PRACTICAL NOTES ON BUILDING MOLECULAR GRAPH GENERATIVE MODELS

Rocío Mercado,[†1] Tobias Rastemo,[†,‡] Edvard Lindelöf,[†,‡] Günter Klambauer,[‖] Ola Engkvist,[†]
Hongming Chen,[§] Esben Jannik Bjerrum[†]

[†] *Molecular AI, Discovery Sciences, BioPharmaceuticals R&D, AstraZeneca, Gothenburg, SE*
[‡] *Chalmers University of Technology, Gothenburg, SE*
[‖] *Institute of Bioinformatics, Johannes Kepler University, Linz, AU*
[§] *Centre of Chemistry and Chemical Biology, Guangzhou Regenerative Medicine and Health, Guangdong Laboratory, Guangzhou, CN*

ABSTRACT

Here are presented technical notes and tips on developing graph generative models for molecular design. This work stems from the development of GraphINVENT, a Python platform for graph-based molecular generation using graph neural networks. In this work, technical details that could be of interest to researchers developing their own molecular generative models are discussed, including strategies for designing new models. Advice on development and debugging tools which were helpful during code development is also provided. Finally, methods that were tested but which ultimately didn't 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.

***K*eywords** deep generative models · graph neural networks · code development · molecular design

1 Introduction

Molecular generative models have emerged as promising methods for exploring the chemical space through *de novo* molecular design [1–15]. Although molecular generative models have largely focused on string-based approaches, graph-based approaches have also emerged in the last 2-3 years [9–21], including a recent approach, GraphINVENT [22], from our group. In GraphINVENT, we introduce graph-based molecular generative models inspired by the works of Li et al. [9] and Li et al. [10]; the models use a tiered deep neural network architecture, where graph neural networks (GNNs) play a key role in learning graph representations.

GNNs were used as they are powerful architectures for modeling patterns in graph-structured data. They come in a variety of flavors, such as message passing neural networks (MPNNs) and graph convolutional networks (GCNs). [23–26] GNN-based models have also shown promise in molecular design applications. [9,10,13,15,22] 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.

Sections 2 and 3 begin by discussing strategies for selecting a generation scheme and a model architecture. This is followed by a discussion on strategies for solving the two biggest challenges faced during the development of GraphINVENT: improving the memory requirements of jobs (Section 4) and improving GPU utilization (Section 5). Details on methods used are discussed, noting specific functions and schemes that were tried but failed, in Section 6. Lastly, tips on development tools are provided in Section 7.

2 Designing a generation scheme

In building a generative model, the first step is designing an action space for how graphs will be constructed. The action space will determine how the molecule is build, 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 doesn't

[1]Corresponding author: rocio.mercado@astrazeneca.com

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 [11, 15].

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, one can imagine splitting up the action space such that nodes and edges are added in separate actions, or such that a single action can 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, it is not necessary to split the action space as in GraphINVENT. For example, if the goal is to generate large molecules, it might be desirable to add many nodes at once (as in Jin et al. [15]) 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; one example of how to do this is given by Li et al. [10]. Recurrence could also be applied during each action e.g. 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 for molecular generation.

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 GraphINVENT, this is done via a separate preprocessing phase (see Sections 4 & 5), but this could also be done on-the-fly.

3 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 factors e.g. if chemical rules will need to be hard-coded into the model, how many hyperparameters will need to be tuned, etc.

A GNN architecture was eventually selected for GraphINVENT due in part to the success of GNNs in recently published graph generative models, and in part due to the good performance of GNNs in molecular property prediction tasks [23, 27–29]. Molecular graph generation can also be framed as a complex property prediction problem, where target "property" for a given input graph becomes the correct action for building a graph. Based on previous work [9, 10], 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 final block is what takes the latent node and graph embeddings and returns the target actions. This is described in detail below.

3.1 Model architecture

The models in GraphINVENT consist of two blocks:

1. a *GNN* block
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 sub-optimal 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.

3.1.1 Node-only global readout block

The node-only multi-layer 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.

$$\begin{aligned} f_{add} &= \texttt{MLP}^{add}(H^L) \\ f_{conn} &= \texttt{MLP}^{conn}(H^L) \\ f_{term} &= \texttt{MLP}^{term}(H^L) \\ APD &= \texttt{SOFTMAX}\left[f_{add}, f_{conn}, f_{term}\right]. \end{aligned}$$

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.

3.1.2 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) Preliminary APD components obtained:

$$\begin{aligned} f'_{add} &= \texttt{MLP}^{add,1}(H^L) \\ f'_{conn} &= \texttt{MLP}^{conn,1}(H^L) \\ f'_{term} &= \texttt{MLP}^{term,1}(H^L). \end{aligned}$$

2) Above output and g used to obtain the final APD:

$$\begin{aligned} f_{add} &= \texttt{MLP}^{add,2}\left(\left[f'^{,add}, g\right]\right) \\ f_{conn} &= \texttt{MLP}^{conn,2}\left([f'^{,conn}, g]\right) \\ f_{term} &= \texttt{MLP}^{term,2}\left(\left[f'^{,term}, g\right]\right) \\ APD &= \texttt{SOFTMAX}\left[f_{add}, f_{conn}, f_{term}\right]. \end{aligned}$$

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

3.1.3 Tiered global readout block B

The best global readout block – and the one reported in [22] – 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) Preliminary APD components obtained:

$$\begin{aligned} f'_{add} &= \texttt{MLP}^{add,1}\left(H^L\right) \\ f'_{conn} &= \texttt{MLP}^{conn,1}\left(H^L\right). \end{aligned}$$

2) Above output and g used to obtain the final APD:

$$\begin{aligned} f_{add} &= \texttt{MLP}^{add,2}\left([f'_{add}, g]\right) \\ f_{conn} &= \texttt{MLP}^{conn,2}\left([f'_{conn}, g]\right) \\ f_{term} &= \texttt{MLP}^{term,2}\left(g\right) \\ APD &= \texttt{SOFTMAX}\left[f_{add}, f_{conn}, f_{term}\right]. \end{aligned}$$

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

4 Strategies for improving memory requirement

4.1 Writing preprocessed data

In order to deal with the large memory requirement of molecular graphs, the code is sectioned 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. Prrocessed data is written to disk using the HDF file format, available in Python via the *h5py* [30] 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 subsection below on *Reading preprocessed data*).

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 5).

4.2 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.

4.3 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.

4.4 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 (i.e. $X_i = X_j$ and $E_i = E_j$) in a group[2] 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. 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, graphs are only compared within groups of fixed size (e.g. 1000) instead of comparison between all graphs in the dataset (which can be millions). Furthermore, only graphs which are *exact* matches in X and E are collected i.e. the comparison is node order dependent. However, as this leads to fewer graphs in the training data, training is faster. Savings in training time are more meaningful as training is done on a GPU.

[2]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.

5 Strategies for improving GPU utilization

5.1 Running separate jobs

Unlike Li et al [9], on-the-fly training data generation was found to be unsuitable for moving on to larger training sets (i.e. millions of structures), 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 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 CPU, doing it as a separate job means that by extension GPU utilization is automatically higher during training jobs, which use GPUs. The second is that, for processes such as hyperparameter optimization (HO), the same training data is used for multiple jobs with different parameters; as such, the training data needs only to be preprocessed once and then saved.

5.2 Reading preprocessed data

As detailed above, all processed training data is 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 (i.e. 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 1) minimize the number of disk reads and 2) better shuffle data between mini-batches. The default block size in the code is 100,000, whereas the default mini-batch size is 1000.

5.3 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. Furthermore, being able to speed up the generation process by taking advantage of GPU operations meant more structures could also be generated during generation and benchmarking runs.

The generation process was non-trivially 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, and the sampled actions are applied to all graphs simultaneously 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 have been generated.

6 Method details

This section contains technical details on processes in GraphINVENT, as well as details on ineffective methods that were not discussed in [22]. 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; $\mathcal{G}_n \subseteq \mathcal{G}$ is a subgraph of $\mathcal{G}$.

6.1 Workflow

6.1.1 Preprocessing

Here, details on the complex data preprocessing scheme 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 $i = \{1, 2, \ldots, |\mathcal{V}|\}$. 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's 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 [22].

Each graph $\mathcal{G}$ in the training set will contribute $|\mathcal{E}| + 2$ subgraphs and $|\mathcal{E}| + 2$ APDs to the training data. The number $|\mathcal{E}| + 2$ 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.

6.1.2 Training

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 won't be properly shuffled and lead 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.

NLL. 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 generation 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 generation sets, then the model is clearly overfitting.

UC-JSD. For a more quantitative comparison of the NLL distributions between training, validation, and generation set structures, the UC-JSD (Uniformity-Completeness Jensen-Shannon Divergence) introduced in Arús-Pous et al [31] 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 is only viable to sample a couple thousand graphs per evaluation epoch during training. 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 generation 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 (e.g. 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-normalizationin 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.

6.1.3 Generation

Here are detailed notes and observations regarding molecular graph generation.

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 *Chem.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 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.

6.2 Hyperparameter optimization

Here are some notes regarding hyperparameter optimization (HO) in GraphINVENT.

6.2.1 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 all have to be sufficiently large in order for the models to learn. The default values 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; for these, 500 is a good value. A strong correlation was not observed between the message size and the model performance (nonetheless, 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 1e-4 was found to be suitable for all models when using a mini-batch size of 1000.

MLP depth. A larger MLP depth (i.e. 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 (i.e. a larger learning rate decay factor, $lrdf$) leads to better learning, up until c.a. 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; 5e-2 was found to work well, meaning that if the initial learning rate was 1e-4, the minimum learning rate allowed in a calculation would be 5e-6.

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 doesn't lead to improved performance. However, the possibility that this is due to sub-optimal 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 (e.g. from 1e-6 to 1e-4) during the initial {10 – 100} minibatches 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.

6.2.2 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.

7 Useful development tools

In this section, packages and development tools which were found to be extremely useful during GraphINVENT development are described, as well as tips for debugging graph generative models.

7.1 Memory profiling

For profiling memory usage in code, PyPi memory-profiler [32] was used.

7.2 GPU profiling

The *torch.utils.bottleneck* tool in PyTorch [33] was very effective for finding GPU bottlenecks in the code and improving GPU utilization.

7.3 Unit testing

Unit tests were heavily used for testing the GNN implementations. The *unittest* framework in Python was used exclusively for this.

7.4 Tips for debugging

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

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

Details on the molecules in each test set can be found in Appendix Section B. To summarize, the molecules in each test case were quasi-randomly selected to have a variety of different atoms. 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 doesn't 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 working correctly.

The reason for the *3 large molecules* set is that some errors (e.g. 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. 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 the code this is 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 (e.g. 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. 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.

8 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 in the development of their own graph-based generative models, providing strategies on 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.

9 Supplementary Information

9.1 Appendices

Abbreviations. Test Cases.

9.2 Acknowledgments

R.M. thanks the Molecular AI group at AstraZeneca for helpful discussions, as well as the AstraZeneca Postdoc Program.

9.3 Author's contributions

All authors provided valuable feedback on methods used, experiments, and results throughout the entire project. R.M. wrote the manuscript and all authors reviewed it and gave excellent feedback.

9.4 Competing interests

The authors declare no competing financial interests.

9.5 Code details

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

References

[1] A. Makhzani, J. Shlens, N. Jaitly, I. Goodfellow, and B. Frey, "Adversarial autoencoders," *arXiv preprint arXiv:1511.05644*, 2015.

[2] A. Kadurin, A. Aliper, A. Kazennov, P. Mamoshina, Q. Vanhaelen, K. Khrabrov, and A. Zhavoronkov, "The cornucopia of meaningful leads: Applying deep adversarial autoencoders for new molecule development in oncology," *Oncotarget*, vol. 8, no. 7, p. 10883, 2017.

[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 preprint doi:10.26434/chemrxiv.5309668.v3*, 2017.

[4] M. Olivecrona, T. Blaschke, O. Engkvist, and H. Chen, "Molecular de-novo design through deep reinforcement learning," *Journal of cheminformatics*, vol. 9, no. 1, p. 48, 2017.

[5] E. J. Bjerrum and R. Threlfall, "Molecular generation with recurrent neural networks (rnns)," *arXiv preprint arXiv:1705.04612*, 2017.

[6] R. Gómez-Bombarelli, J. N. Wei, D. Duvenaud, J. M. Hernández-Lobato, B. Sánchez-Lengeling, D. Sheberla, J. Aguilera-Iparraguirre, T. D. Hirzel, R. P. Adams, and A. Aspuru-Guzik, "Automatic chemical design using a data-driven continuous representation of molecules," *ACS central science*, vol. 4, no. 2, pp. 268–276, 2018.

[7] O. Prykhodko, S. V. Johansson, P.-C. Kotsias, J. Arús-Pous, E. J. Bjerrum, O. Engkvist, and H. Chen, "A de novo molecular generation method using latent vector based generative adversarial network," *Journal of Cheminformatics*, vol. 11, no. 1, p. 74, 2019.

[8] 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 preprint doi:10.26434/chemrxiv.12058026.v2*, 2020.

[9] Y. Li, O. Vinyals, C. Dyer, R. Pascanu, and P. Battaglia, "Learning deep generative models of graphs," *arXiv preprint arXiv:1803.03324*, 2018.

[10] Y. Li, L. Zhang, and Z. Liu, "Multi-objective de novo drug design with conditional graph generative model," *Journal of cheminformatics*, vol. 10, no. 1, p. 33, 2018.

[11] W. Jin, R. Barzilay, and T. Jaakkola, "Junction tree variational autoencoder for molecular graph generation," *arXiv preprint arXiv:1802.04364*, 2018.

[12] Q. Liu, M. Allamanis, M. Brockschmidt, and A. Gaunt, "Constrained graph variational autoencoders for molecule design," in *Advances in neural information processing systems*, pp. 7795–7804, 2018.

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

[14] 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.

[15] W. Jin, R. Barzilay, and T. Jaakkola, "Hierarchical generation of molecular graphs using structural motifs," *arXiv preprint arXiv:2002.03230*, 2020.

[16] 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.

[17] N. De Cao and T. Kipf, "Molgan: An implicit generative model for small molecular graphs," *arXiv preprint arXiv:1805.11973*, 2018.

[18] 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.

[19] M. Simonovsky and N. Komodakis, "Graphvae: Towards generation of small graphs using variational autoencoders," in *International Conference on Artificial Neural Networks*, pp. 412–422, Springer, 2018.

[20] C. Zang and F. Wang, "Moflow: An invertible flow model for generating molecular graphs," *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, Aug 2020.

[21] 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.

[22] R. Mercado, T. Rastemo, E. Lindelöf, G. Klambauer, O. Engkvist, H. Chen, and E. J. Bjerrum, "Graph Networks for Molecular Design," *ChemRxiv preprint doi:10.26434/chemrxiv.12843137.v1*, 2020.

[23] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1263–1272, JMLR. org, 2017.

[24] M. Withnall, E. Lindelöf, O. Engkvist, and H. Chen, "Building attention and edge message passing neural networks for bioactivity and physical–chemical property prediction," *Journal of Cheminformatics*, vol. 12, no. 1, p. 1, 2020.

[25] K. Yang, K. Swanson, W. Jin, C. Coley, P. Eiden, H. Gao, A. Guzman-Perez, T. Hopper, B. Kelley, M. Mathea, *et al.*, "Analyzing learned molecular representations for property prediction," *Journal of chemical information and modeling*, vol. 59, no. 8, pp. 3370–3388, 2019.

[26] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, *et al.*, "Relational inductive biases, deep learning, and graph networks," *arXiv preprint arXiv:1806.01261*, 2018.

[27] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarelli, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," in *Advances in neural information processing systems*, pp. 2224–2232, 2015.

[28] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[29] E. N. Feinberg, D. Sur, Z. Wu, B. E. Husic, H. Mai, Y. Li, S. Sun, J. Yang, B. Ramsundar, and V. S. Pande, "Potentialnet for molecular property prediction," *ACS central science*, vol. 4, no. 11, pp. 1520–1530, 2018.

[30] "Hdf5 for python," 2020.

[31] J. Arús-Pous, S. V. Johansson, O. Prykhodko, E. J. Bjerrum, C. Tyrchan, J.-L. Reymond, H. Chen, and O. Engkvist, "Randomized smiles strings improve the quality of molecular generative models," *Journal of Cheminformatics*, vol. 11, no. 1, pp. 1–13, 2019.

[32] F. Pedregosa and P. Gervais, "PyPi memory-profiler," 2020.

[33] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

A Abbreviations

APD	action probability distribution
GCN	graph convolutional networks
GNN	graph neural network
HO	hyperparameter optimization
MLP	multi-layer perceptron
MPNN	message-passing neural network
PPT	percent properly terminated
PU	percent unique
PV	percent valid
PVPT	percent valid of properly terminated

B 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. As an example, the canonical SMILES strings for the molecules in each example test case are provided below.

B.1 1 benzene

```
c1ccccc1
```

B.2 3 small molecules

```
CCC(C=CC)=CC(C)C
OCC1(CC1)C(=O)OC=C
OCCNCCN(O)C=N
```

B.3 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
```

B.4 3 aromatic rings

```
Cc1ccccc1
Clc1ccccc1
Oc1ccccc1
```