Open-Source Molecular Processing Pipeline for Generating Molecules

Shreyas V[1,2] **Jose Siguenza**[2] **Karan Bania**[1,2] **Bharath Ramsundar**[2]
[1]BITS Pilani Goa Campus [2]Deep Forest Sciences
`{shreyas, bharath}@deepforestsci.com`

Abstract

Generative models for molecules have shown considerable promise for use in computational chemistry, but remain difficult to use for non-experts. For this reason, we introduce open-source infrastructure for easily building generative molecular models into the widely used DeepChem [25] library with the aim of creating a robust and reusable molecular generation pipeline. In particular, we add high quality PyTorch [20] implementations of the Molecular Generative Adversarial Networks (MolGAN) [2] and Normalizing Flows [19]. Our implementations show strong performance comparable with past work [13, 2].

1 Introduction

The discovery of new molecules and materials is key to addressing challenges in chemistry, such as treating diseases and combating climate change [15, 27]. Traditional methods, however, are time-consuming and costly, limiting the exploration of the vast chemical space [22]. Generative models offer a deep learning-based solution, designing molecules with desired properties more efficiently. Despite their potential, these models typically require significant expertise in Python and machine learning.

To address this, we introduce open-source implementations of Molecular Generative Adversarial Networks (MolGAN) [2] and Normalizing Flow models [19] in pytorch into DeepChem[25], a widely used molecular machine learning library. MolGAN utilizes adversarial training [7] to generate novel molecules, while Normalizing Flow models employ exact likelihood methods for molecular generation. Our contributions simplify their use by providing accessible pipelines that require minimal prior knowledge, while also allowing advanced users to modify the models as needed.

2 Methods and Background

2.1 DeepChem and Generative Molecular Models

DeepChem [25] is a versatile open-source Python library tailored for machine learning on molecular and quantum datasets [24]. Its framework supports applications in areas such as drug discovery and biotech [32], breaking down scientific tasks into workflows built from core primitives. DeepChem has facilitated significant advancements, including large-scale molecular machine learning benchmarks via MoleculeNet [32], protein-ligand modeling [5], and generative molecule modeling [4].

While older DeepChem implementations of MolGANs and Normalizing Flows used TensorFlow, we migrate these models to PyTorch, ensuring tighter integration with DeepChem's ecosystem and broader compatibility. This enables users to leverage DeepChem's extensive layer library to experiment and build new models.

2.2 Representation of Molecules

The strength of neural networks lies in their ability to take in a complex input representation and transform it into a latent representation needed to solve a particular task. In this way, the choice of input representation plays a key role in governing how the model learns information about the molecule. Input representations often fall into one of two categories: (1) one-dimensional (e.g., string-based representations), (2) two-dimensional (e.g., molecular graphs).

2.2.1 One-dimensional representations

The most common one-dimensional representation of molecules is SMILES (Simplified Molecular Input Line Entry System) [31], which transforms a molecule into a sequence of characters based on predefined atom ordering rules. This representation enables the use of neural network architectures developed for language processing. For instance, previous work [6, 18] used recurrent neural networks as generative models to create SMILES strings. However, these methods often produce invalid SMILES that cannot be converted to molecular structures due to their disregard for SMILES grammar. To overcome this limitation, [12] introduced SELFIES (Self-Referencing Embedded Strings), an improved string representation. With SELFIES, a recurrent neural network can generate molecules with 100% validity, though validity here pertains to valency rules and does not guarantee molecular stability.

2.2.2 Molecules as Graphs

Molecules can also be represented as graphs, where atoms are nodes and bonds are edges. For MolGAN, molecules are undirected graphs G with edges E and nodes V, we employ the adjacency matrix formulation, A. Each atom is represented by a one-hot vector, and each bond type is represented as an adjacency tensor. This graphical approach captures molecular connectivity directly, unlike 1D representations.

2.3 MolGAN

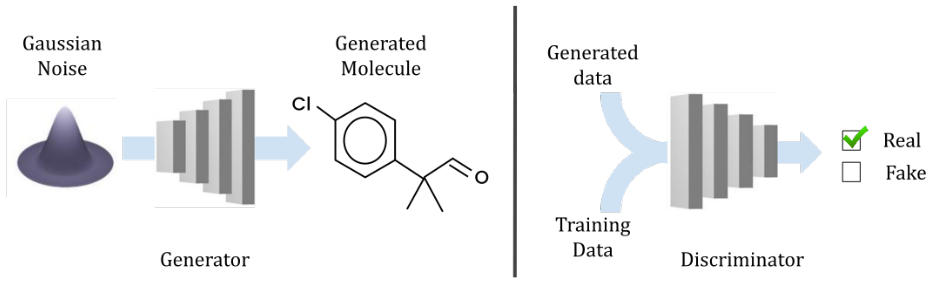

Figure 1: *Model Architecture of MolGAN*

Molecular Generative Adversarial Network (MolGAN) represents a novel approach in the Generative Space for molecules by employing the GAN framework, which allows for an implicit, likelihood-free generative model which overcomes various problems (like graph matching [29] and node ordering heuristics) with previous models. MolGAN performs similarly to current SMILES-based approaches as well, albeit it is more susceptible to model collapse. We have incorporated the following loss used by the paper

$$L(\mathbf{x}^{(i)}, G_\theta(z^{(i)}); \phi) = \underbrace{-D_\phi(\mathbf{x}^{(i)}) + D_\phi(G_\theta(z^{(i)}))}_{\text{Original WGAN Loss}} + \underbrace{\alpha(||\nabla_{\hat{x}^{(i)}} D_\phi(\hat{x}^{(i)})|| - 1)^2}_{\text{Gradient Penalty}} \tag{1}$$

This is an improved form of the WGAN [1] loss, where G_θ is the generator, D_ϕ is the discriminator, $\mathbf{x}^{(i)} \sim p_{data}(\mathbf{x})$ and $\mathbf{z}^{(i)} \sim p_z(\mathbf{z})$ and $\hat{\mathbf{x}}^{(i)}$ is a sampled linear combination as the following equation $\hat{\mathbf{x}}^{(i)} = \epsilon\mathbf{x}^{(i)} + (1-\epsilon)G_\theta(\mathbf{z}^{(i)})$ with $\epsilon \sim U(0,1)$. The model architecture is displayed in Figure 1.

2.4 Normalizing Flows

Normalizing Flows constitute a generative model that uses invertible transformations to model a probability distribution. This methodology enables the computation of likelihoods and the generation of samples by transforming simple base distributions (e.g., gaussian) into more complex ones through a sequence of invertible (& differentiable) transformations. Normalizing flows allow for direct sampling from the target distribution and thus are amazing for property-guided generation. The model architecture is displayed in Figure 2.

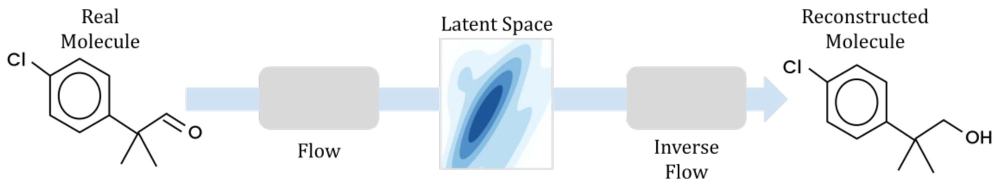

Figure 2: *Model Architecture of Normalizing Flow models*

3 Implementation

The generative models were implemented in three main components: Layers, Base Model, and Molecule Generation Pipeline. We standardize (“deepchemize” [24]) molecule generation by providing `BasicMolGANModel` and `NormalizingFlowModel` which are highly flexible, allowing users to experiment with generators, discriminators, flow layers, and more.

3.1 Layers

MolGAN: The Discriminator in MolGAN begins with an encoder layer composed of multiple convolutional layers, followed by an aggregation layer. The architecture is flexible, allowing easy customization based on user requirements.

Normalizing Flows: Normalizing Flow layers are responsible for both forward and backward computation. In DeepChem, the Normalizing Flows pipeline supports linear layers, which are computationally efficient compared to more complex layers like planar or autoregressive. These layers use linear transformations to model relationships between molecular dimensions: $\mathbf{g}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$ Here, $W \in \mathbb{R}^{D\times D}$ and $b \in \mathbb{R}^{D}$ are parameters, and if W is invertible, the function is invertible as well.

3.2 Base Model

The Base Model consists of three primary components: the Generator, Discriminator, and Normalizing Flow model. Each component is modular and can be adjusted independently for flexibility in experimentation.

The **Generator** is modeled as an MLP [17] with varying units to implicitly model a probability distribution over molecular graphs. It outputs continuous objects representing nodes and edges, which are transformed using the Gumbel Softmax trick [9]. The model can also be adapted to use other methods such as the straight-through estimator [21].

The **Discriminator** scores the generated molecular graph through a series of relational graph convolutional layers [28]. These layers update node representations based on their neighbors and edge types. After several layers of convolution, MLPs extract the final score of the graph. More detailed mathematical formulations of the Generator, Discriminator, and Normalizing Flow model are mentioned in Appendix A

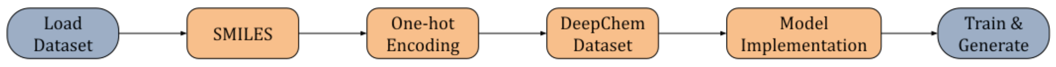

Figure 3: *Molecule generation pipeline for MolGAN*

3.3 Molecule Generation Pipelines

3.3.1 MolGAN

Training / Generating from MolGAN on custom datasets follows a very simple pipeline, which can be achieved in a few lines of code; we describe the pipeline in Figure 3, which involves extracting SMILES [31] representations and wrapping them in a DeepChem dataset that can be used to train the model. The full pipeline is demonstrated in B.1 with a few lines of code.

3.3.2 Normalizing Flows

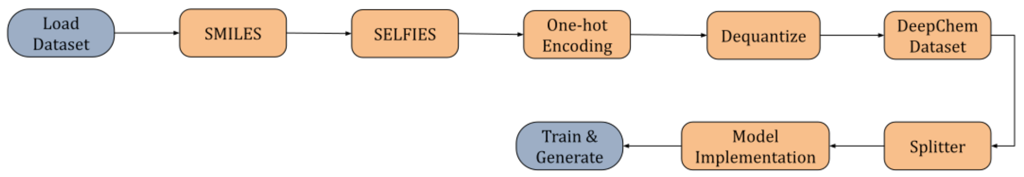

Figure 4: *Molecule generation pipeline for Normalizing flows*

Analogous to the MolGAN pipeline, the Normalizing Flows pipeline (Figure 4) expects a SELF-IES [12] string as input. There are added steps for Dequantization (i.e., adding noise in [0, 1) to every input) The full pipeline is demonstrated in B.2 with a few lines of code.

4 Experiments

4.1 Datasets

We use the following publicly available datasets: QM7 [26], BBBP [16], Lipophilicity [8], PPB [30], and QM9 [23]. QM7 is a subset of GDB-13, consisting of molecules with up to 7 heavy atoms (C, N, O, and S). The BBBP dataset includes around 2,000 molecules with binary labels related to blood-brain barrier permeability. The Lipophilicity dataset, sourced from ChEMBL, contains 4,200 compounds. The PPB dataset, curated from PubChem BioAssay by the Maximum Unbiased Validation (MUV) group, includes around 11,000 compounds. QM9 is a subset of GDB-17, comprising 134,000 organic molecules with up to 9 heavy atoms. All datasets were obtained through MoleculeNet [32].

For nomalizing flows we utilize the full dataset for all experiments, however, following [2], for MolGAN, we only use molecules that only have C, N, O and F in them. The final number of valid molecules in every dataset are shown in 1.

Dataset	**Number of Samples**
BBBP	1631
PPB	1291
QM7	5470
QM9	105984
Lipophilicity	3360

Table 1: Number of samples in each dataset.

4.2 Experimental Setup

For **MolGAN**, we use four edge types (single, double, triple, and no-bond), five node types (C, N, O, F, and PAD) similar to [2], whilst our discriminator discriminates among molecules of all lengths, our generator is restricted to small molecule generation, limited to 9 vertices. To allow this, we would set the maximum number of allowed atoms to the maximum in a dataset. We report mean results across 10 seeds, each involving generation of 6400 molecules, as in [2].

For **Normalizing Flows**, we used two layers of flows with Masked Affine Flows [3] along with an ActNorm layer [11]. We used a Multivariate Normal Distribution from PyTorch [20] to build our flow model. All hyper-parameters are listed in C. All experiments used Python 3.10.

4.3 Results

Dataset	**Model**	**Val(↑)**	**Uni(↑)**	**Nov(↑)**	**SAS(↓)**	**Dru(↑)**
QM7	MolGAN	92.1	4.18	100.0	2.11	70.45
	Norm Flow	100.0	91.87	100.0	5.29	52.19
QM9	MolGAN	89.67	4.46	100.0	2.83	66.03
	Norm Flow	100.0	99.15	100.0	6.58	44.55
	Original Implementation [2]	87.7	2.9	97.7	—*	—*
BBBP	MolGAN	57.18	0.28	100.0	5.24	55.42
	Norm Flow	92.83	98.98	100.0	5.76	49.50
Lipophilicity	MolGAN	49.57	0.08	100.0	5.75	53.58
	Norm Flow	100.0	99.36	100.0	6.18	49.39
PPB	MolGAN	14.04	25.07	100.0	9.17	18.31
	Norm Flow	100.0	98.60	100.0	5.95	50.30

Table 2: Evaluation performance of the open-source models on QM7, BBBP, Lipophilicity, PPB, QM9 datasets. The models are evaluated on the Validity of molecules generated (Val), Uniqueness (Uni), Novelty (Nov), Synthetic Accessibility Score (SAS) out of 10, and Druglikeliness (Dru). SAS is reported out of 10, whereas everything else is a %. (* - not reported in the paper)

Our results 2 demonstrate that our implementation's performance is comparable to existing work ([2, 13]). For a fair comparison, we only compare our implementation to the non-RL results from [2], also [2] only train and evaluate on QM9.

5 Conclusion & Discussion

In this work, we improve DeepChem's generative modeling tools and provide a more standardized and scalable implementation that makes generative molecular methods more accessible to scientists. Standard Machine Learning practices are built within DeepChem (e.g., checkpointing, validation, logging, etc.), which would otherwise need some form of human expertise. Benchmarks show comparable performance with existing implementations, and the tight integration with DeepChem facilitates fast future improvements. This will also allow for Reinforcement Learning [2] based approaches to be incorporated easily. We observed high variance for datasets where the average number of atoms in a molecule was much larger than 9, and leave this for future work. We will provide multi-GPU training support in future work using Distributed Data Parallelism [14], Sharding & PyTorch [20].

Impact Statement

This paper makes Generative Molecular modeling accessible to a broad spectrum of people, and while this is intended, it is not tough to modify these models to make them generate toxic and harmful molecules. However, the synthesis of novel molecules is an area that requires a lot of human expertise, and in general, is not easy to do. In a future where synthesis methods are also equally accessible, these models can be potentially dangerous, but at the present time, the positive outcomes outweigh the negative ones.

References

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan, 2017.

[2] Nicola De Cao and Thomas Kipf. Molgan: An implicit generative model for small molecular graphs, 2022.

[3] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp, 2017.

[4] N. C. Frey, V. Gadepally, and B. Ramsundar. Fast-flows: Flow-based models for molecular graph generation. `https://arxiv.org/abs/2201.12419`, 2023.

[5] Joseph Gomes, Bharath Ramsundar, Evan N. Feinberg, and Vijay S. Pande. Atomic convolutional networks for predicting protein-ligand binding affinity. *CoRR*, abs/1703.10603, 2017.

[6] Rafael Gómez-Bombarelli, Jennifer N. Wei, David Duvenaud, JoséMiguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D. Hirzel, Ryan P. Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS Central Science*, 4(2):268–276, 02 2018.

[7] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[8] Anne Hersey. ChEMBL deposited data set - AZ_dataset. Technical report, February 2015.

[9] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax, 2017.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[11] Diederik P. Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions, 2018.

[12] Mario Krenn, Florian Häse, AkshatKumar Nigam, Pascal Friederich, and Alan Aspuru-Guzik. Self-referencing embedded strings (selfies): A 100*Machine Learning: Science and Technology*, 1(4):045024, October 2020.

[13] Maksim Kuznetsov and Daniil Polykovskiy. Molgrow: A graph normalizing flow for hierarchical molecular generation, 2021.

[14] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training, 2020.

[15] Songtao Liu, Zhengkai Tu, Minkai Xu, Zuobai Zhang, Lu Lin, Rex Ying, Jian Tang, Peilin Zhao, and Dinghao Wu. Fusionretro: Molecule representation fusion via in-context learning for retrosynthetic planning, 2023.

[16] Ines Filipa Martins, Ana L. Teixeira, Luis Pinheiro, and Andre O. Falcao. A bayesian approach to in silico blood-brain barrier penetration modeling. *Journal of Chemical Information and Modeling*, 52(6):1686–1697, 06 2012.

[17] Fionn Murtagh. Multilayer perceptrons for classification and regression. *Neurocomputing*, 2(5):183–197, 1991.

[18] Marcus Olivecrona, Thomas Blaschke, Ola Engkvist, and Hongming Chen. Molecular de-novo design through deep reinforcement learning. *Journal of Cheminformatics*, 9(1):48, 2017.

[19] George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. Normalizing flows for probabilistic modeling and inference, 2021.

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[21] Max B. Paulus, Chris J. Maddison, and Andreas Krause. Rao-blackwellizing the straight-through gumbel-softmax gradient estimator, 2020.

[22] P. G. Polishchuk, T. I. Madzhidov, and A. Varnek. Estimation of the size of drug-like chemical space based on gdb-17 data. *Journal of Computer-Aided Molecular Design*, 27(8):675–679, 2013.

[23] Raghunathan Ramakrishnan, Pavlo O Dral, Matthias Rupp, and O Anatole von Lilienfeld. Quantum chemistry structures and properties of 134 kilo molecules. *Sci. Data*, 1(1):140022, August 2014.

[24] Bharath Ramsundar, Peter Eastman, Aidan MacBride, and Nhan Vu Trang. Making deepchem a better framework for ai-driven science, April 2021.

[25] Bharath Ramsundar, Peter Eastman, Patrick Walters, Vijay Pande, Karl Leswing, and Zhenqin Wu. *Deep Learning for the Life Sciences*. O'Reilly Media, 2019. `https://www.amazon.com/Deep-Learning-Life-Sciences-Microscopy/dp/1492039837`.

[26] Matthias Rupp, Alexandre Tkatchenko, Klaus-Robert Müller, and O. Anatole von Lilienfeld. Fast and accurate modeling of molecular atomization energies with machine learning. *Phys. Rev. Lett.*, 108:058301, Jan 2012.

[27] Benjamin Sanchez and Alán Aspuru-Guzik. Inverse molecular design using machine learning: Generative models for matter engineering. *Science*, 361:360–365, 07 2018.

[28] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks, 2017.

[29] Martin Simonovsky and Nikos Komodakis. Graphvae: Towards generation of small graphs using variational autoencoders, 2018.

[30] Renxiao Wang, Xueliang Fang, Yipin Lu, Chao-Yie Yang, and Shaomeng Wang. The PDBbind database: methodologies and updates. *J. Med. Chem.*, 48(12):4111–4119, June 2005.

[31] David Weininger. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of Chemical Information and Computer Sciences*, 28(1):31–36, 1988.

[32] Zhenqin Wu, Bharath Ramsundar, Evan Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S. Pappu, Keith Leswing, and Vijay Pande. Moleculenet: A benchmark for molecular machine learning. *Chem. Sci.*, 9:513–530, 2018.

Appendix

A Mathematical Formulations

A.1 Molecules as Graphs

For MolGAN, each molecule can be represented as an undirected Graph G with a set of edges E and nodes V. Each atom $v_i \in V$ is associated with a D-dimensional one-hot vector $\mathbf{x}_i$. Each edge $(v_i, v_j) \in E$ is also associated with a bond type $y \in \{1, \dots, Y\}$. Thus, we have a representation of a graph as two objects: a $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_n]^T \in \mathbb{R}^{N \times D}$ and an adjacency tensor $\mathbf{A} \in \mathbb{R}^{N \times N \times Y}$. $\mathbf{A}_{i,j} \in \mathbb{R}^Y$ is a one-hot vector indicating the type of edge between i and j.

A.2 Generator

For any latent variable

$$z \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

the generator $\mathcal{G}_\theta(z)$ outputs two continuous objects:

$$\mathbf{X} \in \mathbb{R}^{N \times D}, \mathbf{A} \in \mathbb{R}^{N \times N \times D}$$

Using the Gumbel Softmax trick [9], categorical sampling is performed as:

$$\tilde{\mathbf{X}} = \mathbf{X} + \text{Gumbel}(\mu = 0, \beta = 1) \tag{2}$$

$$\tilde{\mathbf{A}} = \mathbf{A}_{ijy} + \text{Gumbel}(\mu = 0, \beta = 1) \tag{3}$$

Alternatively, $\tilde{\mathbf{X}}$ and $\tilde{\mathbf{A}}$ can remain as $\mathbf{X}$ and $\mathbf{A}$.

A.3 Discriminator

The discriminator updates node representations through relational graph convolution as:

$$\boldsymbol{h}_i^{\prime(\ell+1)} = f_s^{(\ell)}\left(\boldsymbol{h}_i^{(\ell)}, \boldsymbol{x}_i\right) + \sum_{j=1}^{N}\sum_{y=1}^{Y} \frac{\tilde{\boldsymbol{A}}_{ijy}}{|\mathcal{N}_i|} f_y^{(\ell)}\left(\boldsymbol{h}_j^{(\ell)}, \boldsymbol{x}_j\right) \tag{4}$$

$$\boldsymbol{h}_i^{(\ell+1)} = \tanh\left(\boldsymbol{h}_i^{\prime(\ell+1)}\right) \tag{5}$$

where $\boldsymbol{h}_i^{(l)}$ represents the signal at node i and layer l, and $f_s^{(l)}$ is a self-connection transformation. After several layers of graph convolution, the graph score is calculated as:

$$\boldsymbol{h}'_{\mathcal{G}} = \sum_{v \in \mathcal{V}} \sigma\left(i\left(\boldsymbol{h}_v^{(L)}, \boldsymbol{x}_v\right)\right) \odot \tanh\left(j\left(\boldsymbol{h}_v^{(L)}, \boldsymbol{x}_v\right)\right) \tag{6}$$

$$\boldsymbol{h}_{\mathcal{G}} = \tanh(\boldsymbol{h}'_{\mathcal{G}}) \tag{7}$$

where σ is the sigmoid function $\sigma = \frac{1}{1+e^{-x}}$.

A.4 Normalizing Flow Model

The Normalizing Flow model starts with a base distribution p_z and transforms it into a density function p_x. The transformation $x = L(z)$ is invertible, and the probability density is updated as:

$$\rho_X(x) = \rho_Z(z) \cdot |det(J_L(z))|^{-1}$$

where $J_L(z)$ is the Jacobian determinant of the transformation.

B Code Snippets

B.1 MolGAN Pipeline

Training -

```python
from deepchem.models.torch_models import BasicMolGANModel as MolGAN
from deepchem.models.optimizers import ExponentialDecay
import deepchem as dc
import torch.nn.functional as F
import torch
...
gan = MolGAN(learning_rate=ExponentialDecay(0.001, 0.9, 5000))
dataset = dc.data.NumpyDataset([x.adjacency_matrix for x in features],[x.node_features for x in features])
def iterbatches(epochs):
    for i in range(epochs):
            for batch in dataset.iterbatches(batch_size=gan.batch_size
    , pad_batches=True):
                adjacency_tensor = F.one_hot(
                        torch.Tensor(batch[0]).to(torch.int64),
                        gan.edges).to(torch.float32)
                node_tensor = F.one_hot(
                        torch.Tensor(batch[1]).to(torch.int64),
                        gan.nodes).to(torch.float32)
                yield {gan.data_inputs[0]: adjacency_tensor, gan.
    data_inputs[1]:node_tensor}
    # train model
    gan.fit_gan(iterbatches(8), generator_steps=0.2,
    checkpoint_interval=0)
```

Inference -

```
    generated_data = gan.predict_gan_generator(10)
    # convert graphs to RDKitmolecules
    new_mols = feat.defeaturize(generated_data)
```

B.2 NormalizngFlows Pipeline

Training -

```
    import deepchem as dc
    from deepchem.data import NumpyDataset
    from torch.distributions.multivariate_normal import
    MultivariateNormal
    from rdkit import Chem
    from dc.torch_models.nflows import *

    # Pass data through pipeline (mentioned above)
    ...

    # Construct flow model
    flows = [ActNorm(latent_size)]
    nfm = NormFlow(flows, distribution, dim)
    nfm.fit(max_iterations, optimizer)
```

Inference -

```
    mols = nfm.generate(num_molecules=1000)
    valid_mols = validate_mols(mols)
```

C Hyperparameters

For our MolGAN implementation, similar to [2], we vary the given hyperparameters, and pick the best for each dataset prioritising Validity of generations. Except for QM9, we train each model for 30 epochs on the whole dataset. QM9 is trained similar to [2], i.e., 300 epochs on a random 5k subset, we also perform early stopping if the uniqueness goes below 2%. Apart from this we fix Generator Steps to 0.2, i.e., one step every 5 discriminator steps, a batch size of 32, learning rate of $1e-4$, and an embedding dimension of 32.

Hyperparameter	**Values**
Dropout Rates	$\{0.0, 0.1, 0.25\}$
Sampling Mode	Straight-through, Gumbel, SoftMax

Table 3: Hyperparameter settings for MolGAN.

For the Normalizing Flows implementation, we use a learning rate of 1e-4, and an equal weight decay. We train on a batch size of 1024 for 100 epochs. Both the models were trained on with Adam optimizer [10].