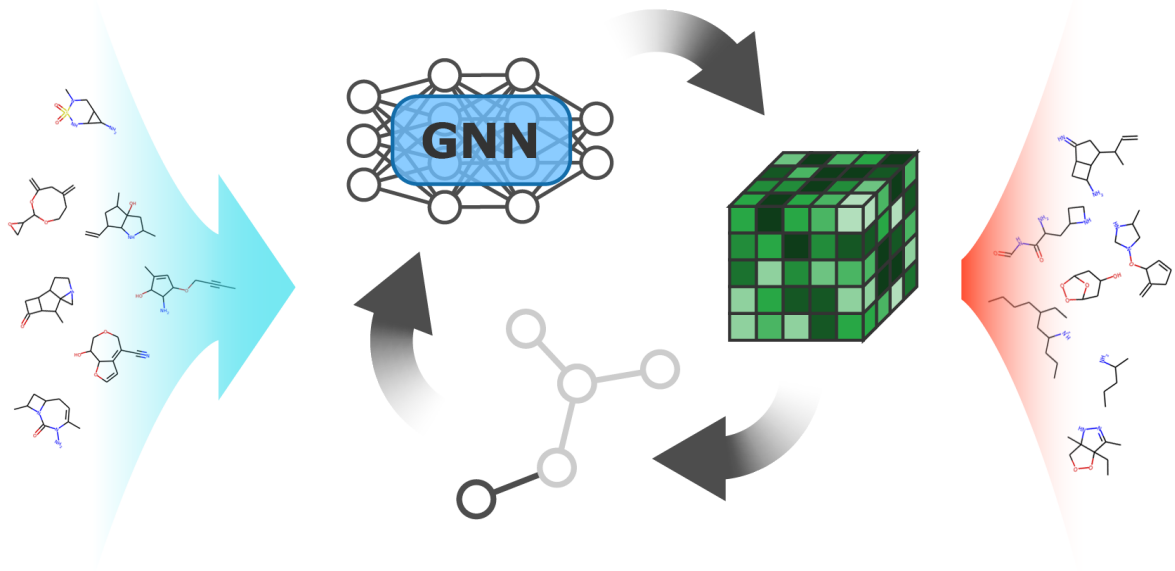
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Sampling a Subset of Chemical Space with GNN-Based Generative Models

Evaluating the Chemical Space Coverage of Molecular Generative Models Using GDB-13

Master's thesis in Computer Science and Engineering

TOBIAS RASTEMO

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Sampling a Subset of Chemical Space with GNN-Based Generative Models

Evaluating the Chemical Space Coverage of Molecular Generative Models Using GDB-13

TOBIAS RASTEMO

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Sampling a Subset of Chemical Space with GNN-Based Generative Models
Evaluating the Chemical Space Coverage of Molecular Generative Models Using GDB-13
TOBIAS RASTEMO

Supervisor: Shirin Tavara, Department of Computer Science and Engineering
Advisor: Rocío Mercado, AstraZeneca
Examiner: Alexander Schliep, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: In the center of the images a depiction of the iterative graph generation process is shown. Here, a GNN takes a graph as input, outputs an action sample space (green-checkered cube), and then an action is is sampled and applied to the graph. This iterative process is repeated until some termination criteria is met. To the left training data from GDB-13 is input into the process, and to the right, generated molecules from trained models is output.

Typeset in LaTeX
Gothenburg, Sweden 2020

Sampling a Subset of Chemical Space with GNN-Based Generative Models
Evaluating the Chemical Space Coverage of Molecular Generative Models Using GDB-13
TOBIAS RASTEMO
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In recent years deep neural network models have been used in the field of drug discovery for *de novo* molecular design. One, somewhat novel, field of deep learning that has seen some use in drug discovery is *graph neural networks* (GNN:s). This thesis evaluates 6 GNN models for use in molecular graph generation. The evaluation is based on a benchmark introduced by Arús-Pous *et al.* [1], which measures how well models sample a subset of chemical space. The models are also compared to existing *recurrent neural network models (RNN:s)*, which use string representation of molecules. The best performing GNN models achieve comparable scores to the RNN models, all though the RNN models score higher. Even though the GNN models score slightly lower on two of the training sets, they still show great potential for future use and merit further research. In addition to this, a data loading scheme for PyTorch is introduced, which increases training speed by loading training data from disk efficiently.

Keywords: machine learning, deep learning, graph neural networks, message passing neural network, *de novo* molecular design, graph generation

Acknowledgements

I would like to thank my supervisor at AstraZeneca, Rocío Mercado for help, guidance, and support during the course of this thesis work. Much of the work done was based on models implemented and coded by her, and she helped me overcome many hurdles and problems. She also helped with many aspects of the report, and helped make it what it is. The entire Molecular AI group at AstraZeneca also provided me with help and insight for many aspects of my thesis, and I would like to especially mention Ola Engkvist and Esben Bjerrum for continuous input that helped shape the project.

I would also like to thank my supervisor at Chalmers, Shirin Tavara for her input and guidance during the project, as well as constructive and well thought out criticism for my final report. My examiner Alexander Schliep, and my opponent Jithinraj Sreekumar also provided me with good input for my report, and I thank them as well.

Lastly I would like to thank Fredrik Ring and Inger Rastemo for proof reading, and providing feed back to my report.

Tobias Rastemo, Gothenburg, August 2020

Contents

List of Figures

List of Tables

1
Introduction

Deep learning (DL) has recently grown in popularity in the fields of drug discovery and *de novo* molecular design. The flexibility of DL models coupled with the increasing amount of available molecular data gives DL many advantages over more traditional machine learning methods, such as support vector machines and random forests. As such, DL is on its way to becoming the new standard in pharmaceutical drug discovery[2].

A rather novel class of models within DL is the graph neural network (GNN), introduced by Gori *et al.* [3] and Scarselli *et al.* [4] in 2005 and 2009, respectively. A GNN takes a graph as input and by utilizing internal neural networks it produces a real-valued output vector. This output vector can be seen as an embedded version of the graph and can be used in other neural networks for global property prediction. Being able to operate on graphs directly is very useful since data in many domains are naturally represented as graphs. One of these domains is drug discovery, where graphs can be used to represent molecules.

Since GNN:s by themselves only output graph embeddings, they are not sufficient to generate new graphs, as is the goal of *de novo* design. However, GNN:s can be used as part of generative network models to create new molecules [5, 6]. This is accomplished by adding nodes and edges iteratively to graphs until some termination criterion is met, and the final graph is output.

The GNN networks show great promise for graph generation, but there is yet to emerge a standardized metric for evaluating how well a generative model performs. This is not only the case for graph generative models, but for generative models in general. One metric, proposed by Arús-Pous *et al.* [7], utilizes a subset of chemical space to evaluate model performance. The authors use this metric to benchmark their RNN-based generative models.

This thesis evaluates GNN-based generative models using the benchmark proposed by Arús-Pous *et al.* [7]. This benchmark evaluates the percent chemical space coverage of an enumerated database of molecules called GDB-13 [8]. GDB-13 consists of small, organic molecules with 13 or fewer heavy atoms. Heavy atoms in the molecules are limited to those in the set $\{C, N, O, S, Cl\}$.

In contrast to this work, the models benchmarked by Arús-Pous *et al.* [7] were based on a string representation of molecules called SMILES [9]; SMILES have shown

great promise in the field of *de novo* design. By employing their benchmark on deep generative models based on GNN:s, this thesis aims to explore if graph-based models are also viable for use in *de novo* molecular design, and to get a qualitative measure of how they compare to string-based models.

2
Technical Background

Here the relevant background and theory regarding the specific type of deep learning networks, as well as the datasets used in this thesis, are presented. In Sections 2.1 and 2.2 the basics of GNN:s and the generative process are covered. Then, in Section 2.3, previous work on string-based generative models is explained. Finally, existing benchmarking metrics for generative models and the GDB-13 dataset are briefly discussed.

It is assumed that the reader has previous knowledge in the domain of machine and deep learning; as such, the general principles behind these concepts are not explained. Basic understanding of biochemistry is also helpful but not required, nor is it covered here.

2.1 Graph Neural Networks

Graph neural networks (GNN:s) are a relatively new addition to deep learning introduced by Gori *et al.* [3] in 2005. GNN:s can be viewed as a network that takes a graph as input and embeds it in some latent real space. Since they operate directly on graphs, GNN:s are applicable in a wide variety of fields and have gained noticeable traction over the last few years. For a brief discussion on some of the GNN:s published in the literature, see Appendix A.

A multitude of different network architectures has been introduced by various researchers and several are nicely summarized in [10]. All of these networks work differently and will not be explained in detail here, but they all share the same underlying GNN architecture. As GNN:s are the main component used in the generative models in this thesis, the following sections are dedicated to explaining the underlying GNN architecture.

2.1.1 General Model Scheme

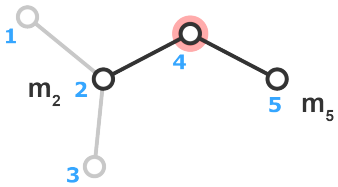
The input graph to the GNN is generally represented as a set of node and edge features. If V and E denote the nodes and edges of the graph, respectively, then the node and edge features are abbreviated as $\{x_v,\ v \in V\}$ and $\{e_{vw},\ (v,w) \in E\}$. In the example of molecular graphs, the nodes represent atoms in the molecule, and the edges the bonds between them. The node features can, for example, include: atom type, formal charge, and the number of implicit hydrogens. The edge features

can, for example, include the bond type (single, double, triple, aromatic) and bond lengths.

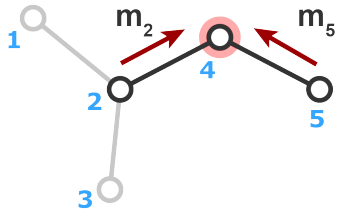
A GNN operates on its graph input by passing messages between connected nodes, which are then used to update a hidden state in each node. The hidden node states are then aggregated into a single graph embedding, which is the output of the GNN. Messages, hidden states, and the output are all represented by real valued vectors.

The message passing and update processes are repeated several times before aggregation, where the exact number depends on the specific GNN implementation. In the original GNN architecture the number of message passes is dynamic, and can change depending on the graph input. This thesis, however, is concerned with a subset of GNN:s called *message passing neural networks* (MPNN), where the number of message passes is a hyperparameter.

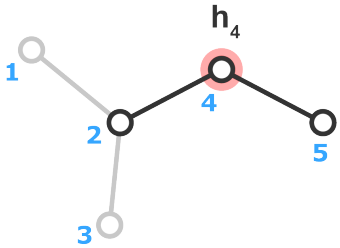
In Figure 2.1, a schematic view of the MPNN message passing is presented. Below, Equation (2.1) describes the mathematics behind this process. In this equation M_t is the message embedding function, U_t the hidden note state update function, and R the global aggregation function.



(a) Create the messages m_2 and m_5.



(b) Pass the messages to node 4.



(c) Update hidden state h_4 using the messages.

Figure 2.1: Illustration of a partial message passing step in an MPNN. When passing messages to node 4, highlighted in red, the neighboring nodes, node 2 and 5, are involved. The messages m_2 and m_5 are first generated using the states of node 2 and 5, as illustrated in Figure (a). These messages are then passed to node 4 and used to update its hidden state, as seen in Figures (b) and (c). This process is repeated for each node in the graph, and for each message passing step.

$$
\begin{aligned}
h_v^0 &= x_v \\
m_v^{t+1} &= \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw}) \\
h_v^{t+1} &= U_t(h_v^t, m_v^{t+1}) \\
y &= R(\{h_v : v \in G\})
\end{aligned}
\tag{2.1}
$$

Here t is the message passing iteration, m_v^t the node message incoming to node v, h_v^t the hidden node state of node v, $N(v)$ the set of all nodes neighboring v, y the target property, and G the graph in question. By making the functions U_t, M_t, and R learnable deep neural networks, this becomes a deep model.

2.2 Graph-based Generative Models

The graph generative model used in this thesis is based on the concept of iteratively constructing graphs. Starting from a base graph, the model adds nodes and edges at each iteration step and decides what construction action to apply based on the current graph. Figure 2.2 shows a schematic of this process.

The construction of graphs is not inherently connected to deep learning. However, in this thesis GNN:s are utilized for graph generation. This refers to the fact that, in each step of the graph construction process, a GNN is used to determine what action is taken.

2.2.1 General Construction Scheme

In the deep generative models presented here, the construction action is decided using GNN:s. More precisely, a GNN is used to embed the graph into a latent space, and this embedding is then fed into another neural network. This neural network then outputs an *action probability distribution* (APD), which can be used to sample a construction action, and the action is then applied to the graph. In the APD every element represents the probability of that action being sampled. A view of this process is presented in Figure 2.3

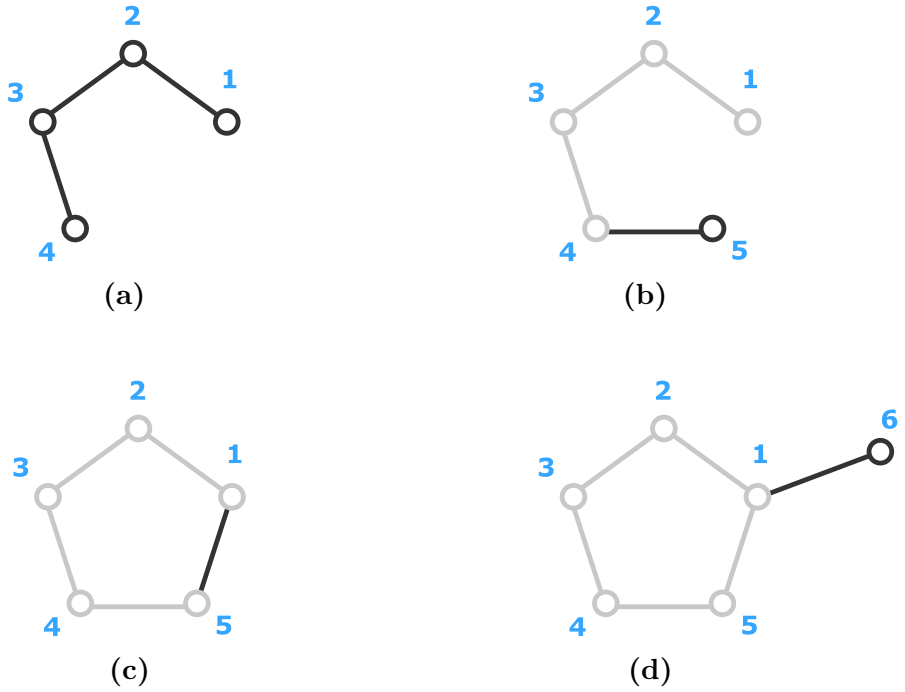


Figure 2.2: This figure illustrates the iterative graph generation process. In Figure (a) we start with a graph consisting of four nodes and three edges. Then in Figure (b) a node along with an edge is added. In Figure (c) this newly added node is then connected to another node with an edge. Finally in Figure (d) the last node and edge are added. These sort of graph generation steps are referred to as *actions* in the thesis.

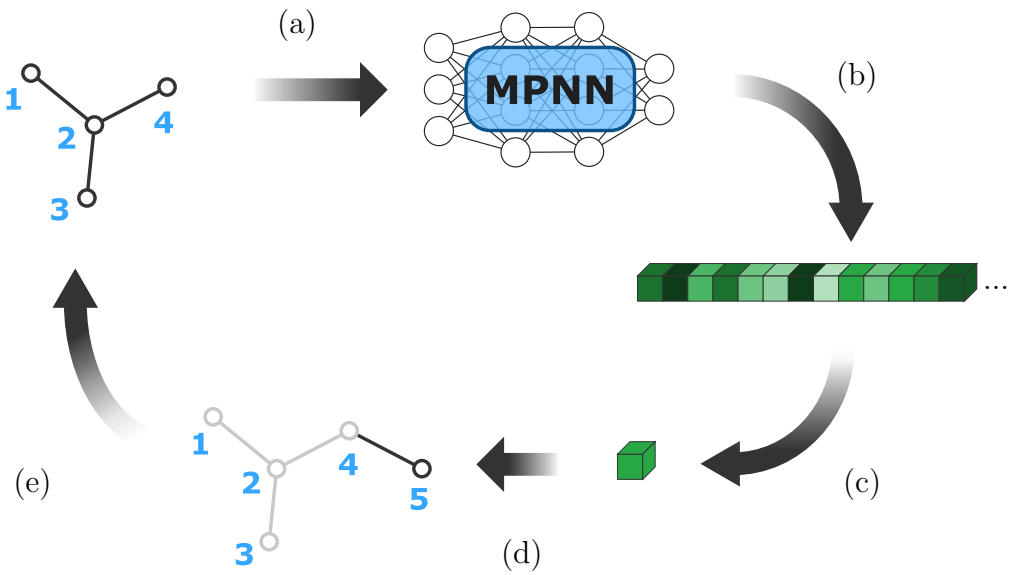


Figure 2.3: In this figure the process of generating graphs using MPNN:s is described. (a) The graph is passed through the MPNN. (b) This produces an APD, containing all possible actions. (c) An action is sampled from the APD, in this case to add a node and connect it to node 4. (d) The action is applied, resulting in a new graph. (e) The new graph replaces the old graph and the process repeats.

As an algorithm, the process is described as follows:

1. Initialize an empty graph $\mathcal{G}_0$
2. Embed the current graph with an embedding function

$$y^t = \texttt{GNN}(\mathcal{G}_t) \tag{2.2}$$

3. Pass the embedded graph through another function that outputs an action probability distribution

$$APD_t = F(y^t) \tag{2.3}$$

4. Sample an action probabalistically from A_t and apply it to the graph

$$\mathcal{G}_{t+1} = \text{APPLY}(\mathcal{G}_t, a \sim APD_t) \tag{2.4}$$

5. Repeat steps (2)-(4) until a terminate action has been sampled

Here F is learnable network. Note that this graph construction process is stochastic, since actions are sampled probabilistically.

2.2.2 Action Sampling

There are several potential schemes for adding edges and nodes when constructing a graph. When a node is added it can always be connected to an existing node in the graph, for example. This node could also be allowed to be added without an edge, thus making disconnected graphs a possible output. There are also choices for how to add edges: can edges always be added between all nodes, or is it only possible to add edges to the most newly added node? In the scheme explained here, and which is used in the thesis, a newly added node is also connected to an existing node in the graph, and edges can only be connected to the most recently added node.

When constructing the graphs it is also necessary to determine how to represent the different construction actions in the APD. When adding a node, for example, there are several features that need to be chosen: the features of the node added, to what node it is being connected, and the features of the edge used to connect it. Note that the newly added node is also connected to an existing node in the graph. The set of different node features, abbreviated by A, includes all different combinations of features. The same applies to the edge features B. What previous node the new node is added to is represented by the set of all nodes V. In the end, the APD for adding a new node can be viewed as a three dimensional tensor with dimensions $|A|, |B|, |V|$. Each element in the tensor represents a combination of the different choices for node features, edges, and connecting nodes. This tensor is flattened, which produces a vector of length $|A| \times |B| \times |V|$, which is the APD for adding nodes. Similarly, adding edges results in a tensor with dimensions $|V| \times |B|$, which can also be flattened. An illustration of these tensors is displayed in Figure 2.4[1].

[1]This image was heavily inspired by a similar figure from Li *et al.* [6]

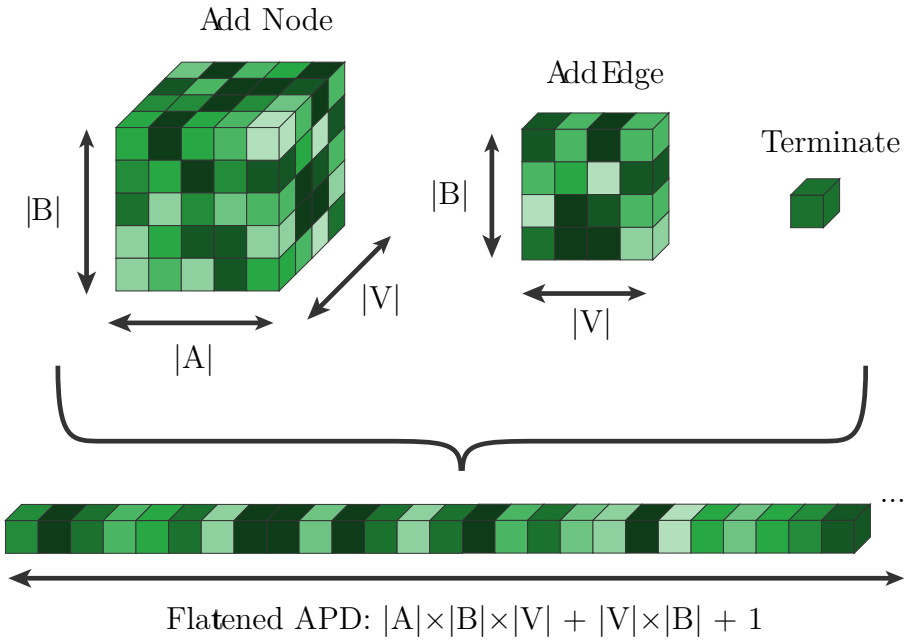


Figure 2.4: Illustration of the different tensors used to represent construction actions. The "Add Node" tensor has three dimensions corresponding to what node is connected and how. This consists of: its node features, what node it gets connected to, and the features of the edge used to connect it. Similarly the "Add Edge" tensor has two dimensions: one representing to what node to connect, and the other what edge is used to connect the nodes. Note that "Add Edge" always connects from the most recently added node. The last action is the termination action which is a single element. These tensors are flattened and put into the final APD.

Concatenating the two APD tensors and a single element for termination gives us the final APD. Thus, the index sampled from this flattened APD corresponds to one of the elements in the tensors presented, and, consequently, a construction action.

Something worth noting about this action sampling scheme is that it is possible to sample invalid actions. The reason for this is that the output APD from the model will always be of fixed size, since the neural networks have fixed output size. Thus, the indices corresponding to, for example, connecting newly added nodes to non-existing nodes will be present in the APD. Note that the value of an element in the APD is the probability to sample this action, thus, it can be sampled unless it is exactly zero. If these indices are sampled, then the action is invalid. In the models used in this thesis sampling invalid actions result in a termination of the construction process. It is also possible to mask invalid actions, and only sample valid actions, but this was not done in this thesis. Instead, when an invalid action is sampled the current graph is taken as the final output.

2.2.3 Training Graph Generative Models

The training of graph generative models can seen somewhat confusing at first, but is very similar to that of RNN:s. Training loss is calculated for each APD a model gen-

erates, i.e., each APD that is output from the GNN model is compared to a ground truth APD. This ground truth APD corresponds to a sub-graph in the training set, which is part of a complete graph. Since this sub-graph belongs to a full graph it is known what the next correct action should be, in order to construct the full graph. This single action corresponds to an APD of all zeros, except for the correct action, which has a value, and consequently a probability, of 1.

The loss during training is calculated for each APD output at each construction step. Given an APD output, APD_{out}, from the model, the *Kullback–Leibler divergence* (D_{KL}) is used as the loss function. Conceptually this means that the model is trained to generate APD:s that look like the ones in the training set. The definition of the loss function can be seen in equation (2.5).

$$D_{KL}(APD_{target}||APD_{out}) = \sum_{x} APD_{target}(x) \log\left(\frac{APD_{out}}{APD_{target}}\right) \tag{2.5}$$

For a given graph in the training set there exist multiple construction orderings. Thus, an ordering needs to be chosen. This can be done in several ways. For example, starting from an initial node, the graph can be deconstructed by a breadth- or depth-first algorithm. When an ordering has been chosen, a construction path for the graph is computed, and for each subgraph in the construction process we can compute the ground truth APD, APD_{target}.

2.2.4 Existing Implementations

Li *et al.* [5] proposes a generative model for iteratively building graphs. This scheme is very similar to the one that has been presented so far in the thesis, since it was from this publication a lot of inspiration was taken. They evaluate their model on several tests. For generating molecules, the graph-based approach perform better than string-based approaches on some metrics.

Li *et al.* [6] introduce two networks architectures for generating molecules: MolMP, and MolRNN. Both closely follow the scheme of Li *et al.* [5], and this thesis. The biggest difference is that they use a *graph convolutional network* (GCN) [11] as their GNN. MolMP uses a *multilayer perceptron* (MLP) to get the action probability distributions from a graph embedding, while MolRNN uses a *gated recurrent unit* (GRU) instead. This GRU keeps track of a hidden construction state. The authors report that the best performing model is MolRNN and that it outperforms string-based networks on several generative measurements.

2.3 String-based Generative Models

For generation of molecular graphs, there exist networks that utilize RNN:s and string representations of molecules such as SMILES. Previous work done by Arús-Pous *et al.* [1] uses such models, and the results there are relevant for comparison

to the GNN based models evaluated in this thesis. In this section we give a short overview of the concepts relevant to understanding the string-based models.

2.3.1 SMILES

Simplified molecular input line entry specification (SMILES) is a string representation of molecular graphs that was introduced by Weininger [9]. SMILES can represent a large amount of drug-like molecules and offer a very concise and compact notation. In Figure 2.5 a selection of molecules and an example SMILES for each is displayed.

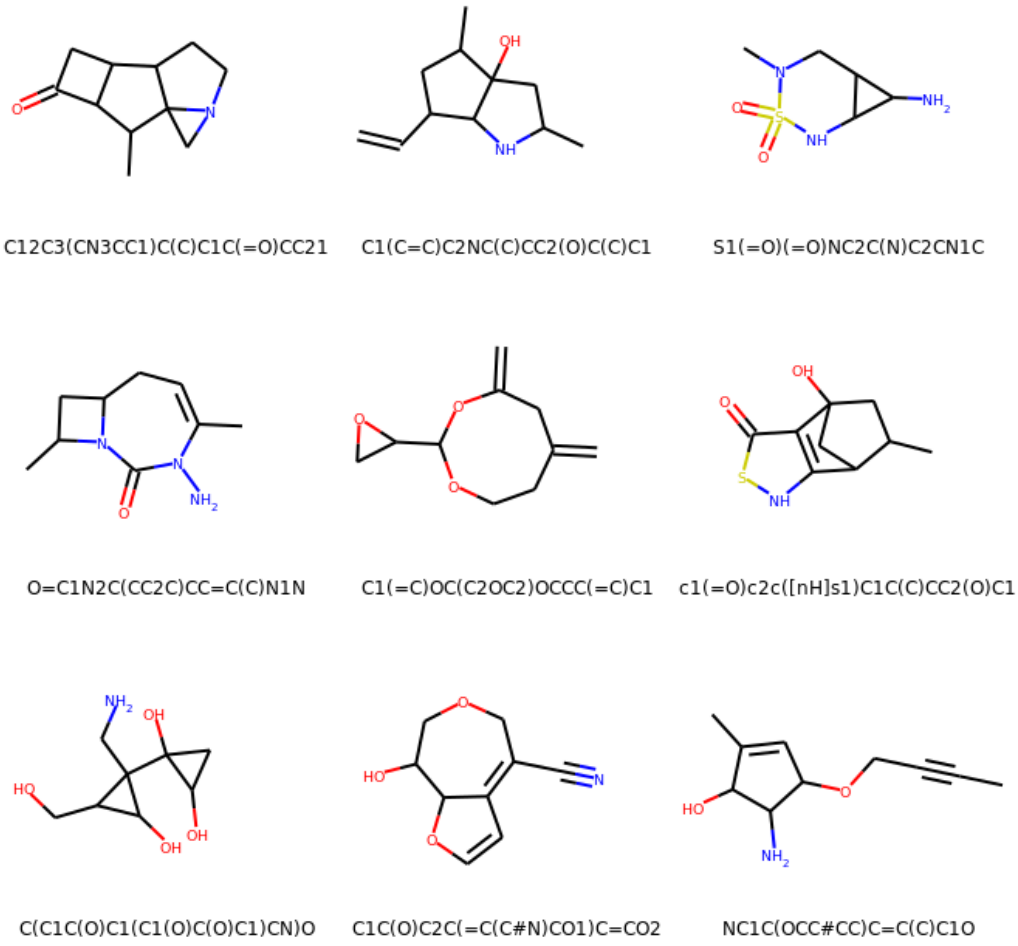


Figure 2.5: Various molecules from GDB-13 and corresponding examples of one SMILES representation for each.

It is worth noting that a molecular graph does not have a unique SMILES representation, and can be represented by many different SMILES. There exist canonical SMILES representations, one of which is used in GDB-13; see Section 2.5 for an explanation of GDB-13. One advantage of having multiple SMILES per molecule is that it enables augmentation of training sets by using several SMILES per sample.

2.3.2 Model Structure

One of the most common architectures to use for string data is the RNN. Both LSTM:s and GRU:s are examples of RNNs, and are very popular. This is also the case for processing SMILES where models use layers of LSTM:s and GRU:s.

RNN-based models operate similarly to graph-based models, but, instead of constructing graphs directly, they construct strings. They do this by generating an APD from the current string and sampling what token to add. A token here can be several characters, since for example some atoms types, like Cl, contain several characters but represent one token in the SMILES.

Although RNN:s have shown a lot of success in molecule generation they might have some disadvantages. One is the fact that LSTM models using SMILES not only have to learn the underlying rules of atom connectivity, but also the SMILES grammar. For example, the strings have syntax for creating rings where brackets are used and must be closed. This can also lead to the LSTM "forgetting" to close a bracket and creating an invalid SMILES string.

2.3.3 Existing Implementations

Arús-Pous *et al.* [7] utilize an RNN-based model for generating molecules. The model is implemented using GRU:s [12] and operates on SMILES [9]. Training data is a subset (0.001) of GDB-13 [8], an enumerated dataset of all drug-like molecules with 13 or fewer heavy atoms.

To measure the performance of the generative model, the authors calculate what percentage of the entire GDB-13 dataset the model covers when generating 2 billion structures. This is compared to an ideal model that samples GDB-13 uniformly, which is expected to cover 0.8712 of GDB-13 when sampling 2 billion molecules. The model is reported to cover 0.689 of GDB-13 and the authors also conclude that it is more difficult for the model to sample some advanced molecules, such as molecules with complex ring systems.

Arús-Pous *et al.* [1] use the same approach to generating molecules as in [7]. The main difference is the training data representation used. Instead of using canonical SMILES, the atom ordering is instead randomized. Randomizing ordering also allows for easy data augmentation by including several orderings for a single molecule.

The authors calculate the chemical space coverage as in [7] and find that using randomized SMILES, the GDB-13 coverage is 0.830 when sampling 2 billion molecules, compared 0.871 with the ideal model.

2.4 Evaluating Generative Models

Here is presented some of the existing metrics that are currently used to evaluate generative models. The simplest metrics are:

- counting the percentage of valid molecules
- counting the amount of unique molecules generated
- counting how many of the generated molecules are novel, i.e., not in the training set.

While these metrics give useful information and are valuable, they are not perfect. For example a model could generate only one molecule which could both be valid and novel, but these metrics wouldn't capture the uniqueness. A model could also generate only novel and unique molecules that could all be invalid.

There exist proposed frameworks for evaluating the generative power of *de novo* molecular models. One such example is *GuacaMol* proposed by Brown *et al.* [13]. This benchmark is a collection of aforementioned metrics, two more general metrics, and several target-directed metrics. These two additional metrics are more complicated and measure how alike the generated set is to the training set and how well the generated set approximates a variety of chemical descriptors. The target-directed metrics measure performance for a set of tasks where specific molecules are used. The authors conclude that the non-targeted parts of their proposed benchmark are too easy for modern deep generative models.

2.5 Enumerated Database of Molecules (GDB-13)

Blum and Reymond [8] published an enumerated database of all drug like molecules with 13 or fewer heavy atoms in the set

$$C, N, Cl, O, S.$$

Enumerated here refers to the fact that all possible valid molecules have been included. This was done by constructing molecules according to deterministic construction rules. The database is available online and contains roughly 1 billion SMILES. For an explanation of SMILES see 2.3.1.

There also exists an enumerated database of molecules of 17 heavy atoms or fewer [14]. This dataset contains roughly 166 billion molecules.

3
Methods

Here we present the methods used to train and benchmark the models in the thesis. In addition to this we also discuss some steps used to improve training speed and performance and how previous models were trained, as well as how the GDB-13 benchmark is defined and calculated.

3.1 Choice of GNN Models

Six different models were trained and evaluated for this thesis. Here we present a brief overview of what they look like and how they are defined. For a more detailed description, see Appendix B. Note also that there exists some ambiguity in the naming of the different models in regard to the publications they are based on.

MNN: This is the simplest network used in this thesis. The message passing function M_t is a single learnable linear layer, and the update function U_t is a GRU. This part of the model is the same as the GGNN introduced by Li *et al.* [15]. As an aggregation function R a simple summation is used.

GGNN: This model is also similar to the GGNN introduced by Li *et al.* [15], but instead of a single layer in M_t an MLP is used instead. A GRU is still used as an update function and the same readout as Li *et al.* [15] proposed, which consists of a kind of soft attention on the nodes.

S2V: Here again an MLP is utilized in M_t and a GRU is used as U_t. The biggest addition here is an *attention network*, introduced by Vaswani *et al.* [16], as a readout function R.

AttGGNN: This network is the same as the GGNN but adds an additional attention network on top of the message passing function M_t.

AttS2V: Just as with the AttGGNN an attention layer is added on top of M_t. The rest of the network is the same as the S2V.

EMN: The EMN is similar to the D-MPNN presented by Yang *et al.* [17] in that it uses hidden edge states, and messages between edges. But, as M_t, U_t, and R more complex networks are used. The main part of M_t is made up of an attention

network, and a GRU is used as an update. The readout is very similar to that of the GGNN but operates on edge features instead of node features.

3.2 Training

Both the models proposed in this thesis, as well as earlier string-based models, were trained on a number of different data sets, and used a variety of hyperparameters. This is covered in the following sections.

3.2.1 Training Sets

When training the graph-based models, three different datasets were used. All of these were subsets of GDB-13 and contained: 1K, 10K, and 100K randomly sampled molecules respectively. The original goal was to also train on a 1M subset, but due to time constraints this was not possible, as training the slowest models on the 100K subset took several days.

The graphs contained in each training set was not only randomly chosen from GDB-13, but their construction ordering was also randomized, as discussed in Section 2.2.3. In this work only a single ordering was chosen for each graph, but by choosing several orderings the data would have been augmented, i.e., more samples would have been available during training. Data augmentation done in this manner is very similar to that done by Arús-Pous *et al.* [1] where it improved model performance. It was not doable in this project due to the scope of the project.

The main idea behind using different splits of the GDB-13 set is to not only evaluate how well the models perform on each set, but also how changing the size of the dataset affects performance. The point of using differently sized training sets was not to find the best training set. Smaller sets were expected to perform worse and overfit more, but the aim was to get insight to how model performance was affected by training set size. Even if samples are abundant in a large dataset such as GDB-13, this is not always the case for real world applications.

3.2.2 Hyperparameters

For the different MPNN models that were used there exist a number of different common hyperparameters that are associated with the deep network structures. These had to be fine-tuned in order to ensure that the models trained properly and could learn the distribution of properties in the training set. Below we give a brief overview of the different common hyperparameters.

- **Message Passes:** The number of message passes done by the MPNN networks.

Hyperparameter	Value
Message Passes	3
Hidden Node Feature Size	100
MLP Depth	4
MLP Hidden Dimension	500

Table 3.1: The values of the common hyperparameters used in the different GNN networks.

- **Hidden Node/Edge Feature Size:** The size of the hidden feature vector associated with the nodes, and in the case of EMN with the edges.
- **MLP Depth:** Throughout each MPNN, a number of MLP:s are used, and this hyperparameter controls the depth of all of these networks.
- **MLP Hidden Dimension:** The number of nodes in each hidden layer in the MLP:s of the different models.

In Table 3.1, a list of these hyperparameters and their values are shown.

Learning rate (lr) was also tuned in order to improve training and convergence, and in addition to this a custom learning rate decay scheme was adopted. This scheme is very similar to exponential learning rate decay but not entirely the same. At a set interval of batches the lr is updated by multiplying it with the learning rate decay factor (lrdf) taken to the power of the number of batches processed. In Equation (3.1) this scheme is described.

$$\begin{aligned} r_{i+1} &= r_i f^{n_i} = r_i f^{i \cdot b}, \quad n_i = i \cdot b \\ r_{i+1} &= r_0 \prod_{k=1}^{i} f^{k \cdot b} = r_0 f^{\left(\sum_{k=1}^{i} k \cdot b\right)} = r_0 f^{b \frac{i(i+1)}{2}} \end{aligned} \tag{3.1}$$

Here n_i is the number of batches processed at update step i, r_i is the learning rate at update i, b the interval between updates, and f the decay factor. From the equations it appears that the decay scheme follows a Gaussian curve, since the exponent of f is proportional to the square of i. This is also visible in Figure 3.1, where an example of the learning rate decay with three different decay factors is plotted. Here the 10K subset of GDB-13 was used.

In order for the training on the different GDB-13 subsets to converge, the initial learning rate and decay factors were tuned depending on the training set. In Table 3.2 these different learning rates and decay factors are listed for the respective subsets. To choose a decay factor, different values were tested for each subset, and the factor that led to the lowest and smoothest converged loss for the MNN model was chosen and used for all different GNN models. A more thorough search could be done but due to time constraints only the MNN model was considered during optimization.

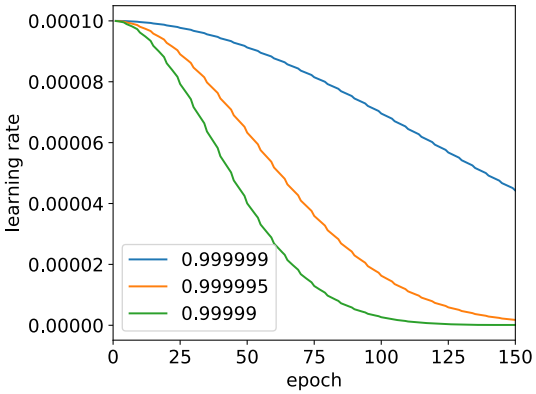


Figure 3.1: An example of how the learning rate changes over time for three learning rate decay factors. Here the MNN model and 10K GDB-13 subset was used.

GDB-13 Subset	Initial LR	LRDF
1K	0.0001	0.9999
10K	0.0001	0.999995
100K	0.00005	0.999999995

Table 3.2: The different learning rates and learning rate decay factors used for the different GDB-13 subsets. Note that the same initial learning rate was used for all models on a given subset.

3.2.3 String based Models

When training the string-based models, Arús-Pous *et al.* [7] also used 3 subsets of the GDB-13 database. These contained 1K, 10K, and 1M molecules each; a 100K set as is used in this thesis was not included. In addition to this the model was also trained using an augmented data set [1]. The authors used several different randomized SMILES representations for each molecule, thus increasing the data set size. This lead to great performance improvements.

They also generated 2 billion SMILES every 5 epochs to see how much of the chemical space could be sampled. As a baseline for how well chemical space was being sampled, they also calculated the performance of an ideal model. This ideal model would sample chemical space uniformly and thus when sampling 2 billion molecules with this model the expected fraction of chemical space found should have been ≈ 0.8712. For a more in depth discussion of the GDB-13 benchmark see 3.5

3.2.4 Hardware and Software

Training was done on two different platforms: a desktop workstation, and remotely via a computer cluster. Both setups were using linux and the hardware specification can be seen in Table 3.3

Resource	Desktop	Cluster
OS	CentOS Linux 7	CentOS Linux 7
GPU	GeForce RTX 2080 Ti	Tesla K80
CPU	Intel(R) Xeon(R) W-2125	Intel(R) Xeon(R) CPU E5-2683
Ram	64 GB	32 GB

Table 3.3: The hardware used for training and benchmarking. Note that two different platforms were used for this task: a dekstop, and a remote cluster.

3.3 Training Data and Data Storage

It is not obvious how training data for the graph models was stored and represented. Thus, the following sections explain the data structures used, how it was stored in memory, and what preprocessing was necessary.

3.3.1 Graph Representation

To store a molecule as a graph all features of its atoms and bonds need to represented as nodes and edges in a graph. Each atom is considered to be a node, and the bonds between atoms are the edges. As common with graphs the data is then stored as two tensors: one representing node features, and the other connections between nodes. The node and edge features that are represented in the feature tensors are presented in Tables 3.4 and 3.5 respectively.

Type	Features
atom	C, N, O, S, Cl
hydrogen	$0, 1, 2, 3$
charge	$-1, 0, 1$
chirality	None, R, S

Table 3.4: The node features represented in the graph data.

Type	Features
bond	single, double, triple, aromatic

Table 3.5: The edge features represented in the graph data.

Each node is represented as a matrix where each row is a concatenation of one-hot encoded features. The edge features are represented by an adjacency tensor, which is just an adjacency matrix with one-hot encoded features as entries. To illustrate this, consider the molecule presented in Figure 3.2. In this molecule, the set of different node features is:

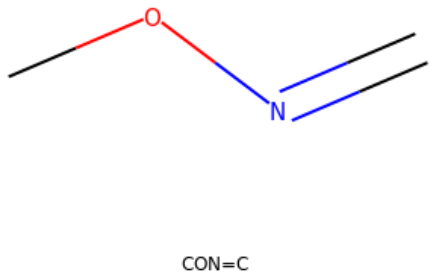


Figure 3.2: A small sample molecule from GDB-13.

$$\begin{aligned} \text{atoms} &= \{C, O, N, S, Cl\} \\ \text{hydrogen} &= \{0, 1, 2, 3\} \\ \text{charge} &= \{-1, 0, 1\} \\ \text{chirality} &= \{None, S, R\} \end{aligned}$$

As an example of the encoded node features, let's look at the oxygen atom in the middle left. It has no hydrogen atoms, neutral charge, and no chirality. So the one hot encoded features become:

$$\begin{aligned} \text{atom} &: [0, 1, 0, 0, 0] \\ \text{hydrogen} &: [1, 0, 0, 0] \\ \text{charge} &: [0, 1, 0] \\ \text{chirality} &: [1, 0, 0] \end{aligned}$$

and their concatenation will become a row in our node features tensor like below:

$$[\overbrace{0, 1, 0, 0, 0}^{atom}, \overbrace{1, 0, 0, 0}^{hydrogen}, \overbrace{0, 1, 0}^{charge}, \overbrace{1, 0, 0}^{chirality}].$$

If we again consider our sample molecule and enumerate the atoms from left to right we can write the complete node feature tensor, T_v, as

$$T_v = \left[\begin{array}{ccccc|cccc|ccc|ccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}\right] \tag{3.2}$$

where vertical lines have been added to illustrate which part of each row corresponds to a node feature. Note that we could have chosen to one-hot encode all of the features but this would have required a tensor with a dimensionality greater than the number of node features, and would have resulted in even more sparse data and consequently greater memory usage.

Moving onto edges, in Equation (3.3) the adjacency matrix, A, for this same molecule can be seen.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.3}$$

Note that as we are working with undirected graphs the adjacency matrix is symmetric.

The adjacency matrix does not contain all the information about the bonds between atoms, we also need to represent the bond type. The set of bond types is:

$$bonds = \{Single, Double, Triple, Aromatic\}.$$

Thus the single bond between the oxygen and nitrogen is encoded as $[1, 0, 0, 0]$ and the triple bond between nitrogen and the right most carbon is $[0, 0, 1, 0]$. If no bond is present we use $[0, 0, 0, 0]$. If we put this all into the adjacency matrix we end up with the edge feature tensor, T_e, which can be seen in Equation (3.4). If we reduce the tensor by summation of the innermost dimension we end up with the adjacency matrix.

$$T_e = \begin{bmatrix} [0,0,0,0] & [1,0,0,0] & [0,0,0,0] & [0,0,0,0] \\ [1,0,0,0] & [0,0,0,0] & [1,0,0,0] & [0,0,0,0] \\ [0,0,0,0] & [1,0,0,0] & [0,0,0,0] & [0,0,1,0] \\ [0,0,0,0] & [0,0,0,0] & [0,0,1,0] & [0,0,0,0] \end{bmatrix} \tag{3.4}$$

3.3.2 Data Preprocessing and Data Representation

All the data used is present in GDB-13, which contains roughly 1 billion molecules. This means that the amount of data is more than sufficient for training deep generative models. However data preprocessing was necessary to extract the relevant data for each molecule.

The molecules in the training sets, taken from GDB-13, were originally represented as SMILES. In order to use these in the GNN generative models, the data were converted into tensors containing the relevant node and edge features. In addition to this, a node construction ordering was calculated for each molecule, which was then taken as the ground truth during training.

3.3.3 Padding Data

A single input to the GNN of a model consists of the pair of feature tensors (T_v, T_e). For a given molecular graph the shape of the feature tensors will be dependent on the number of nodes in the graph. This is not compatible with the GNN which has a fixed shape of its input. It was thus necessary to pad the feature tensors of all data samples to match the dimensions of the largest molecular graph in the dataset. Once the model has been specified this input shape cannot be changed and thus a trained model is restricted to handle graphs smaller or equal to the largest graph it was trained on. Note that padding data does increase the memory usage, but, it also enables data to be stored with consistent spacing in memory. In theory this should enable faster reads from disk, since the data layout on disk is then known.

The node features are padded by appending rows of zeros. In total there should be as many rows in the node feature tensors as the maximum number of nodes in the largest molecular graph in the training set. The edge feature tensor is padded by appending rows and columns of zeros.

3.4 Data Loading

During development of the graph based models it became apparent that the largest bottleneck when training GNN models was reading training data from disk. This was most noticable when looking at GPU utilization, which was jumping between 10% and 20% during model training. Further profiling revealed that a majority of the time (up to 70%, during each training epoch) was spent on loading data from disk. Thus, in order to improve training speed and resource usage, some time was spent investigating why this problem appeared, and how it could be fixed. In the end a satisfactory solution was found.

In the following sections the solution to the problem is explained. In order to understand the bottleneck it is necessary to know how training data was stored on and loaded from disk previously, therefor this is also discussed.

3.4.1 HDF

A very common way to store training data for deep learning, especially image-base tasks, is the Hierarchical Data Format (HDF5) [18]. Some advantages are that it allows for simple partitioning of data within a single file. This means that in order to access parts of the dataset it is not necessary to load all data into memory. HDF5 is easily accessible in Python through the package `h5py` [19]. All data was stored using HDF5 and loaded using data loaders from the Python package `torch`.

3.4.2 Disk Bottleneck

The reason for the slow reading speeds was the way data was loaded from disk into memory. All data loading in Python was originally managed by the `DataLoader`

class from the package `torch`. This is common practice as the `DataLoader` class allows for simple management of batches, and shuffling during training. It also turned out that this was where the bottleneck was. According to the PyTorch documentation data is loaded according to the scheme in Listing 3.1 [20] below.

Listing 3.1: Semantics for the PyTorch `DataLoader`.

```
for indices in batch_sampler:
    yield collate_fn([dataset[i] for i in indices])
```

This shows that the data is accessed once for each index in every batch. In the case with HDF5 this leads to reading new data from disk every time we access a new index. Reading data from disk is a time consuming process and in general it is advantageous to avoid multiple reads, and instead read data contiguously. Thus, the source of the problem seemed to be this disk access scheme.

The proposed solution was to read data from disk in contiguous batches in such a way that only a single read per batch was required. This was accomplished by writing code that circumvented the data loading scheme in the `DataLoader` class, and forced the data loader to not iterate over the indices. This seemed to alleviate the disk reading bottleneck, but there were some disadvantages associated with this approach. For example it did not allow the shuffling of data dynamically between batches, since the batches were read contiguously. To solve this problem a new scheme was adopted where data was loaded in blocks of batches. This is discussed in the next section.

3.4.3 Loading in Blocks and Shuffling

Shuffling data can have a huge impact when training a model. If a model only sees the same batches over and over it is more prone to overfit to the training data. Therefore it was necessary to enable shuffling of training data, while still benefiting from the performance gain associated with reading blocks of data.

The simplest solution is to only read data from disk once and store the entire training set in memory. Memory wise this is possible with the smaller datasets consisting of 1K, 10K, and 100K molecules each. As can be seen in Table 3.6, the largest of these three takes up roughly 1.3 GB in memory, which is manageable. To possibly allow for training on the larger 1M dataset this was not considered an option, as its size is roughly 9 GB. Nonetheless, in the end this 1M dataset was not used for training in this thesis.

The final approach was to write a custom `DataLoader` where data was read contiguously in blocks of batches that fit into memory. Once the data block was loaded the samples were shuffled and batched as shown in Listing 3.2.

Dataset	# Graphs	# Subgraphs	Size [Mb]
1K	979	10,764	13
10K	9988	109,262	129
100K	100289	1,056,561	1300
1M	998922	13,844,921	9000

Table 3.6: The table shows size and number of samples for the different GDB-13 datasets. Each dataset is a subset of the entire GDB-13 set.

Listing 3.2: Pseudo code illustrating the semantics of the custom `BlockDataloader`. Note here that the main difference compared to the Pytorch `Dataloader` is that the blocks are loaded from disk into memory contiguously. Shuffling was managed by the inner `batch_sampler`.

```
for block_indices in block_sampler:
    block = dataset[block_indices]
    batch_sampler = BatchSampler(block)
    for indices in batch_sampler:
        yield collate_fn([block[i] for i in indices])
```

This resulted in shuffling only parts of the dataset but was deemed to be sufficient given large datasets. This method resulted in fast load times, while still retaining the ability to shuffle.

3.5 GDB-13 as a Benchmark

The idea of using GDB-13 as a benchmark is to measure how well models sample chemical space when trained on only a subset of it. This is accomplished by measuring what percentage of GDB-13 is found when sampling a set number of molecules. To understand the motivation for why this is a good approach, we need to look at what an ideal model would be.

3.5.1 The Ideal Model

The ideal model samples chemical space uniformly. Such a model is ideal since it shows no bias towards a specific part of chemical space, and thus will sample more of it. In order to measure how well this ideal model performs, the *coupon collector's problem* is considered, i.e., what is the expected number of samples required to sample the entire space. In the case of GDB-13 the entire space consists of roughly 1 billion points. If N is the number of samples required to sample the entire space, then

$$\mathbf{E}_{ideal}[N] \approx 2 \cdot 10^{10}.$$

This means that the ideal model needs to sample roughly 20 billion molecules in order to find the entire chemical space. This gives an upper bound for performance:

in theory, 20 billion molecules can be sampled by the model.

A problem that arises is that sampling 20 billion molecules and calculating what percentage of chemical space they cover is a very computationally heavy task. This is infeasible and instead 100K molecules are sampled. This means that it is also necessary to calculate what percentage of GDB-13 the ideal model covers when sampling 100K molecules. The ideal models coverage when sampling 100K molecules is given by the expression

$$\mathbf{E}_{ideal}[X] = 1 - (1 - p)^k = \varphi(k),$$

where k is the number of sampled molecules and p is the probability of sampling any molecule. Note also that the expression has been assigned to the function $\varphi(k)$. Given this expression the ideal model is expected to sample

$$\mathbf{E}_{ideal}[X] = 1 - (1 - 1.023 \cdot 10^{-9})^{10^5} \approx 1.023 \cdot 10^{-4}$$

of chemical space. For a more in depth explanation of the ideal model, see Appendix D

3.5.2 Benchmark Metrics

The benchmark is the same as proposed and used by [21] and consists of several metrics based on GDB-13. To calculate the metrics, the following quantities are used:

$$ratio_{in} = \frac{|in|}{k},$$
$$ratio_{unique} = \frac{|unique|}{k}.$$

Here, k is the number of sampled molecules, $|in|$ the number of sampled molecules in GDB-13, and $|unique|$ the number of distinct sampled molecules in GDB-13. In Figure 3.3 a depiction of the different sampled sets is presented. Using these, the metrics are defined as:

$$completeness = \frac{ratio_{unique}}{\varphi(k)},$$

$$uniformity = \frac{ratio_{unique}}{\varphi(|in|)},$$

$$closedness = ratio_{in}.$$

Another metric, the most relevant in this thesis, is the product of the three aforementioned metrics. This is called *ucc* and is defined as follows:

$$ucc = completeness \cdot uniformity \cdot closedness.$$

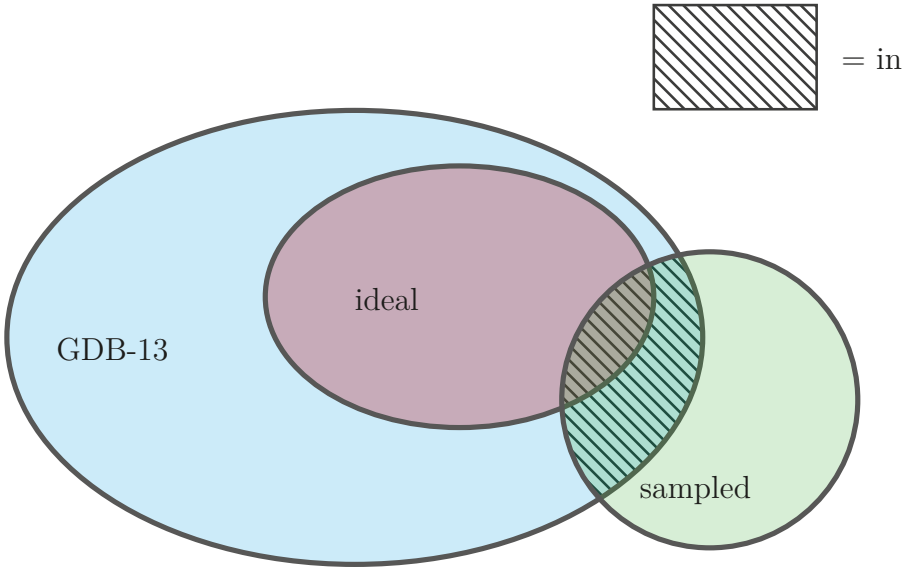


Figure 3.3: A depiction of GDB-13, samples from the ideal model, and samples from a generative model. Note that all samples from the ideal model are valid GDB-13 molecules, while the generative model can sample molecules outside of GDB-13. Also the striped intersection between *sampled* and GDB-13 is denoted *in*. Note also that the sets depicted are not to scale.

The relevance of this metric comes from the fact that it captures all three aspects of GDB-13 coverage. This can allow for models that score highly for different metrics to still be comparable. It is also one of the metrics used by Arús-Pous *et al.* [1].

3.5.3 Performance Considerations when Processing SMILES

In order to measure what percentage of GDB-13 a model covers, a fixed number of molecules are generated, and then the intersection of these molecules with the GDB-13 dataset is computed. Below is a discussion of the time complexity of this task, and how it was done in this thesis. Note that the generated molecules are always converted to SMILES strings, since these are information dense, and it is also the format used by GDB-13.

The intersection of two lists, both of length n, of SMILES can be calculated in several ways. The naïve approach would be to compare all elements in list one to all elements in list two, which can be accomplished in $\mathcal{O}(n^2)$ time.

A more refined approach would be to first sort these lists and find the unique elements, which takes $\mathcal{O}(n \log n)$ time. The lists can then be traversed only once in $\mathcal{O}(n)$ time resulting in a total time of $\mathcal{O}(n \log n)$. Even more useful is the fact that the GDB-13 set is fixed, and can thus be presorted.

The processing time can be decreased even further by separating the different SMILES into different partitions, one for each SMILES length, and then comparing the partitions one by one. This is possible since two SMILES can only be equal if

they are of equal length. If the number of partitions is m, and all partitions are of equal size, this results in a time complexity of

$$m \times \mathcal{O}\left(\frac{n}{m} \log \frac{n}{m}\right) < \mathcal{O}(n \log n).$$

The partitioning of SMILES also enables easy data parallellization, which is another possible improvement, since several CPU cores were available.

The assumption made that the size of all partitions are equal is not valid, however. The bar plot in Figure 3.4 shows the distribution of SMILES lengths in GDB-13. Even though all partitions will not be of equal length it is apparent that several of the partitions around length 23 and many are similarly sized. Thus, it was still advantageous to split the data into partitions.

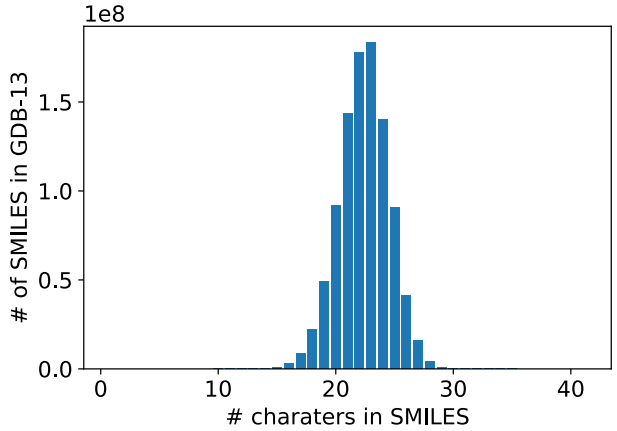


Figure 3.4: The distribution of the length of SMILES in GDB-13.

In order to sort, find unique, and calculate the intersection of the lists, Python was used. Within Python, the `numpy` package, which has well optimized functions `unique` and `intersection1d`, was used. Running on 4 CPUs, a sample of 2 billion SMILES took roughly 15 minutes to benchmark.

4
Results

In this chapter various results from training and benchmarking are presented. In addition to the results for the GNN models, we also present some metrics for string-based models. More specifically we present results from one string-based model trained on sets of canonical SMILES, and one trained on sets of randomized SMILES. These models are referred to as CanonCRNN and RandCRNN, where CRNN refers to *Character Recurrent Neural Network*. These RNN models are the ones used by Arús-Pous *et al.* [1].

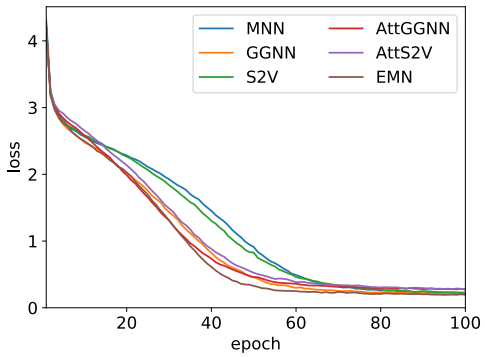
4.1 Model Convergence

In Figure 4.1 the training loss for the different GNN models and datasets is plotted. These figures show that all the different trained models on all the different datasets seem to converge.

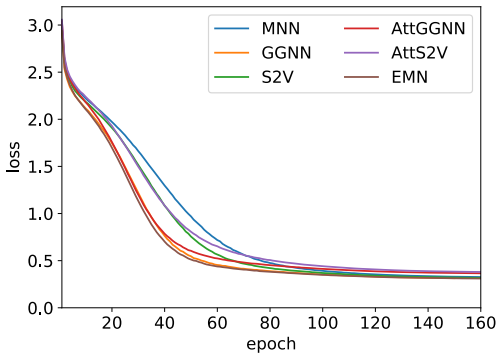
Though note that for the models trained on the 100K set the loss is not fully converged at the final epoch (see Subfigure 4.1(c)). At these epochs the models started to display over fitting at some metrics, so no further training was done. Also, the 1K and 10K models appear to converge around epoch 80.

4.2 Model Validation

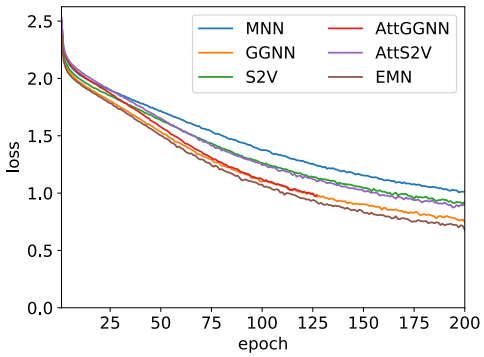
As a rough evaluation metric, the percentage of valid and unique molecules was used during training. These metrics were also calculated for the sampled molecules used for benchmarking. In Figures 4.2, 4.3, and 4.4, these metrics are presented for the different GNN models and GDB-13 training sets. The metrics were calculated on samples of 100,000 molecules every 10 epochs.



(a) Trained on GDB-13 1K subset

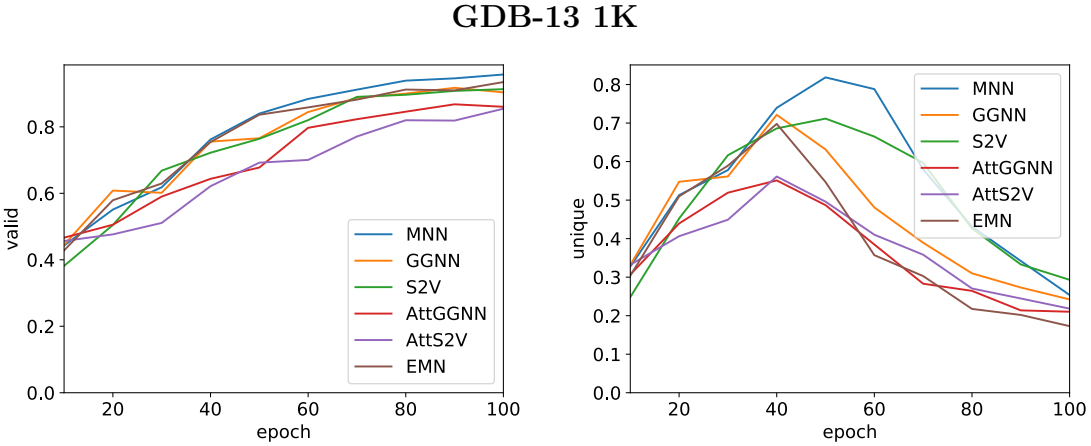


(b) Trained on GDB-13 10K subset



(c) Trained on GDB-13 100K subset

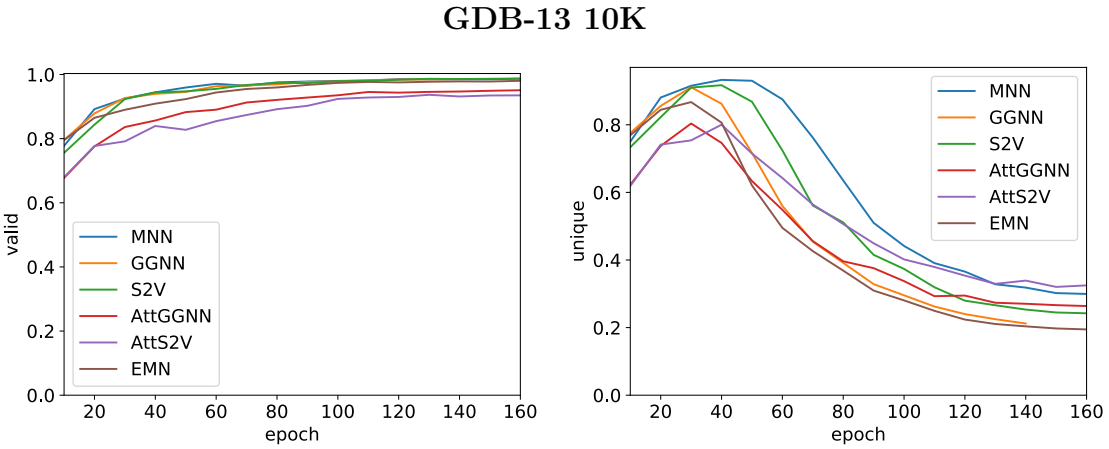
Figure 4.1: The training loss plotted for every training epoch. Each plot shows the loss for a different subset of GDB-13, and in each plot the loss is displayed for all trained GNN models.



(a) valid

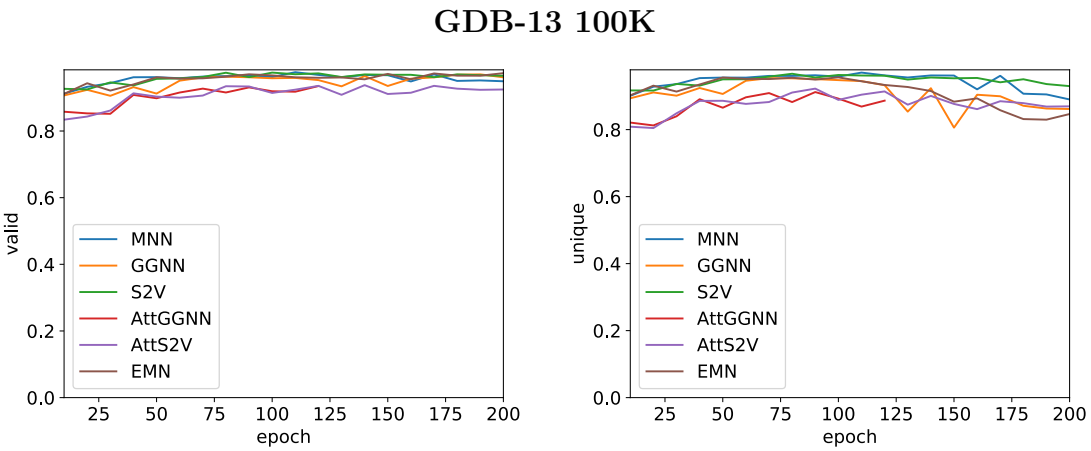
(b) unique

Figure 4.2: The graphs show the ratio of validity and uniqueness of molecules sampled by the different GNN models. The GNN models sampled 100,000 molecules every 10 epochs and were trained on the GDB-13 1K subset.



(a) valid **(b)** unique

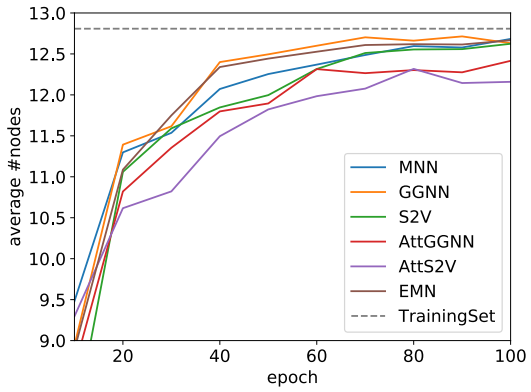
Figure 4.3: Analogous plots to what is presented in Figure 4.2 but for models trained on the GDB-13 10K subset.



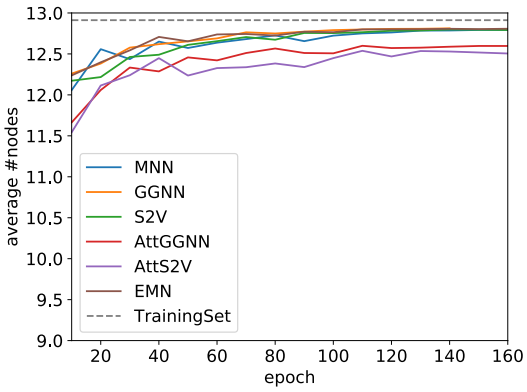
(a) valid **(b)** unique

Figure 4.4: Analogous plots to what is presented in Figure 4.2 but for models trained on the GDB-13 100K subset.

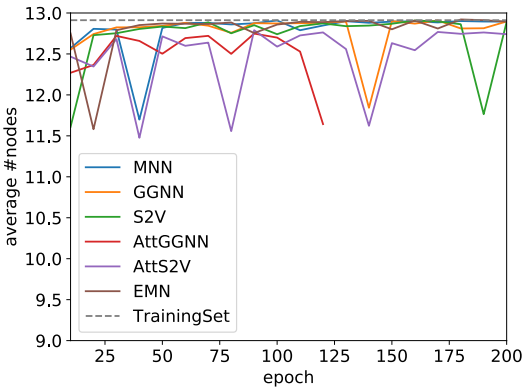
Another metric used during training to evaluate models was the average number of nodes of the sampled molecules. This metric was also calculated for the generated samples and the results are presented in Figure 4.5



(a) 1K training set



(b) 10K training set

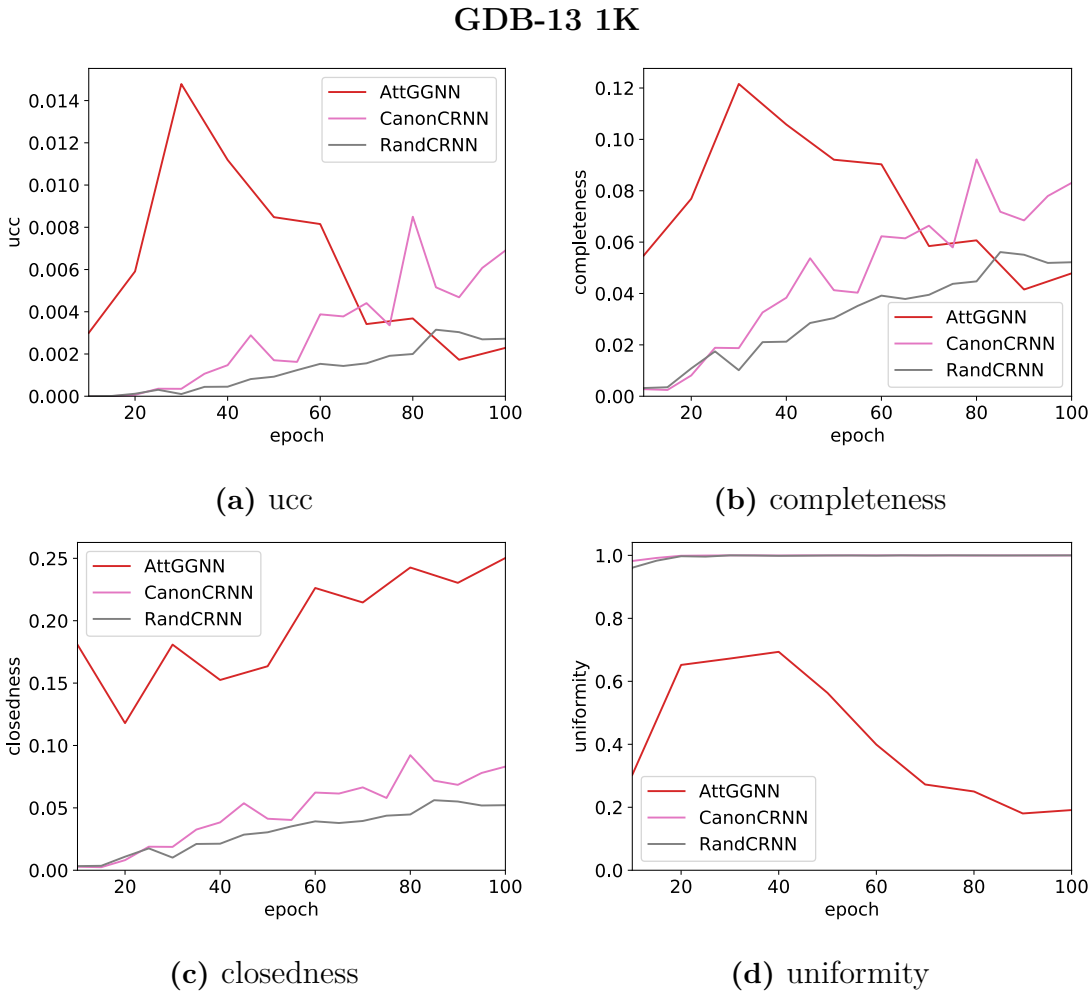


(c) 100K training set

Figure 4.5: The average number of nodes in each graph of the 100,000 molecules sampled every 10 epochs. Included is also the average number of nodes in the molecules in the training set used.

4.3 GDB-13 Benchmark

All models trained on the different subsets of GDB-13 were benchmarked using the presented GDB-13 metrics (see Section 3.5 for more details). The models were saved every 10 epochs and benchmarked with 100,000 generated molecules. In Figures 4.6, 4.7, and 4.8 the metrics of the best performing GNN model on each training set have been plotted. Drawn is also the performance of the string based models: CanonCRNN and RandCRNN. Best performing refers to the model that reached the highest ucc value. For the results of all GNN based models, see Appendix C

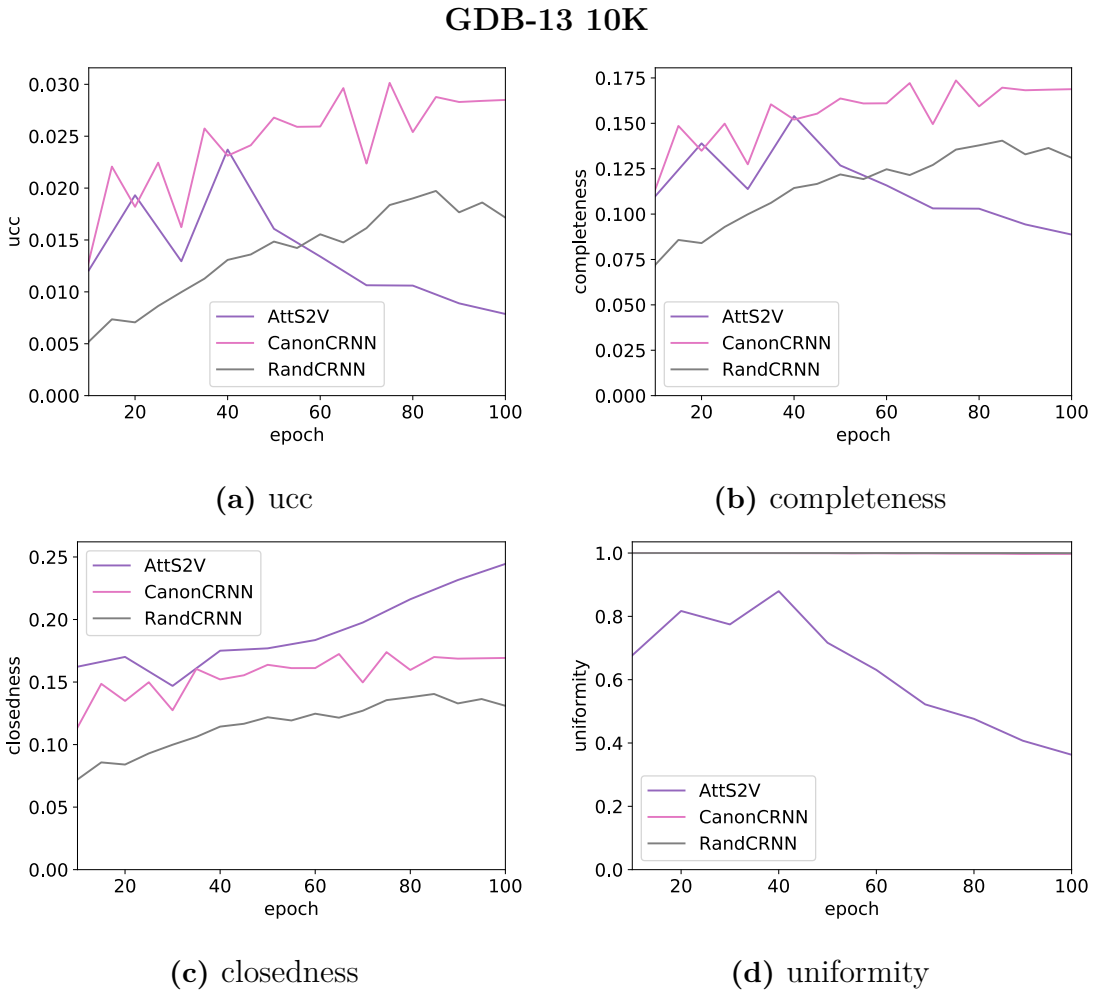


(a) ucc **(b)** completeness

(c) closedness **(d)** uniformity

Figure 4.6: GDB-13 benchmarks for some models trained on the GDB-13 1K subset. The only GNN model presented is the one that reached the highest ucc score, AttGNN, but the results for both CRNN models are plotted for comparison. Both GNN and CRNN models sampled 100,000 molecules.

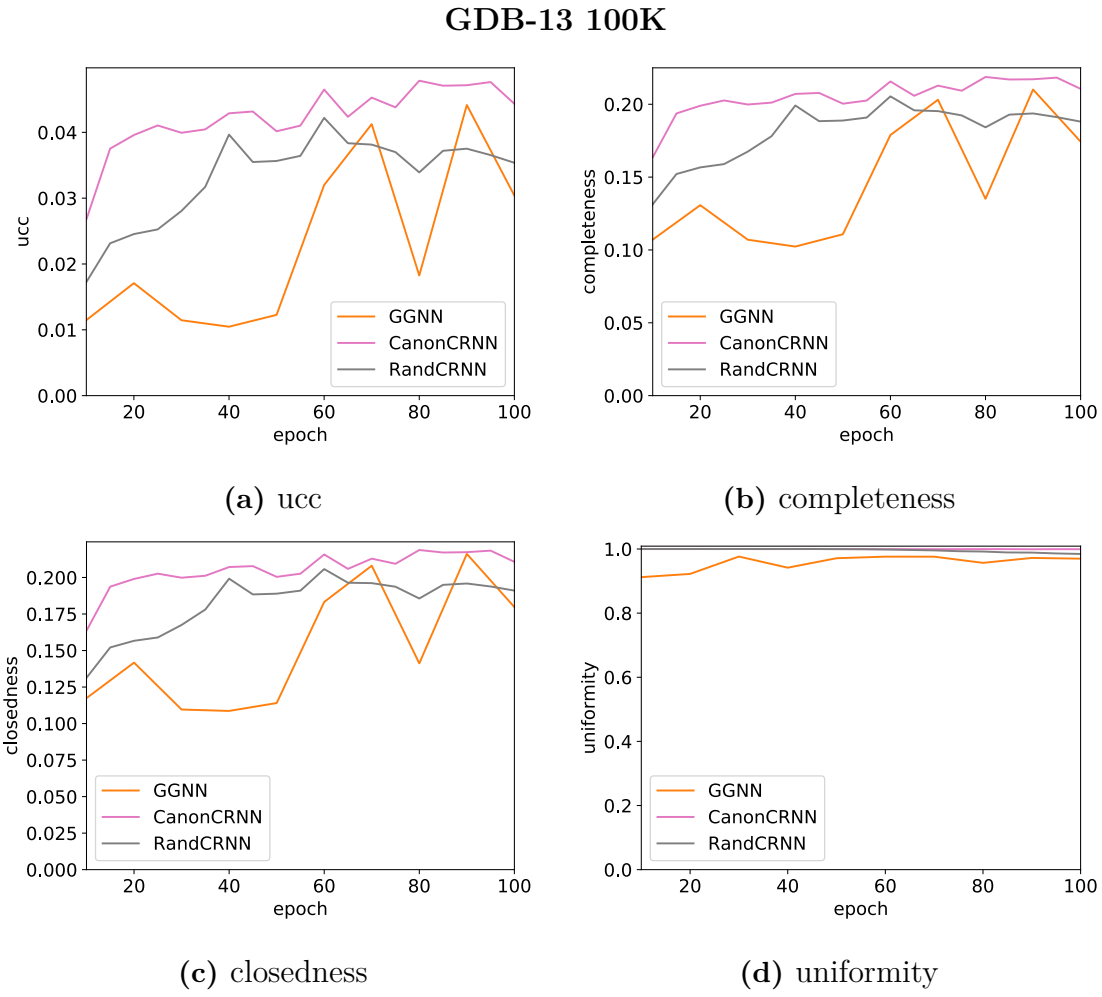
For all benchmarks the metrics for the best epoch were extracted, and the results can be seen in Table 4.1. Here, the epoch where the ucc was the highest was considered the best. All other measurements for this epoch are also printed, including validity and uniqueness.



(a) ucc **(b)** completeness

(c) closedness **(d)** uniformity

Figure 4.7: Here the results for the GDB-13 benchmarks analogous to those Figure 4.6 are presented, but for models trained on the GDB-13 10K subset. The best GNN model for this training set was AttS2V.



(a) ucc **(b)** completeness

(c) closedness **(d)** uniformity

Figure 4.8: Here the results for the GDB-13 benchmarks analogously to those in Figure 4.6 are presented, but for models trained on the GDB-13 100K subset. The best GNN model for this training set was GGNN.

4.3.1 Sampled Molecules

In Figures 4.9, 4.10, and 4.11, an assortment of sampled molecules have been displayed. These molecules are taken from the best epoch of the best performing GNN models for each GDB-13 dataset.

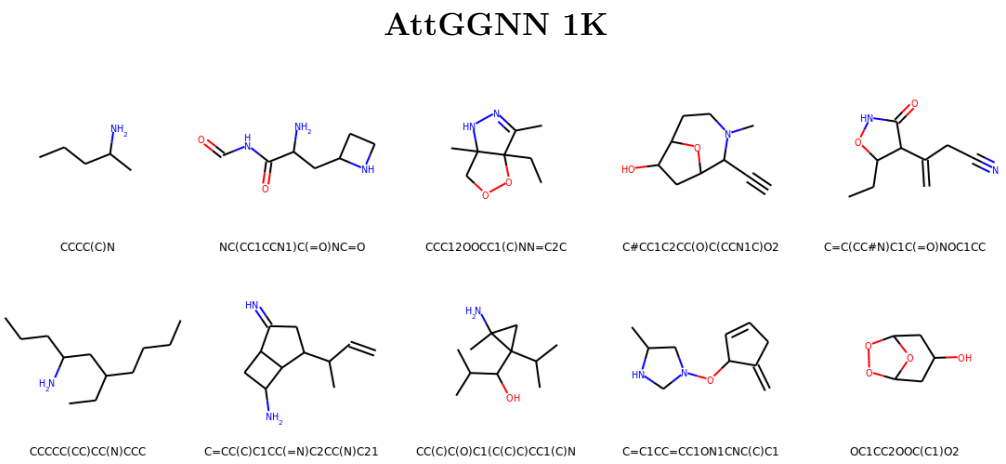


Figure 4.9: Molecules sampled with the AttGGNN model at epoch 30 trained on the 1K GDB-13 subset.

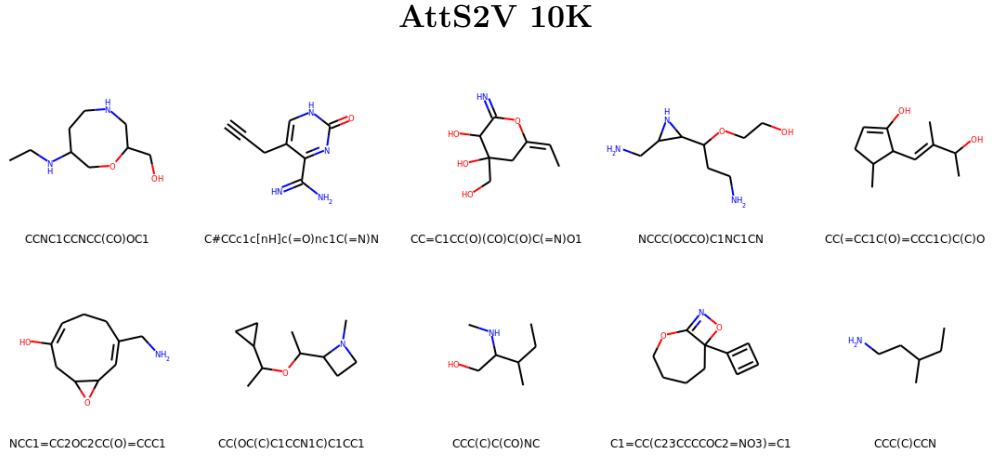


Figure 4.10: Molecules sampled with the AttS2V model at epoch 40 trained on the 1K GDB-13 subset.

4.4 Computational Performance

When designing, training, and optimizing the models, some choices were made in regards to performance. To make these choices some rough performance tests were carried out. Here, the results of these tests are presented.

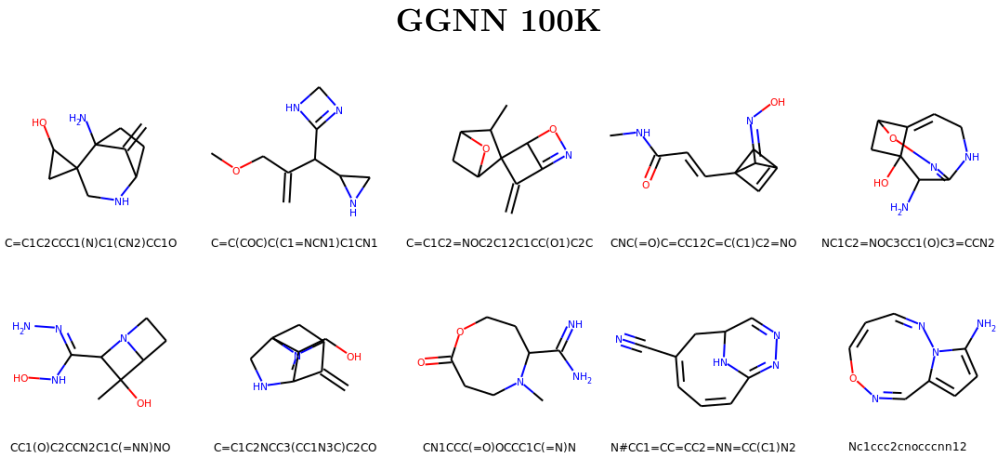


Figure 4.11: Molecules sampled with the GGNN model at epoch 90 trained on the 100K GDB-13 subset.

4.4.1 Memory usage

An obstacle when working with huge datasets such as GDB-13 is memory usage. In order to process and manage the large data set it was necessary to either have vast amounts of RAM available, or devise strategies to overcome memory limitations.

The entire GDB-13 dataset consists of SMILES and their respective IDs (integers), which were all stored in SMI files. This is a text file where each row contains one SMILES and its enumerated ID separated by a whitespace. The full dataset is roughly 31 GB, but if the ID:s, which are irrelevant for our purposes, are discarded, this shrinks to 22 GB. Such sizes are possible to handle and load into memory but can still become a problem.

When loading large data into memory great care was taken on the choice of datatypes. In Table 4.2 the memory usage for different data types is compared. This shows that binary strings stored in NumPy arrays are roughly as memory efficient as C. Thus this data type was used in loading and managing SMILES for the benchmark.

When calculating the chemical space coverage benchmarks the total memory usage was even greater. The entire GDB-13 dataset is roughly 22 GB in size; in addition to this, 100,000 generated SMILES, were loaded into memory. Furthermore, extra memory was required when sorting strings, finding unique strings, and the intersection of arrays.

This problem was averted by splitting the SMILES into partitions depending on their length, which made it possible to not load all at once. This approach is discussed in Section 3.5.3.

4.4.2 Computational Speed

Although widely used in the sciences, Python is a relatively slow language for working with large datasets and huge computations compared to languages like C or C++. Thus, some effort was put into investigating whether an alternative approach was warranted, such as using C. Writing code in C is much slower and more prone to errors but it is a much faster language.

In order to evaluate whether to use C some tests were done on reading and sorting SMILES. When reading roughly 1 million SMILES from disk, C outperforms Python by a factor of roughly 10, and for sorting the same amount of SMILES roughly 2. The benchmark computations in Python take roughly 20 minutes, however, compared to the training of models this time is negligible. Python was ultimately chosen as it is easier to work with.

dataset	model	ucc	compl.	closed.	unif.	valid	unique	epoch
100K	MNN	0.0321	0.179	0.184	0.976	0.968	0.962	90
	GGNN	0.0442	0.21	0.216	0.972	0.961	0.951	90
	S2V	0.0393	0.198	0.205	0.969	0.973	0.961	120
	AttGGNN	0.0407	0.202	0.218	0.926	0.927	0.909	70
	AttS2V	0.0388	0.197	**0.219**	0.898	0.938	0.901	140
	EMN	0.0424	0.206	0.211	0.976	0.963	0.955	80
	CanonCRNN	**0.0478**	**0.219**	**0.219**	**1.0**	**0.992**	**0.991**	80
	RandCRNN	0.0422	0.205	0.206	0.998	0.972	0.969	80
10K	MNN	0.018	0.134	0.164	0.821	**0.971**	0.875	60
	GGNN	0.0203	0.142	0.154	0.922	0.927	0.911	30
	S2V	0.0205	0.143	0.16	0.895	0.944	0.917	40
	AttGGNN	0.0227	0.151	**0.196**	0.767	0.857	0.746	40
	AttS2V	0.0237	0.154	0.175	0.88	0.839	0.8	40
	EMN	0.0188	0.137	0.155	0.883	0.865	0.844	20
	CanonCRNN	**0.0301**	**0.174**	0.174	0.998	0.964	**0.962**	75
	RandCRNN	0.0197	0.14	0.14	**1.0**	0.926	0.926	85
1K	MNN	0.0118	0.109	0.14	0.774	**0.884**	**0.788**	60
	GGNN	0.0073	0.0854	0.102	0.837	0.756	0.721	40
	S2V	0.00785	0.0886	0.123	0.719	0.764	0.712	50
	AttGGNN	**0.0148**	**0.122**	**0.181**	0.672	0.59	0.519	30
	AttS2V	0.011	0.105	0.146	0.717	0.621	0.561	40
	EMN	0.0106	0.103	0.131	0.785	0.754	0.698	40
	CanonCRNN	0.0085	0.0922	0.0922	0.999	0.762	0.762	80
	RandCRNN	0.00315	0.0561	0.0561	**1.0**	0.586	0.586	85

Table 4.1: The table shows the different benchmark metrics, validity, and uniqueness for the trained models. All metrics were calculated from a sample of 100,000 molecules, and the benchmark values were taken from the epoch with the highest ucc. The best model for each metric and dataset has been marked with bold face, and the row of the best GNN and CRNN model for each dataset has been highlighted with blue and red respectively.

Storage Method	Memory usage [MiB]
disk	4100
C array	4100
numpy binary string array	4200
python binary string list	14200

Table 4.2: Data usage for different data types in Python and C when loading a SMI file containing roughly 180 million smiles of length 23. The binary data indicated that the data was loaded and managed as binary strings instead of regular strings. The advantage of binary strings is that only 1 byte is required per character as opposed to the 4 bytes needed for unicode characters.

5
Discussion

In this chapter we start by discussing the results of the GDB-13 benchmark. Then we move on to future work based on this thesis. This includes both further benchmarking we did not have time for, as well as improvements that could be made to the models.

5.1 GDB-13 Benchmark

Here we discuss the results for the GDB-13 benchmark, including the viability of the GNN models as well as some discrepancies in the data.

5.1.1 Fluctuations

When trained on the 100K subset the GNN models displayed some aberrant behavior. As can be seen in Figure 4.5(c), the number of nodes of the sampled graphs made regular jumps to lower values. This behavior was present in all benchmarked GNN models. In addition to this, the benchmarking metrics for some of the GNN models were also behaving differently for the 100K training set, compared to the 1K and 10K. This is most apparent when looking at the full benchmarks, which can be seen in Appendix C. For the 1K, and 10K models, the different metrics seem to increase and then decrease, while for the 100K they reach a peak very early and then start to rapidly fluctuate. This might indicate that there are problems with the models, and the GNN results for the 100K set should be considered less telling.

It is difficult to assess why these problems arise. One explanation could be that the learning rate for the 100K set is not optimal, and that it converges too quickly. But, lowering the learning rate did not seems to affect the the benchmarks.

5.1.2 Viability of GNN Based Models

Looking at the results in Table 4.1 we can see that for the 100K and 10K training sets the CharRNN:s have the highest scores on the GDB-13 metrics. For the 1K set, it seems to perform significantly worse, but this is likely due to the models not being fully trained. In Figure 4.6, we can see that the ucc score for the CharRNN:s is still increasing, and thus they should have been trained longer. Retraining was unfortunately not possible due to lack of time. However, for the 10K and 100K metrics, we can see that some GNN based models are still performing rather well,

and the only score that is consistently lower for all GNN models is the uniformity.

It is also worth noting that the 100K CharRNN models are not fully optimized. The hyperparameters used were taken from the models optimized for the 1M subset, since the 100K subset was not used by Arús-Pous *et al.* [1]. On the other hand, the GNN models were not optimized for the 10K or 100K subset. All hyper parameter optimization was done on the 1K set, which was due to time constraints. Since the 1K set could be trained rather quickly it was deemed suitable for optimization.

From Table 4.1, we can also observe that the GNN based models trained on the 1K and 10K set seem to reach a maximum ucc score earlier than the CRNN:s during training. The models trained on the 100K reach this maximum later, but the score almost peaks during early epochs, as discussed in 5.1.1. That the ucc score of the GNN models peak earlier might indicate that the models are faster to train. On the other hand, the uniformity score of the CharRNN:s peaks during the first 5 epochs, even for the 1K set, as can be seen in Figure 4.6. Considering all of this it might be the case that the string based and GNN based models learn rather differently.

Another consideration is that the GNN models are much slower to train and sample. The fastest GNN model, MNN, is still slower than the CharRNN:s, and the slowest GNN model, EMN, is slower than MNN by at least an order of magnitude.

Taking all of this into consideration, GNN:s appear promising for molecular graph generation. Even if the models are not performing as well as the CharRNN:s they still have a lot of potential, and further research could make them very useful as a tool for *de novo* molecular design.

5.1.3 Randomized and Canonical CharRNN Performance

The results presented for the GDB-13 benchmarks show that the CharRNN trained on the canonical SMILES outperforms the one trained on randomized SMILES. At first glance this seems to contradict the results presented by Arús-Pous *et al.* [1] in the paper "Randomized SMILES Strings Improve the Quality of Molecular Generative Models". However, this is not the case. Arús-Pous *et al.* [1] utilized data augmentation by using several randomized SMILES representations for each SMILES sample, in contrast to this thesis where only one randomization was used. The authors only used non-augmented canonical SMILES when training models on the 1M subset. However, in this thesis models were only trained on the 1K, 10K, and 100K subsets. Thus, a direct comparison of these results is not indicative of the models' performances.

A possible explanation for the low performance of the RandCRNN is that training data in the three smaller sets is not sufficient for learning the randomized SMILES space. Randomized SMILES space is vastly greater than canonical space, and up towards 1M samples might be needed to learn the space distribution correctly.

5.1.4 Sample Size

When evaluating the models, a sample size of 100,000 molecules was used, which is much smaller than the sample size of 2 billion used in Arús-Pous *et al.* [1]. The reason for this was time constraints, since 100,000 molecules can be generated and evaluated in significantly less time than 2 billion. This decrease in sample size might have introduced noise in the metrics calculated, but 100,000 was the maximum number of structures that was feasible.

To evaluate the impact of the decreased sample size, the variance of the ideal model's coverage of GDB-13 can be analyzed. Let N_k denote the number of unique samples in GDB-13 when sampling k molecules form the ideal model. Then the expected number and variance of the samples are:

$$\mathbf{E}[N_{100K}] = 99,994.876$$

$$\mathrm{Var}(N_{100K}) = 5.123.$$

For a more thorough explanation of how these numbers were calculated, see Appendix D.

The variance in the number of unique molecules found by the ideal model is very small, which means that an ideal model should always generate almost the same amount of unique molecules when we sample 100K. So for the GBD-13 measurements we should see rather low fluctuations even when sampling so few molecules. This indicates that even for the non-ideal models, 100,000 should be sufficient.

5.2 Continuing Evaluations

Due to lack of time, only the 1K, 10K, and 100K GDB-13 training sets were considered, but in previous works by Arús-Pous *et al.* [7] and Arús-Pous *et al.* [1] models were also trained on a 1M subset. The models used in these publications were all CharRNN:s, which are much faster to train than the GNN-based models. Training GNN models on the 1M subset, partly for comparison reasons, is still desirable, and hopefully this can be done in future work.

In addition to this, the GNN-based models showed some unexpected behavior when trained on the largest dataset, 100K, as discussed in Section 5.1.1. Trying to evaluate why these fluctuations appeared and making changes to correct this behaviour is also of interest.

5.3 Model Improvements

Here we present some suggestions for how the GNN models in this thesis could be improved in future work.

5.3.1 Incorporating RNN:s

One possible improvement is the inclusion of RNN:s in the graph generative process. Currently the APD for a subgraph is generated using a combination of MLP:s, but some publications (see for example Li *et al.* [6]) report a performance improvement when instead using RNN:s.

Exchanging the MLP:s in the current models for RNN:s would also affect training data. Currently samples in the training data are made up of subgraphs and APD:s, but subsequent subgraphs in a molecules construction path are not necessarily stored in order. This would need to change if RNN:s were implemented, since each step in an RNN depends on iteration.

These changes were never implemented during the course of this thesis, simply due to lack of time. But implementing this should be straight forward, and is a possible improvement that is instead left for future work.

5.3.2 Further Hyperparameter Optimization

Some time was spent on optimizing the different GNN models used in this thesis, and in the end the parameters presented in Tables 3.1 (Repeated) and 3.2 (Repeated) were used; these tables have been repeated here for convenience. Note that these parameters were the same for all different GNN models, and were based on optimization done on the MNN model. A much more thorough hyperparameter optimization could have been performed, and each model could have been more finely tuned. This could be achieved by optimizing hyperparameters for each separate model and training set, as was done by Arús-Pous *et al.* [1]. This also includes learning rate and learning rate decay.

Hyperparameter	**Value**
Message Passes	3
Hidden Node Feature Size	100
MLP Depth	4
MLP Hidden Dimension	500

Table 3.1 (Repeated): The values of the common hyperparameters used in the different GNN networks.

For the optimization that was done the number of *valid* and *unique* structure generated were used to measure performance. But different metrics could also be used. For example, the GDB-13 benchmarking metrics could have been used as well, since optimizing these scores is of interest.

The optimization method used in this thesis was a grid search, but there exist other methods. An example is a random walk of hyperparameters, and another is Bayesian hyperparameter optimization. In addition to this, the models were only optimized

GDB-13 Subset	Initial LR	LRDF
1K	0.0001	0.9999
10K	0.0001	0.999995
100K	0.00005	0.999999995

Table 3.2 (Repeated): The different learning rates and learning rate decay factors used for the different GDB-13 subsets. Note that the same initial learning rate was used for all models on a given subset.

using the 1K training set, and were not fine-tuned for the larger subsets. Arús-Pous *et al.* [1] optimized the hyperparameters for their different models on each training set size. They also used optimized additional hyperparameters such as batch size. These further optimizations could also be applied to the GNN models and are left for future work.

5.3.3 Newer and Different Architectures

None of the models tested and used in this thesis fall under the category of GCN:s. GCN models have already seen applications in the domain of *de novo* molecular design, and also in other fields such as social networking, where significantly larger graphs are used [6, 11]. Since the MPNN models used in this thesis are quite slow, it could be of interest to see how well models based on GCN:s work, and if they improve computational speed while still scoring high on the GDB-13 metrics.

Since the field of GNN:s is still rather new there is also a constant flow of newer models, and during the course of this thesis work new models have been published in the literature. One such model is the SAMPN by Tang *et al.* [22], which is based on the D-MPNN model introduced by Yang *et al.* [17]. New architectures such as this one are also of interest for future work, since they could provide improvements to the generative models.

5.3.4 Canonical Representation and Data Augmentation

The graphs used for training were all randomly chosen, as was their construction orderings. This is in contrast to what was done with SMILES by Arús-Pous *et al.* [1], where they used a canonical representation, as well as several randomized ones. The latter case here is data augmentation. To make the comparison between SMILES-based and GNN-based models better a canonical construction ordering could have been used instead for the molecular graphs, where the graph traversal would have been based on the atom ordering of the SMILES.

As mentioned earlier in this thesis, data augmentation is rather easily achieved by using several construction orderings for each graph. This is also something that could be of interest to investigate, to see if it improves model performance, and to what degree.

6
Conclusion

This thesis presents an evaluation of deep graph based generative models for *de novo* molecular design. The evaluation is based on metrics that measure how well models sample a subset of chemical space, which consists of small molecules with 13 or fewer heavy atoms taken from the set C, N, O, S, Cl. Four metrics are used, one of which, called *ucc*, is the product of the other three.

The models are all based on graph neural networks, which are incorporated in the generative process. In total, 6 different models are trained and evaluated on three different training sets, all three being different subsets of the target chemical space subset GDB-13. The three training sets consist of 1K, 10K, and 100K molecules respectively. In addition to the 6 graph models, two string based models are also trained and evaluated for comparison reasons. These models are based on earlier work done by Arús-Pous *et al.* [1] and operate on string representation of molecules called SMILES. Time was also spent on optimizing the graph based model structures.

In addition the work done on models a data loading scheme for `pytorch` is introduced. This scheme loads data in large blocks from disk into memory, and the data block is then shuffled and separated into batches. This prevents slow random access reads from disk, while still maintaining the possibility for shuffling large datasets that don't fit into memory.

For the 100K set the best performing graph model scored a ucc of 0.442, compared to the best string model which scored 0.478. For the 10K set the best graph model scored 0.237, and the best string model 0.301. Lastly for the 1K set the best graph model scored 0.0148 and the best string model 0.0085. Different graph models scored the highest on all three subsets, while the same string model performed the best. Note that for the 1K set the string based models were not converged, and thus it is possible they could have scored higher.

The results indicate that graph based models can be viable for *de novo* molecular design. Even if the performance is worse than the string based models for two of the sets, they are still performing similarly. It is also worth noting that the graph based models were not as optimized as some of the string models, and another thing to consider is the computational complexity of the different models. All graph based models are slower than the string based models, both during training and generation.

There are also improvements that can be made for the graph based models. One

such is the inclusion of recurrent neural networks in the generation process. This has been done by other authors and has improved performance of their models, but it is left for future work.

Bibliography

[1] 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", Jul. 2019.

[2] H. Chen, O. Engkvist, Y. Wang, M. Olivecrona, and T. Blaschke, "The rise of deep learning in drug discovery", *Drug Discovery Today*, vol. 23, no. 6, pp. 1241 –1250, 2018, ISSN: 1359-6446.

[3] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains", in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 2, Jul. 2005, 729–734 vol. 2.

[4] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model", *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, Jan. 2009.

[5] Y. Li, O. Vinyals, C. Dyer, R. Pascanu, and P. W. Battaglia, "Learning deep generative models of graphs", *CoRR*, vol. abs/1803.03324, 2018.

[6] 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, ISSN: 1758-2946.

[7] J. Arús-Pous, T. Blaschke, S. Ulander, J.-L. Reymond, H. Chen, and O. Engkvist, "Exploring the gdb-13 chemical space using deep generative models", *Journal of Cheminformatics*, vol. 11, no. 1, p. 20, 2019, ISSN: 1758-2946.

[8] L. C. Blum and J.-L. Reymond, "970 million druglike small molecules for virtual screening in the chemical universe database gdb-13", *Journal of the American Chemical Society*, vol. 131, no. 25, pp. 8732–8733, 2009, PMID: 19505099.

[9] D. Weininger, "Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules", *Journal of Chemical Information and Computer Sciences*, vol. 28, no. 1, pp. 31–36, 1988.

[10] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks", *CoRR*, vol. abs/1901.00596, 2019.

[11] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks", *CoRR*, vol. abs/1609.02907, 2016.

[12] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation", *arXiv preprint arXiv:1406.1078*, 2014.

[13] N. Brown, M. Fiscato, M. H. Segler, and A. C. Vaucher, "Guacamol: Benchmarking models for de novo molecular design", *Journal of Chemical Information and Modeling*, vol. 59, no. 3, pp. 1096–1108, 2019.

[14] L. Ruddigkeit, R. van Deursen, L. C. Blum, and J.-L. Reymond, "Enumeration of 166 billion organic small molecules in the chemical universe database gdb-17", *Journal of Chemical Information and Modeling*, vol. 52, no. 11, pp. 2864–2875, 2012, PMID: 23088335.

[15] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow, "Gated graph sequence neural networks", in *Proceedings of ICLR'16*, Apr. 2016.

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

[17] K. Yang, K. Swanson, W. Jin, C. Coley, P. Eiden, H. Gao, A. Guzman-Perez, T. Hopper, B. Kelley, M. Mathea, A. Palmer, V. Settels, T. Jaakkola, K. Jensen, and R. Barzilay, "Analyzing learned molecular representations for property prediction", *Journal of Chemical Information and Modeling*, vol. 59, no. 8, pp. 3370–3388, 2019, PMID: 31361484.

[18] (2020). Hdf5 support page, [Online]. Available: `https://portal.hdfgroup.org/display/HDF5/HDF5` (visited on 06/01/2020).

[19] A. Collette and contributors. (2014). H5py documentation, [Online]. Available: `http://docs.h5py.org/en/stable/` (visited on 06/01/2020).

[20] T. contributors. (2019). Pytorch documentation, [Online]. Available: `https://pytorch.org/docs/stable/index.html` (visited on 06/01/2020).

[21] 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.

[22] B. Tang, S. T. Kramer, M. Fang, Y. Qiu, Z. Wu, and D. Xu, "A self-attention based message passing neural network for predicting molecular lipophilicity and aqueous solubility", *Journal of Cheminformatics*, vol. 12, no. 1, p. 15, 2020, ISSN: 1758-2946.

[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*, ser. ICML'17, Sydney, NSW, Australia: JMLR.org, 2017, pp. 1263–1272.

[24] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. F. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, Ç. Gülçehre, H. F. Song, A. J. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. R. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks", *CoRR*, vol. abs/1806.01261, 2018.

[25] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?", *CoRR*, vol. abs/1810.00826, 2018.

[26] B. Weisfeiler and A. A. Lehman, "A reduction of a graph to a canonical form and an algebra arising during this reduction", *Nauchno-Technicheskaya Informatsia*, vol. 2, no. 9, pp. 12–16, 1968.

A

GNN Publications

Here we give a brief overview of some of the different GNN:s published in the literature. For a more in depth survey of GNN:s, see Wu *et al.* [10].

A.1 Original Graph Neural Networks

Gori *et al.* [3] introduces the novel neural network architecture GNN, which is an extension of the RNN. The GNN is essentially a mapping from a graph to a real valued vectors $y_v \in \mathbb{R}^m$, one for each node v. This mapping is defined by the propagation scheme in Equation (A.1).

$$\begin{aligned} X_v &= \{x_w : w \in N(v)\} \\ H_v^t &= \{h_w^t : w \in N(v)\} \\ h_v^{t+1} &= f_v(x_v, X_v, H_v^t) \\ y_v &= g_v(x_v, h_v) \end{aligned} \tag{A.1}$$

Here, X_v and H_v^t are the set of node and hidden features in the neighborhood of v. The mapping from graph to latent vector is then carried out by applying the function f_v in Equation (A.1) repeatedly until the hidden states h_v^t converge to fixed points. The authors argue that, by the fixed-point theorem, if f is defined in such a way that it is a contraction mapping, then applying f will always converge to a unique point. The authors present two implementations of GNN:s and evaluate them on a set of small benchmarks, where both GNN:s show promising results.

Scarselli *et al.* [4] continue the work on GNN:s introduced by Gori *et al.* [3]. The main contribution the authors provide is a rigorous theoretical foundation for the GNN model. They prove that if the model is differentiable w.r.t. its parameters, then back propagation is possible.

A.2 Graph Convolutional Networks

Kipf and Welling [11] present an efficient approximation of the graph neural network; they call their networks *graph convolutional networks* (GCN:s). Because their approach scales linearly with the number of graph edges, and is applicable on very large graphs.

They define the t:th layer of their graph convolutional step as

$$H^{t+1} = \sigma\left(\hat{A}H^tW^t\right), \tag{A.2}$$

where H are hidden graph states, W learnable matrices, and σ a non-linear activation function. The matrix $\hat{A}$ is a normalized adjacency matrix defined by

$$\tilde{A} = A + I, \tag{A.3}$$

$$\tilde{D} = \sum_j \tilde{A}_{ij}, \tag{A.4}$$

$$\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}. \tag{A.5}$$

The paper presents a rigorous mathematical motivation for these approximations.

The main goal of the paper is to apply GCN:s on sparsely-labeled graph-structured data using a semi-supervised learning approach. The model is applied on several large graphs ($|V|, |E| \in [1000, 1000000]$) for benchmarking and compared to previously published approaches.

Although the GCN:s have shown promise and scale well to large data, they were not specifically used in this thesis. It is also worth noting that GCN:s are a subset of the MPNN architecture discussed in Section A.3.

A.3 Message Passing Neural Networks

Gilmer *et al.* [23] introduce a graph neural network framework they call *message passing neural network* (MPNN). MPNN:s operate by using messages which are passed from a node to its neighbors and then aggregated into a hidden node state. This message passing step is applied a fixed number of times. This allows information to propagate through the graph via connected nodes, as at each step information is only transmitted to neighboring nodes.

At a given message passing step t and for a given node v, the messages m_v^{t+1} are computed using the message passing function, M_t, and the hidden states h_v^{t+1} are computed with the aggregation function, U_t. These are defined as can be seen in Equations (A.6) and (A.7).

$$m_v^{t+1} = \sum_{w\in,N(v)} M_t(h_v^t, h_w^t, e_{vw}), \tag{A.6}$$

$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1}). \tag{A.7}$$

The authors show that several previously published architectures [15] can be represented using this framework. They also benchmark several models on property prediction tasks, and thus identify the best model. They found that the best model is a combination of a previously published model and a novel message passing function.

Li *et al.* [15] build upon general GNN:s by using a *gated recurrent unit* (GRU) as a part of their node propagation model. They call this architecture *Gated Graph Neural Network* (GG-NN) but also apply it for generating sequential data from graphs, which they call *Gated Graph Sequence Neural Network*. Another significant difference is that the propagation, or message passing, is not repeated until convergence, as is the case for the general GNN. Instead, a fixed number of message passing steps are used. In the context of MPNN:s, the GG-NN can be described by the message passing and update functions in Equations (A.8) and (A.9). A more advanced readout function is also applied; see Appendix B for an in-depth explanation.

$$M_t = A_{e_{vw}} h_w^t \tag{A.8}$$

$$U_t = \mathrm{GRU}(h_v^t, m_v^{t+1}) \tag{A.9}$$

The main goal of the authors is to use graph networks for program verification. Consequently, the benchmarks and tests they present are for generating sequential output, and they find the model performs well compared to previous work.

Inspired by the MPNN model framework, Yang *et al.* [17] present *directed message passing neural networks* (D-MPNN), a similar architecture which also utilizes message passing. Instead of using hidden node states in the message passing and aggregation functions, hidden directed edge states are used. This means that a message is passed to an edge from all edges directed at its base node. The message passing and aggregation schemes are explicitely written in Equations (A.10) and (A.11).

$$m_{vw}^{t+1} = \sum_{k \in N(v) \setminus w} M_t(x_v, x_k, h_{kv}^t) \tag{A.10}$$

$$h_{vw}^{t+1} = U_t(h_v^t, m_v^{t+1}) \tag{A.11}$$

Since messages in this network are passed along the direction of edges, messages are not propagated back and forth between nodes. The authors argue that this reduces the amount of noisy data generated during the message passing step. A concrete implementation of the framework is also presented and evaluated. The results show that this model performs on par with, or better than, existing networks at a large number of the benchmarks.

Battaglia *et al.* [24] present a framework they call the *graph network* (GN), which is a further generalization of the MPNN framework. In the GN framework, the message passing and aggregation steps are each modeled using something they refer to as GN blocks. Each block passes information according to the scheme in Equations (A.12) and (A.13).

$$\mathbf{e}_k' = \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}) \tag{A.12}$$

$$h_{vw}^{t+1} = \phi^e(h_{vw}, h_v^t, h_w^t, h) \tag{A.13}$$

Several GN blocks can be put together to form more complicated propagation schemes, allowing for the easy creation of more advanced models.

Xu *et al.* [25] present a variant of the MPNN architecture, which is defined by Equations (A.14) and (A.15).

$$m_v^{t+1} = F_t\left(\left\{h_w^t : w \in N(v)\right\}\right) \tag{A.14}$$

$$h_v^{t+1} = U_t\left(h_v^t, m_v^{t+1}\right), \tag{A.15}$$

Here, U_t is required to be an injective function. The authors also specify an implementation of this scheme which they call the *graph isomorphism network* (GIN). The implementations of the message passing and aggregations functions for GIN are presented in Equations (A.16) and (A.17) .

$$m_v^{t+1} = \sum_{w \in N(v)} h_w^t, \tag{A.16}$$

$$h_v^{t+1} = \mathrm{MLP}^t\left(\left(1 + \epsilon^t\right) h_v^t + m_v^{t+1}\right), \tag{A.17}$$

ϵ is a learnable parameter.

The authors also lay a theoretical foundation for understanding why some GNN:s are more expressive than others. They do this by comparing GNN:s to the Weisfeiler-Lehman test [26], which is a traditional algorithm for testing graph isomorphism. Furthermore, the authors provide proofs that the scheme in Equations (A.16) and (A.17) can provide a test as strong as the WL-test under the right conditions. They further prove that their own propagation and aggregation functions satisfy these conditions.

B

GNN Architectures

In this appendix, the 6 different GNN architectures used in this thesis are presented in detail. For all models, the message passing step as well as readout function is presented, as these set apart the different models.

$$
\begin{aligned}
h_v^0 &= x_v \\
m_v^{t+1} &= \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw}) \\
h_v^{t+1} &= U_t(h_v^t, m_v^{t+1}) \\
g &= R(\{h_v : v \in G\})
\end{aligned}
$$

Some of the models also introduce an additional attention step to the message passing. The message passing for these models is defined by:

$$
\begin{aligned}
\bar{m}_v^{t+1} &= \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw}) \\
b_v^{t+1} &= \sum_{w \in N(v)} B_t(h_v^t, h_w^t, e_{vw}) \\
m_v^{t+1} &= \texttt{ATTENTION}(\bar{m}_v^{t+1}, b_v^{t+1}).
\end{aligned}
$$

MNN

The MNN model uses a very simple linear M_t. The update function consists of a `GRU` and the aggregation function is simply a sum.

$$m_v^{t+1} = \sum_{w_\in N(v)} W_{e_{vw}} h_w^t \tag{B.1}$$

$$h_v^{t+1} = \texttt{GRU}(m_v^{t+1}, h_v^t) \tag{B.2}$$

$$g = \sum_{v \in G} h_v^T \tag{B.3}$$

GGNN

In the GGNN model, the linear layer in the MNN has been replaced my an `MLP`, and the readout function is the one presented by Li *et al.* [15]. Note here that $\odot$ refers

to elementwise multiplication, also known as the Hadamard product.

$$m_v^{t+1} = \sum_{w \in N(v)} \texttt{MLP}_{e_{vw}}(h_w^t)\ e_{vw} \tag{B.4}$$

$$h_v^{t+1} = \texttt{GRU}(m_v^{t+1}, h_v^t) \tag{B.5}$$

$$g = \sum_{v \in G} \texttt{MLP}(h_v^t) \odot \tanh\left(\texttt{MLP}\left([h_v^t, h_v^0]\right)\right) \tag{B.6}$$

S2V

The S2V network uses a very similar architecture to the GGNN, but a more advanced readout function.

$$m_v^{t+1} = \sum_{w \in N(v)} \texttt{MLP}(e_{vw})\ h_w^t \tag{B.7}$$

$$h_v^{t+1} = \texttt{GRU}(m_v^{t+1}, h_v^t) \tag{B.8}$$

The readout function here utilizes an attention mechanism as presented by Vaswani *et al.* [16], and an `RNN`. T here refers to the number of message passes.

$$\begin{aligned} a &= \texttt{embedding}([h_v^0, h_v^T]) \\ r^0, q^0, c^0 &= [0, \ldots, 0] \\ q^{t+1}, c^{t+1} &= \texttt{LSTM}(r^t, q^t, c^t) \\ b^{t+1} &= q^{t+1} \cdot a \\ r^{t+1} &= \texttt{ATTENTION}^t(a, b^{t+1}) \\ g &= [q^T, r^T] \end{aligned} \tag{B.9}$$

AttGGNN

The AttGGNN builds upon the GGNN and introduces an attention mechanism as an extra step between M_t and U_t. The additional step is defined as

$$b_v^{t+1} = \sum_{w \in N(v)} \texttt{MLP}(h_w^t) e_{vw} \tag{B.10}$$

$$\tag{B.11}$$

The rest of the model is identical to GGNN.

AttS2V

As with AttGGNN, the AttS2V model builds upon S2V by adding an attention layer. The additional message passing step is defined as:

$$b_v^{t+1} = \sum_{w \in N(v)} \texttt{MLP}([e_{vw}, h_w^t]) \tag{B.12}$$

$$\tag{B.13}$$

EMN

The final model is the EMN. This model passes messages between edges instead of nodes, and utilizes a GRU and attention. The scheme is very similar to that of the other GNN models but here indexing is done on edges vw instead of of nodes v.

$$h_{vw}^0 = [0, \ldots, 0] \tag{B.14}$$

$$\hat{e}_{vw} = \sum_{w \in N(v)} \texttt{tanh}(\texttt{MLP}([x_v, x_w, e_{vw}])) \tag{B.15}$$

$$\bar{m}_{vw}^{t+1} = \texttt{MLP}([\hat{e}_{vw}, h_{vw}^t]) \tag{B.16}$$

$$b_{vw}^{t+1} = \sum_{w \in N(v)} [\texttt{MLP}(\hat{e}_{vw}), \texttt{MLP}(h_{vw}^t)] \tag{B.17}$$

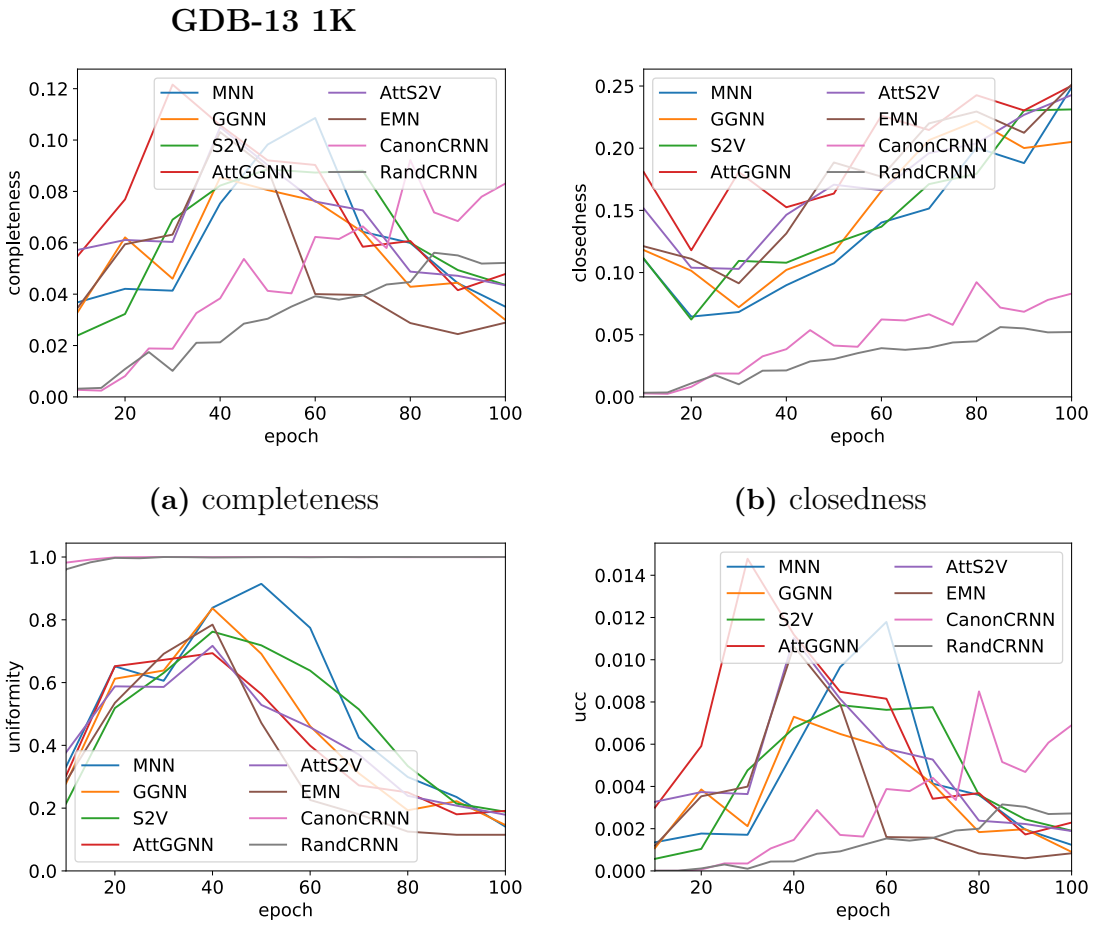
$$m_{vw}^{t+1} = \texttt{ATTENTION}^t(\bar{m}_{vw}^{t+1}, b_{vw}^{t+1} \tag{B.18}$$

$$h_{vw}^{t+1} = \texttt{GRU}(m_{vw}^{t+1}) \tag{B.19}$$

C

Complete GDB-13 Benchmark

Here, the complete GDB-13 benchmarking runs for all GNN and CharRNN models are presented. The results for models trained on the 1K, 10K, and 100K sets can be seen in Figures C.1, C.2, and C.3, respectively. Included in the figures are the four metrics: ucc, uniformity, closedness, and completeness.



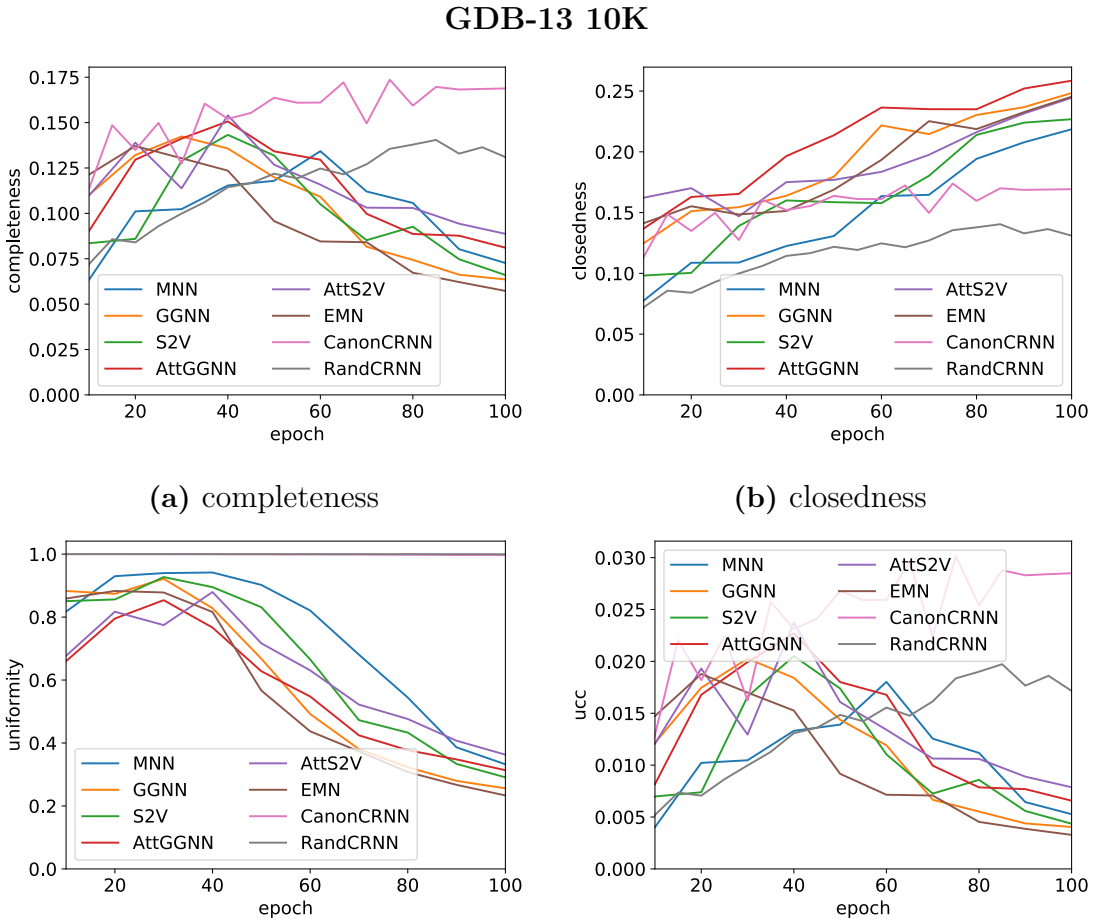
(a) completeness

(b) closedness

(c) uniformity

(d) ucc

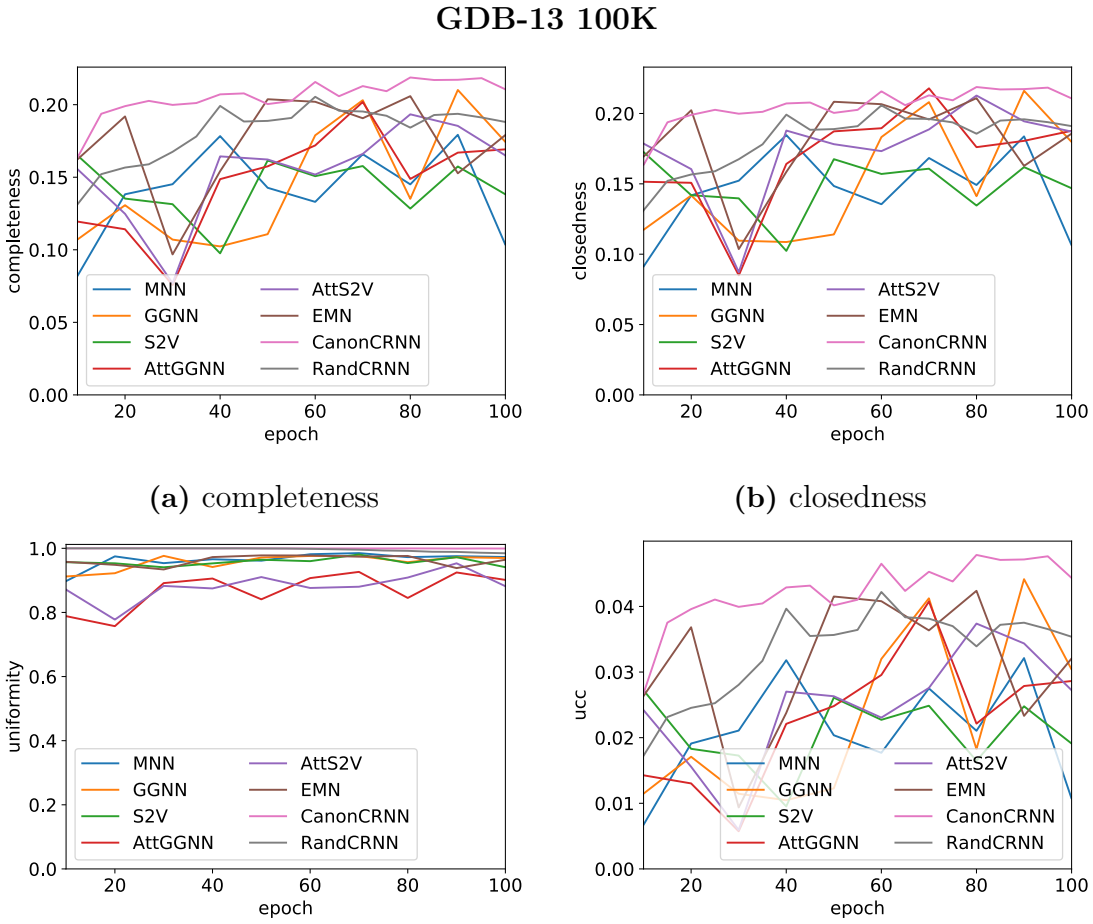
Figure C.1: The full benchmark of all GNN and CRNN models for the 1K training set.



(a) completeness **(b)** closedness

(c) uniformity **(d)** ucc

Figure C.2: The full benchmark of all GNN and CRNN models for the 10K training set.



(a) completeness

(b) closedness

(c) uniformity

(d) ucc

Figure C.3: The full benchmark of all GNN and CRNN models for the 100K training set.

D

Ideal Generative Model

The ideal model samples molecules without bias, i.e, uniformly. This can be used to derive an expression for the number of distinct molecules sampled. Denote the number of distinct molecules by N, the sample size by k, and the number of molecules in the samples space by n. Note here that N is a stochastic variable, while n and k are constants. The expected number of samples can be seen in Equation (D.1)[1].

$$\mathbf{E}[N] = n\left[1 - \left(1 - \frac{1}{n}\right)^k\right] \tag{D.1}$$

Of interest is also the variance of the variable N. This can be seen in Equation (D.2)[2].

$$\mathrm{Var}(N) = n\left(1 - \frac{1}{n}\right)^k + n^2\left(1 - \frac{1}{n}\right)\left(1 - \frac{2}{n}\right)^k - n^2\left(1 - \frac{1}{n}\right)^{2k} \tag{D.2}$$

Note here that for the GDB-13 set $n \approx 10^9$ and $k = 10^5$. For such large n the expression in Equation (D.2) can be hard to compute since $1 - \frac{1}{n}$ is quite close to 1. This problem was solved by utilizing Python and the package `mpmath`, which is a package for arbitrary-precision floating-point arithmetic. Using only the built in Python arithmetic is not sufficient, since this causes significant floating-point errors.

In addition to these two measurements, it is also of interest to look at the *Coupon collector's problem*, i.e., what is the expected number of samples before the entire target space has been sampled. Denote by K this number of samples, then the expected value of K can be computed using the expression in Equation (D.3) [3].

$$\mathbf{E}[K] = n\left(\frac{1}{1} + \frac{1}{2} + \ldots\frac{1}{n}\right) = nH_n \tag{D.3}$$

Above, H_n is the *harmonic series*, which can be easily approximated numerically.

[1]For a discussion on the derivation of Equations (D.1) and (D.2), see `https://math.stackexchange.com/questions/32800/probability-distribution-of-coverage-of-a-set-after-x-independently-randomly`.

[2]See Footnote 1.

[3]For an explanation of the expression in Equation (D.3), see `https://en.wikipedia.org/wiki/Coupon_collector%27s_problem`.