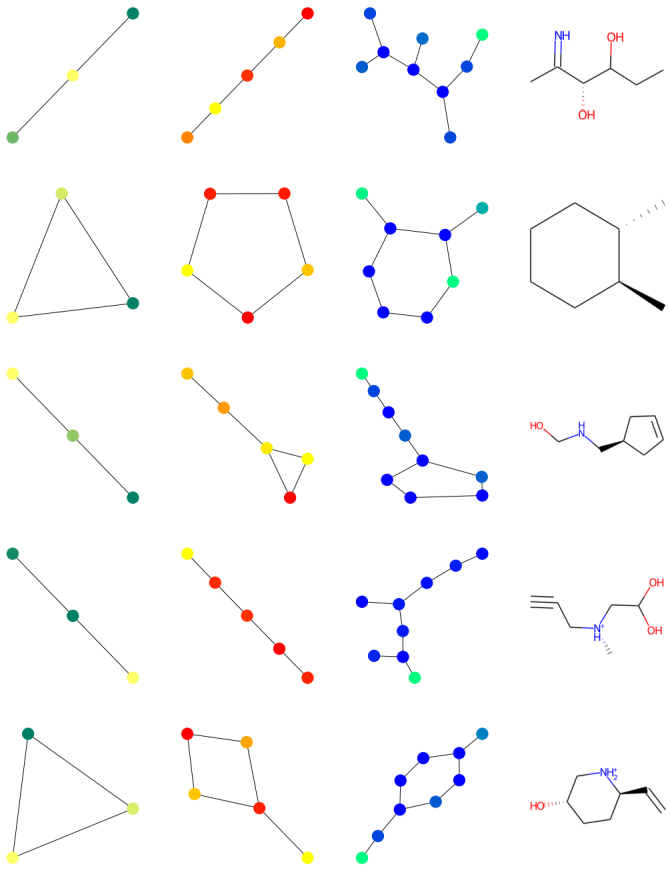

Figure 13: Five examples of generated graphs using UL GAN trained with QM9. Each row represents one example, showing intermediate graphs in the generation process. Left column: initial 3-nodes graph; Middle 2–3 columns: intermediate graphs after unpooling layers; Right column: the final generated molecule. The color represents one dimension of the node features.

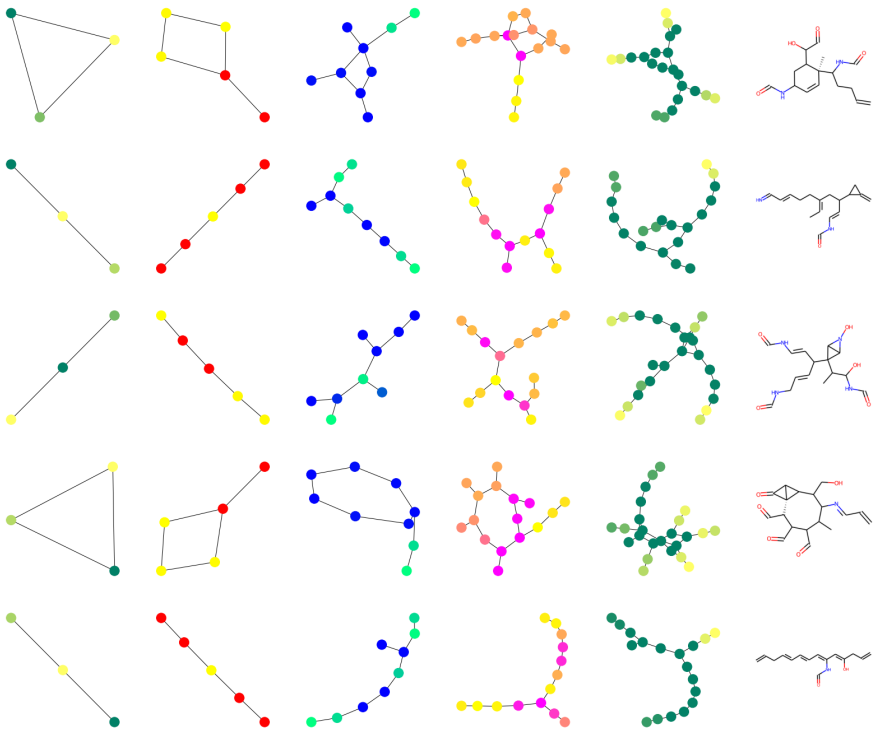

Figure 14: Five examples of generated graphs using UL GAN trained with ZINC dataset. Each row represents one example, showing intermediate graphs in the generation process. Left column: initial 3-nodes graph; Middle 2–5 columns: intermediate graphs after unpooling layers; Right column: the final generated molecule. The color represents one dimension of the node features.