MolGrow: A Graph Normalizing Flow for Hierarchical Molecular Generation

Maksim Kuznetsov, Daniil Polykovskiy
Insilico Medicine
kuznetsov@insilico.com, daniil@insilico.com

Abstract

We propose a hierarchical normalizing flow model for generating molecular graphs. The model produces new molecular structures from a single-node graph by recursively splitting every node into two. All operations are invertible and can be used as plug-and-play modules. The hierarchical nature of the latent codes allows for precise changes in the resulting graph: perturbations in the top layer cause global structural changes, while perturbations in the consequent layers change the resulting molecule marginally. The proposed model outperforms existing generative graph models on the distribution learning task. We also show successful experiments on global and constrained optimization of chemical properties using latent codes of the model.

Introduction

Drug discovery is a challenging multidisciplinary task that combines domain knowledge in chemistry, biology, and computational science. Recent works demonstrated successful applications of machine learning to the drug development process, including synthesis planning (Segler, Preuss, and Waller 2018), protein folding (Senior et al. 2020), and hit discovery (Merk et al. 2018; Zhavoronkov et al. 2019). Advances in generative models enabled applications of machine learning to drug discovery, such as distribution learning and molecular property optimization. Distribution learning models train on a large dataset to produce novel compounds (Polykovskiy et al. 2020); property optimization models search the chemical space for molecules with desirable properties (Brown et al. 2019). Often researchers combine these tasks: they first train a distribution learning model and then use its latent codes to optimize molecular properties (Gómez-Bombarelli et al. 2018). For such models, proper latent codes are crucial for molecular space navigation.

We propose a new graph generative model—MolGrow. Starting with a single node, it iteratively splits every node into two. Our model is invertible and maps molecular structures onto a fixed-size hierarchical manifold. Top levels of the manifold define global structure, while the bottom levels influence local features.

Our contributions are three-fold:

- We propose a hierarchical normalizing flow model for generating molecular graphs. The model gradually increases graph size during sampling, starting with a single node;
- We propose a fragment-oriented atom ordering that improves our model over commonly used breadth-first search ordering;
- We apply our model to distribution learning and property optimization tasks. We report distribution learning metrics (Fréchet ChemNet distance and fragment distribution) for graph generative models besides providing standard uniqueness and validity measures.

Background: Normalizing Flows

Normalizing flows are generative models that transform a prior distribution $p(z)$ into a target distribution $p(x)$ by composing invertible functions f_k:

$$z = f_K \circ ... \circ f_2 \circ f_1(x), \tag{1}$$

$$x = f_1^{-1} \circ ... \circ f_{K-1}^{-1} \circ f_K^{-1}(z). \tag{2}$$

We call Equation 1 a forward path, and Equation 2 an inverse path. The prior distribution $p(z)$ is often a standard multivariate normal distribution $\mathcal{N}(0, I)$. Such models are trained by maximizing training set log-likelihood using the change of variables formula:

$$\log p(x) = \log p(z) + \sum_{i=1}^{K} \log \left| \det \left(\frac{dh_i}{dh_{i-1}} \right) \right|, \tag{3}$$

where $h_i = f_i(h_{i-1})$, $h_0 = x$. To efficiently train the model and sample from it, inverse transformations and Jacobian determinants should be tractable and computationally efficient. In this work, we consider three types of layers: invertible linear layer, actnorm, and real-valued non-volume preserving transformation (RealNVP) (Dinh, Sohl-Dickstein, and Bengio 2017). We define these layers below for arbitrary d-dimensional vectors, and extend these layers for graph-structured data in the next section.

We consider an invertible linear layer parameterization by Hoogeboom, Van Den Berg, and Welling (2019) that uses QR decomposition of a weight matrix: $h = QR \cdot z$, where Q is an orthogonal matrix ($Q^T = Q^{-1}$), and R is an upper

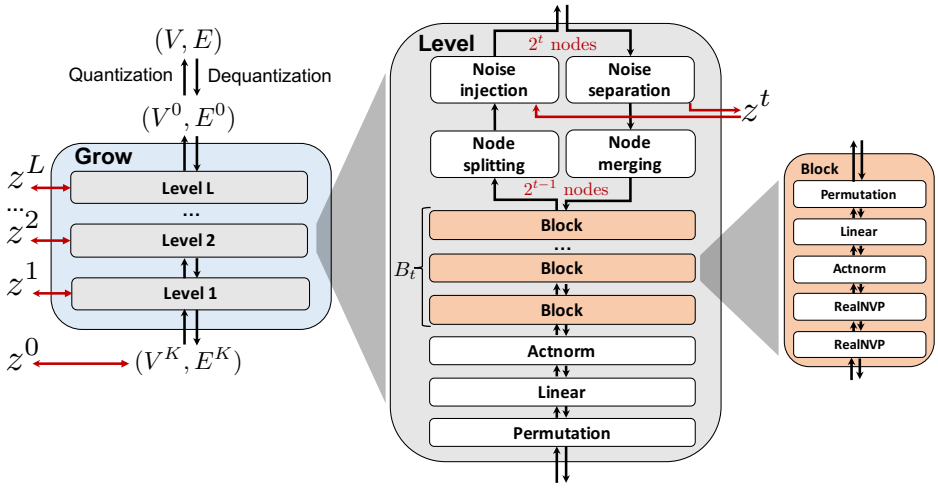

Figure 1: MolGrow architecture. Left: Full architecture combines multiple levels to generate latent codes $z^L, \dots, z^0$ from a graph (V, E) and vice versa. Middle: Each level separates noise, merges node pairs, applies multiple blocks and linear transformations; Right: Each block applies three channel-wise transformations and two RealNVP layers.

triangular matrix with ones on the main diagonal. We use Householder reflections to parameterize Q:

$$Q = \prod_{i=1}^{d'} \left(I - 2 \frac{v_i v_i^T}{\|v_i\|^2} \right), \tag{4}$$

where v_i are learnable column-vectors. The Jacobian determinant of a linear layer is 1. There is an alternative way to formulate a linear layer with LU decomposition (Kingma and Dhariwal 2018). However, in our experiments, QR decomposition showed more numerically stable results.

Actnorm layer (Kingma and Dhariwal 2018) is a linear layer with a diagonal weight matrix: $h = s \odot z + m$, where $\odot$ is an element-wise multiplication. Vectors s and m are initialized so that the output activations from this layer have zero mean and unit variance at the beginning of training. We use the first training batch for initialization. The Jacobian determinant of this layer is $\prod_{i=1}^{d} s_i$.

RealNVP layer (Dinh, Sohl-Dickstein, and Bengio 2017) is a nonlinear invertible transformation. Consider a vector z of length $d = 2t$ with first half of the components denoted as z_a, and the second half as z_b. Then, RealNVP and its inverse transformations are:

$$\begin{pmatrix} h_a \\ h_b \end{pmatrix} = \begin{pmatrix} z_b \\ e^{s_\theta(z_b)} \odot z_a + t_\theta(z_b) \end{pmatrix} \tag{5}$$

$$\begin{pmatrix} z_a \\ z_b \end{pmatrix} = \begin{pmatrix} (h_b - t_\theta(h_a)) / e^{s_\theta(h_a)} \\ h_a \end{pmatrix} \tag{6}$$

Functions s_θ and t_θ do not have to be invertible, and usually take form of a neural network. The Jacobian determinant of the RealNVP layer is $\prod_{i=1}^{d} e^{s_{\theta,i}(z_b)}$. We sequentially apply two RealNVP layers to transform both components of z. We also use permutation layer that deterministically shuffles input dimensions before RealNVP—this is equivalent to randomly splitting data into a and b parts.

MolGrow (Molecular Graph Flow)

In this section, we present our generative model—MolGrow. MolGrow is a hierarchical normalizing flow model (Figure 1): it produces new molecular graphs from a single-node graph by recursively dividing every node into two. The final graph has $N = 2^L$ nodes, where L is a number of node-splitting layers in the model. To generate graphs with fewer nodes, we add special padding atoms. We choose N to be large enough to fit any graph from the training dataset.

We represent a graph with node attribute matrix $V \in \mathbb{R}^{N \times d_v}$ and edge attribute tensor $E \in \mathbb{R}^{N \times N \times d_e}$, where d_v and d_e are feature dimensions. For the input data, V_i defines atom type and charge, $E_{i,j}$ defines edge type. Since molecular graphs are non-oriented, we preserve the symmetry constraint on all intermediate layers: $E_{i,j,k} = E_{j,i,k}$.

We illustrate MolGrow's architecture in Figure 1. MolGrow consists of L invertible *levels*, each *level* has its own latent code with a Gaussian prior. On a forward path, each *level* extracts the latent code and halves the graph size by merging node pairs. On the inverse path, each *level* does the opposite: it splits each node into two and adds additional noise. The final output on the forward path is a single-node graph graph $z^0 = (V^K, E^K)$ and latent codes from each *level*: $z^L, \dots, z^1$. We call z^0 a top level latent code.

Dequantization

To avoid fitting discrete graphs into a continuous density model, we dequantize the data using a uniform noise (Kingma and Dhariwal 2018):

$$V^0_{i,j} = V_{i,j} + u^v_{i,j}, \tag{7}$$

$$E^0_{i,j,k} = E^0_{j,i,k} = \begin{cases} E_{i,j,k} + u^e_{i,j,k}, & i < j \\ 0, & i = j \end{cases}. \tag{8}$$

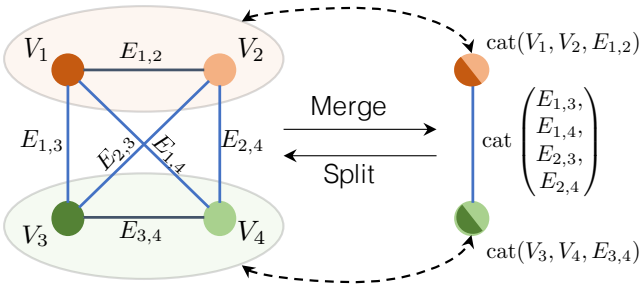

Figure 2: Node merging and splitting example for a 4-node graph. We concatenate features of nodes V_1 and V_2 and edge $E_{1,2}$ to get new node features. We also concatenate edge features $E_{1,3}$, $E_{1,4}$, $E_{2,3}$, and $E_{2,4}$. Splitting operation slices merged graph's node and edge features.

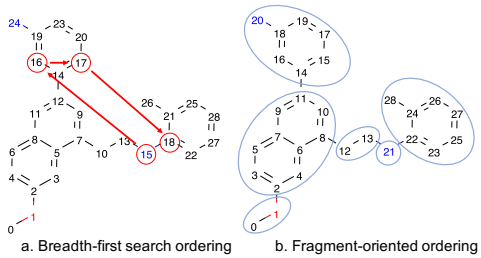

Figure 3: Different atom orderings. Numbers are atom's indices in a particular ordering. Note that BFS ordering (a) generates two fragments in parallel (see nodes 15-18), while our method completes a fragment before transitioning to the next one. For fragment-oriented ordering (b), we circled extracted fragments.

Elements of u^v and u^e are independent samples from a uniform distribution $\mathcal{U}[0, c]$. Such dequantization is invertible for $c \in [0, 1)$—original data can be reconstructed by rounding down the elements of $V^0_{i,j}$ and $E^0_{i,j,k}$. We dequantize the data for each training batch independently and train the model on (V^0, E^0). Dequantizated graph (V^0, E^0) is a complete graph.

Node Merging and Splitting

We use node merging and splitting operations to control the graph size. These operations are inverse of each other, and both operate by rearranging node and edge features. Consider a graph (V^k, E^k) with N_k nodes. Node merging operation joins nodes $2i$ and $2i+1$ into a single node by concatenating their features and features of the edge between them. We concatenate edge features connecting the merged nodes:

$$\underbrace{V_i^{k+1}}_{2d_v+d_e} = \mathrm{cat}\Big(\underbrace{V_{2i}^k}_{d_v}, \underbrace{V_{2i+1}^k}_{d_v}, \underbrace{E_{2i,2i+1}^k}_{d_e}\Big), \quad (9)$$

$$\underbrace{E_{i,j}^{k+1}}_{4d_e} = \mathrm{cat}\Big(\underbrace{E_{2i,2j}^k}_{d_e}, \underbrace{E_{2i,2j+1}^k}_{d_e}, \underbrace{E_{2i+1,2j}^k}_{d_e}, \underbrace{E_{2i+1,2j+1}^k}_{d_e}\Big). \quad (10)$$

Node splitting is the inverse of node merging layer: it slices features into original components. See an example in Figure 2.

Noise Separation and Injection

MolGrow produces a latent vector for each *level*. We derive the latent codes by separating half of the node and edge features before node merging and impose Gaussian prior on these latent codes. During generation, we sample the latent code from the prior and concatenate it with node and edge features. As we show in the experiments, latent codes on different levels affect the generated structure differently. Latent codes from smaller intermediate graphs (top level) influence global structure, while bottom *level* features define local structure.

Block Architecture

The basic building block in MolGrow (denoted *block* in Figure 1) consists of five layers. The first three layers (permutation, linear, and actnorm) serve as 1×1 convolutions. Each layer contains two transformations: one transforms every node and the other transforms every edge. The number of linear layer's Housholder reflections in matrix Q is smaller than the dimension of Q. Hence, a combination of linear and permutation layers is not equivalent to a single linear layer.

The final two layers of the *block* are RealNVP layers. RealNVP layer splits its input graph (V^k, E^k) with N_k nodes into $(V^{k,a}, E^{k,a})$ and $(V^{k,b}, E^{k,b})$ along features dimension. We transform $(V^{k,b}, E^{k,b})$ by projecting node and edge features onto a low-dimensional manifold and applying attention on complete graph edges (CAGE) architecture (Algorithm 1). We compute the final output of RealNVP layer by applying fully-connected neural networks s^v_θ, t^v_θ, s^e_θ, and t^e_θ to each node and edge independently:

$$(\overline{V}^{k,b}, \overline{E}^{k,b}) = \mathrm{CAGE}(V^{k,b}W_v, E^{k,b}W_e) \quad (11)$$

$$V_i^{k+1,b} = \exp\left(s^v_\theta\left(\overline{V}_i^{k,b}\right)\right) \odot V_i^{k,a} + t^v_\theta\left(\overline{V}_i^{k,b}\right) \quad (12)$$

$$V_i^{k+1,a} = V_i^{k,b} \quad (13)$$

$$E_{i,j}^{k+1,b} = \exp\left(s^e_\theta\left(\overline{E}_{i,j}^{k,b}\right)\right) \odot E_{i,j}^{k,a} + t^e_\theta\left(\overline{E}_{i,j}^{k,b}\right) \quad (14)$$

$$E_{i,j}^{k+1,a} = E_{i,j}^{k,b} \quad (15)$$

Similar to other attentive graph convolutions (Veličković et al. 2017; Guo, Zhang, and Lu 2019), CAGE architecture uses a multi-head attention (Vaswani et al. 2017). It also uses gated recurrent unit update function to stabilize training (Parisotto et al. 2020). Positional encoding in CAGE consists of two parts. First $d_v - \lceil \log_2 N_k \rceil$ dimensions are standard sinusoidal positional encoding (Vaswani et al. 2017):

$$\mathrm{pos}_{i,2j} = \sin\left(i/10000^{2j/d_v}\right), \quad (16)$$

$$\mathrm{pos}_{i,2j+1} = \cos\left(i/10000^{2j/d_v}\right). \quad (17)$$

Algorithm 1 Attention on complete graph edges (CAGE)

Input: Complete graph (V, E) with node feature matrix $V \in \mathbb{R}^{n \times d_v}$ and edge feature tensor $E \in \mathbb{R}^{n \times n \times d_e}$.
Output: Transformed complete graph $(\overline{V}, \overline{E})$ of the same dimension as (V, E).
Compute positional encodings matrix $\mathrm{pos} \in \mathbb{R}^{n \times d_v}$
for $i = 1$ **to** n **do**
  Allocate $\mu \in \mathbb{R}^{n \times d_v}$
  for $j = 1$ **to** n **do**
    $\mu_j = f_\theta\left(\mathrm{cat}\left(E_{i,j}, V_i, V_j\right)\right) + \mathrm{pos}_j$ — compute message $j \to i$ of dimension d_v using a fully-connected neural network f_θ
  end for
  $q = V_i + \mathrm{pos}_i$ — compute attention query
  $r = \text{Multi-Head Attention}(\text{query}{=}q, \text{keys}{=}\mu, \text{values}{=}\mu)$ — aggregate messages
  $\overline{V}_i = \mathrm{GRU}_V(x{=}r,\ h{=}V_i)$ — update node feature matrix using a GRU cell
  $\nu_{i,j} = \mathrm{cat}\left(E_{i,j}, \overline{V}_i, \overline{V}_j\right),\ \forall j$ — compute edge update vectors
  $\overline{E}_{i,j} = \frac{1}{2}\mathrm{GRU}_E(x{=}\nu_{i,j},\ h{=}E_{i,j}) + \frac{1}{2}\mathrm{GRU}_E(x{=}\nu_{j,i}, h{=}E_{j,i}) \quad \forall j$ — update edge features
end for

Algorithm 2 Balanced padding (function "pad")

Input: List of fragments $f_{1:K} = [f_1 \dots f_K]$, where f_i contains atom indices in the i-th fragment; N—target graph size, power of 2.
Output: Atom order with $*$ indicating padding positions ($*_N$ indicates N sequential paddings).
if $K = 1$ and $|f_1| \le N/2$ **then**
  Randomly add padding: with 50% probability **return** $\mathrm{cat}(\mathrm{pad}(f_1, N/2), *_{N/2})$, otherwise **return** $\mathrm{cat}(*_{N/2}, \mathrm{pad}(f_1, N/2))$
end if
Find possible splitting positions B: $b \in B$ if left and right parts fit into subtrees: $|f_{1:b}| \le N/2$ and $|f_{b+1:K}| \le N/2$
if $|B| > 0$ **then**
  Sample any index $b \in B$ that minimizes the number of bonds between $f_{1:b}$ and $f_{b+1:K}$.
  Recursively add padding to left and right parts:
  return $\mathrm{cat}\Big(\mathrm{pad}(f_{1:b}, N/2), \mathrm{pad}(f_{b+1:K}, N/2)\Big)$
else
  Add padding to the right
  return $\mathrm{cat}(f_{1:K}, *_{N-|f_{1:K}|})$
end if

The last $\lceil \log_2 N_k \rceil$ components of pos_i contain a binary code of i. We add multiple blocks before the first and after the last *level* in the full architecture.

Layout and Padding

Similar to the previous works (Shi et al. 2020; You et al. 2018b), we achieved better results when learning on a fixed atom ordering, instead of learning a distribution over all permutations. Previous works used breadth-first search (BFS) atom ordering, since it avoids long-range dependencies. However, BFS does not incorporate the knowledge of common fragments and can mix their atoms (Figure 3a). We propose a new atom ordering to incorporate prior knowledge about frequent fragments. Our ordering better organizes the latent space and simplifies generation.

We break the molecule into fragments by removing BRICS (Degen et al. 2008) bonds and bonds connecting rings, linkers, and decorations in the Bemis-Murcko scaffold (Bemis and Murcko 1996). We then enumerate the fragments and atoms in each fragment using BFS ordering (Figure 3b). We recursively choose padding positions, minimizing the number of edges after node merging layers (Algorithm 2).

Related Work

Many well-known generative models work out of the box for molecular generation task. By representing molecules as strings, one can apply any sequence generation model: language models (Segler et al. 2018), variational autoencoders (Gómez-Bombarelli et al. 2018), and generative adversarial networks (Sanchez-Lengeling et al. 2017). Molecular graphs satisfy a formal set of rules: all atoms must have a proper valency, and a graph must have only one component. These constraints can be learned implicitly from the data or explicitly by specifying grammar rules (Kusner, Paige, and Hernández-Lobato 2017; O'Boyle and Dalke 2018; Krenn et al. 2019).

Multiple generative models for molecular graphs were proposed. Graph recurrent neural network (GraphRNN) (You et al. 2018b) and molecular recurrent neural network (MolecularRNN) (Popova et al. 2019) use node and edge generators: node generator sequentially produces nodes; edge generator sequentially predicts edge types for all the previous nodes from the hidden states of a node generator. Molecular generative adversarial network (MolGAN) (De Cao and Kipf 2018) trains a critic on generated graphs and passes the gradient to the generator using deep deterministic policy gradient (Lillicrap et al. 2015). Graph variational autoencoder (GraphVAE) (Simonovsky and Komodakis 2018) encodes and decodes molecules using edge-conditioned graph convolutions (Simonovsky and Komodakis 2017). Graph autoregressive flow (GraphAF) (Shi et al. 2020) iteratively produces nodes and edges; discrete one-hot vectors are dequantized, and tokens are decoded using argmax. The most similar work to ours is a graph non-volume preserving transformation (GraphNVP) (Madhawa et al. 2019) model. GraphNVP generation is not autoregressive: the model produces a dequantized adjacency matrix using normalizing flow, turns it into a discrete set of edges by computing argmax, and obtains atom types using a normalizing flow. MoFlow (Zang and Wang 2020) exploits a two-stage graph generation process similar to GraphNVP: first, it generates an adjacency matrix with Glow architecture and then recovers node attributes with additional coupling lay-

Method	FCD/Test (↓)	Frag/Test (↑)	Unique@10k (↑)	Novelty (↑)
Graph-based models				
MolecularRNN	23.13	0.56	98.6%	99.9%
GraphVAE	49.39	0.0	5%	**100%**
GraphNVP	29.95	0.62	99.7 %	99.9%
GraphAF (BFS)	21.84	0.651	97%	99.9%
MoFlow	28.05	0.685	100%	99.99%
Proposed model				
MolGrow (fragment-oriented)	$\mathbf{6.284 \pm 0.986}$	$\mathbf{0.9294 \pm 0.025}$	99.28 ± 0.62 %	99.26 ± 0.12 %
MolGrow (BFS)	9.96 ± 0.795	0.933 ± 0.01	100 ± 0.0 %	99.4 ± 0.08 %
MolGrow (BFS on fragments)	16.1 ± 1.03	0.868 ± 0.02	100 ± 0.0 %	100 ± 0.0%
MolGrow (random permutation)	40.2 ± 4.71	0.05 ± 0.04	59 ± 38.1%	100 ± 0.0 %
MolGrow (GAT instead of CAGE)	6.52 ± 0.3	0.941 ± 0.013	99.4 ± 0.3 %	99.3 ± 0.06 %
MolGrow (No positional embedding)	6.77 ± 0.555	0.937 ± 0.006	99.5 ± 0.18 %	99.4 ± 0.06 %
SMILES and fragment-based models				
CharRNN (from MOSES benchmark)	$\mathbf{0.073 \pm 0.024}$	$\mathbf{0.9998 \pm 0.000}$	99.73 ± 0.03%	84.19 ± 5.09%
VAE (from MOSES benchmark)	0.099 ± 0.012	0.9994 ± 0.000	99.84 ± 0.12 %	69.49 ± 0.69%
JTN-VAE (from MOSES benchmark)	0.422 ± 0.023	0.9962 ± 0.000	$\mathbf{100 \pm 0.0}$%	$\mathbf{91.53 \pm 0.58}$%

Table 1: Distribution learning metrics on MOSES dataset. Baseline models from (Popova et al. 2019; Simonovsky and Komodakis 2018; Madhawa et al. 2019; Shi et al. 2020; Zang and Wang 2020)

ers. Unlike GraphNVP and MoFlow, we generate the graph hierarchically and inject noise on multiple levels. We also produce nodes and edges simultaneously. Tran et al. (2019) model discrete data directly with straight-through gradient estimators. Application of such model to graph structured data is yet to be explored.

All graph generative models mentioned above are not permutation invariant, and most of these models employ a fixed atom order. GraphVAE aligns generated and target graphs using Hungarian algorithm, but has high computational complexity. Vinyals, Bengio, and Kudlur (2016) report that order matters in set to set transformation problems with specific orderings giving better results. Most models learn on a breadth-first search (BFS) atom ordering (You et al. 2018b). In BFS, new nodes are connected only to the nodes produced on the current or previous BFS layers, avoiding long range dependencies. Alternatively, graph generative models could use a canonical depth-first search (DFS) order (Weininger, Weininger, and Weininger 1989). Note that unlike graph-based generators, string-based generators seem to improve from augmenting atom orders (Bjerrum 2017).

Several works studied fragment-based molecular generation. Jin, Barzilay, and Jaakkola (2018) replace molecular fragments with nodes to form a junction tree. They then produce molecules by sampling a junction tree and then expanding it into a full molecule. The authors expanded this approach to incorporate larger fragments as tokens (motifs) (Jin, Barzilay, and Jaakkola 2020).

Experiments

We consider three problems: distribution learning, global molecular property optimization and constrained optimization. For all the experiments, we provide model and optimization hyperparameters in supplementary material A; the source code for reproducing all the experiments is provided in supplementary materials. We consider hydrogen-depleted graphs, since hydrogens can be deduced from atom valence.

Distribution Learning

In distribution learning task, we assess how well models capture the data distribution. We compare generated and a test sets using Fréchet ChemNet distance (FCD/Test) (Preuer et al. 2018). FCD/Test is a Wasserstein-1 distance between Gaussian approximations of ChemNet's penultimate layer activations. We also computed cosine similarity between fragment frequency vectors in the generated and test sets. We report the results on MOSES (Polykovskiy et al. 2020) dataset in Table 1. MolGrow outperforms previous node-level graph generators by a large margin. Note that SMILES-based generators (CharRNN and VAE) and fragment-level generator (JTN-VAE) outperform all node-level graph models. We hypothesize that such representations impose strong prior on the generated structures. We provide samples from graph-based models in Figure 4. Note that baseline models tend to produce macrocycles which were not present in the training set; molecules produced with GraphNVP contain too few rings. Ablation study demonstrates the advantage of fragment-oriented ordering and CAGE over standard graph attention network GAT (Veličković et al. 2017). We provide the results on QM9 (Ramakrishnan et al. 2014) and ZINC250k (Kusner, Paige, and Hernández-Lobato 2017) datasets in supplementary material B. In Figure 6 of Supplementary materials, we show how resampling different latent codes affect the generated structure.

Method	Penalized logP			QED		
	1st	2nd	3rd	1st	2nd	3rd
ZINC250k	4.52	4.30	4.23	**0.948**	**0.948**	**0.948**
Graph-based models						
GCPN	7.98	7.85	7.80	**0.948**	0.947	0.946
MolecularRNN	8.63	6.08	4.73	0.844	0.796	0.736
GraphNVP	-	-	-	0.833	0.723	0.706
GraphAF	12.23	11.29	11.05	**0.948**	**0.948**	**0.948**
MoFlow	-	-	-	**0.948**	**0.948**	**0.948**
Proposed model						
Genetic	$\mathbf{14.01 \pm 0.36}$	$\mathbf{13.95 \pm 0.42}$	$\mathbf{13.92 \pm 0.42}$	$\mathbf{0.948 \pm 0.0}$	$\mathbf{0.948 \pm 0.0}$	$\mathbf{0.948 \pm 0.0}$
Genetic, Top	11.66 ± 0.31	11.65 ± 0.31	11.63 ± 0.31	$\mathbf{0.948 \pm 0.0}$	$\mathbf{0.948 \pm 0.0}$	$\mathbf{0.948 \pm 0.0}$
Genetic, Bottom	10.29 ± 3.32	10.29 ± 3.33	10.28 ± 3.32	$\mathbf{0.948 \pm 0.0}$	$\mathbf{0.948 \pm 0.0}$	$\mathbf{0.948 \pm 0.0}$
Predictor-guided	5.2 ± 0.34	4.94 ± 0.26	4.84 ± 0.22	$\mathbf{0.948 \pm 0.0}$	$\mathbf{0.948 \pm 0.0}$	$\mathbf{0.948 \pm 0.0}$
REINFORCE	4.81 ± 0.28	4.47 ± 0.14	4.39 ± 0.13	0.947 ± 0.001	0.946 ± 0.001	0.946 ± 0.001
SMILES and fragment-based models						
DD-VAE	5.86	5.77	5.64	-	-	-
Grammar VAE	2.94	2.88	2.80	-	-	-
SD-VAE	4.04	3.50	2.96	-	-	-
JT-VAE	5.30	4.93	4.49	**0.948**	0.947	0.947

Table 2: Molecular property optimization: penalized octanol-water partition coefficient (penalized logP) and quantitative estimation of drug-likeness (QED). Results for baseline models from (You et al. 2018a; Popova et al. 2019; Madhawa et al. 2019; Shi et al. 2020; Zang and Wang 2020; Polykovskiy and Vetrov 2020; Kusner, Paige, and Hernández-Lobato 2017; Dai et al. 2018; Jin, Barzilay, and Jaakkola 2018).

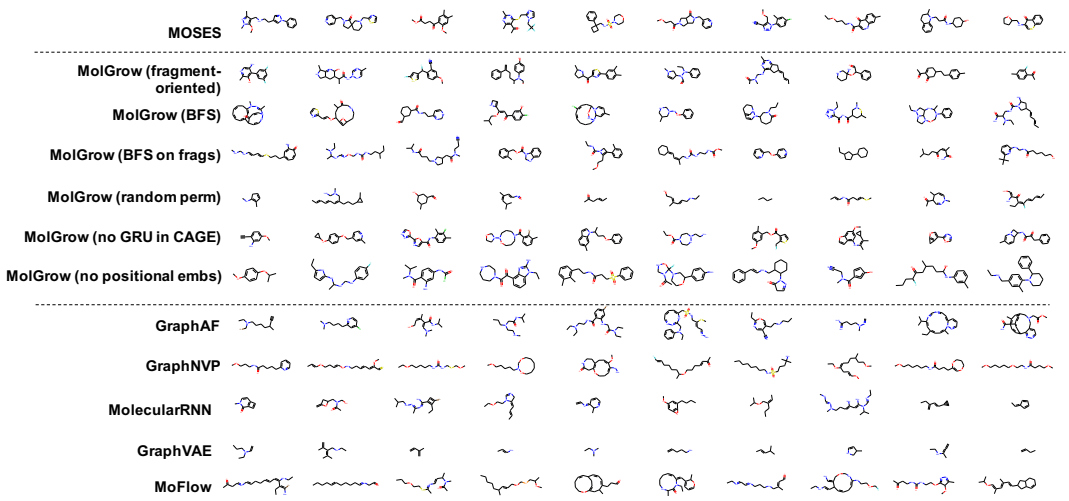

Figure 4: Samples from molecular graph generative models trained on MOSES.

Global Optimization

The goal of a global optimization task is to produce new molecules that maximize a given chemical property. Similar to the previous works, we selected two commonly used properties: penalized octanol-water partition coefficient (penalized logP) (Kusner, Paige, and Hernández-Lobato 2017) and quantitative estimation of drug-likeness (QED) (Gómez-Bombarelli et al. 2018). We considered genetic and predictor-guided optimization strategies.

For genetic optimization, we start by sampling 256 random molecules from ZINC250k dataset and computing their latent codes. Then we hierarchically optimize latent codes

δ	GCPN			GraphAF		
	Improvement	Similarity	Success	Improvement	Similarity	Success
0.0	4.20 ± 1.28	0.32 ± 0.12	100%	13.13 ± 6.89	0.29 ± 0.15	100%
0.2	4.12 ± 1.19	0.32 ± 0.11	100%	11.90 ± 6.86	0.33 ± 0.12	100%
0.4	2.49 ± 1.30	0.47 ± 0.08	100%	8.21 ± 6.51	0.49 ± 0.09	99.88%
0.6	0.79 ± 0.63	0.68 ± 0.08	100%	$\mathbf{4.98 \pm 6.49}$	0.66 ± 0.05	96.88%
δ	MoFlow			MolGrow		
	Improvement	Similarity	Success	Improvement	Similarity	Success
0.0	8.61 ± 5.44	0.30 ± 0.20	98.88%	$\mathbf{14.84 \pm 5.786}$	0.048 ± 0.038	100%
0.2	7.06 ± 5.04	0.43 ± 0.20	96.75%	$\mathbf{11.99 \pm 6.45}$	0.23 ± 0.045	99.88%
0.4	4.71 ± 4.55	0.61 ± 0.18	85.75%	$\mathbf{8.337 \pm 6.85}$	0.44 ± 0.048	99.88%
0.6	2.10 ± 2.86	0.79 ± 0.14	58.25%	4.063 ± 5.609	0.65 ± 0.068	97.78%

Table 3: Constrained optimization of penalized octanol-water partition coefficient (logP). Mean±std over 800 initial molecules with the worst penalized logP in ZINC250k.

for 3000 iterations. At each iteration we generate a new population using crossing-over and mutation and keep 256 molecules with the highest reward. In crossing-over, we randomly permute all molecules in the population to form 256 pairs. For each pair, we uniformly sample latent codes from spherical linear interpolation (Slerp) trajectory (White 2016) and reconstruct the resulting molecule. We mutate one level's latent code at each iteration. Starting with a top level, we resample 10% of the components from a Gaussian distribution. For genetic optimization, we compare different mutation and crossing-over strategies, including top level optimization with fixed bottom layers and vice versa.

In predictor-guided optimization, we followed the approach proposed by Jin, Barzilay, and Jaakkola (2018): we fine-tuned the pre-trained model jointly with a penalized logP predictor from the high-level latent codes for one epoch (MAE=0.41 for penalized logP, MAE=0.07 for QED). We randomly sampled 2560 molecules from a prior distribution and took 200 constrained gradient ascent steps along the predictor's gradient to modify the high-level latent codes; we resample low-level latent codes from the prior. We decrease the learning rate after each iteration and keep the best reconstructed molecule that falls into the constrained region. Intuitively, the gradient ascent over high-level latent codes guides the search towards better global structure, while low-level latent codes produce a diverse set of molecules with the same global structure and similar predicted values.

We report the scores of the best molecules found during optimization in Table 2 and provide optimization trajectories in supplementary information C.

Constrained Optimization

In this section, we apply MolGrow to constrained molecular optimization. In this task, we optimize a chemical property in proximity of the initial molecule. Following the previous works (Jin, Barzilay, and Jaakkola 2018; You et al. 2018a), we selected 800 molecules with the lowest penalized octanol-water partition coefficient (logP) and constrain minimum Tanimoto similarity δ between Morgan fingerprints (Rogers and Hahn 2010) of the initial and final molecules. For constrained optimization, we followed the predictor-guided approach described above and optimize each of 800 starting molecules for 200 steps. In Table 3, we report average penalized logP improvement and similarity to the initial molecule. We also report a fraction of molecules for which we successfully discovered a new molecule with higher penalized logP. Note that unlike GCPN and GraphAF baselines, we do not fine-tune the model for each starting molecule, reducing time and memory costs for optimization.

Conclusion

In this paper, we presented a new hierarchical molecular graph generative model and outperformed existing node-level models on distribution learning and molecular property optimization tasks. On distribution learning, string- and fragment-based generators still perform better than node-level models, since they explicitly handle valency and connectivity constraints. Similar to the previous models, we obtained better performance when learning on a fixed atom ordering. Our fragment-oriented ordering further improves the results over BFS. In this work, we compared generated and test sets using standard distribution learning metrics and found out that the distributions produced by previous node-level graph generators differ significantly from the test set, although these models were trained for distribution learning.

References

Bemis, G. W.; and Murcko, M. A. 1996. The properties of known drugs. 1. Molecular frameworks. *J. Med. Chem.* 39(15): 2887–2893.

Bjerrum, E. J. 2017. Smiles enumeration as data augmentation for neural network modeling of molecules. *arXiv preprint arXiv:1703.07076* .

Brown, N.; Fiscato, M.; Segler, M. H.; and Vaucher, A. C. 2019. GuacaMol: benchmarking models for de novo molecular design. *Journal of chemical information and modeling* 59(3): 1096–1108.

Dai, H.; Tian, Y.; Dai, B.; Skiena, S.; and Song, L. 2018. Syntax-Directed Variational Autoencoder for Structured Data. *arXiv preprint arXiv:1802.08786* .

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* .

Degen, J.; Wegscheid-Gerlach, C.; Zaliani, A.; and Rarey, M. 2008. On the art of compiling and using 'drug-like' chemical fragment spaces. *ChemMedChem* 3(10): 1503–1507.

Dinh, L.; Sohl-Dickstein, J.; and Bengio, S. 2017. Density estimation using real nvp. *International Conference on Learning Representations* .

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.

Guo, Z.; Zhang, Y.; and Lu, W. 2019. Attention Guided Graph Convolutional Networks for Relation Extraction. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 241–251. Florence, Italy: Association for Computational Linguistics. doi:10.18653/v1/P19-1024. URL https://www.aclweb.org/anthology/P19-1024.

Hoogeboom, E.; Van Den Berg, R.; and Welling, M. 2019. Emerging Convolutions for Generative Normalizing Flows. In Chaudhuri, K.; and Salakhutdinov, R., eds., *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, 2771–2780. Long Beach, California, USA: PMLR. URL http://proceedings.mlr.press/v97/hoogeboom19a.html.

Jin, W.; Barzilay, R.; and Jaakkola, T. 2018. Junction Tree Variational Autoencoder for Molecular Graph Generation. In Dy, J.; and Krause, A., eds., *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, 2323–2332. Stockholmsmässan, Stockholm Sweden: PMLR.

Jin, W.; Barzilay, R.; and Jaakkola, T. 2020. Hierarchical Generation of Molecular Graphs using Structural Motifs. *International Conference on Machine Learning* .

Kingma, D. P.; and Dhariwal, P. 2018. Glow: Generative flow with invertible 1x1 convolutions. In *Advances in Neural Information Processing Systems*, 10215–10224.

Krenn, M.; Häse, F.; Nigam, A.; Friederich, P.; and Aspuru-Guzik, A. 2019. SELFIES: a robust representation of semantically constrained graphs with an example application in chemistry. *arXiv preprint arXiv:1905.13741* .

Kusner, M. J.; Paige, B.; and Hernández-Lobato, J. M. 2017. Grammar variational autoencoder. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 1945–1954. JMLR. org.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* .

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

Merk, D.; Friedrich, L.; Grisoni, F.; and Schneider, G. 2018. De novo design of bioactive small molecules by artificial intelligence. *Molecular informatics* 37(1-2): 1700153.

O'Boyle, N.; and Dalke, A. 2018. DeepSMILES: An Adaptation of SMILES for Use in Machine-Learning of Chemical Structures. *ChemRxiv* .

Parisotto, E.; Song, H. F.; Rae, J. W.; Pascanu, R.; Gulcehre, C.; Jayakumar, S. M.; Jaderberg, M.; Kaufman, R. L.; Clark, A.; Noury, S.; et al. 2020. Stabilizing transformers for reinforcement learning. *International Conference on Machine Learning* .

Polykovskiy, D.; and Vetrov, D. 2020. Deterministic Decoding for Discrete Data in Variational Autoencoders. In Chiappa, S.; and Calandra, R., eds., *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, 3046–3056. PMLR.

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.

Popova, M.; Shvets, M.; Oliva, J.; and Isayev, O. 2019. MolecularRNN: Generating realistic molecular graphs with optimized properties. *arXiv preprint arXiv:1905.13372* .

Preuer, K.; Renz, P.; Unterthiner, T.; Hochreiter, S.; and Klambauer, G. 2018. Fréchet ChemNet Distance: A Metric for Generative Models for Molecules in Drug Discovery. *J. Chem. Inf. Model.* 58(9): 1736–1741.

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: 140022.

Rogers, D.; and Hahn, M. 2010. Extended-connectivity fingerprints. *Journal of chemical information and modeling* 50(5): 742–754.

Sanchez-Lengeling, B.; Outeiral, C.; Guimaraes, G. L.; and Aspuru-Guzik, A. 2017. Optimizing distributions over molecular space. An Objective-Reinforced Generative Adversarial Network for Inverse-design Chemistry (ORGANIC). *ChemRxiv* doi:10.26434/chemrxiv.5309668.v3.

Segler, M. H.; Preuss, M.; and Waller, M. P. 2018. Planning chemical syntheses with deep neural networks and symbolic AI. *Nature* 555(7698): 604–610.

Segler, M. H. S.; Kogej, T.; Tyrchan, C.; and Waller, M. P. 2018. Generating Focused Molecule Libraries for Drug Discovery with Recurrent Neural Networks. *ACS Cent Sci* 4(1): 120–131.

Senior, A. W.; Evans, R.; Jumper, J.; Kirkpatrick, J.; Sifre, L.; Green, T.; Qin, C.; Žídek, A.; Nelson, A. W.; Bridgland, A.; et al. 2020. Improved protein structure prediction using potentials from deep learning. *Nature* 1–5.

Shi, C.; Xu, M.; Zhu, Z.; Zhang, W.; Zhang, M.; and Tang, J. 2020. GraphAF: a flow-based autoregressive model for molecular graph generation. *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*, 3693–3702.

Simonovsky, M.; and Komodakis, N. 2018. GraphVAE: Towards Generation of Small Graphs Using Variational Autoencoders. In Kůrková, V.; Manolopoulos, Y.; Hammer, B.; Iliadis, L.; and Maglogiannis, I., eds., *Artificial Neural Networks and Machine Learning – ICANN 2018*, 412–422. Cham: Springer International Publishing. ISBN 978-3-030-01418-6.

Tran, D.; Vafa, K.; Agrawal, K.; Dinh, L.; and Poole, B. 2019. Discrete flows: Invertible generative models of discrete data. In *Advances in Neural Information Processing Systems*, 14719–14728.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. In *Advances in neural information processing systems*, 5998–6008.

Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio, P.; and Bengio, Y. 2017. Graph attention networks. *International Conference on Learning Representations* .

Vinyals, O.; Bengio, S.; and Kudlur, M. 2016. Order matters: Sequence to sequence for sets. *International Conference on Learning Representations* .

Weininger, D.; Weininger, A.; and Weininger, J. L. 1989. SMILES. 2. Algorithm for generation of unique SMILES notation. *Journal of chemical information and computer sciences* 29(2): 97–101.

White, T. 2016. Sampling generative networks. *arXiv preprint arXiv:1609.04468* .

You, J.; Liu, B.; Ying, Z.; Pande, V.; and Leskovec, J. 2018a. Graph convolutional policy network for goal-directed molecular graph generation. In *Advances in neural information processing systems*, 6410–6421.

You, J.; Ying, R.; Ren, X.; Hamilton, W.; and Leskovec, J. 2018b. GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models. In Dy, J.; and Krause, A., eds., *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, 5708–5717. Stockholmsmässan, Stockholm Sweden: PMLR. URL http://proceedings.mlr.press/v80/you18a.html.

Zang, C.; and Wang, F. 2020. MoFlow: An Invertible Flow Model for Generating Molecular Graphs. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* doi:10.1145/3394486.3403104. URL http://dx.doi.org/10.1145/3394486.3403104.

Zhavoronkov, A.; Ivanenkov, Y. A.; Aliper, A.; Veselov, M. S.; Aladinskiy, V. A.; Aladinskaya, A. V.; Terentiev, V. A.; Polykovskiy, D. A.; Kuznetsov, M. D.; Asadulaev, A.; et al. 2019. Deep learning enables rapid identification of potent DDR1 kinase inhibitors. *Nature biotechnology* 1–4.