Study of Deep Generative Adversarial Network Generation of Molecular Graphs

Anthony J Vasquez
Johns Hopkins University
Whiting School of Engineering
Baltimore, MD
avasque1@jh.edu

***Abstract*—**

Deep Generative Adversarial Networks (DGANs) have demonstrated remarkable effectiveness across various fields, producing high-quality samples that can often deceive both humans and neural networks. For instance, DGANs have been employed in anomaly detection within banking, creating synthetic driving environments for autonomous vehicles, generating synthetic audio and facilitating audio style-transfer, restoring images, and many other applications. One topic of interest, is in the field of chemistry where the challenging problem of generating molecular graphs exists. Many solutions have been tried with varying degrees of success, such as using a variational auto-encoder VAE (Bidisha, 2019). The work presented here compares synthetic GAN generations using recurrent and multi-head attention, named Synthetic Molecule GAN (SMGAN), to data synthesis of a MolGAN (Nicola De Cao, 2018). by training the networks to generate realistic molecular graphs similar to Simplified Molecular Input Line Entry System (SMILES), a database that generates string representations of molecules and reactions (National Library of Medicine, 2024).

Keywords - DGANS, VAE, Generative, SMILES, Medicine, Chemistry, Molecule

Introduction

The generation of new molecules is of interest in medicine, where designing new drugs is long, costly, laborious, and yields few new molecular discoveries (Bidisha, 2019). Some estimates suggest that it takes an average of 10-15 years to develop a single new medicine and comes with a $2.6 dollar price tag (Phrma, 2024). There is much interest in deriving or automating a way to generate physically viable molecules which could potentially speed up the discovery-to-shelf pipeline of pharmaceuticals at a fraction of the price.

Among the many difficulties with working with molecular data, is that it is not readily consumable by machine learning algorithms. This is the case with most data, for example Natural Language Processing (NLP) and the use of term frequency-inverse document frequency (TF-IDF), which is a well-known encoding algorithm (Das, 2018). However, there are few vectorizations transforms for molecular data. One molecular vectorizer is SMILES (Weininger, 1987), a flexible fragment based molecular representation framework. SMILES uses various characters in a sequence which conveys relationships among the different atoms in molecules. It captures the molecular graph which is used to create a vector representation as seen in NLP. Another much newer approach designed to extend semantic complexity of the traditional SMILES strings is that of t-SMILES. This method contains additional information by increasing the unique number of characters used to capture more relational information needed for high-fidelity string representations of molecular graphs (Jaun-Ni Wu, 2024).

For the sake of simplicity, and to make this work more relevant to previous work on generative molecule synthesis, the SMILES representation was used. Many tools have been created and extensively tested on SMILES strings for years, making this approach stable and trusted. In this work, a GAN architecture that uses recurrence and multi-head attention is used to generate molecules that are in

some documented cases, valid, novel, unique, soluble, druglike, and synthesizable.

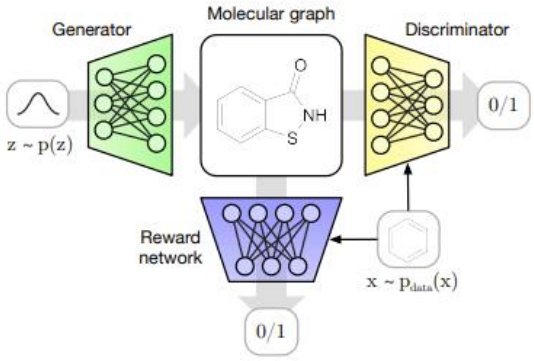
Related work

Two generative methods that stand out, explores similar generative strategies. One method is the VAE, from which many models have been implemented, like NeVAE (Bidisha, 2019), GraphVAE (Simonosvsky, 2018), GrammarVAE (Kusner, 2017), CVAE (Pagnomi, 2018), SDVAE (Hankun Dai, 2018), JTVAE (Jin, 2019), and CGVAE (Rigoni, 2020), to name a few. This is by no means an exhaustive list, and it can be safely assumed that newer models have been created. For the sake of brevity, and among the VAE architectures, only NeVAE will be discussed, as it is well documented.

The claims of NeVAE are that, "[their] model guarantee[s] a set of valid generated molecules", and their model can "discover plausible, diverse and novel molecules more effectively than several state-of-the-art methods" (Bidisha, 2019). An innovation used in their experimentations was to use a VAE combined with a data representation that captures structural properties of a molecule. Their assumption is that the vector representation embeds N molecular graphs, composed of various sets of atoms (nodes) and bonds (edges), where each may contain various lengths; $\{G_i = (V_i \epsilon_i)\}_{i\epsilon[N]}$, where V_i and ε_i are nodes and edges respectively, there exists a set of features $F = \{f_u\}_{u\epsilon V}$ and edge weighs, $Y = \{y_{uv}\}_{(u,v)\epsilon\varepsilon}$, that can be learned (Bidisha, 2019). Their technique creates a continuous latent space, enabling users to extract useful and novel chemical compounds (Bidisha, 2019). Their model is evaluated by the quality of decoder generations, namely, Novelty, Uniqueness, and Validity. These metrics will be discussed in more depth in Evaluation Metrics.

Another method used with molecule synthesis include the deep generative paradigm of GANS. Like the VAE, the GAN approach is many and varied. For example, MolGAN that combines deep reinforcement learning (RL) by rewarding an agent with high objective scores based on several metrics (Nicola De Cao, 2018). Among the objective scores of this method are adversarial loss, to measure the quality of the generated images from the real ones, solubility loss, and drug-likeness loss (Nicola De Cao, 2018). An objective loss is also calculated for the RL agent during training. The approach used here follows the GAN paradigm to novel molecule synthesis, though there are some key differences that are expanded on in the Network Architecture section.

Figure 1. The MolGAN Network has three main components, a generator, discriminator, and an RL component.



Experiment

The experiment section contains all of the components needed to produce results. This section contains a section on the data set used, preprocessing, the SMGAN architecture, hardware, software, initial training, hyperparameter tuning, inference tuning, final training, and evaluation metrics.

Data set:

SMILES as used in this work, a molecular graph string generation methodology and a database. SMILES strings combine molecular graph theory with natural language processing to produce unique character strings, which explain the underlying structure of molecules, such as, atoms, bonds, and connectivity (Weininger, 1987). In this work, a small sample of the SMILES dataset was used. Of the 249,456 molecular strings, only 10 percent of the dataset was used to train. It is unclear of whether adding more data would significantly improve model convergence, but this training set percentage is mostly arbitrary and is influenced by time-to-train constraints. These SMILES strings were randomly sampled from the entire dataset

using scikit-learn' s train-test-split method (Pedregosa, 2011).

Preprocessing

The SMILES strings can be converted to bit-map representations for numeric modeling and molecule-to-molecule nearest neighbor modeling. However, because bit-map conversion is not lossless, it is difficult to cross reference between other SMILES molecules. For example, a bitmap cannot be directly converted back to a SMILES string, which means that indirect methods need to be used for molecule comparisons, such as similarity and/or distance metrics. One could create a lookup table to generate bit map representations beforehand, but attempts to do so, were abandoned early on, due to timeliness and large file generation. Instead, a SMILES string tokenization approach was used.

Each sample string from the trainset was tokenized by using regular expressions to split the string into segments by using special characters. For example, "-" is a single bond, "=" is a double bond, and "#" is a triple bond. More symbols can be found in Table 1.

Table 1. This table shows common SMILES symbols (Kim, 2017).

Single Bond	-
Double Bond	=
Triple Bond	#
Arom. Bond	:
Pos. Charge	[C+]
Neg. Charge	[C-]
Arom. Carb.	c (lowercase)

The original strings are characterized by each sample with no spaces, letters representing atoms, and special characters that relate the relationship of an atom to other atoms in their respective molecules. Examples of this notation are found in the below Table 2, which shows four examples of SMILES strings.

Table 2. Example of SMILES Strings

```
0. COc1ccc(C(=O)Nc2ccc(Cl)cc2C(F)(F)F)cc1S(=O)(=O)N1CCCCC1COc1ccc(C(=O)Nc2ccc(Cl)cc2C(F)(F)F)cc1S(=O)(=O)N1CCCCC1COc1ccc(C(=O)Nc2ccc(Cl)cc2C(F)(F)
F)cc1S(=O)(=O)N1CCCCC1COc1ccc(C(=O)Nc2ccc(Cl)cc2C(F)(F)F)cc1S(=O)(=O)N1CCCCC1COc1ccc(C(=O)Nc2ccc(Cl)cc2C(F)(F)F)cc1S(=O)(=O)N1CCCCC1COc1ccc(C(=O)Nc
2ccc(Cl)cc2C(F)(F)F)cc1S(=O)(=O)N1CCCCC1COc1ccc(C(=O)Nc2ccc(Cl)cc2C(F)(F)F)cc1S(=O)(=O)N1CCCCC1COc1ccc(C(=O)Nc2ccc(Cl)cc2C(F)(F)F)cc1S(=O)(=O)N1CCC
CC1COc1ccc(C(=O)Nc2ccc(Cl)cc2C(F)(F)F)cc1S(=O)(=O)N1CCCCC1COc1ccc(C(=O)Nc2ccc(Cl)cc2C(F)(F)F)cc1S(=O)(=O)N1CCCCC1

1. Cn1cnnc1Sc1ccccc1Cn1cnnc1Sc1ccccc1Cn1cnnc1Sc1ccccc1Cn1cnnc1Sc1ccccc1Cn1cnnc1Sc1ccccc1Cn1cnnc1Sc1ccccc1Cn1cnnc1Sc1ccccc1Cn1cnnc1
Sc1ccccc1Cn1cnnc1Sc1ccccc1

2. CCOC(=O)C(C)(C)N(C)Cc1c(N)ncnc1OCCCOC(=O)C(C)(C)N(C)Cc1c(N)ncnc1OCCCOC(=O)C(C)(C)N(C)Cc1c(N)ncnc1OCCCOC(=O)C(C)(C)N(C)Cc1c(N)ncnc1OCCCOC(=O)C(C)
(C)N(C)Cc1c(N)ncnc1OCCCOC(=O)C(C)(C)N(C)Cc1c(N)ncnc1OCCCOC(=O)C(C)(C)N(C)Cc1c(N)ncnc1OCCCOC(=O)C(C)(C)N(C)Cc1c(N)ncnc1OCCCOC(=O)C(C)(C)N(C)Cc1c(N)n
cnc1OCCCOC(=O)C(C)(C)N(C)Cc1c(N)ncnc1OC

3. COc1ncccc1C(=O)NCCCOc1ncccc1C(=O)NCCCOc1ncccc1C(=O)NCCCOc1ncccc1C(=O)NCCCOc1ncccc1C(=O)NCCCOc1ncccc1C(=O)NCCCOc1ncccc1C(=O)NCCCOc1ncccc1C(=O)NCC
COc1ncccc1C(=O)NCCCOc1ncccc1C(=O)NCC
```

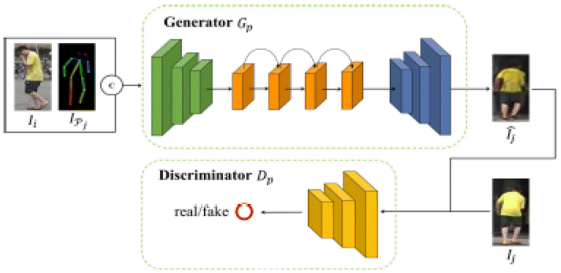
The SMILES strings were tokenized by using regular expressions to split the strings by symbol, for example, *pattern* = *r*"(\[[^\[\]] *\])" for each SMILES string. This encoding allows an embedding layer to be used when modeling.

Network Architecture

The SMGAN architecture consists of a generative model with adversarial discriminator and generator networks. More specifically, SMGAN uses embedding, recurrence, multi-head-attention, and fully connected layers. It is hypothesized that each of these layers play a role in the ability of the network to generate high quality molecules. However, unique layer relies on various unique hyperparameters to be tuned. This subject is mentioned in more detail in Hyperparameter Tuning. For now, a brief introduction is given to each the architecture layers.

GAN: The GAN architecture was discovered by Ian Goodfellow in 2016, when he described a model consisting of a sample generator G_θ and discriminator D_φ networks that trained according to an adversarial policy (Goodfellow, 2014). G_θ attempts to generate realistic looking samples and D_θ attempts to classify if the image is real of fake. At each iteration, the network that loses updates the weights based on the real data distribution. The network optimizes according to its respective loss function, and the model is said to converge when G_θ can no longer fool D_φ.

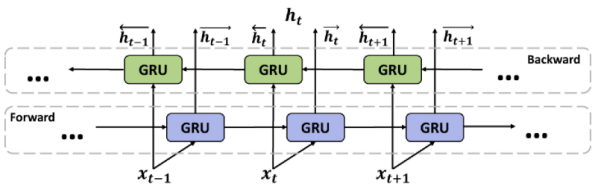
Figure 2. Basic GAN Architecture (Alqahtani, 2019)



Though GANs were innovative and fast adopted, there was one big problem – mode collapse (Mangalam, 2021). Mode collapse is when the generator fails to produce diverse samples. To combat this, a major improvement to the GAN was proposed with the contributions of WGAN (Arjovsky, 2017). WGAN improved the original GAN by updating the loss function to the Wasserstein loss, weight clipping, and gradient regularization (Arjovsky, 2017). For this reason, SMGAN uses the WGAN paradigm to address potential stability issues during training.

BiGRU: The Gated Recurrent Unit (GRU) is a simplified version of a Long-Short-Term-Memory network (LSTM) with less gating and better time complexity. For example, the GRU only has two gates, a reset gate that allows the model to determine how much of a previous hidden state to use, and an update gate that determines how much of a previous hidden state to pass to the next timestep. GRUs have a bidirectional mechanism, that allows it to learn sequences from start to end, and end to start in parallel. In this work, the bidirectional GRU is referred to as BiGRU.

Figure 3. The BiGRU is able to learn sequential data by maintaining memory. Unlike the traditional RNN, the BiGRU attempts to use data from both the past and future directions. Put another way, it attempts to learn the sequence from beginning to end and end to beginning simultaneously (Liu, 2021).



Multi-head-attention (MHA): is an extension of self-attention that allows the network to focus on segments input sequences. For example, a network that uses MHA with four-heads means that the data is attended by four attention mechanisms. This attention mechanism allows the network to take advantage of both syntactic and semantic features learned from the linear projections of each segment of a sequence (Cordeonnier, 2021). Given the marked success of MHS, it is assumed that it has a positive impact on both training and generation modes of the SMGAN.

Hardware/Software

The operating system was Linux, with CUDA 12.2 and Python 3.8 installed (Nickolls, 2008). The initial hardware was an NVIDIA RTX 3080 Ti with 16 GB of memory and 14 logical cores, but it was discovered to be inadequate for this task because training and metrics testing were time intensive. Instead, two NVIDIA RTX A4000s with 16GB of memory, with 24 logical processors were used for all stages of experimentation. The increased logical processor count proved to helpful for parallelization metric calculations, which are discussed in more detail in Evaluation Metrics. Two GPUs were helpful with training larger batch-sizes, but they did not contribute much in speeding up training, since using a GRU is a sequential task so GPU clock speeds are more important.

Initial Training

The purpose of the initial training was to create a baseline model, so most, if not all picked hyperparameters were arbitrarily picked but constrained by hardware. The initial training used binary cross entropy loss (BCEL) loss, the Adam algorithm (Pytorch, 2024), and a learning rate of 0.00025 for both generator and discriminator optimizers. A basic learning rate scheduler was used to update the learning rate with $\gamma = 0.99$ (update coefficient) incremented at every five epochs. The momentum moving average was set to 0.5 to 0.99 with L2 regularization of 1e-5.

All layers contained dropout with a probability of 0.25 except for the fully connected layers. The

embedding dimension was set to the length of the vocabulary x the initial embedding dimension of 32. The network hidden dimension was set to 64, which was used in the BiGRU, attention, and fully connected layers. However, the BiGRU required a doubling of hidden dimension size, and therefore an increase in computational resources. As mentioned previously, WGAN methodology was used during training and the weights were clipped to ± 0.01 at each batch iteration. Furthermore, the discriminator was allowed to optimize three times for every one generator update.

The network was trained for 50 epochs and both generator and discriminator losses were monitored every two epochs for signs of mode collapse. For example, if either the generator or discriminator displayed signs of exploding or diminishing gradients, the experiment was stopped. However, training never stopped while using the WGAN paradigm.

Hyperparameter Tuning

RayTune is a scalable and flexible hyperparameter tuning API (Ray, 2024). It was used for hyperparameter tuning because of its robust logging functionality and ease of use. Using Ray Tune, the Asynchronous Successive Halving Algorithm (ASHA) (Ray, 2024) was used to search a hyperparameter grid with predetermined boundaries. The ASHA tuning schedule was shown to be an effective tuning scheduler, as it effectively implements successive halving and asynchronous execution using a Rung method (Li, 2018). An added benefit of using Ray Tune is that it schedules the experiments, automatically and handles all hardware resource. For example, the resources per experiment parameter were set to 2 CPUs and 1 GPU per run, with a maximum concurrent trial of four experiments at any one time.

The hyperparameter search space initially started with the Adam, NAdam, and RMSprop optimizers, but was reduced to just RMSprop because it tended to converge to a lower training loss faster. Other searched hyperparameters include, generator learning rate, discriminator learning rate, batch size, hidden layer dimension, embedding dimension, number of GRU layers, dropout probability, number of attention heads, bidirectional, number of extra discriminator updates, and weight clipping value. The search was conducted with a max number of epochs of 25 with only ten percent of the data.

Table 3. Hyperparameter Search Space

Hyperparameter	Search Space	Selection
Optimizer	Adam, Nadam, RMSProp	Choice
Learning Rate	0.00002 - 0.002	Loguniform
Batch Size	32, 64, 128, 256, 384	Choice
Hidden Layer Dim	16, 32, 64, 128	Choice
Embedding Dim	16, 32, 64	Choice
Num GRU Layers	2, 3	Choice
Dropout Prob	0.25, 0.5	Choice
Num Attn. Heads	2, 4, 8	Choice
Bidirectional	True, False	Choice
Num Discr. Updates	1, 2, 3, 4, 5	Choice
Weight Clip Value	0.01 - 0.001	Loguniform

As previously mentioned, an ASHA scheduler was used for efficient hyperparameter tuning. The ASHA parameters used include loss metric: generator loss, mode: minimum, maximum time to train of 2500 seconds, an early stopping grace period of 5, and a reduction factor of 2. The reduction factor determines which hyperparameters make it to the next round using a rung policy (Ray, 2024). A total of 300 experiments were run over the span of two tuning sessions, and optimal network hyperparameters are reported in Table 4.

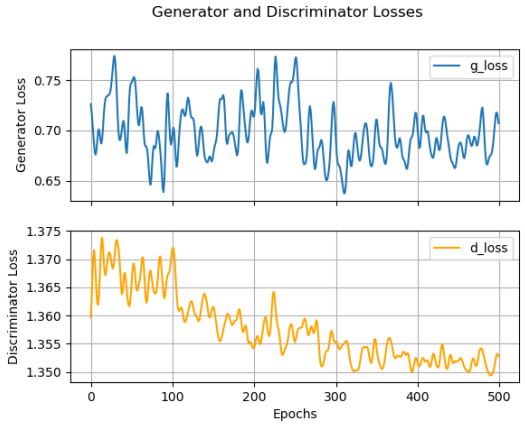
Table 4. Tuned Hyperparameters

Hyperparameter	Value
Optimizer	RMSProp
Gen. LR	0.00408680002539221
Disc. LR	0.0134480709026601
Batch Size	128
Hidden Layer Dim	128
Embedding Dim	32
Num GRU Layers	2
Dropout Prob	0.5
Num Attn. Heads	4
Bidirectional	TRUE
Num Discr. Updates	3
Weight Clip Value	0.00810198750825

Final Training

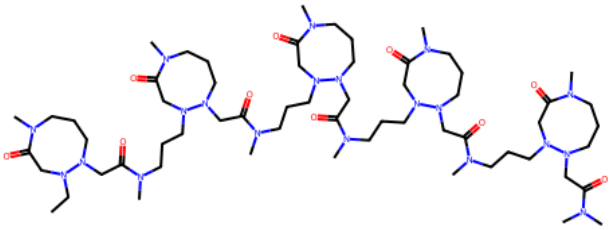
Configured with the optimal hyperparameters, the final model was trained on 10 percent of the full dataset (25K samples). Both generator and discriminator losses were monitored for signs of instability or mode collapse. The discriminator loss appeared to be more stable than the generator loss, but both stayed within a small range, implying network performance matching, and therefore exhibited no evidence of mode collapse. See the Matplotlib (Hunter, 2007) generated training losses of Figure 4.

Figure 4. Shown here are generator and discriminator training losses.



Using the trained network and a maximum generator output length of 10, the network can generate valid molecules, one of which is shown in Figure 5.

Figure 5. Valid and unique SMGAN generated image.



Inference Tuning

The best trained model was used to run a simple loop over various values of maximum generator output length. Though the model was optimized for a length output of 10, that does not rule out the possibility that lengths smaller or larger than this number could produce more viable molecules. Given hardware restraints, the values picked were the counting numbers in the interval [3, 14].

Evaluation Metrics

A molecule's quality and viability were judged by using several performance metrics that were also used in MolGAN (Nicola De Cao, 2018). These metrics include generator BCEL, generated molecule validity, uniqueness, novelty, solubility druglikeness, and synthesizability. For simplification, GAN generated strings are measured against quality metrics, such as Validity, Uniqueness, and Novelty. In addition, viability of the molecules was estimated, such as, solubility, druglikeness, and synthesizability. The latter metrics were estimated using RDKit database descriptors. All metrics were calculated using Python (Van Rossum, 2009), basic statistics, and descriptors obtained through the RDKit API and Database molecule descriptors (RDKit, 2024). The specific implementations are also included in this work (Vasquez J, 2024).

Validity: is the ratio of the number of valid molecules to the number of all generated molecules. A molecule is said to be valid if it is recognized as valid by an open-source cheminformatics software (RDKit, 2024).

Uniqueness: is the percentage of unique generations to all valid generations, and is an attempt to quantify the diversity of molecule generations (Nicola De Cao, 2018).

Novelty: is the ability of the network to generate non-existent samples, that are never seen by the model. The generated SMILES string is canonicalized, then compared to other known molecules (RDKit, 2023). If the molecule exists but is not in-sample, then it is said to be novel.

Solubility: is an approximation of the degree of molecule hydrophilicity, such as high reactivity to water or dissolvability (Comer, 2001).

Druglikeness: Is calculated using the Quantitative Estimate of Druglikeness (QED) (Bickerton, 2012). A high druglikeness score is $QED \geq 0.7$, whereas a value of 0.4 to 0.69 is considered moderate, and anything less is considered low (Bickerton, 2012).

Synthesizability: is a probabilistic estimation of the ease of synthesizing a molecule (Ertl, Estimation of synthetic accessibility score of drug-like molecules based on molecular complexity and fragment contributions, 2009). This metric is fairly complex, but code written for synthesizability calculations were found to be external to the RDKit API, but have been used for over a decade (Ertl, sascorer, 2009).

Results

Molecules were generated using the optimal inference hyperparameters with the optimal generator output length of 10, followed by 13, 12, 6, and 14. This result shows evidence that better performance is possible with larger max generator output lengths, and should be further investigated in future work. For the purpose of this study, the best values were picked from any of the output length generations. Solubility, druglikeness, and synthesizability were all normalized between [0, 1].

Table 5. MolGAN and SMGAN Best Scoring Results

Metric	SMGAN	MolGAN
Valid (%)	31.20	100.00
Unique (%)	0.05	0.03
Novel (%)	100.00	100.00
Solubility	0.60	0.89
Druglikeness	0.02	0.62
Synthesizability	0.46	0.95

Though MolGAN uses a different dataset and network structure, both networks are able to generate valid, unique, novel, soluble, and synthesizable molecules. One performance problem with SMGAN is the druglikeness score. Table 5 shows a very small druglikeness, which could be improved by including the druglikeness score into the loss function of the generator during training. Examples of unique and valid molecules are shown in Figure 6 and Figure 7.

Figure 6. Valid SMGAN Generated Molecule w/ Max Length = 13

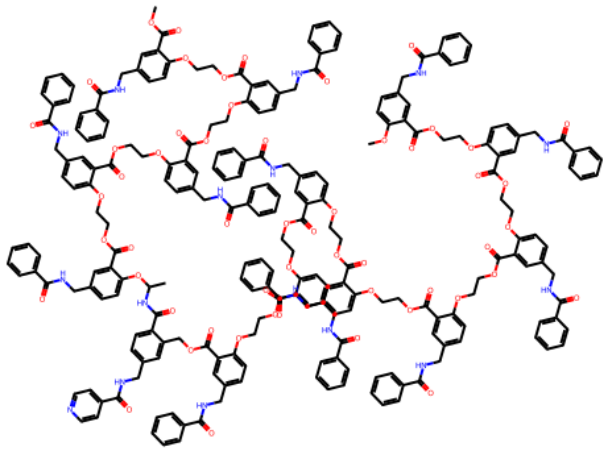
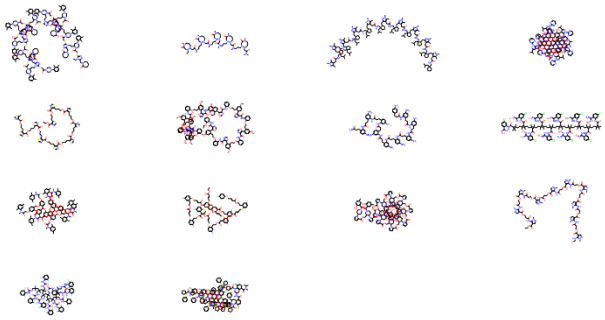


Figure 7. Several Valid and Unique SMGAN Generated Molecules with Max Length = 10.



Future Work

Besides time, another constraint is hardware availability. More GPU memory could allow for optimization on longer string generations. As was shown in the section on inference tuning, the string length of the generator is an important hyperparameter, and though the network was optimized on a string length of 10, during inference, lengths of 13, 12, 6, and 14 also performed relatively well. Unfortunately, 14 is the maximum string currently available given the hardware setup of this experiment.

Furthermore, training for more epochs should be investigated. MolGAN used 5k carefully picked molecules, rather than 25K randomly sampled molecules. Perhaps focusing in on certain molecules with a maximum or minimum complexity could

help convergence. Other sample discriminations can be used, such as mol weight, charge profile, bond count, and many others molecule characteristics.

Another improvement is to add a loss function that considers one, some, or all of the viability metrics. Along the same grain, perhaps splitting the data into a training set consisting of high scoring viability metrics would result in positive performance gains. It is possible that adding both of these viability methods would be ideal.

Still another potential improvement could be to add quality metrics to the loss function. For example, adding validity and uniqueness to the loss function could aid with generating a higher number of valid molecules. Novelty might not be a good candidate for this, because it was seen that the unique valid generations were all novel. This makes sense, because statistically, there are many possible molecule configurations that could be generated.

Conclusion

In this work, it was shown that valid SMILES strings could be synthesized through the use of SMGAN. With the best performing SMGAN model, synthesis of non-valid molecules is more than five time likely than valid molecules. However, some generations are valid, and score reasonably well according to the quality and viability metrics defined in Evaluation Metrics. In addition, MolGAN scored better on novelty, solubility, druglikeness and synthesizability. But SMGAN scored nearly twice as good on the uniqueness score, and as good as MolGAN on novelty. However, there are many ways to improve SMGAN performance that are low hanging fruit so to speak. For example, adding viability metrics to the loss function. Perhaps using deep reinforcement learning like MolGAN could drastically raise SMGAN's overall molecule synthesis performance.

References

Alqahtani, H. E. (19 de Dec de 2019). Applications of Generative Adversarial Networks (GANS): An Updated Review.

Arjovsky, M. E. (26 de Jan de 2017). Wasserstein GAN.

Bickerton, G. R. (2012). Quantifying the chemical beauty of drugs.

Bidisha, S. (2019). NeVAE: A Deep Generative Model for Molecular Graphs.

Comer, J. a. (2001). Lipophiicity profiles: theory and measurement. Zurick, Switzerland.

Cordeonnier, J.-B. E.-A. (20 de May de 2021). Multi-Head Attention: Collaborate Instead of Concatenate.

Das, B. (2018). An Improved Text Sentiment Classification Model Using TF-IDF and Next Word Negation.

Ertl, P. a. (2009). Estimation of synthetic accessibility score of drug-like molecules based on molecular complexity and fragment contributions. *Journal of Cheminformatics*, 1-8.

Ertl, P. a. (1 de 1 de 2009). *sascorer*. Obtenido de github: https://github.com/rdkit/rdkit/blob/master/Contrib/SA_Score/sascorer.py

Goodfellow, E. A. (10 de June de 2014). Generative Adversarial Networks.

Hankun Dai, E. A. (24 de Febuary de 2018). Syntax-Directed Variational Autoencoder for Structured Data.

Hunter, J. D. (2007). Matplotlib is a 2D graphics package used for Python for application development, interactive scripting, and publication-quality image generation across user interfaces and operating systems. *Computing in Science \& Engineering*, 90-95.

Jaun-Ni Wu, E. A. (2024). t-SMILES: A Fragment-based Molecular Representation Framework for De Novo Ligand Design. Hunan, China.

Jin, W. E. (29 de March de 2019). Junction Tree Variational Autoencoder for Molecular Graph Generation.

Kim, S. (2017). *Line Notation (SNUKES and InCHI)*. Obtenido de LibreTexts Chemistry.

Kusner, M. E. (6 de March de 2017). Grammer Variational Autoencoder.

Li, L. E. (16 de March de 2018). A System for Massively Parallel Hyperparameter Tuning.

Liu, X. E. (2021). Bi-directional gated recurrent unit neural network based nonlinear equalizer for coherent optical communication systems.

Mangalam, K. G. (29 de Dec de 2021). Overcoming Mode Collapse with Adaptive Multi Adversarial Training.

National Library of Medicine. (22 de July de 2024). *National Center for Biotechnology Information*. Obtenido de PubChem: https://pubchem.ncbi.nlm.nih.gov/

Nickolls, J. E. (2 de March de 2008). Scalable Parallel Programming with CUDA.

Nicola De Cao, T. K. (2018). MolGAN: An Implicit Generative Model for Small Molecular Graphs. Stochholm, Sweden.

Pagnomi, A. E. (11 de December de 2018). Conditional Variational Autoencoder for Neural Machine Translation.

Pedregosa, F. E. (2011). Scikit-learn: Machine Learning in {P}ython}. *Journal of Machine Learning Research*, 2825-2830.

Phrma. (22 de July de 2024). *Research & Development Policy Framework*. Obtenido de phrma.org: https://phrma.org/policy-issues/Research-and-Development-Policy-Framework

Pytorch. (20 de August de 2024). *Adam*. Obtenido de Pytorch: https://pytorch.org/docs/stable/generated/torch.optim.Adam.html

Ray. (20 de 8 de 2024). *Tune*. Obtenido de Ray: https://docs.ray.io/en/latest/tune/index.html

Ray. (20 de 8 de 2024). *Tune Trial Schedulers*. Obtenido de Ray: https://docs.ray.io/en/latest/tune/api/schedulers.html

RDKit. (2023). *rdkit.Chem.rdMolTransforms module*. Obtenido de rdkit.org: https://www.rdkit.org/docs/source/rdkit.Chem.rdMolTransforms.html

RDKit. (20 de August de 2024). *RDKit: Open-Source Cheminformatics Software*. Obtenido de rdkit.org: 2024

Rigoni, D. E. (1 de Sep de 2020). Conditional Constrained Graph Variational Autoencoders for Molecule Design.

Simonosvsky, M. K. (9 de Feb de 2018). GraphVAE: Towards Generation of Small Graphs Using Variational Autoencoders.

Van Rossum, G. a. (2009). Python 3 Reference Manual. Scotts Valley, CA, United States.

Vasquez J, V. (22 de August de 2024). *SMGAN*. Obtenido de github: https://github.com/vanthony715/SMGAN

Vaswani, A. E. (12 de June de 2017). Attention is All You Need.

Weininger, D. (1987 de June de 1987). SMILES, a Chemical Language and Information System. 1. Introduction to Methodology and Encoding Rules. Claremont, California, United States.