Practical notes on building molecular graph generative models

Received: 28 August 2020 | Revised: 11 November 2020 | Accepted: 1 December 2020

DOI: 10.1002/ail2.18

LETTER

WILEY

Practical notes on building molecular graph generative models

Rocío Mercado[1] | **Tobias Rastemo**[1,2] | **Edvard Lindelöf**[1,2] | **Günter Klambauer**[3] | **Ola Engkvist**[1] | **Hongming Chen**[4] | **Esben Jannik Bjerrum**[1]

[1]Molecular AI, Discovery Sciences, BioPharmaceuticals R&D, AstraZeneca, Gothenburg, Sweden

[2]Chalmers University of Technology, Gothenburg, Sweden

[3]Institute of Bioinformatics, Johannes Kepler University, Linz, Austria

[4]Centre of Chemistry and Chemical Biology, Guangzhou Regenerative Medicine and Health, Guangdong Laboratory, Guangzhou, China

Correspondence
Rocío Mercado, Molecular AI, Discovery Sciences, BioPharmaceuticals R&D, AstraZeneca, Gothenburg, Sweden.
Email: rocio.mercado@astrazeneca.com

Abstract

Here are presented technical notes and tips on developing graph generative models for molecular design. Although this work stems from the development of GraphINVENT, a Python platform for iterative molecular generation using graph neural networks, this work is relevant to researchers studying other architectures for graph-based molecular design. In this work, technical details that could be of interest to researchers developing their own molecular generative models are discussed, including an overview of previous work in graph-based molecular design and strategies for designing new models. Advice on development and debugging tools which are helpful during code development is also provided. Finally, methods that were tested but which ultimately did not lead to promising results in the development of GraphINVENT are described here in the hope that this will help other researchers avoid pitfalls in development and instead focus their efforts on more promising strategies for graph-based molecular generation.

KEYWORDS

code development, code optimization, deep learning, drug discovery, generative models, graph neural networks, molecular design

1 | INTRODUCTION

Molecular generative models have emerged as promising methods for exploring the chemical space through de novo molecular design.[1-16] Although molecular generative models have largely focused on string-based approaches, graph-based approaches have also emerged in the last 3 years,[10-24] including a recent approach, GraphINVENT,[25] from our group. Like GraphINVENT, many graph-based generative models are built using graph neural networks (GNNs), while many others are built using variational autoencoders (VAEs), generative adversarial networks (GANs), and combinations thereof.

Abbreviations: AE, autoencoder; APD, action probability distribution; DNN, deep neural network; GAN, generative adversarial network; GCN, graph convolutional network; GCF, graph conditional flow; GNN, graph neural network; GRAN, Graph Recurrent Attention Network; HO, hyperparameter optimization; MLP, multilayer perceptron; MPNN, message-passing neural network; PPT, percent properly terminated; PU, percent unique; PV, percent valid; PVPT, percent valid of properly terminated; RL, reinforcement learning; RNN, recurrent neural network; SA, synthetic accessibility; SGD, stochastic gradient descent; SOTA, state-of-the-art; VAE, variational autoencoder.

All of the aforementioned methods can be powerful architectures for modeling patterns in graph-structured data. GNN-, VAE-, and GAN-based models have shown recent promise in molecular design applications,[10,11,14,16,25] just as recurrent neural networks (RNNs) transformed string-based molecular generation in 2016.[26] However, when it comes to development tools, there is a lack of practical information surrounding the construction of molecular graph generative models; for example, things like why a specific architecture/method was chosen, or if the authors tried any other methods unsuccessfully, are rarely, if ever, explained. This motivated the creation of this text, where choices made during development are detailed in the hopes of helping other researchers developing their own generative models.

In sections 2 and 3, we begin by introducing the most common graph formulation and summarizing some important work in the field of graph-based molecular generation. Here, the key differences between graph-based generative models schemes are highlighted. Then, sections 4 to 6 go into strategies for selecting a graph representation, a generation scheme, and a model architecture. This is followed by a discussion on the strategies for solving the two biggest challenges faced during the development of graph-based molecular generative models: improving the memory requirements of jobs (section 7) and improving GPU utilization (section 8). As this work stems from the development of GraphINVENT, details on methods tested in GraphINVENT are discussed in section 9, and specific functions and schemes that were tried but failed are highlighted here. Lastly, tips on general development tools are provided in section 10.

2 | FORMULATION

Throughout this work, we use the notation $\mathcal{G} = (A,X)$ to represent a molecular graph $\mathcal{G}$, where the matrix graph representation consists of an adjacency tensor $A \in \{0, 1\}^{N\times N\times B}$ and a node feature matrix $X \in \{0, 1\}^{N\times M}$. Here, N is the number of nodes in the graph, M is the number of types of nodes, and B is the number of types of edges. In other words, M represents the types of atoms (eg, carbon and nitrogen) in a molecule, and B represents the types of bonds (eg, single, double, triple, etc).

This is by far the most common molecular representation used in graph-based generative models. For a discussion on how to choose a representation, see section 4.

3 | PREVIOUS WORK

This section provides an overview of previous work in deep graph-based molecular generation. In general, graph-based molecular generative models can be classified into two types: *single-shot* and *iterative* models. In single-shot molecular graph generation, new graphs $\mathcal{G} = (A,X)$ will typically be generated in one or two steps; most often, this is done by first generating an adjacency tensor A from a corresponding latent graph representation, and then generating a node features tensor X which "fills in" the identities of the nodes in A. On the other hand, in iterative molecular graph generation, the molecular graph is generated in a step-by-step fashion, typically starting from an empty graph $\mathcal{G}_0 = (A_0,X_0)$, then adding nodes and/or edges sequentially by sampling from a learned distribution until the graph is complete (eg, $\mathcal{G}_0 \rightarrow \mathcal{G}_1 \rightarrow \cdots \rightarrow \mathcal{G}_{N-1} \rightarrow \mathcal{G}_N$).

There are advantages to each scheme. For example, although single-shot graph generation methods are generally easier to implement, iterative methods are better equipped for dealing with and generating large molecules.

The first works on deep molecular graph generation using deep learning focused on VAE and GAN architectures, to learn mappings between the graph representation and their latent vector representations.[17,18] These methods learned distributions for single-shot graph generation using deep neural networks (DNNs) from a set of training examples. However, these were quickly followed by the publication of methods using GNNs and iterative generation schemes.[10,11,14] Nonetheless, the emergence of these methods can almost be said to be simultaneous, as all of the aforementioned works were first reported in 2018.

Although initial methods were focused on the generation of small molecular graphs (ie, <10 atoms) due to computational limitations,[17,18] developments in the field have made it such that recently published methods can easily generate molecular graphs with 10 to 100 nodes.[12,16,23,25] Developments in the field of graph-based molecular design happen very quickly, as they closely follow developments in graph representation learning, just as string-based molecular generation quickly follows developments in natural language processing.

The methods discussed in this section are for "2D" molecular graph generation in the sense that the models generate molecules without any coordinate information. Nonetheless, there do exist promising methods for 3D molecular

generation, such as CVGAE,[22] GraphDG,[27] G-SchNet,[28,29] WGAN,[30] and CGCF,[31] for which we point the reader to the aforementioned references.

3.1 | Commonly used datasets

Throughout this section, we refer to datasets that are widely used within the deep molecular generation community. These datasets are:

1. ZINC[32]: free database of commercially available, drug-like compounds for virtual screening (>230M structures).
2. ChEMBL[33,34]: large, open-access drug discovery database containing molecules, assays, etc, obtained from the literature and other databases (>1.6M structures).
3. GDB-13[35]: enumerated database of small organic molecules of up to 13 heavy atoms of C, N, O, S, and Cl, following simple chemical stability and synthetic feasibility rules (977M molecules).
4. GDB-17[36]: enumerated database of molecules of up to 17 heavy atoms of C, N, O, S, and halogens, covering a drug-like range and including millions of isomers (166B molecules).

Additionally, many datasets used are subsets of the datasets mentioned above, such as the MOSES[37] dataset, which is a subset of the ZINC Clean Leads collection. Similarly, ZINC-250K is a subset of 250K randomly selected molecules from the ZINC database, containing up to 38 heavy atoms per molecule. Finally, QM9[38] is a subset (134K molecules) of the massive GDB-17 dataset, where structures in QM9 contain up to nine heavy atoms of C, N, O, and F. Especially QM9 and ZINC-250K have been widely used in deep molecular generation using graphs.

3.2 | Single-shot graph generation

Single-shot graph generation is the idea of generating graphs in a "single" step, in contrast to using an iterative scheme to build graphs. This is similar to how image generation works, where images can be thought of as regular graphs. Below, we summarize a few models that follow this scheme: MolGAN,[18] GraphNVP,[8] RVAE,[39] and MoFlow.[23] For additional single-shot models, see GRF,[40] GraphVAE,[21] GraphVAE-approx,[41] DEFactor,[20] GCAE,[42] and LF-MolGAN.[43]

3.2.1 | MolGAN: De Cao et al (2018)[18]

GANs consist of two main components: a generative model, or generator, G_θ, and a discriminative model, or discriminator, D_θ.[44] While G_θ learns to generate new data-points from a prior, D_θ evaluates them and learns to classify whether the samples come from the data distribution or from G_θ. The two models are implemented using DNNs and trained simultaneously using techniques such as stochastic gradient descent (SGD), such that the goal of G_θ is to fool D_θ, and the goal of D_θ is to correctly classify the samples. The two models can be seen as players in a zero-sum game, as one agent's loss is the other's gain.

MolGAN is an implicit, likelihood-free generative model for small molecular graphs that uses GANs on graph-structured data. The authors use their model in a reinforcement learning (RL) framework to generate molecules with specific desired properties. Although MolGAN generates nearly 100% valid molecules and is the first GAN-based model used for graph-based molecular design, it is susceptible to mode collapse, which results in a large number of repeats in the generated set (∼2% unique).

The MolGAN architecture consists of three components: the generator G_θ, the discriminator D_θ, and the reward network $\hat{R}_\psi$. While G_θ learns to generate new molecules and D_θ learns to distinguish between generated and training set compounds, $\hat{R}_\psi$ learns to assign a reward to each molecule that matches either the synthetic accessibility (SA) score[45] calculated by RDKit, the log octanol-water partition coefficient (logP), or the Quantitative Estimate of Druglikeness[46] score. These are the three metrics used for evaluating the model for goal-directed generation and compare to published state-of-the-art (SOTA) models.

This work is similar to the SMILES-based ORGAN model by Guimaraes et al,[26] although MolGAN not only uses the graph representation instead of strings, but also a different RL algorithm (DDPG instead of REINFORCE). The authors

observe that MolGAN outperforms ORGAN in terms of the validity of samples, as well as in the three RL objective scores studied; nonetheless, ORGAN does not suffer from mode collapse.

The source code for MolGAN is publicly available at https://github.com/nicola-decao/MolGAN.

Datasets used in this work: QM9.

3.2.2 | RVAE: Ma et al (2018)[39]

VAE models consist of a stochastic encoder and an imperfect decoder, where the decoder learns to generate molecules from latent representations by minimizing the reconstruction loss. For graph-based models, calculating the reconstruction loss often involves computationally expensive graph comparisons. VAE-based models assume a simple variational distribution for latent representation vectors.

In this work, the authors propose a regularization framework for VAEs by introducing penalty terms for the output distribution of the decoder using a Lagrangian function, which biases the model toward the generation of valid molecular graphs (ie, validity constraints), thus presenting RVAEs, or *regularization* VAEs. The penalties regularize the distributions of the types of nodes and edges which can exist in a molecular graph (eg, possible valence states, elements, etc). The authors demonstrate that introducing regularization increases the probability of generating valid graphs compared to the standard VAE; the reported percent validities for the generated sets were 97% for RVAE trained on QM9 and 35% for RVAE trained on ZINC.

The source code for RVAE was not made publicly available.

Datasets used in this work: QM9, ZINC-250K.

3.2.3 | GraphNVP: Madhawa et al (2019)[8]

The goal of normalizing flow-based models is to learn mappings between complex distributions and simple prior distributions through invertible neural networks; such models benefit from exact and tractable likelihood estimation for training, efficient single-shot inference, and invertible mapping—thus ensuring for the faithful reconstruction of all the training data. Flow-based invertible transformations on graphs can be used for tasks such as node and graph classification, as well as graph generation.

GraphNVP is a model for graph-based molecular generation using normalizing flows (a typical invertible flow). Normalizing flows were originally developed for image generation tasks; however, as image pixels fall on regular grids, the normalizing flow architecture has to be modified to apply to molecular graphs, which are irregular. GraphNVP works by generating graphs in two steps: first, it generates an adjacency tensor A, followed by the generation of the node attributes X. It does this by learning two latent representations, one for A, and one for X conditioned on A. The objective of GraphNVP is to learn an invertible model $f_\theta : \mathcal{G} \mapsto g \in \mathbb{R}^D$, with learnable parameters θ and $D = N \times N \times B + N \times M$.

Flow-based models are invertible by design, and as such perfect reconstruction of a graph from a latent representation is guaranteed in GraphNVP. The models generate almost no repeat graphs (irrespective of the number of samples). Optimizations along the latent space can be used to generate molecular graphs with specific desired properties without any expert/domain knowledge.

The source code for GraphNVP is available at https://github.com/pfnet-research/graph-nvp.

Datasets used in this work: QM9, ZINC-250K.

3.2.4 | MoFlow: Zang et al (2020)[23]

MoFlow is a flow-based graph generative model for learning invertible mappings between molecular graphs and their latent representations. In MoFlow, a graph conditional flow (GCF) for atoms $f_{X \mid A}$ transforms X using A into a conditional latent vector $Z_{X \mid A}$, whereas the GCF for bonds f_A transforms A into latent vector Z_A. The generation process uses the reverse transformations of the inference operations, followed by validity correction.

Like GraphNVP,[8] MoFlow works by first generating an adjacency tensor A through a Glow-based model,[47] then generating X by a novel GCF which depends on A, and finally applying a validity correction to a generated graph.

However, MoFlow is different from GraphNVP[8] and GRF,[40] in that all structures generated by MoFlow are guaranteed to be chemically valid thanks to the post hoc validity corrections.

The authors validate their model by looking at the percent validity of generated structures (both with and without the validity correction), percent uniqueness, percent novel, and percent NUV (novel, unique, and valid), as well as the reconstruction accuracy. The authors show that MoFlow not only memorizes and reconstructs all training molecular graphs, but also generates novel, unique, and valid molecular graphs at a higher rate than existing SOTA methods. The models generalize well from the training data, making this a promising tool for chemical space exploration.

The source code for MoFlow is available at https://github.com/calvin-zcx/moflow.

Datasets used in this work: QM9, ZINC-250K.

3.3 | Iterative graph generation

In iterative graph generation, molecules are generated in a step-by-step manner by adding components (eg, atoms) to an incomplete molecular graph one step or "action" at a time. Below we summarize a few models which follow this scheme: NeVAE,[17] a model from DeepMind,[10] MolMP and MolRNN,[11] JT-VAE,[12] HierVAE,[16] and GraphINVENT.[25] For additional examples of iterative molecular graph generation, see GCPN,[14] MolecularRNN[48] (extension of GraphRNN[15] to molecules), CGVAE,[13] RL-VAE,[49] GraphAF,[24] and ScaffoldVAE.[19]

Additionally, while Graph Recurrent Attention Networks (GRANs)[50] have not yet, to our knowledge, been used to generate molecular graphs, they have been used with great success to generate protein graphs, where each node represents an individual amino acid. Furthermore, the authors claim that they can use GRANs to generate graphs of up to 5K nodes with good quality, making it the first deep graph generative model to scale to such large chemical graphs.

3.3.1 | NeVAE: Samanta et al (2018)[17]

To our knowledge, NeVAE was the first model published to carry out deep *graph*-based molecular generation. NeVAE applies a VAE architecture to molecular graphs, whose encoder and decoder are specially designed to account for (a) irregularity in graphs, (b) node order invariance, and (c) variable size. The decoder is also able to provide spatial coordinates for the atoms in the generated molecules, such that the model is able to optimize spatial configurations for the generated molecules.

The authors benchmark their model in terms of the percent validity. Notably, invalid edges are masked (using domain knowledge) during the decoding process, and without the mask, the generated molecules have a relatively low percent validity (∼60%-70%) compared to SOTA methods.

The source code for NeVAE is available at https://github.com/Networks-Learning/nevae.

Datasets used in this work: QM9, 10K samples from ZINC.

3.3.2 | DeepMind: Li et al (2018)[10]

In Reference 10, a team from DeepMind use a GNN-based model to generate new graphs in an iterative fashion. Their models can generate both synthetic graphs and real molecular graphs, unconditionally and conditioned on data.

The decision space is split into three possible actions for building the graphs: f_{addnode}, which determines whether to terminate the graph building process or add a new node to the graph; f_{addedge}, which determines whether to add a new edge to a newly created node; and f_{nodes}, which determines a score for each node and thus determines which node to build upon next. All of these actions use the learned node and graph embeddings, which are computed using message-passing neural networks (MPNNs), a class of GNNs. The model generates molecules with high percent validity and novelty (>90% for both) and can be used for arbitrary graphs.

Li et al[10] did not publish any code.

Datasets used in this work: ZINC-250K, 30K samples from ChEMBL.

3.3.3 | MolMP and MolRNN: Li et al (2018)[11]

In Reference 11, molecular graphs are built by iteratively sampling actions and applying them to incomplete graphs until the "terminate" action is sampled. As opposed to Reference 10, in MolMP and MolRNN the action space is split into four possible actions for building subgraphs: *initialization*, which adds a single node to an empty graph; *append*, which adds a new node and connects it to an existing node in the graph; *connect*, which connects the last appended node to another node in the graph; and *termination*, which ends the graph generation process. The first model, MolMP, uses a graph convolution network (GCN) by Wu et al[51] to learn the action space for building new graphs, and treats graph generation as a Markov process. In the MolRNN model, a recurrent unit is added to MolMP such that graph generation is no longer Markovian.

The authors found that introducing recurrence into their models (MolRNN) led to significant improvement in the drugs generated. MolRNN outperforms MolMP in both general molecular generation metrics (validity, uniqueness, and novelty), as well as in goal-directed molecular generation, where the generative model is used to produce molecules with specific druglikeness and SA scores. The models are used to generate molecules with up to 50 heavy atoms, whereas many of the previously discussed methods were limited to smaller graphs (<20 heavy atoms).

Notably, the term *decoding route* is introduced in this work to refer to the specific route, r, that is taken to construct a particular graph such that $r = ((\mathcal{G}_0, t_0), (\mathcal{G}_1, t_1), ..., (\mathcal{G}_N, t_N))$.

The source code for this work is available at https://github.com/kevinid/molecule_generator.

Dataset used in this work: ChEMBL subset of molecules containing up to 50 heavy atoms.

3.3.4 | JT-VAE: Jin et al (2018)[12]

This work introduces the junction tree variational encoder, or JT-VAE, a graph-based generative model that generates molecules in two iterative steps. The model works by first generating a junction tree (a tree-structured object) representing a scaffold of molecular components and their coarse relative arrangements. These substructures are valid chemical building blocks automatically extracted from the training set using tree decomposition. The model then assembles the substructures (nodes in a tree) into a molecule using a GNN. The validity of all sampled molecules is guaranteed by nature of the algorithm.

In summary, a graph $\mathcal{G}$ is mapped into a junction tree $\mathcal{T}_\mathcal{G}$ via a tree decomposition algorithm which contracts certain vertices into a single node, such that $\mathcal{T}_\mathcal{G}$ contains no cycles. The latent representations of $\mathcal{G}$ and $\mathcal{T}_\mathcal{G}$ are then encoded using two separate GNNs. Then, the junction tree $\mathcal{T}$ is decoded from its encoding $z_\mathcal{T}$ using a tree structured decoder. The final step of the model is to decode the molecular graph $\mathcal{G}$ that underlies the predicted junction tree $\mathcal{T}$.

The authors use Bayesian optimization (BO) in the latent space of the junction tree to search for molecules with specific properties. The JT-VAE outperforms other VAE-based molecular generation methods both in terms of validity of generated molecules (100%) and molecular property scores using BO.

The source code for this model is available at https://github.com/wengong-jin/icml18-jtnn.

Dataset used in this work: ZINC-250K.

3.3.5 | HierVAE: Jin et al (2020)[16]

This work follows up on the JT-VAE and introduces a hierarchical graph encoder-decoder architecture for sequentially generating molecular graphs using GNNs. In HierVAE (also referred to as hgraph2graph), first, the encoder creates a multi-resolution representation for each molecule in a fine-to-coarse fashion, from the atom layer, to the attachment layer, to the motif layer. Then, the autoregressive decoder works in a coarse-to-fine fashion to generate first the motif layer, one motif at a time, then determining the attachment points, and finishing up with the atom layer.

This hierarchical GNN architecture enables the authors to generate larger molecular graphs (eg, polymers) than are typical for deep molecular generative models, as the building blocks (molecular motifs) are large and thus fewer steps are required to build a large molecule as would be required if the building blocks were individual atoms. HierVAE outperforms previous generative models, including the JT-VAE, in terms of translation accuracy (translation from less desirable to more desirable compounds), reconstruction accuracy, and output diversity. However, the low training speed (25 molecules per second) still leaves room for improvement.

The source code for this model is available at https://github.com/wengong-jin/hgraph2graph.

Dataset used in this work: St. Johns et al[52] polymer dataset (86K molecules with >80 atoms).

3.3.6 | GraphINVENT: Mercado et al (2020)[25]

GraphINVENT is a model introduced by our group for iteratively generating molecular graphs using GNNs. It uses atomistic-level building blocks (atoms/bonds) and has a tiered DNN structure, where a GNN is first used to learn latent node and graph representations for a molecular graph, and the learned representations are used as input to a series of feed-forward networks to predict the next "action" to apply to the input graph. The models were benchmarked using the open-source MOSES[37] benchmarks, and the best GraphINVENT model was shown to compare favorably with SOTA methods, especially in terms of the diversity of the generated structures. GraphINVENT applies no mask to the sampling space, such that the percent validity of the generated structures (96% in the best reported model) is decent, although leaves room for improvement. GraphINVENT has yet to be used for goal-directed molecular generation.

Notably, the training speed of GraphINVENT is seemingly 10 times faster than HierVAE (eg, 287 molecules per second vs 25), although comparisons on the same dataset and equipment have yet to be done.

The source code for GraphINVENT is available at https://github.com/MolecularAI/GraphINVENT.

Datasets used in this work: GDB-13, MOSES.

4 | CHOOSING A REPRESENTATION

As introduced in section 2, molecular graphs in deep generative models are typically represented using two tensors: one for representing the edges and connectivity in a graph, such as an adjacency tensor A, and the other for representing the identity of the nodes in a graph, such as a node features matrix X. However, this is by no means the only reasonable molecular graph representation. As such, it could be worthwhile to spend some time here to find a representation that works well for a specific generative task, as there is no unique or standard way to represent a molecular graph.

What information should you include and what information should you ignore when describing nodes? It seems natural to include information such as the atom types (eg, carbon, nitrogen, etc) and the bond types (eg, single, double, etc), but what about information such as formal charge, valency, implicit hydrogens, hybridization, aromaticity, or chirality? This depends on what types of molecules you want to generate. Generally speaking, atom type, formal charge, valency, and chirality are enough to satisfactorily represent small organic molecules. However, the more information is included, the more information a model will have to learn patters from. Nonetheless, keep in mind that larger graph representations also mean the computational requirements of models will increase.

Information related to quantum mechanical properties of an atom, or spatial information corresponding to a molecular structure, could also be included. Whether or not to include it depends on whether it will improve the quality of the structures generated by the models. Is the information relevant to the types of graphs you are interested in building?

Finally, there is no reason that the nodes must represent atoms in a molecule and the edges must represent bonds. This is just an arbitrary choice based on our intuition of what bonds and atoms look like, and it can in fact work just as well to do the opposite.[53]

5 | DESIGNING A GENERATION SCHEME

In building a generative model, the next step is deciding on a scheme; that is, will the model generate graphs in a single-shot way, or using an iterative approach?

If choosing to generate graphs using a single-shot approach, then it will be important to decide in which order to generate components of the graph representation (eg, A and X), and whether or not any component will depend on another. As shown in section 3, the most common method for generating graphs in a single-shot way is to first generate A, and then generate X using the information predicted in A. However, this is not the only possible scheme, and it could be that there are better sampling schemes.

If choosing to generate graphs using an iterative approach, the first step in designing a new model is selecting an action space for how graphs will be constructed. The action space will determine how the molecule is built, such as a single atom at a time (atomistic) or many atoms at a time (fragments). As each additional action required to build a molecule adds computational expense to a model, it is important to choose actions which suit the problem. Commonly, when representing molecules as graphs, the nodes represent individual atoms in the graphs, and the pairwise edges represent bonds. Nonetheless, this does not have to be the case; for example, molecular generative models have been developed where a single node in a graph can represent a group of atoms.[12,16]

For example, in GraphINVENT, the action space is split into three atomistic actions: *add*, *connect*, and *terminate*. The *add* action adds a new node and connects it to the graph with a new edge. The *connect* action connects two existing nodes in the graph. The *terminate* action ends the graph generation. These actions, encoded as vectors, then become the target "properties" to fit during training, and are the properties which are sampled during the graph generation process. However, we showed in section 3 that there are other ways to split an action space, as in the DeepMind and MolMP models.

In analogy to the single-shot approach, one can imagine splitting up the action space such that nodes and edges are added in separate actions. Similarly, a single action can also be used to add more than a single node. As GraphINVENT was to be applied to small molecule generation, an atomistic approach for building graphs a single node/edge at a time was a good way to sample chemical space while minimizing unwanted bias. However, if the goal is to generate large molecules, it might be desirable to add many nodes at once (as in JT-VAE[12] or HierVAE[16]) because the number of actions required to build any molecule thus become fewer and the models can build large molecules faster.

One can also incorporate elements of recurrence in designing the action space; an example of how to do this is given by MolRNN.[11] Recurrence could also be applied during each action, for example, selecting the *add* action leads to another network for predicting the atom type to add, which then leads to another network for predicting the formal charge of the new atom, etc. However, to our knowledge, this has not yet been done in deep graph-based molecular generative models.

Once the action space is determined, the training data must be processed in such a way that the model can build each of these atoms using the defined actions. In models such as the DeepMind approach,[10] training data are seemingly generated on-the-fly. However, in GraphINVENT, this is done via a separate preprocessing phase, which was necessary in order to minimize the GPU memory requirement (section 7) of training jobs and increase GPU utilization (section 8), thus making it possible to scale to larger datasets.

6 | SELECTING A NETWORK ARCHITECTURE

Selecting an architecture for graph generative models means anticipating what class of models will perform best, while also taking into consideration other factor, for example, if chemical rules will need to be hard-coded into the model, how many parameters will need to be tuned, and so on.

A few examples of models using different architectures for molecular design have been provided in section 3. We encourage the interested reader to check out those references for details on why those architectures were chosen; often, an architecture is chosen because it has not been investigated previously for molecular generation tasks. Below, we provide a few insights into why the GraphINVENT architecture was chosen, including architectures that were experimented with but ultimately discarded due to their failures to learn the action space and generate reasonable molecular graphs.

A GNN architecture was eventually selected for GraphINVENT due in part to the success of GNNs in recently published graph generative models,[10,11,14] and in part due to the good performance of GNNs in molecular property prediction tasks.[54-57] Molecular graph generation can also be framed as a complex property prediction problem, where the target "property" for a given input graph is the correct action for building a graph. Based on previous work,[10,11] a tiered approach to molecular graph generation was selected, where a GNN is first used to generate node and graph embeddings, followed by a second network which converts the said node and graph embeddings into properties.

Besides exploring a variety of GNNs, different architectures were also experimented with in the second (and final) block of the networks. This second block is what takes the latent node and graph embeddings and returns the target actions. This is described in detail below.

6.1 | Model architecture in GraphINVENT

The models in GraphINVENT consist of two blocks:

1. a GNN block and
2. a global readout block.

The output of the first block is used as input to the second block. Implementing various GNNs in the GNN block was straightforward using the MPNN framework, as one can easily experiment with different message passing and message update functions. More challenging was finding a suitable global readout block, as having a suboptimal block here leads to models which generate too many invalid structures (even if the initial GNN is properly trained); this makes it difficult to compare GNN blocks.

Various global readout blocks were thus experimented with to find what worked best; these are described below.

6.1.1 | Node-only global readout block

The node-only multilayer perceptron (MLP) block below has a very simple functional form as it simply takes as input the final transformed node feature states (no graph embedding) and uses a unique MLP to predict each action probability distribution (APD) component: f_{add}, f_{conn}, and f_{term}. The output is then concatenated and normalized using the softmax to get the final APD.

$$f_{\text{add}} = \text{MLP}^{\text{add}}\left(H^L\right)$$

$$f_{\text{conn}} = \text{MLP}^{\text{conn}}\left(H^L\right)$$

$$f_{\text{term}} = \text{MLP}^{\text{term}}\left(H^L\right)$$

$$\text{APD} = \text{SOFTMAX}[f_{\text{add}}, f_{\text{conn}}, f_{\text{term}}].$$

This global readout block did not work as well as the others. The biggest issue with using this global readout block was that the models had difficulty learning to form rings. As this global readout block does not make use of the learned graph embedding, g, this was unsurprising.

6.1.2 | Tiered global readout block A

A tiered MLP block that uses both the node and graph embeddings was found to work significantly better. The block consists of three MLPs, as in the node-only block, to generate a *preliminary* APD, where the preliminary APD components are then concatenated with the graph embedding and used as input to a final series of MLPs to obtain the final APD components.

1. Obtain preliminary APD components:

$$f'_{\text{add}} = \text{MLP}^{\text{add},1}\left(H^L\right)$$

$$f'_{\text{conn}} = \text{MLP}^{\text{conn},1}\left(H^L\right)$$

$$f'_{\text{term}} = \text{MLP}^{\text{term},1}\left(H^L\right).$$

2. Obtain the final APD using above output and g:

$$f_{\text{add}} = \text{MLP}^{\text{add},2}\left(\left[f^{',\text{add}}, g\right]\right)$$

$$f_{\text{conn}} = \text{MLP}^{\text{conn},2}\left(\left[f^{',\text{conn}}, g\right]\right)$$

$$f_{\text{term}} = \text{MLP}^{\text{term},2}\left(\left[f^{',\text{term}}, g\right]\right)$$

$$\text{APD} = \text{SOFTMAX}[f_{\text{add}}, f_{\text{conn}}, f_{\text{term}}].$$

This readout block worked very well, but could be made more efficient by removing redundancies, leading to the next readout block.

6.1.3 | Tiered global readout block B

The best global readout block—and the one reported in Reference 25—was the following tiered MLP block. In this block, two MLPs are first used to generate a *preliminary* APD. The preliminary APD components are then concatenated with the graph embedding and used as input to a final series of MLPs to get the final APD components. Note that, as opposed to the tiered MLP block above, f_{term} only depends on the graph embedding.

1. Obtain preliminary APD components:

$$f'_{\text{add}} = \text{MLP}^{\text{add},1}\left(H^L\right)$$

$$f'_{\text{conn}} = \text{MLP}^{\text{conn},1}\left(H^L\right).$$

2. Obtain the final APD using above output and g:

$$f_{\text{add}} = \text{MLP}^{\text{add},2}\left(\left[f'_{\text{add}}, g\right]\right)$$

$$f_{\text{conn}} = \text{MLP}^{\text{conn},2}\left(\left[f'_{\text{conn}}, g\right]\right)$$

$$f_{\text{term}} = \text{MLP}^{\text{term},2}(g)$$

$$\text{APD} = \text{SOFTMAX}[f_{\text{add}}, f_{\text{conn}}, f_{\text{term}}].$$

This MLP block was best as it performed on par with the previous tiered global readout block but was faster to train.

7 | STRATEGIES FOR IMPROVING MEMORY REQUIREMENT

Below we detail strategies that were used to improve the memory requirement of jobs during the development of GraphINVENT. Improving the memory requirement was especially important for any jobs which ran on a GPU, as the

cluster we used had (at the time) a limited number of high-memory GPU nodes. While also important to reduce the memory requirement for CPU jobs, this was in our case less crucial as CPU nodes were easier to come by.

Nonetheless, reducing the memory requirement of jobs is important in general, as it means that a program will be able to run using larger mini-batch sizes (and thus train faster). It also means a program can scale to larger datasets, either in terms of overall number of structures in the dataset or molecule size (eg, number of nodes).

We do not discuss here strategies discussed in other works for improving memory requirement as we found this information hard to come by in the literature. Nonetheless, reading other people's source code is a good way to get an idea of programming strategies that others have employed to improve the efficiency of their code.

7.1 | Writing preprocessed data

In order to deal with the large memory requirement of molecular graphs, GraphINVENT was designed such that *preprocessing* and *training* jobs can be run separately, working with the data in chunks and thus maintaining a relatively low RAM requirement throughout a given job. Processed data is written to disk using the HDF file format, available in Python via the *h5py*[58] package. Although initially memory chunking was used as implemented in *h5py*, this scheme was eventually discarded, leading to the development of a custom PyTorch Dataloader which could read contiguous data blocks (see section 8.2 below).

This strategy works well for large datasets, where the entire processed training data does not fit in GPU memory at once. Nonetheless, in GraphINVENT the same preprocessing scheme is used for all datasets, including small datasets with low GPU memory requirements, so as to keep the same workflow. This has the additional benefit of maximizing GPU utilization during training, as preprocessing is done separately on the CPU (see section 8).

7.2 | Sparse data structures

NumPy arrays are used to handle all matrix representations during preprocessing. Both sparse SciPy arrays and sparse PyTorch tensor representations were experimented with, but no significant decrease in disk space requirement was observed. Furthermore, using sparse data structures led to noticeably longer processing times due to overhead.

7.3 | Using smaller data types

Setting the data type of the graph representation arrays (X and E) to *int8* was found to be more useful than using sparse data structures when it comes to reducing the memory requirement during both preprocessing and training. However, *float32* tensors were used for APDs during training, as *int8* tensors were not fully supported in PyTorch 1.3 for GPU operations.

7.4 | Collecting identical graphs

During preprocessing jobs, the fact that many graphs share common subgraphs can be used to save a significant amount of disk space when writing data. This is done by *collecting* identical graph representations.

Graphs which have the exact same matrix representation (ie, $X_i = X_j$ and $E_i = E_j$) in a group[1] are collected such that only one copy of X_n and E_n is kept in the group while the APDs are summed and normalized. In other words, the duplicate X_n and E_n are dropped but the corresponding APDs are aggregated as they could encode different actions (depending on the decoding routes). The memory requirement is thus reduced because a single APD can encode for multiple viable actions.

This naturally increases the preprocessing time, because graphs must be compared and these comparisons are expensive; as such, graph comparisons are limited to groups of fixed size (eg, 1000) instead of comparisons between all graphs in the dataset (which can be millions). Furthermore, only graphs which are *exact* matches in X and E are collected (ie, the comparison is node order dependent). However, as this leads to fewer graphs in the training data, training is faster.

For example, preprocessing the MOSES training set,[37] which consists of 1.5M molecules, using GraphINVENT without utilizing the above graph collection procedure leads to roughly 72M subgraphs. Another way of putting this is that

if we were to sum up all the subgraphs in the training set molecules' decoding routes, there would be 72M subgraphs in this set. Now, if we collect the subgraphs using the above procedure so as to drop duplicate subgraphs while combining their APDs, the number of subgraphs is significantly reduced from 72M. The specific number depends on which specific graph traversal algorithm and which features are used. For example, using the parameters from the three best MOSES models in Reference 25, 72M subgraphs can be reduced to 33M subgraphs, on average (cGGNN: 26M, rGGNN: 33M, and aGGNN: 40M). This is clearly a significant reduction.

8 | STRATEGIES FOR IMPROVING GPU UTILIZATION

Anyone who has ever tried to run a graph-based deep generative model has probably noticed that they unfortunately suffer from slow training and generation speeds, especially when compared to string-based methods. One of the best ways to improve on slow training and generation speeds, as we observed during the development of GraphINVENT, was to identify bottlenecks in the GPU utilization and take better advantage of things which could be parallelized in the code using tensor operations. While this concept is obvious to a programmer, the field of deep molecular generation lies at the cross-section of AI and chemistry, and thus attracts researchers from a variety of fields who may not be aware of all standard practices in software development. With the goal of encouraging transparency between fields, below we provide some tips that greatly improved GPU utilization in GraphINVENT.

8.1 | Running separate jobs

Unlike the DeepMind approach,[10] on-the-fly training data generation was found to be unsuitable for moving on to larger training sets (ie, millions of structures) in GraphINVENT, as it significantly slowed down training and decreased GPU utilization. As such, the workflow was split in a way that allowed training data preprocessing and model training to be done in separate jobs.

There are two advantages to separating training data preprocessing from training. The first is that, because data preprocessing is all done on CPUs in GraphINVENT, doing it as a separate job means that by extension GPU utilization is automatically higher during training jobs, which use GPUs. Then, because processes such as hyperparameter optimization (HO) use the same training data for multiple jobs with different parameters, the training data really only needs to be preprocessed once and saved.

8.2 | Reading preprocessed data

As detailed above, all processed training data are written to disk as HDF files. This training data must then be read from the HDF files during training jobs; to do this efficiently, custom wrappers were created for the standard PyTorch DataLoader and Dataset classes.

By default, the standard PyTorch DataLoader accesses one "item" at a time from different locations on disk, which in the case of HDF files means one data point (ie, one graph and APD). This is extremely inefficient for HDF files and leads to low GPU utilization during training because the DataLoader must read from disk as many times as there are data-points. The aforementioned custom data structures were thus created so as to minimize the number of disk reads while still allowing for shuffling of training data.

The custom DataLoaders read contiguous *blocks* of data at once. During training jobs, the block size is fixed to be much larger than the mini-batch size so as to (a) minimize the number of disk reads and (b) better shuffle data between mini-batches. The default block size in the code is 100 000, whereas the default mini-batch size is 1000.

8.3 | Generating structures on the GPU

Models do not only train during training jobs, but are also evaluated at periodic intervals so as to understand how training is proceeding. During model evaluation, a small set of graphs is generated using the trained model and analyzed. As such, optimizing GPU utilization during graph generation was also important to keep GPU utilization high.

Furthermore, being able to speed up the generation process by taking advantage of GPU operations meant more structures could also be generated overall. The generation process was parallelized in the code so that it could be carried out for a batch of subgraphs simultaneously on the GPU, thus maximizing GPU utilization during both training and generation jobs.

During GPU-optimized graph generation, a batch of empty graphs (tuples of zero matrices) is input to a trained model, which outputs a batch of APDs. The batch of APDs is then sampled simultaneously, and the sampled actions are applied to all graphs using matrix operations. To achieve this, a dummy action was created that incorporates all the indices corresponding to both *add* and *connect* actions simultaneously, but in practice only applies the sampled action. If either the *terminate* action or an invalid action is sampled for a given graph, then the graph is saved (before the dummy action is applied) and replaced in the mini-batch with an empty graph (after the dummy action). The process is repeated until the desired number of structures has been generated.

9 | FURTHER DETAILS ON THE DEVELOPMENT OF GraphINVENT

This section contains technical details on each of the specific workflows in GraphINVENT, including details on ineffective methods that were not discussed in the original work.[25] However, we believe other researchers might still benefit from knowing what we tried and did not work. Throughout this section, the following notation is used: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a molecular graph, where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges, and $\mathcal{G}_n \subseteq \mathcal{G}$ is a subgraph of $\mathcal{G}$.

9.1 | Workflow

9.1.1 | Preprocessing details

Here, details on the complex data preprocessing scheme in GraphINVENT are provided.

Graph fragmentation

In order for the model to learn how to build molecular graphs, molecules in the training set must be fragmented in a way that they can be reconstructed by the model. A key part of data preprocessing thus involves calculating the graph decoding route, *r*, which is determined by iteratively removing nodes and edges from the graph.

The order of the node/edge removal is determined by reversing a (modified) breadth-first search (BFS). The BFS algorithm was modified so as to never create any disconnected fragments in the graph after removing any node/edge. Disconnected fragments are avoided since disconnected fragments cannot pass messages to each other unless connected by an "artificial" edge, which was not used here (to minimize the size of graph representations).

Graph traversal

The modified BFS graph traversal proceeds as follows. First, all nodes $v_i \in \mathcal{V}$ are *randomly* assigned an index from $i = \{1, 2, ..., |\mathcal{V}|\}$. The graph is then traversed starting at v_1, followed by all the nearest neighbors of v_1, denoted as $\mathcal{N}(v_1)$, in order from lowest index to highest index. Note that the specific choice of which nearest neighbors to traverse first (lower or highest index) is arbitrary so long as it is consistent throughout the preprocessing scheme. The process is repeated for the nearest neighbors of the nearest neighbors, $\mathcal{N}(\mathcal{N}(v_1))$, and so on, until all nodes in a graph have been traversed.

Graph deconstruction

To get the decoding route, *r*, the *reverse* order of the node traversal is followed during graph deconstruction; in other words, the last node to be visited during the graph traversal is the first node to be removed. If the node has more than one edge linking it to the graph, then first each additional edge is removed one by one (meaning each edge removal is a single action), until there is a single edge left, and then the node and final edge are removed in a single action.

Calculating the APD

For each intermediate subgraph along *r*, an APD is created which describes how to get back to the graph from which the edge or node+edge removal was performed. The APD is a tensor and is discussed in detail in Reference 25.

Each graph $\mathcal{G}$ in the training set will contribute $|\mathscr{E}| + 2$ subgraphs and $|\mathscr{E}| + 2$ APDs to the training data. The number $|\mathscr{E}| + 2$ corresponds to one action for each edge addition, plus one for adding the first node to an empty graph and one for terminating the graph. Each individual data-point in the processed dataset thus corresponds to a tuple (X_n, E_n, APD_n), where X_n is the node features matrix and E_n is the adjacency tensor that describe $\mathcal{G}_n$.

Empty graphs

The APD of an empty graph (no nodes or edges) is a special case, as a separate action was not created for adding a node without an edge to an empty graph. Instead, the APD for empty subgraphs is nonzero in f_{add} at indices indicating the node to be added; indices indicating which node to connect to and with what bond type are ignored when applying an action to an empty graph.

9.1.2 | Training details

Here are detailed notes regarding model training strategies.

Reading preprocessed data

The size of the blocks which are read from disk at a single time should not be set to an integer that is close to the total number of subgraphs in the training set. This runs the risk of leaving too few mini-batches in the final block that will not be properly shuffled, thus leading to spikes in the loss during training. Alternatively, the final block can be dropped.

Model stability

Initially, a common issue faced when training these models was a lack of robustness. However, it was observed that if certain hyperparameters are in the wrong range, then the models will be unstable and converge to different solutions every time. Nonetheless, once an adequate set of hyperparameters was identified, the models were very robust.

Optimizer

All models were trained using the Adam optimizer with the default PyTorch parameters (except for weight decay in certain specified cases). SGD was experimented with to see if it would lead to more stable training, but it converged too slowly to be practical. No other optimizers were experimented with.

Avoiding early stopping

Early stopping was originally used as a criterion for ending training and avoiding overfitting. However, early stopping frequently led to inconsistent results, and was overall unsatisfactory. Instead, increasing the size of the networks, finding an appropriate learning rate decay scheme, and training to convergence led to more robust models.

Sampling the best epoch of a model

Instead of early stopping criteria, terminating training when the loss converged within three significant figures worked well for GraphINVENT models.

Nonetheless, if the goal is to generate *novel* molecules, then there is a benefit to training models for fewer epochs. However, besides the observed model instability described above, models not trained to convergence will generate a higher fraction of invalid structures, so early stopping must be used carefully. If the goal is to generate a high percentage of valid structures highly resembling the training set, then longer training is desirable. Slight overtraining is not problematic for GraphINVENT models.

Tracking training status

Four methods were used to track training, each discussed below.

Loss. As long as the loss continues monotonically decreasing during training, then a model is still learning. However, there are clear signs of inadequate hyperparameters reflected in the loss: too fast initial learning rates will lead to sharp peaks in the loss, and will also cause the loss to plateau at large values (large being >2.0 in GraphINVENT models). These should be avoided.

Negative log-likelihood. Another method used when evaluating models was the negative log-likelihood (NLL) of generated molecules. When each molecular graph is constructed, the associated probabilities for the sampled actions are saved. Summing the NLLs for the sampled actions gives the total NLL of generating any given molecule. NLL distributions can be calculated for graphs in the training, validation, and generated sets. To calculate the NLL of training or validation set structures, training/validation graphs are fed into the network and the NLL is sampled for the *correct* action, which is not necessarily the most likely action if the model is not well trained. Having a larger probability for the correct actions in the validation set and thus—a smaller (non-negative) NLL for those actions—means that the model is learning to build molecules correctly. However, if the NLL corresponding to the validation set is much larger than that of the training and generated sets, then the model is clearly overfitting.

Uniformity-completeness Jensen-Shannon divergence. For a more quantitative comparison of the NLL distributions between training, validation, and generated set structures, the UC-JSD (Uniformity-Completeness Jensen-Shannon Divergence) introduced in Arús-Pous et al[59] is computed during evaluation epochs. This is a measure of the Jensen-Shannon divergence between distributions of the NLL *per action* for the three datasets. However, the UC-JSD was often too noisy in GraphINVENT models to be informative, as it was only viable to sample a couple thousand graphs per evaluation epoch during training and the UC-JSD converges as the number of samples increases (ideally, n_samples $\gg 10^4$). The other metrics listed here worked better with fewer samples.

Prior distribution. The simplest and most intuitive method of analyzing how the model is training is to compare the property distributions of the generated molecules with those of the training set; if a model is good then the properties of the training set will be reflected in the generated set and gradually converge to the correct prior. These properties are, for example, the distributions in atom types, formal charges, and bond types. With bad hyperparameters, the distributions do not converge but rather fluctuate around incorrect values (sometimes close to the correct values). This metric was found to be very informative.

A combination of the methods discussed above were used to determine how models were training, as certain metrics were more useful for small datasets (*loss*, *NLL*) and others for large datasets (*loss*, *prior*).

Observations

Below are a few additional observations made during model training.

Batch size. The size of the mini-batch is not too important for training so long as it is large enough and the learning rate is adequately adjusted. Striking a good balance between computation time and GPU memory is more important, where a larger mini-batch size means faster training but also an increased memory requirement for jobs. A batch size of 1000 was typically used.

Model robustness. Models without suitable hyperparameters (eg, too small MLP width and depth, or small feature vector size) will converge to different solutions every time. However, with adequate hyperparameters, GraphINVENT models are robust and stable.

Proper termination of structures. When using inadequate hyperparameters, one of the biggest problems observed in the models is learning to "properly" terminate graphs. That is, all too often an invalid action will be sampled before the *terminate* action is sampled during graph generation.

Dropout. The effect of adding dropout to every MLP in each model was investigated by varying the probability of dropout. Specifically, torch.AlphaDropout() was used to maintain self-normalization in the MLPs, which all use the SELU activation function. The values of dropout investigated ranged between {0.05-0.5}. In all models, adding the lowest probability dropout (0.05) significantly lowered the probability of generating valid structures as well as the percentage of properly terminated structures. Adding the largest probability dropout had disastrous effects on the percent validity and uniqueness of all model.

Weight decay. The effect of using weight decay in the optimizer (Adam) was investigated by varying the weight decay value. The values of weight decay investigated were 0.001 and 0.005. In all models, setting the weight decay to 0.001 noticeably increased the percentage of unique structures generated while only slightly decreasing the percent valid.

However, a weight decay of 0.005 seems to be too large (for all models) as it has a strong negative effect on the percentage of valid structures generated, although a positive effect on the uniqueness.

9.1.3 | Generation details

Here are detailed notes and observations regarding molecular graph generation in GraphINVENT.

Storing graphs as SMILES
Generated molecular graphs are converted to SMILES for saving, as SMILES require less disk space than the matrix representations. To do this, a molecular graph is first converted into an RDKit Mol object, then to SMILES using the RDKit MolToSmiles() function. For graphs that cannot be converted to SMILES, a placeholder string, "Xe," is used in the output SMILES file, so as to keep track of how often this happens while also being able to easily filter these out later if needed.

Percent validity (PV)
When well trained, the best models sample ≥95% valid actions, but may not reach 100 PV depending on the size of the training set. This is due to the probabilistic nature of sampling the APD and the large size of the action space. In other words, if there are many extremely low probability actions which are invalid then they (potentially) have a non-negligible probability of being sampled.

9.2 | Hyperparameter optimization details

Here are some notes regarding HO in GraphINVENT.

9.2.1 | Initial optimization on GDB-13 subset

Initial HO was carried out on the GDB-13 1K training set using a grid search. Model depth, learning rate, and hidden node features size were found to be some of the most important parameters. Below is a summary of observations.

Vector widths
A key observation that applies to all models, regardless of GNN block, is that the message size, hidden node features size, graph gather width, and edge embedding width have to be sufficiently large for the models to learn. The default value for all these qualities in GraphINVENT is 100 for all models. Similarly, the hidden dimensions of all MLPs in the models have to be sufficiently large; 500 is a good value. A strong correlation was not observed between the message size and the model performance (this was fixed to 100). Once a suitable value was found for the aforementioned vector widths, these were fixed for all models and not further optimized so as to narrow down the hyperparameter search space.

Initial learning rate
If the initial learning rate is too high, the models will not learn. This will be evident in the loss flattening within a few training epochs. If it is too small, models will train unbearably slowly. An initial learning rate of 0.0001 was found to be suitable for all models when using a mini-batch size of 1000.

MLP depth
A larger MLP depth (ie, a greater number of layers) and slower rate of learning rate decay lead to better learning. However, introducing too many layers leads to less unique molecules, as models are more prone to overfit. An MLP depth of 4 was found to work best coupled with a hidden layer dimension of 500.

Rate of learning rate decay
A slower rate of learning rate decay (ie, a larger learning rate decay factor, *lrdf*) leads to better learning, up until ca. Epoch 80-100 for the GDB-13 1K subset, when the second loss drop has occurred. After the second loss drop has occurred, the learning rate should be rapidly decreased to avoid the uniqueness of the structures generated from significantly decreasing.

To tackle this, an exponential learning rate decay scheme was implemented (see *Experimenting with learning rate decay* below), where an $lrdf = 0.9999$ and $lrdi = 100$ were found to work best with this scheme. Note that *lrdi* and *lrdf* depend on the training set size; for larger training sets, a larger *lrdi* in the range {100-10 000} is recommended.

Learning rate decay schemes

The learning rate decay scheme was found to be one of the most sensitive parameters during model training.

The learning rate decay scheme found to work best multiplies the learning rate by $lrdf^{epochs}$ every *lrdi* epochs, where *epochs* is the number of epochs elapsed. When visualized against the number of elapsed epochs, the learning rate resembles a smoothed step function, meaning the learning rate is slowly decreased in the beginning and rapidly in the middle/end of training, eventually flattening out when the minimum learning rate is reached.

The minimum learning rate is defined by setting the minimum *relative* learning rate; 0.05 was found to work well, meaning that if the initial learning rate was 0.0001, the minimum learning rate allowed in a calculation would be 0.000005.

Loss

A lower loss generally corresponds to a greater overall validity of molecules that will be generated by a model at that epoch.

Properly terminated structures

More layers and a slower learning rate decay lead to a greater percentage of properly terminated structures. In order to have close to 100% proper termination of generated structures, a large hidden node features size is necessary; 100 works well.

Attention

All else being the same, adding attention to the models does not lead to improved performance. However, the possibility that this is due to suboptimal hyperparameters in these models cannot be ruled out, as given how much slower the Attention models train, less hyperparameter combinations could be tried for these in a fixed amount of time during HO.

Loss function

The KL divergence was found to work best for training GraphINVENT models. The MSE and SmoothL1 loss also work adequately, although a lower percentage of valid and properly terminated molecules are observed with these. The L1 and BCE loss do not work so well in GraphINVENT.

Weight initialization scheme

Both normal and uniform weight initialization lead to indistinguishable model performance. However, using no weight initialization scheme led to noticeably slower training (more than twice as long all else being the same).

Learning rate warm up

Ramping up the learning rate (eg, from 0.000001 to 0.0001) during the initial {10-100} mini-batches has no effect other than to delay training.

Bias

All MLP biases are set to True by default. This is necessary for the models to learn to grow on empty graphs, which are all zeros.

9.2.2 | Experimenting with learning rate decay

As previously mentioned, the learning rate decay scheme was found to be one of the most sensitive parameters when it comes to training; so, various schemes were experimented with to find what works best. Here is a summary of observations.

The terms *lrdf* and *lrdi* are frequently used throughout this section. These are the learning rate decay parameters. $lrdf \in \{0.0 - 1.0\}$ is the learning rate decay *factor*, and $lrdi \in \mathbb{Z}^+$ is the learning rate decay *interval*. Both are user defined hyperparameters.

Multiplying learning rate by lrdf *every* lrdi *epochs*
This works fine, but leads to too rapid learning rate decay in early epochs and too slow learning rate decay in later training epochs.

Subtracting a constant amount from the learning rate every lrdi *epochs*
This also leads to too rapid learning rate decay in early epochs and too slow learning rate decay in later epochs.

Multiplying learning rate by an exponentially smaller lrdf *every* lrdi *epochs*
Multiplying the learning rate by $lrdf^{epochs}$ ever *lrdi* epochs works the best of any of the learning rate decay schemes tried. The learning rate is slowly decreased in the beginning and rapidly in the middle/end of training.

Multiplying learning rate by lrdf *whenever a convergence criteria is met*
This generally lead to too rapid learning rate decay and was sensitive to the convergence criteria.

10 | OTHER USEFUL DEVELOPMENT TOOLS

In this section, standard Python packages and development tools which researchers may find useful in the development of their own models are described, as well as tips for debugging graph generative models.

10.1 | Why Python?

All of the graph-based generative models introduced in section 3 have used Python for the development of their models. This is likely due to the wide availability of frameworks, packages, and libraries for deep learning in Python, as well as wide availability of helpful resources and low barrier to entry. Furthermore, Python has exploded in popularity amongst the scientific community in recent years such that is has become the de facto scientific programming language for AI applications.

We should mention that although Python 3 is generally considered the standard for software development, some recently published graph generative models have used Python 2 (not recommended).

10.2 | Python frameworks

The most common framework used in the development of graph-based deep generative models are by far PyTorch[60] and TensorFlow,[61] although Chainer[62] has also been used. Recently, libraries for working with graph-structured data in DNNs have also been published which may be of interest to researchers working in the field; these are PyTorch Geometric (PyG)[63] and Deep Graph Library (DGL).[64]

10.3 | Memory profiling

For profiling memory usage in code, PyPi memory-profiler[65] can be used.

10.4 | GPU profiling

The *torch.utils.bottleneck* tool in PyTorch[60] is effective for finding GPU bottlenecks in the code and improving GPU utilization. We found this a very valuable tool during the development of GraphINVENT.

10.5 | Unit testing

Unit tests were heavily used for testing the GNN implementations. The *unittest* framework in Python was used exclusively for this, although other powerful frameworks also exist for unit testing in Python.

10.6 | Tips for debugging

10.6.1 | Test cases

Besides regular unit testing, it was useful throughout GraphINVENT development to test the code with the following test cases:

1. 1 benzene;
2. 3 small molecules;
3. 3 large molecules; and
4. 3 aromatic rings.

Details on the molecules in each test set can be found in the Appendix. To summarize, the molecules in each test case were quasi-randomly selected to have a variety of different atoms types. The test sets were kept extremely small at three molecules so that tests could be run quickly.

The point of the *1 benzene* set is that it is very quick to preprocess and train on, and building a benzene molecule uses all the actions for building graphs (add, connect, and terminate). It is a good way to test that the different aspects of the code are working. Furthermore, if training the models on 1 benzene does not eventually lead to (overtrained) models which exclusively generate benzene, then that is a red flag. This could point to either (a) bugs in the code or (b) inadequate parameters/hyperparameters in the models.

The point of the *3 small molecules* set is that it is also very quick to preprocess and train on, but a step up from having a single molecule in the training set. Testing with multiple molecules is a way to test that mini-batches are loading correctly.

The reason for the *3 large molecules* set is that some errors (eg, in graph traversal/fragmentation/construction) will not pop up for small molecules simply because they have fewer atoms and bonds and there are fewer places to make mistakes. As such, low-probability actions might not be sampled for a dataset of small molecules, but they could be for a set of large molecules. Large molecules with complicated bonding patterns are also useful for checking that the deconstruction algorithm does not create disconnected fragments. Using a large molecule dataset can also give an idea of how much slower the code will run as one increases the size of the molecules in the dataset.

Finally, the *3 aromatic rings* set is recommended as there are some bugs that are easier to spot during ring formation in batches. This was the most useful test case for finding bugs in GraphINVENT.

While the specific molecules in the test cases are not important, it is very useful to have a such sets of test cases during development. It is extremely difficult to debug code with a real-life dataset, as it is much easier to spot bugs when the molecules in the training set are few.

Warning signs

An important metric for identifying bugs in GraphINVENT was the PVPT metric (*percent valid of properly terminated*). If there are bugs at any point in the preprocessing/training/generation schemes, this will manifest itself as a low PVPT in the generated structures. This is because a low PVPT means that structures are not being properly terminated, and that could be for a variety of reasons (eg, bugs in deconstruction path, bugs in action sampling, etc).

Another good overall check for any generative model is testing that the model can indeed overfit to the training set (ie, reconstruction accuracy). Naturally, it is not the goal of a molecular generative model to regenerate the training set, but if a model cannot learn to memorize the training set, that is a huge red flag. This could point to an inadequate model architecture, inadequate hyperparameters, or bugs in the code.

11 | CONCLUSION

Development of molecular generative models is still a relatively young field. We have written this technical note with the hope that it helps researchers understand previous work in the field and develop their own graph-based generative models, and we provide strategies for efficiently reading and writing data, improving model training time, and efficiently debugging models. Graph-based generative models are promising methods for molecular discovery, and we hope that this work lowers the barrier of entry for other researchers looking to move into the exciting field of graph-based molecular design.

ACKNOWLEDGMENTS

Rocío Mercado thanks the Molecular AI group at AstraZeneca for helpful discussions, as well as the AstraZeneca Postdoc Program. The authors would also like to thank the reviewers for their thoughtful review of this work.

CONFLICT OF INTEREST

The authors declared no potential conflicts of interest.

AUTHOR CONTRIBUTIONS

Rocío Mercado: Wrote the manuscript and all authors reviewed it and gave excellent feedback. All authors provided valuable feedback on methods used, experiments, and results throughout the entire project.

CODE DETAILS

This technical note is based on code available at https://github.com/MolecularAI/GraphINVENT.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ORCID

Rocío Mercado https://orcid.org/0000-0002-6170-6088
Esben Jannik Bjerrum https://orcid.org/0000-0003-1614-7376

ENDNOTE

[1] A *group* here is a mini-batch of graphs that are processed together. We wanted to avoid using the word mini-batch outside of the training context, as a group and a mini-batch can be different sizes in GraphINVENT.

REFERENCES

1. A. Makhzani, J. Shlens, N. Jaitly, I. Goodfellow, and B. Frey. Adversarial autoencoders. *arXiv preprint arXiv:1511.05644*; 2015.
2. Kadurin A, Aliper A, Kazennov A, et al. The cornucopia of meaningful leads: applying deep adversarial autoencoders for new molecule development in oncology. *Oncotarget*. 2017;8(7):10883.
3. B. Sanchez-Lengeling, C. Outeiral, G. L. Guimaraes, and A. Aspuru-Guzik. Optimizing distributions over molecular space. An objective-reinforced generative adversarial network for inverse-design chemistry (organic). *ChemRxiv*. 2017; doi:https://doi.org/10.26434/chemrxiv.5309668.v3.
4. Olivecrona M, Blaschke T, Engkvist O, Chen H. Molecular de-novo design through deep reinforcement learning. *J Chem*. 2017;9(1):48.
5. E. J. Bjerrum and R. Threlfall. Molecular generation with recurrent neural networks (RNNS). *arXiv preprint arXiv:1705.04612*; 2017.
6. Gómez-Bombarelli R, Wei JN, Duvenaud D, et al. Automatic chemical design using a data-driven continuous representation of molecules. *ACS Cent Sci*. 2018;4(2):268-276.
7. Prykhodko O, Johansson SV, Kotsias P-C, et al. A de novo molecular generation method using latent vector based generative adversarial network. *J Chem*. 2019;11(1):74.
8. K. Madhawa, K. Ishiguro, K. Nakago, and M. Abe GraphNVP: an invertible flow model for generating molecular graphs. *arXiv preprint arXiv:1905.11600*; 2019.
9. T. Blaschke, J. Arús-Pous, H. Chen, C. Margreitter, C. Tyrchan, O. Engkvist, K. Papadopoulos, and A. Patronov. REINVENT 2.0—an AI tool for de novo drug design. *ChemRxiv*. 2020; doi:https://doi.org/10.26434/chemrxiv.12058026.v2.
10. Y. Li, O. Vinyals, C. Dyer, R. Pascanu, and P. Battaglia, Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324*; 2018.
11. Li Y, Zhang L, Liu Z. Multi-objective de novo drug design with conditional graph generative model. *J Cheminformatics*. 2018;10(1):33.
12. W. Jin, R. Barzilay, and T. Jaakkola, Junction tree variational autoencoder for molecular graph generation. *arXiv preprint arXiv:1802.04364*; 2018.
13. Liu Q, Allamanis M, Brockschmidt M, Gaunt A. Constrained graph variational autoencoders for molecule design. *Advances in Neural Information Processing Systems*; 2018:7795-7804.
14. You J, Liu B, Ying Z, Pande V, Leskovec J. Graph convolutional policy network for goal-directed molecular graph generation. *Advances in Neural Information Processing Systems*, Red Hook, NY, US: Curran Associates, Inc.; 2018:6410-6421. https://proceedings.neurips.cc/paper/2018/hash/d60678e8f2ba9c540798ebbde31177e8-Abstract.html. Accessed December 22, 2020.
15. J. You, R. Ying, X. Ren, W. L. Hamilton, and J. Leskovec, GraphRNN: generating realistic graphs with deep auto-regressive models. *arXiv preprint arXiv:1802.08773*; 2018.
16. W. Jin, R. Barzilay, and T. Jaakkola, Hierarchical generation of molecular graphs using structural motifs. *arXiv preprint arXiv:2002.03230*; 2020.
17. B. Samanta, A. De, G. Jana, P. K. Chattaraj, N. Ganguly, and M. Gomez-Rodriguez, NeVAE: a deep generative model for molecular graphs. *arXiv preprint arXiv:1802.05283*; 2018.
18. N. De Cao and T. Kipf, MolGAN: An implicit generative model for small molecular graphs. *arXiv preprint arXiv:1805.11973*; 2018.

19. J. Lim, S.-Y. Hwang, S. Kim, S. Moon, and W. Y. Kim, Scaffold-based molecular design using graph generative model. *arXiv preprint arXiv:1905.13639*; 2019.
20. R. Assouel, M. Ahmed, M. H. Segler, A. Saffari, and Y. Bengio, DEFactor: differentiable edge factorization-based probabilistic graph generation. *arXiv preprint arXiv:1811.09766*; 2018.
21. Simonovsky M, Komodakis N. GraphVAE: towards generation of small graphs using variational autoencoders. *arXiv preprint arXiv: 1802.03480*. Cham, CH: Springer International Publishing; 2018. https://link.springer.com/chapter/10.1007/978-3-030-01418-6_41. Accessed 22 December 2020.
22. Mansimov E, Mahmood O, Kang S, Cho K. Molecular geometry prediction using a deep generative graph neural network. *Sci Rep*. 2019; 9(1):1-13.
23. C. Zang and F. Wang. MoFlow: an invertible flow model for generating molecular graphs. Paper presented at: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining; August 2020.
24. C. Shi, M. Xu, Z. Zhu, W. Zhang, M. Zhang, and J. Tang, GraphAF: a flow-based autoregressive model for molecular graph generation. *arXiv preprint arXiv:2001.09382*; 2020.
25. Mercado R, Rastemo T, Lindelöf E, et al. Graph networks for molecular design. *Mach Learn: Sci Technol*. Bristol, UK: IOP Publishing Ltd.; 2020. https://doi.org/10.1088/2632-2153/abcf91. Accessed 22 December 2020.
26. G. L. Guimaraes, B. Sanchez-Lengeling, C. Outeiral, P. L. C. Farias, and A. Aspuru-Guzik, Objective-reinforced generative adversarial networks (ORGAN) for sequence generation models. *arXiv preprint arXiv:1705.10843*; 2017.
27. G. N. Simm and J. M. Hernández-Lobato, A generative model for molecular distance geometry. *arXiv preprint arXiv:1909.11459*; 2019.
28. N. W. Gebauer, M. Gastegger, and K. T. Schütt, Generating equilibrium molecules with deep neural networks. *arXiv preprint arXiv: 1810.11347*; 2018.
29. Gebauer N, Gastegger M, Schütt K. Symmetry-adapted generation of 3d point sets for the targeted discovery of molecules. *Advances in Neural Information Processing Systems*. Red Hook, NY, US: Curran Associates, Inc.; 2019:7566-7578. https://papers.nips.cc/paper/2019/hash/a4d8e2a7e0d0c102339f97716d2fdfb6-Abstract.html. Accessed December 22, 2020.
30. M. Hoffmann and F. Noé, Generating valid Euclidean distance matrices. *arXiv preprint arXiv:1910.03131*; 2019.
31. Anonymous authors (paper under double-blind review). Learning neural generative dynamics for molecular conformation generation. ICLR; 2021.
32. Sterling T, Irwin JJ. Zinc 15–ligand discovery for everyone. *J Chem Inf Model*. 2015;55(11):2324-2337.
33. Gaulton A, Hersey A, Nowotka M, et al. The ChEMBL database in 2017. *Nucleic Acids Res*. 2017;45(D1):D945-D954.
34. Mendez D, Gaulton A, Bento AP, et al. ChEMBL: towards direct deposition of bioassay data. *Nucleic Acids Res*. 2019;47(D1):D930-D940.
35. Blum LC, Reymond J-L. 970 million druglike small molecules for virtual screening in the chemical universe database gdb-13. *J Am Chem Soc*. 2009;131(25):8732-8733.
36. Ruddigkeit L, Van Deursen R, Blum LC, Reymond J-L. Enumeration of 166 billion organic small molecules in the chemical universe database GDB-17. *J Chem Inf Model*. 2012;52(11):2864-2875.
37. D. Polykovskiy, A. Zhebrak, B. Sanchez-Lengeling, S. Golovanov, O. Tatanov, S. Belyaev, R. Kurbanov, A. Artamonov, V. Aladinskiy, M. Veselov, A. Kadurin, S. Nikolenko, A. Aspuru-Guzik, and A. Zhavoronkov, Molecular sets (MOSES): a benchmarking platform for molecular generation models. *arXiv preprint arXiv:1811.12823*; 2018.
38. Ramakrishnan R, Dral PO, Rupp M, Von Lilienfeld OA. Quantum chemistry structures and properties of 134 kilo molecules. *Sci Data*. 2014;1(1):1-7.
39. Ma T, Chen J, Xiao C. Constrained generation of semantically valid graphs via regularizing variational autoencoders. *Advances in Neural Information Processing Systems*. Red Hook, NY, US: Curran Associates, Inc.; 2018:7113-7124. https://proceedings.neurips.cc/paper/2018/hash/1458e7509aa5f47ecfb92536e7dd1dc7-Abstract.html. Accessed December 22, 2020.
40. S. Honda, H. Akita, K. Ishiguro, T. Nakanishi, and K. Oono, Graph residual flow for molecular graph generation. *arXiv preprint arXiv: 1909.13521*; 2019.
41. Kwon Y, Yoo J, Choi Y-S, Son W-J, Lee D, Kang S. Efficient learning of non-autoregressive graph variational autoencoders for molecular graph generation. *J Chem*. 2019;11(1):70.
42. X. Bresson and T. Laurent, A two-step graph convolutional decoder for molecule generation. *arXiv preprint arXiv:1906.03412*; 2019.
43. S. Pölsterl and C. Wachinger, Likelihood-free inference and generation of molecular graphs. *arXiv preprint arXiv:1905.10310*; 2019.
44. Goodfellow I, Pouget-Abadie J, Mirza M, et al. Generative adversarial nets. *Advances in Neural Information Processing Systems*. Red Hook, NY, US: Curran Associates, Inc.; 2014:2672-2680. https://papers.nips.cc/paper/2014/hash/5ca3e9b122f61f8f06494c97b1afccf3-Abstract.html. Accessed December 22, 2020.
45. Ertl P, Schuffenhauer A. Estimation of synthetic accessibility score of drug-like molecules based on molecular complexity and fragment contributions. *J Chem*. 2009;1(1):8.
46. Bickerton GR, Paolini GV, Besnard J, Muresan S, Hopkins AL. Quantifying the chemical beauty of drugs. *Nat Chem*. 2012;4(2):90-98.
47. Kingma DP, Dhariwal P. Glow: generative flow with invertible 1x1 convolutions. *Advances in Neural Information Processing Systems*. Red Hook, NY, US: Curran Associates, Inc.; 2018:10215-10224. https://papers.nips.cc/paper/2018/hash/d139db6a236200b21cc7f752979132d0-Abstract.html. Accessed December 22, 2020.
48. M. Popova, M. Shvets, J. Oliva, and O. Isayev, MolecularRNN: generating realistic molecular graphs with optimized properties. *arXiv preprint arXiv:1905.13372*; 2019.
49. S. Kearnes, L. Li, and P. Riley, Decoding molecular graph embeddings with reinforcement learning. *arXiv preprint arXiv:1904.08915*; 2019.
50. Liao R, Li Y, Song Y, et al. Efficient graph generation with Graph Recurrent Attention Networks. *Advances in Neural Information Processing Systems*. Red Hook, NY, US: Curran Associates, Inc.; 2019:4257-4267. https://papers.nips.cc/paper/2019/hash/d0921d442ee91b896ad95059d13df618-Abstract.html. Accessed December 22, 2020.

51. Wu Z, Ramsundar B, Feinberg EN, et al. MoleculeNet: a benchmark for molecular machine learning. *Chem Sci*. 2018;9(2):513-530.
52. John PCS, Phillips C, Kemper TW, et al. Message-passing neural networks for high-throughput polymer screening. *J Chem Phys*. 2019; 150(23):234111.
53. Yang K, Swanson K, Jin W, et al. Analyzing learned molecular representations for property prediction. *J Chem Inf Model*. 2019;59(8): 3370-3388.
54. Duvenaud DK, Maclaurin D, Iparraguirre J, et al. Convolutional networks on graphs for learning molecular fingerprints. *Advances in Neural Information Processing Systems*. Red Hook, NY, US: Curran Associates, Inc.; 2015:2224-2232. https://papers.nips.cc/paper/2015/hash/f9be311e65d81a9ad8150a60844bb94c-Abstract.html. Accessed December 22, 2020.
55. Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*; 2015.
56. Gilmer J, Schoenholz SS, Riley PF, Vinyals O, Dahl GE. Neural message passing for quantum chemistry. Paper presented at: Proceedings of the 34th International Conference on Machine Learning-Volume 70; 2017:1263–1272. JMLR.org.
57. Feinberg EN, Sur D, Wu Z, et al. PotentialNet for molecular property prediction. *ACS Cent Sci*. 2018;4(11):1520-1530.
58. "HDF5 for Python." https://www.h5py.org/. Accessed June 18, 2020.
59. Arús-Pous J, Johansson SV, Prykhodko O, et al. Randomized smiles strings improve the quality of molecular generative models. *J Chem*. 2019;11(1):1-13.
60. Paszke A, Gross S, Massa F, et al. PyTorch: an imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*. Red Hook, NY, US: Curran Associates, Inc.;. Vol 32. 2019:8024-8035. https://papers.nips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html. Accessed December 22, 2020.
61. "TensorFlow." https://www.tensorflow.org/. Accessed October 31, 2020.
62. "Chainer." https://chainer.org/. Accessed October 31, 2020.
63. "PyG: PyTorch geometric." https://pytorch-geometric.readthedocs.io/. Accessed October 31, 2020.
64. "DGL: Deep graph library." https://www.dgl.ai/. Accessed October 31, 2020.
65. F. Pedregosa and P. Gervais, "PyPi memory-profiler." https://pypi.org/project/memory-profiler/. Accessed February 5, 2020.

How to cite this article: Mercado R, Rastemo T, Lindelöf E, et al. Practical notes on building molecular graph generative models. *Applied AI Letters*. 2020;1:e18. https://doi.org/10.1002/ail2.18

APPENDIX: TEST CASES

A few test cases were created for use not only in debugging the code, but also in understanding how the models learn different features. Examples are provided below using their canonical SMILES strings.

1 benzene

c1ccccc1

3 small molecules

CCC(C═CC)═CC(C)C
OCC1(CC1)C(═O)OC═C
OCCNCCN(O)C═N

3 large molecules

CC1C2CCC(C2)C1CN(CCO)C(═O)c1ccc(Cl)cc1
COc1ccc(—c2cc(═O)c3c(O)c(OC)c(OC)cc3o2)cc1O
CCOC(═O)c1ncn2c1CN(C)C(═O)c1cc(F)ccc1-2

3 aromatic rings

Cc1ccccc1
Clc1ccccc1
Oc1ccccc1