CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Differentiable Neural Computers for in silico molecular design

Benchmarks of architectures in generative modeling of molecules

Master's thesis in Computer science and engineering

OLEKSII PRYKHODKO
SIMON JOHANSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Differentiable Neural Computers for in silico molecular design

Benchmarks of architectures in generative modeling of molecules

OLEKSII PRYKHODKO
SIMON JOHANSSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Differentiable Neural Computers for in silico molecular design
Benchmarks of architectures in generative modeling of molecules
OLEKSII PRYKHODKO
SIMON JOHANSSON

Advisor: Hongming Chen, AstraZeneca
Supervisor: Graham Kemp, Department of Computer Science & Engineering
Examiner: Alexander Schliep, Department of Computer Science & Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Differentiable Neural Computers for in silico molecular design
Benchmarks of architectures in generative modeling of molecules
OLEKSII PRYKHODKO
SIMON JOHANSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In the area of *in silico* drug discovery, deep learning has grown immensely as a field of research. Recurrent neural networks (RNN) is one of the most common approaches used for generative modeling, but recently the differentiable neural computer (DNC) has been shown to give considerable improvement over the RNN for modeling of sequential data. In this thesis, a DNC has been implemented as an extension to REINVENT, an RNN based model that has already been successfully shown to generate molecules with high validity. The model was benchmarked on its capacity to learn the SMILES language on the GDB-13 and ChEMBL datasets. The DNC shows some improvement on all tests conducted at the cost of greatly increased computational time and memory consumption, which puts its practical use into question. This project also gives some insight into the effect of the DNC hyperparameters for the task of generative modeling of molecules.

Keywords:Computer science, machine learning, recurrent neural networks, GRU, LSTM, differentiable neural computer, engineering, project, thesis.

Acknowledgements

We want to thank Hongming Chen for his supervision on-site at AstraZeneca and Graham Kemp for supervising us in all academical matters. We thank Josep Arus-Pous for the many discussions regarding his earlier work and the many insights on the GDB-13 chemical database. We thank Amol Thakkar for taking an advisory role in everything since our first day of the project. We thank the Molecular AI team for welcoming us and giving us feedback on the design team meetings. We thank AstraZeneca for providing the resources required to perform this project. Finally, we want to thank Ola Engkvist believing in us and enabling us to work on this project.

Oleksii Prykhodko and Simon Johansson, Gothenburg, July 2019

Contents

List of Figures

List of Tables

1
Introduction

Drug discovery is a vital part of improving the quality of life by treating diseases, whether it be by improving upon existing drugs or by targeting new ailments.

The process of discovering a new drug is long and expensive; sometimes taking decades and costing billions of dollars [1]. The chemical space is big; it is estimated that the space of drug-like molecules measures between 10^{60} to 10^{100} compounds [2]. During drug development, there are many stages of testing that filter away inadequate chemical candidates. There are several approaches to estimate the effect of a drug prior to clinical trials, such as *in vitro* (in laboratories), *in vivo* (in living organisms) and *in silico* (tests using computers). Classical in silico testing requires the expertise of medicinal chemists or sets of rules for possible chemical building blocks, which severely limits the chemical space that can be explored. Using deep learning methods to train models provides a way to understand what makes a molecule drug-like, shortens the time required for a development of a new drug and makes the whole process cheaper [3].

There are multiple ways to represent a molecule in a computer e.g. by the means of a graph or a list of it's atoms and bonds, but these are costly in terms of space as the number of molecules grow. A string-based notation named **S**implified **M**olecular **I**dentification and **L**ine **E**ntry **S**ystem (SMILES) [4] was developed to create a compact, yet unambiguous way to describe chemical structures (see 2.5.2). Since this notation can be viewed as a language, with the rules of chemistry as its grammar, text-processing methods have successfully been used in molecular design for drug discovery [3]. The most common way to perform language recognition in deep learning is using recurrent neural networks (RNNs). RNNs usually use either Long short-term memory (LSTM) units or gated recurrent units (GRUs). Their performance against each other has previously been studied by several parties without reaching a consensus. They were found to be similar in output results for polyphonic music modeling and speech signal modeling, but that GRUs were faster both in CPU time per iteration and in convergence [5]. Another study on language recognition found that LSTM units had a strictly higher potential to learn [6]. Currently there is no comparative study (to our knowledge) conducted specifically for the purpose of generative models for molecules.

GRU units have successfully been used in a recent architecture called REINVENT. It was shown to able to generate a high level of valid chemical structures during training and, using reinforcement learning, was able to fine-tune its generation against desired biological targets [7].

The Differentiable Neural Computer (DNC) was developed to be an extension to the regular RNN architectures. With the RNN as controller, the output was written

and stored in an external memory with trainable weightings. By reading from this memory, the DNC could remember information over a longer period of time and vastly outperform regular LSTMs. In a demonstration with graph traversal experiments, LSTMs reached an average accuracy of 37% after two million training examples, while the DNCs had up to 98.8% accuracy while requiring half the training samples [8].

The first reported architecture using DNCs in molecular design was called the reinforced adversarial neural computer (RANC) [2] which was compared against its predecessor ORGANIC [9], both of them based on the generative adversarial network (GAN). The RANC architecture was shown to produce longer sequences more similar to that of the training set, which was attributed to having a DNC generator. It was followed by the same research team by the Adversarial Threshold Neural Computer (ATNC), which again used a DNC [10].

At the moment of writing, there has not been an empirical study comparing DNCs against regular RNN architectures for the molecular generation task. In this study, we implement the DNC with both GRU and LSTM controllers and benchmark the performance on molecular generation. We also highlight the influence of the DNC hyperparameters on generation.

1.1 Potential technical issues

Recurrent neural networks are subject to numeric instability. This can occur both by a vanishing gradient when the weights in the network are small (see 2.2.2) or by gradient explosion if they are too big (which is most commonly mitigated by gradient clipping, see 2.1.4.3).

While the different models of RNN are suited for sequences, it has been noted that for complex molecules containing several rings, or a high amount of heavy atoms, the information passed in the network too far back gets blurred and the probability of the model making a mistake such as "forgetting to close a ring" increases (see sections 2.5.2). This results in a 'tail' on the molecule which would generally be a R' chain of 5 heavy atoms. This can be confused with the regular mode collapse of GAN architectures [11].

Finally, there's an inherent weakness in the computer representations of molecules; there is no guarantee that the feasibility in the generator logic and the real world chemistry feasibility overlap, and as such the assessment of model output isn't as simple as looking at the training or validation loss (see section 2.1). This is related to problems mentioned above; a model can have a low loss, still generate useless molecules.

1.2 Project Goal

This project goal is to implement the DNC model and answer the following questions

- Can the external memory of the DNC assist in the generation of 'complex' molecules?

- Can we show by benchmarking that the DNC outperforms regular RNN for the task of molecule generation?
- How sensitive is the DNC to changes in the memory settings?
- Is a benchmark result translatable into better quality of generation i.e complexity or diversity?

1.3 Delimitations

The REINVENT architecture has two stages, training a *prior* RNN and fine-tuning towards specific targets (see 3.4,2.3). During training of the prior, we noted that the DNC architecture required too much time to train per model (see 4.1.1), such that an extensive optimization of parameters for fine-tuning would be infeasible, especially if the prior were to be trained on the complete ChEMBL dataset (see 3.3.1,4.2.1). We concluded then that a benchmark on the reinforcement learning would be out of scope. We are instead limiting our experiments to tests for our different models' capacity to learn during the training of the prior. We are choosing to still mention the reinforcement learning procedure, because it is a central part of REINVENT and its usage in practice.

1.4 Thesis outline

This report is written with the intention to be understandable by engineers of multiple fields and as such has a theory chapter (Chapter 2) that acts as a primer to bridge the knowledge gap that some might have. Sections 2.1-2.3 address the machine learning and neural network terminology that is used for the project. Section 2.4 give the mathematical theory that is used in the project. Section 2.5 describes the different ways that a molecule is represented as data for the computer.

The method chapter first introduces tools used for the project in sections 3.1.1 to 3.1.3. It further describes the data used in 3.3. Section 3.4 describes the REINVENT model and how it is used. Sections 3.5 to 3.7 outlines the methodology and experiments performed for this project. Chapter 4 shows the project results, separated by the datasets used and Chapter 5 attempts to answer the questions listed in 1.2. A summary of the conclusions is given in Chapter 6.

2

Theory

The field of Chemoinformatics lies in the intersection between chemistry, mathematics and computer science. As such, its users and contributors will inevitably come from a vast variety of backgrounds with different focuses from these three fundamental branches of science. This theoretical background aims to define the tools used for the project. It will provide an introduction to the areas of machine learning used, followed by a mathematical background to functions used in these algorithms with brief motivations. Lastly, this chapter will describe how chemical molecules and features can be represented in computer language in an efficient way to make the machine learning possible.

2.1 General Machine Learning Terms and Tools

In the field of data science and machine learning there are many tools and concepts that are used in several models. The goal of this section is to act as a short primer to features used in the project that is not necessarily part of a particular architecture, as well as to give definitions to terms that will be commonly used during the methodology of the report.

Machine learning (ML) provides systems the ability to automatically learn and improve from experience without being explicitly programmed. An ML system is usually called a model and at its core lies a learning algorithm. The learning algorithm has a loss function and an optimization algorithm. The former defines how good the model performed on the data and the later tries to make model learn the function that minimizes the error. Learning this function can be seen as an optimization problem. Therefore many concepts seen in optimization theory are also inherited by machine learning methods. The structure of the section will refer first to the terms used in optimization, followed by the machine learning counterparts in brackets.

2.1.1 Terminology

The process of training a machine learning model consists of repeatedly showing it the same data, and in each iteration step (an epoch) the trainable parameters (known as weights) are optimized using optimization algorithms. The most common and widely known optimization algorithm is a gradient descent.

Gradient descent is a first order optimization algorithm, which means that it uses first order derivative. There are also second order optimization algorithms that use

second order derivative. But since they are costly to compute and thus not widely used, they will not be described here.

The error between the predicted solution and the true solution (cost) is differentiated with respects to the weights in the model and then changed by a defined step length (learning rate) in the direction of negative gradient. The model is considered converged if the cost has reached a minimum. By selecting the cost function such that it is continuous and letting the learning rate decay over time it is ensured that the model always converges and stops learning, although the conditions are not sufficient to guarantee that the minimum is global.

2.1.2 Training, Validation, Test and overfitting

In machine learning the data is commonly divided into three subsets: the training set, the validation set and the test set. All three sets are constructed using real data or ground truth.

The training data is the only set shown to the model during the learning phase of the model construction and it is for this data that the model is optimized. The data is characterised by features, which are specific properties of the data that the model can learn and base its classification upon. If the model learns *too much* of the features of the training data it may predict the output perfectly for the training data and be much less accurate on the test set. This is called *overfitting*, because the weights of the model have become too biased towards the training set.

The validation set is typically data of the same type as the training data, but contains no intersecting elements that the model has already seen. This data will be shown to the model at each epoch, but without any optimization performed towards lowering the cost. A typical training period will show the cost decrease for both data sets as the prediction accuracy increases until the validation set reaches a minimum, while the training cost continues to decrease. It is at this point where the training should be stopped.

The test set is then a third separate set showing data not previously seen to verify how strong the model predictions actually are. In general, one wants to avoid overfitting since a model without generalization would typically be a more computationally expensive solution to a regular optimization problem. On the contrary, there also exist *underfitting*, where the model fails to learn the features of training set, which can be likened to making random guesses.

2.1.3 Batch

Data sets are usually too big for the model to input at once due to technical limitations such as memory. To handle such big data, the data is split into smaller subsets and processed by the model in some arbitrary (usually completely random) order. The update to the training is then performed after each batch instead of the complete epoch (also to preserve memory). The batch size can thus impact the way the model learns as smaller batches leans towards each individual object in the batch having a larger influence on the gradient for that training step, which makes the model more volatile and unstable. Generally, the batch size tends to scale linearly

with memory consumption but not in computational time per batch. If there is a time constraint to the model one generally wants to keep the batches as large as possible.

2.1.4 Regularization

Regularization is a collective name for methods that aims to reduce the variance of the impact of different inputs to the model. This is primarily used to reduce overfitting, but can also have other uses (see 3.5.4). The idea is that if a model has less extreme gradients during its optimization steps, the training can be stopped at a much more precise step. Regularization methods will by extension have the drawback of an increased number of required epochs to reach a converged state.

2.1.4.1 Batch shuffling

During training, the updating (and the end result) of the gradients by stochastic gradient decent (as is common) will differ between the two possible ways one can order the input of any number of batches ≥ 2. It follows that the training result of one epoch has a variance that is dependent not only on the way batches was ordered but also in what configuration the elements was assigned to their batch (consider the case of two batches with training performed between them, one can easily see that there can exist a batch item x_i that with two different sets of weights will be an outlier if loaded into the first batch, but not if loaded into the second). To reduce the influence of this factor, another randomization is introduced between each epoch, where the training set is split into a new random configuration every time.

2.1.4.2 Batch/layer normalization

For some models, there is a high variation between individual training runs on the same training data with different convergence behaviours. The normalization approach attempts to make the training more robust to outliers. This method can be applied at any point before the input passes through a layer of weights. Consider an input x_t with corresponding mean μ_t and variance σ_t^2. Then the normalization takes the form

$$Norm(x_t) = g * \frac{x_t - \mu_t}{\sqrt{\sigma_t^2 - \varepsilon}} + b, \tag{2.1}$$

where the gain, g and bias b are trainable weights and ε an arbitrarily small non-zero element. The operator $*$ denotes element-wise multiplication. This has the drawback of increased computational time and memory consumption [12].

2.1.4.3 Gradient clipping

Sometimes during training if the learning rate is too high or if, due to some numerical instability, the gradients on any number of weights during the update is unreasonably large, the model can experience an explosion in its values towards $\pm\infty$. This effect can be dampened by clipping the gradients in one of two ways. The first is a simple cutoff method using a defined interval $[x_{min}, x_{max}]$, where if any weight has

a gradient outside of the interval, this value is ignored and the defined threshold is used instead. The second method is to normalize the vector of gradient elements such that the vector norm equals a defined value x, this is referred to as gradient scaling or norm gradient scaling and is the method used in this project.

2.1.4.4 Dropout

Dropout is a method developed to reduce overfitting [13] by making the observation that in many cases during training, some weights are activated more than others and will thus end up the only trained weights in a given network. The goal is to force all weights in the network to have to activate in a more normalized way. This is accomplished by introducing a random chance for each unit in the network to be ignored in a forward pass. The idea is that this prevents the model from relying on any given subset of units in the network, giving a more uniform training. The dropout method works in the following way. A dropout counterpart of the feed-forward operation (see eq. 2.4) is given by

$$\begin{aligned} r_i^{(l)} &\sim \text{Bernoulli}(p), \\ \tilde{y}^{(l)} &= r^{(l)} * y^{(l)}, \\ z_i^{(l+1)} &= w_i^{(l+1)} \tilde{y}^l + b_i^{(l+1)}, and \\ y_i^{(l+1)} &= f(z_i^{(l+1)}). \end{aligned} \tag{2.2}$$

The retention rate p corresponds to the probability that unit i will be used in any given forward pass.

2.2 Neural networks

A Neural network (NN) is a combination of mathematical functions and operations that are used to learn and predict a pattern. NNs are inspired by the neurons that constitute a brain. Like in the biological brain, neurons in an artificial NN are connected and the output of each individual neuron is used to form a final output. Neural networks are found in a wide variety of applications: from image comprehension to stock value prediction. The most common neural network types are feed-forward networks, convolutional networks, recurrent neural networks and assemblies. Nevertheless, they have similar parts like activation functions and use similar algorithms to be trained and make predictions [14].

2.2.1 Deep neural networks

Feedforward neural networks, or multilayer perceptrons, are the essential deep learning models. The goal of a feedforward network is to approximate some function f. For example, a classifier $y = f(x)$ maps an input x to a category y. A feedforward network defines a mapping $y = f(x, \theta)$ and learns the value of the parameters θ that result in the best function approximation. These models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y [14].

2.2.1.1 Perceptron

The neuron in the deep feedforward network is sometimes called a Perceptron and was first described in 1958 [15]. The equation for the perceptron is

$$\begin{aligned} z &= wx + b, \\ y &= f(z), \end{aligned} \tag{2.3}$$

where x is the input data, w and b are the *weight* and the *bias* respectively, f is the activation function and y is the output. Having weight and bias as parameters, the first step of a perceptron can be viewed as an arbitrary linear function with slope w and intercept b. The activation function scales the output to $[0, 1]$ which tells how strong a neuron reacted to the input.

A simple feed-forward neural network is illustrated in Figure 2.1

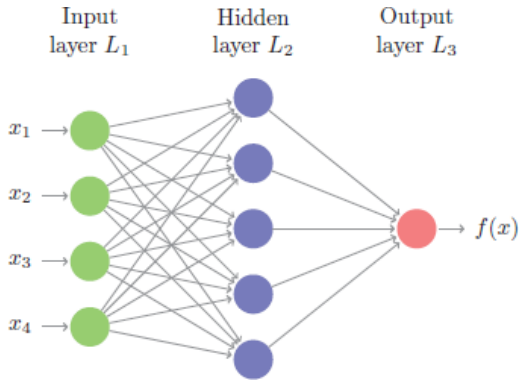

Figure 2.1: A simple feed forward neural network [16]

The input layer L_1 reads the data from a user provided input. Each neuron in the hidden layer L_2 is a perceptron that processes the input data and passes information to the next layer L_3. Output layer may consist of one or several neurons depending on the task and it's role is to project the result.

2.2.1.2 Forward propagation

Passing the input through each layer and projecting an output is called a forward pass. A feed-forward pass through layer $l \in [1, ...L-1]$ of input $y^{(l)}$ with hidden unit denoted by i will give output $y^{(l)}$ according to the recursive scheme

$$\begin{aligned} z_i^{(l+1)} &= W_i^{(l+1)} y^l + b_i^{(l+1)}, \\ y_i^{(l+1)} &= f(z_i^{(l+1)}), \end{aligned} \tag{2.4}$$

where f is the *activation function* of the layer and $W^{(l)}$, $b^{(l)}$ the weights and biases, respectively. Here, $y^{(0)}$ is the original input x and $y^{(L)}$ the final output.

2.2.1.3 Backward propagation

After forward propagation the network outputs a prediction. At this step the accuracy of the network should be assessed and its performance improved. To do that, a loss function needs to be defined e.g. mean square error or cross entropy. Loss gives a numerical representation of the error the model has. The objective is to minimize that error. This is done by calculating the gradient for every unit in the network and updating the weights in the network. The following formula illustrates how the weights are updated

$$W^l_{i,t+1} \quad = \quad W^l_{i,t} - \lambda(\frac{\partial Loss}{\partial W^l_{i,t}}), \tag{2.5}$$

where W^l_i is the weight of the neuron i in the layer l and λ is the learning rate. Learning rate is generally in the range $(0, 1]$.

2.2.2 Recurrent neural networks

Recurrent neural networks (RNNs) is a special type of NNs designed to work with sequential data. RNNs are commonly used in e.g. speech recognition, machine translation or music composition. Sequential data is represented by sequence of tokens x_t where $t = 0, 1, \ldots, T$. Interpretation of the token x_t depends on the token x_{t-1}. To deal with that, RNNs propagate through time.

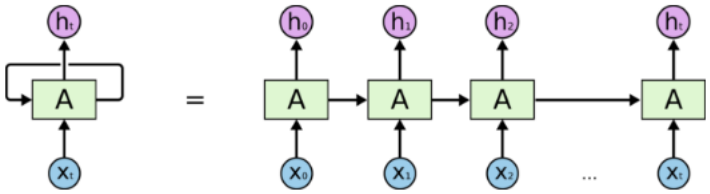

Figure 2.2: General scheme of a recurrent neural network [17]

In Figure 2.2 A is a RNN, x_t is the sequence of tokens and h_t is the output of the network produced at each time step for $t = 0, 1, \ldots T$. The output of the network on the time step t is called a hidden state. The hidden state from the previous time step h_{t-1} and data for the current time step x_t together form an input to the network (x_t, h_{t-1}) at the time step t. The final prediction of the network is the output at the last time step. It can be said that a recurrent neural network has a memory, which is represented by the hidden state. The hidden state contains information about the part of the sequence seen before the current timestep to affect processing of every next token. This allows recurrent neural networks to perform the tasks that feed forward neural network struggle with [14].

A basic recurrent neural network uses a fully connected neural network to perform propagation through time. The total error for the RNN model is the sum of errors at each time step. Calculating the error and updating weights for the RNN model is called Back propagation. A problem that can occur when deriving the gradient of the error for the time step i as a function of the output from the time step $i-1$ is that

the weights can be very small i.e. $\ll 1$. Due to the nature of recurrence, this weight is multiplied for each time step, resulting in values close to 0. That prevents model from training and is called a vanishing gradient problem. Approaches that attempt to solve this are Long Short Term Memory (LSTM) [18] and Gated Recurrent Units (GRU) [19].

2.2.2.1 LSTM cell

The vanishing gradient problem is partially solved by the LSTM architecture.

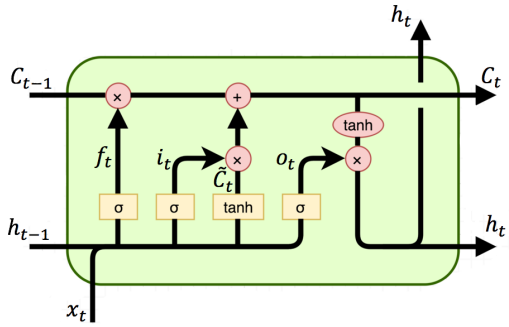

Figure 2.3: Schematic representation of the LSTM cell [17]

As can be observed in 2.3, the LSTM cell uses the following set of equations:

$$\begin{aligned} i_t &= \sigma\Big(x_t U^i + h_{t-1} W^i\Big), \\ f_t &= \sigma\Big(x_t U^f + h_{t-1} W^f\Big), \\ o_t &= \sigma\Big(x_t U^o + h_{t-1} W^o\Big), \\ \tilde{C}_t &= \tanh\Big(x_t U^g + h_{t-1} W^g\Big), \\ C_t &= \sigma\Big(f_t * C_{t-1} + i_t * \tilde{C}_t\Big), \\ h_t &= \tanh(C_t) * o_t, \end{aligned} \tag{2.6}$$

where bias terms are omitted. Here i, f, o are called the input, forget and output gates, U and W denote set of weights for the input and the hidden state respectively. They are called gates because the sigmoid function squashes the values of these vectors between 0 and 1, and by multiplying them elementwise with another vector you define how big the affect of that vector on the output and the internal memory of the cell will be. The input gate defines how much the newly computed state will alter the internal memory of the cell. The forget gate defines how important the previous state at the current timestep is. Finally, the output gate defines how much of the internal state you want to expose to the external network (higher layers and the next time step). All the gates have the same dimensions d_h, the size of the hidden state.

$\tilde{C}$ is a "candidate" hidden state that is computed based on the current input and the previous hidden state. C is the internal memory of the unit. It is a combination of the previous memory, multiplied by the forget gate, and the newly computed hidden state, multiplied by the input gate. Thus, intuitively it is a combination of

how we want to combine previous memory and the new input. We could choose to ignore the old memory completely (forget gate all 0's) or ignore the newly computed state completely (input gate all 0's), but most likely we want something in between these two extremes. The output hidden state, h_t, is computed by multiplying the memory with the output gate. Not all of the internal memory might be relevant to the hidden state used by other units in the network.

2.2.2.2 GRU cell

Gated recurrent units may be called a simplified version of the LSTM as they have fewer parameters but follow a similar principle.

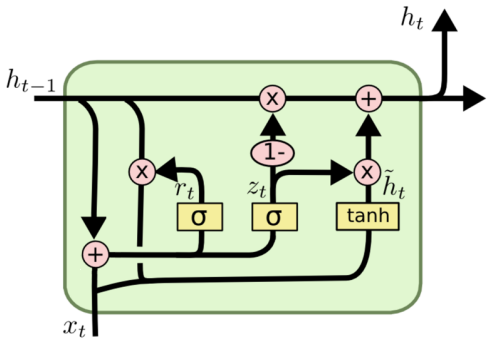

Figure 2.4: Schematic representation of the GRU cell [17]

For GRU, the hidden state h_t is computed as

$$\begin{aligned} z_t &= \sigma\Big(x_t U^z + h_{t-1} W^z\Big), \\ r_t &= \sigma\Big(x_t U^r + h_{t-1} W^r\Big), \\ \tilde{h}_t &= \tanh\Big(x_t U^h + (r_t * h_{t-1}) W^h\Big), \textit{and} \\ h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t. \end{aligned} \tag{2.7}$$

Here r is a reset gate, and z is an update gate. Intuitively, the reset gate determines how to combine the new input with the previous memory, and the update gate defines how much of the previous memory to keep around. If the reset gate is set to a vector of 1's and update gate to 0's, it will simplify to the base RNN model.

2.2.3 Differentiable neural computer

In 2014 a principally new model called Neural Turing Machine (NTM) was introduced [20]. The model was inspired by the architecture of the modern computer. It was shown, that a neural turing machine is turing complete, meaning that it can solve any algorithmic task. Moreover, it outperformed the LSTM models on a range of tasks that require the model to store information for a long period of time and recall it with precision [20].

In 2016 an extension of the NTM was released. The model is called a Differentiable Neural computer or DNC for short [8]. The model is similar to the NTM,

but it has an improved mechanism of managing the memory. An overview of the schematic is shown in Figure 2.5.

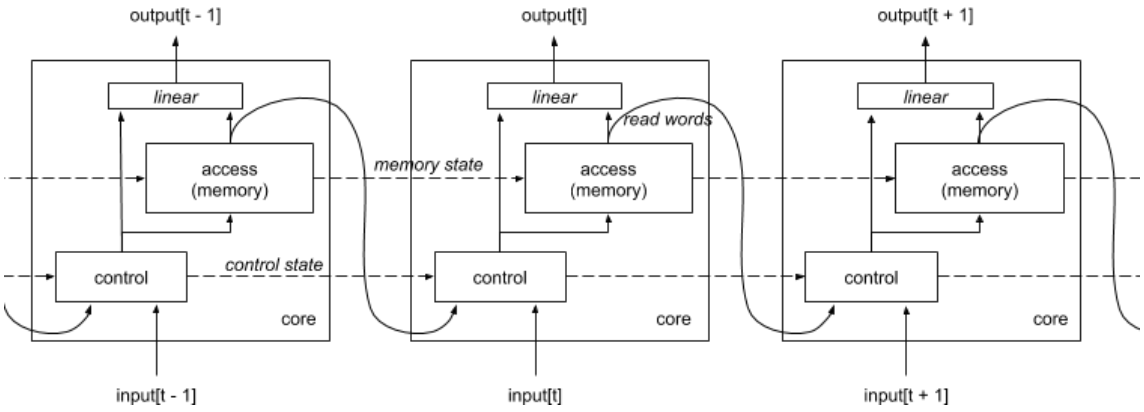

Figure 2.5: Schematic representation of the differential neural computer [21]. Note that the controller can be any regular RNN unit, like GRU or LSTM.

A differential neural computer has two essential parts: a controller and a memory with read and write heads. The controller has the same role as a processor in a computer to perform operations on the input data and store the results into the memory. Storing is done using the write heads. Later whenever a DNC needs results of previous computations it can load them from the memory using read heads. The model learns what to write and erase from the memory as well as what and when to read from it.

2.2.3.1 Controller

The controller is responsible for taking an input, reading from and writing to the memory, and producing an output that can be interpreted as an answer. Any neural network can be defined as a controller for a DNC. A controller can perform several operations on the memory. At every time step, it chooses whether to write to the memory or not. If it chooses to write, it can choose to store information at a new, unused location or at a location that already contains information. This allows the controller to update what is stored at the location. If all the locations in the memory are used up, the controller can decide to free the locations, much like how a computer can reallocate memory that is no longer needed. When the controller does write, it sends a vector of information to the chosen location in the memory. Every time information is written, the locations are connected by links of association, which represent the order in which information was stored. As an input the controller receives a concatenated vector $\chi_t = [x_t; r^1_{t-1}; \ldots; r^R_{t-1}]$, where t is the current time step, R is the number of read heads, x is the user input. After the input is processed by the controller, an output vector υ_t and an interface vector ξ_t are emitted;

$$\begin{aligned} \upsilon_t &= W_y[h^l_t; \ldots; h^L_t], \\ \xi_t &= W_\xi[h^l_t; \ldots; h^L_t], \end{aligned} \tag{2.8}$$

where h^l_t is a hidden state of a layer l and W_y,W_ξ are learnable weighting matrices.

The interface vector is used to parameterize memory interactions. It is subdivided as follows

$$\xi_t = [k_t^{1,R}; \ldots; k_t^{r,R}, \tilde{\beta}_t 1, R; \ldots; \tilde{\beta}_t^{r,R}; k_t^w; \tilde{\beta}_t^w; \tilde{e}_t; \upsilon_t; \tilde{f}_t^1; \ldots; \tilde{f}_t^R; \tilde{g}_t^a; \tilde{g}_t^w; \tilde{\pi}_t^1; \ldots; \tilde{\pi}_t^R] \tag{2.9}$$

where $k_t^{r,i}, \beta_t^{r,i}, k_t^w, \beta_t^w, e_t, \upsilon_t, f_t^i, g_t^a, g_t^w, \pi_t^i$ are read keys, read strength, write keys, write strength, erase vector, write vector, free gates, allocate gate, write gate and read modes respectively.

The final output of the model is defined as a concatenation of the current read vectors through the $RW \times Y$ weighting matrix W_r

$$y_t = \upsilon_t + W_r[r_t^1; \ldots; r_t^R;]. \tag{2.10}$$

2.2.3.2 Memory

A memory of DNC is represented by an $N \times M$ matrix where a value may be stored for a later retrieval. N denotes the number of memory cells and M their respective length. The memory can be searched based on the content of each location. Writing into the memory creates associative temporal links that can be followed forward and backward to recall information written in a sequence or in reverse. The read out information can be used to produce answers to questions or actions to take in an environment. Together, these operations give DNCs the ability to make choices about how they allocate memory, store information in the memory, and easily find it once there. An illustration of how the memory works can be seen in Figure 2.6.

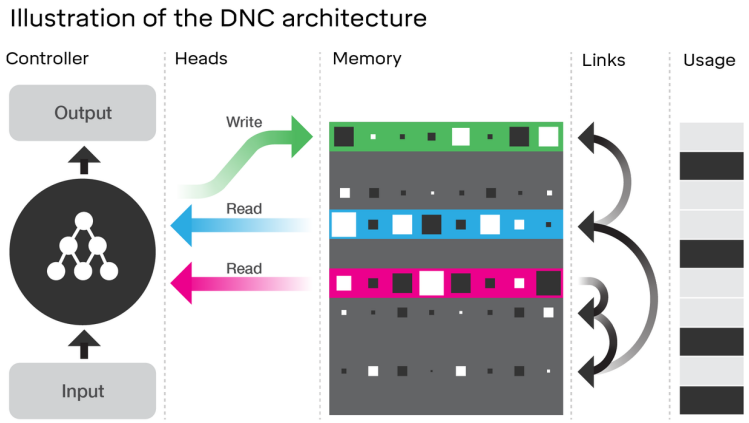

Figure 2.6: Schematic representation of the memory of the differential neural computer. [8]

How DNC reads and writes from the memory is going to be explained in the following section.

2.2.3.3 Read and write operations

The role of the heads is to store the information in the memory and recall it later when it's needed. To do that the heads use 3 different forms of differential attention.

The first one is content lookup. The goal is to provide a flexible mechanism of navigating through the memory by comparing a key vector emitted from the controller to the content of each memory location using a similarity score. The score may later be used by the read heads for associative recall or by the write heads to modify the content of the memory. A second attention mechanism records transition between consequently written memory locations in a $N \times N$ temporal link matrix. L[i,j] is close to 1 if i was the next location written after j and close to 0 otherwise. This allows DNC to recover sequences in the order that it wrote them. The third form of the attention is designed to allocate the memory for writing. The usage of each memory cell is represented as a number between 0 and 1 and a weighting, that picks up unused locations is delivered to the write head. After each write the usage vector increases its value, while not writing results in a decay over time. This allows the controller to reallocate and delete the information that is no longer required. When reading from the memory R read weightings $\{w^{r,1}; \ldots; w^{r,R}\}$ are defined, that average the content of locations. The read vectors are defined as

$$r_t^i = M_t^\top w_t^{r,i}. \tag{2.11}$$

The read concatenated with the input for the next time step gives the controller access to the memory. At every time step the memory is modified by the write operation

$$M_t = M_{t-1} * (E - w_t^w e_t^\top) + w_t^w v_t^\top, \tag{2.12}$$

where E is a N $\times$W matrix of ones. e_t and w_t are erase and write vectors respectively emitted by the controller. The write operation erases unused data from the memory and writes new values.

2.3 Reinforcement Learning

Supervised learning is a class of machine learning tasks when the model basically learns how to solve the problem based on input-output pairs from the training set. It is called supervised learning because the process of an algorithm training may be described as a teacher supervising the learning. This approach has several disadvantages. Not every problem can be solved using supervised learning. There are cases when there is no or very little labeled data, so the model has nothing to train on. In that case information must be inferred from the input data itself. This is called unsupervised learning. Another class of problems has no training data at all. Instead an environment is given and the training data is generated by the model interacting with it. Those tasks are solved using reinforcement learning [22].

The two main components of the reinforcement learning are the agent and the environment. The agent is a model that learns to solve the problem. The agent exists in, has prior knowledge of, and can interact with some environment. The agent has a state $s_i \in S$ and at every timestep t performs an action $a_j \in A$ in the environment. Performing any action in the environment will transition the agent to a new state and might change the state of the environment. Performing an action a_j in the state s_i will result in agent getting a reward $R_{i,j}$. The end-state $e \in S$ is a final state

that an agent tries to reach in order to finish an epoch. The goal of reinforcement learning is usually defined to reach an end-state accumulating the maximum reward. The main difference of the reinforcement learning from the supervised learning is that the feedback the model gets is not instructive, but evaluative. Instead of being told "how to solve the problem" the feedback tells "how well the problem was solved".

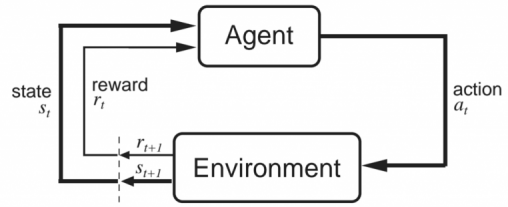

Figure 2.7: Reinforcement learning scheme

If in an environment a state s_i fully summarizes all previous states than making an action is called a Markov decision process and every state in it has a Markov property. Reinforcement learning most often models a Markov decision process (MPD) (see 2.4.5) [22].

The cumulative future reward R for a series of actions that an agent performed is called return R and given by

$$R_t = r_{t+1} + r_{t+2} \cdots \sum_{k=1}^{\infty} r_{t+k+1}. \tag{2.13}$$

It is easy to see from the equation, that the longer agent is in the environment the more reward it gets and when the goal is set to accumulate a maximum return the agent will never reach the end step. Therefore the equation only makes sense if the task is episodic and the series of rewards will end. More common than using future cumulative reward as return is using future cumulative discounted reward,

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} \cdots \sum_{k=1}^{\infty} \gamma^k r_{t+k+1} \tag{2.14}$$

where $\gamma \in [0, 1]$. If γ is set to 0, the agent will be getting only the immediate reward without concern of what comes after it. On the other hand, if γ is set to 1, than agent cares about all rewards equally.

The next concept in the reinforcement learning is Policy. Policy $\pi(s, a)$ describes how an agent is going to act in a state s. The probabilities of taking every action in a state are $\sum_a \pi(s, a) = 1$. The goal of the reinforcement learning is to learn an optimal policy π_*, which maximises the return. When following a policy π it is possible to evaluate how much reward the agent will get from being in the state s and taking an action a from that state. This is done using state and action value functions.

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[R_t | s_t = s] \\ Q^\pi(s, a) &= \mathbb{E}_\pi[R_t | s_t = s, a_t = a] \end{aligned} \tag{2.15}$$

$V(s)$ is the expected return when starting in the state s following the environment policy π, which may be stochastic. The same is true for the action value function $Q^{\pi}(s,a)$, that defines how much return is expected if an agent takes an action a while in the state s and following the policy π.

The transition probability between states can be defined as

$$\mathcal{P}^{a}_{ss'} = Pr(s_{t+1} = s' | s_t = s, a_t = a). \tag{2.16}$$

And the expected reward for taking an action a and moving to a state s' from the state s can be rewritten as

$$\mathcal{R}^{a}_{ss'} + E[r_{t+1} | s_t = s, s_{t+1} + s, a_t = a]. \tag{2.17}$$

Now, the Bellman equations (full derivation can be found in [14]) for state value function and action value are respectively

$$\begin{aligned} V_{\pi}(s) &= \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'} [\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s')], and \\ Q^{\pi}(s) &= \sum_{s'} \mathcal{P}^{a}_{ss'} [\mathcal{R}^{a}_{ss'} + \gamma \sum_{a'} \pi(s',a') Q^{\pi}(s',a')]. \end{aligned} \tag{2.18}$$

These Bellman equations allows calculation of state value functions as values of other states. For example, knowing the value for state s_{t+1} we can calculate the value for s_n. Calculating state value function is called policy evaluation.

At this point there are several ways forward depending on the environment. If the MDP is known, then generalized policy iteration using dynamic programming is one of several ways to solve the problem of finding $\pi*$. In this thesis MDP is unknown, and as such Monte Carlo (MC) methods are used. Monte Carlo methods require only experience—sample sequences of states, actions, and rewards from actual or simulated interaction with an environment. The methods are based on averaging sample returns. Let us assume experience is divided into episodes, and that all episodes eventually terminate no matter what actions are selected. Only on the completion of an episode are value estimate and policies changed.

The first step is to estimate $V_{\pi}(s)$ under policy π, given a set of episodes obtained by following π and passing through s. The estimation can be done using two different MC methods: first-visit MC and every-visit MC. These two methods are very similar but have slightly different theoretical properties. In the first-visit MC estimation is done simply by averaging returns following the first visits to s, yielding

$$V\pi(s) = \frac{\sum_{i}^{K} V_{\pi,i}(s)}{K}. \tag{2.19}$$

An important fact about Monte Carlo methods is that the estimates for each state are independent. Estimating action values $Q_{\pi}(s,a)$ in MC methods is essentially the same as estimating state values. The action value is the average return for every

time the state s was visited and action a was taken. It is important to note, that policy should be stochastic in order for model to learn. Otherwise some actions will never be taken and the agent might never find the optimal solution. Policy improvement is done by making the policy greedy with respect to the current value function. For any action-value function q, the corresponding greedy policy is the one that, for each $s \in S$, deterministically chooses an action with maximal action-value

$$\pi(s) = argmax_a q(s, a). \tag{2.20}$$

Policy improvement then can be done by constructing each π_{k+1} as the greedy policy with respect to q_{π_k}

$$q_{\pi_k}(s, \pi_{k+1}(s)) = q_{\pi_k}(s, argmax_a q_{\pi_k}(s, a)). \tag{2.21}$$

For Monte Carlo policy evaluation it is natural to alternate between evaluation and improvement on an episode-by-episode basis. After each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode [14].

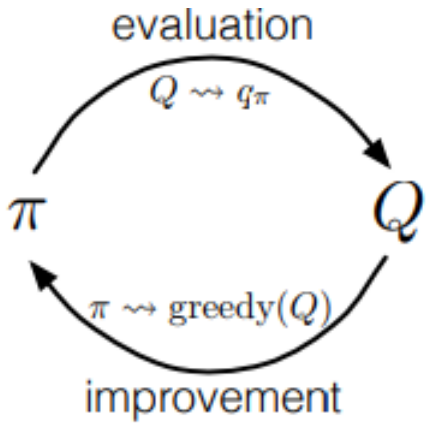

Figure 2.8: Monte Carlo approximation of the optimal policy

2.4 Mathematical Background

This section aims to define functions and metrics used in the project. This section is aimed towards an audience with less mathematical background and as such will not focus on rigorousness but rather on ease of reading while also relating it to the context of machine learning.

2.4.1 Softmax

During classification tasks the aim is to translate a vector of outputs (with size equal to the amount of possible classes) into a vector of probabilities corresponding to the classes [23]. One can think of this as a re-scaling of the values in the vector such

that all values sum to 1. For classes $j \in [1, 2, \ldots, K]$ the probabilities is computed by

$$softmax(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}. \tag{2.22}$$

2.4.2 Activation functions

Activation functions are non-linear transformations that are used inside the perceptrons of a neural network. After a perceptron has computed its weighted inputs, the activation output decides whether or not the cell is 'used'. Activation functions are non-linear by necessity in order to be 'stackable' (using more than one layer of perceptrons connected after one another). Consider the case of two linear activation functions that are stacked. Then, for any two such activation functions a and b it is possible to construct a third linear activation function, such that it takes the same input as a and gives the output of b.

2.4.2.1 The sigmoid function

The sigmoid function behaves like a smoothed heaviside step function [24] and satisfies the equation

$$y = \frac{1}{1 + e^{-x}}. \tag{2.23}$$

The motivation behind using the sigmoid function is that the output is always bounded in $[0, 1]$.

2.4.2.2 The hyperbolic tangent

Defined with

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \tag{2.24}$$

the hyperbolic tangent function has a very similar shape to that of the sigmoid function, save that it is instead bounded in $[-1, 1]$[25].

2.4.2.3 ReLU

An increasingly more popular activation function is the Reticified linear unit (ReLU). This unit uses the function

$$Relu(x) = max(0, x), \tag{2.25}$$

and is a very successful way to mitigate the risk of a vanishing gradient. Since it does not have an upper bound, it can not be implemented into the LSTM unit in a stable fashion (because the gates needs to stay bounded) due to an amplifying behaviour. Since it lacks the gates that makes the hidden state store information from further back in the sequence, ReLU does not see much usage within RNN architectures.

2.4.3 Maximum Likelihood Estimation and Negative log likelihood loss

The maximum likelihood estimation (MLE) method is a way to find the most likely value for a parameter to generate a given distribution. It can be shown by the following example

2.4.3.1 Example: MLE for variance of a spherical gaussian

The likelihood function for a spherical Gaussian $\mathcal{N}(\mu, \sigma^2 I)$ distributed variables in 2 dimensions (denoted by (u, v)) is

$$f(x|\mu, \sigma^2) = \Pi_{i=1}^{n} \frac{1}{2\pi\sigma^2} exp(-\frac{||x-\mu||^2}{2\sigma^2}). \tag{2.26}$$

Now, since the logarithmic function is a monotone function, its maximum is achieved at the same point as the original function.

$$f(x|\mu, \sigma^2) = \sum_{i=1}^{n} log(\frac{1}{2\pi\sigma^2} exp(-\frac{||x-\mu||^2}{2\sigma^2})) = n \ log(\frac{1}{2\pi\sigma^2}) - \sum_{i=1}^{n} \frac{||x-\mu||^2}{2\sigma^2}. \tag{2.27}$$

Now, by differentiating $f(x|\mu, \sigma^2)$ and setting the derivative to 0, we obtain

$$\frac{n(4\pi\sigma)}{2\pi\sigma^2} = \sum_{i=1}^{n} \frac{||x-\mu||^2}{\sigma^3} \tag{2.28}$$

and by rearranging to get σ^2 on the left hand side we get

$$\sigma^2 = \frac{1}{2n} \sum_{i=1}^{n} ||x-\mu||^2 = \frac{\sigma_u^2 + \sigma_v^2}{2} \tag{2.29}$$

which is the average between the variances in each dimension.

2.4.3.2 Negative Log Likelihood Loss (NLL)

By the above example, it is apparent that finding the MLE of a parameter and finding the logarithmic MLE are equivalent, but maximizing a function (the reward) is not consistent with general machine learning algorithms, where the loss is minimized. Let the likelihood be the probability of correctly predicting the target output given the input. Now, consider that any likelihood is bounded by $[0, 1]$, since probability 0 is no occurrence and probability 1 represent a correct prediction always happening. Then, the logarithm of the likelihood will be in $(-\infty, 0]$. We can then define the Negative Log likelihood loss of model A as $NLL_A = -log(P(x)_A)$. This loss will then take values $[0, \infty)$, where the value gets higher as the prediction gets unlikelier and approaches 0 when the predictive power of the model is strong.

2.4.4 Jensen-Shannon Divergence

Then Jensen-Shannon divergence (JSD) is a measure for the distance between a number of probability distributions using a combination of the Shannon Entropy and Jensen inequality. We will here not here list the complete derivation of the measure but refer the reader to the original paper [26]. The generalized Jensen-Shannon divergence can be defined as

$$JS_{\pi}(p_1, p_2, \ldots, p_n) = H(\sum_{i=1}^{n} \pi_i p_i) - \sum_{i=1}^{n} \pi_i H(p_i), \tag{2.30}$$

where $\pi_1, \pi_2, \ldots, \pi_n$ are weights corresponding to distributions $p_1, p_2, \ldots, p_n$ and H is the Shannon entropy function has the following definition: Let $\boldsymbol{A} = (A, \theta)$ be a discrete probability space i.e. $A = \{a_1, a_2, \ldots a_n\}$ is a set of finite size, with a corresponding probability θ_i for each element. Then, the Shannon entropy of $\boldsymbol{A}$ can be described with

$$H(\boldsymbol{A}) = -\sum_{i=1}^{n} \theta_i log_2 \theta_i. \tag{2.31}$$

This measure is used to compare the likelihood distributions between the training, validation and the sampled molecules. The absolute value of this measure is not indicative of predictive strength. The purpose of the measure is to give an indication to when overfitting begins to occur, which we consider to happen when the trend of the JSD value has a positive gradient.

2.4.5 Markov Property and Decision Process

Consider $\boldsymbol{A}$ to be the set of all possible actions to perform, and $\boldsymbol{S}$ the set of all possible states of the environment in which the action is taken. For each state $s \in \boldsymbol{S}$, a subset of actions $a \subset \boldsymbol{A}$ is available that each transition s to some new state $s' \in \boldsymbol{S}$. If a process has the Markov property, then given state s, the probability distribution of the actions a are independent of the history of states leading up to s. A Markov decision process is defined using a reward function $r_t(a, s)$ that assigns a reward for each possible action taken at time t, and $p_t(.|a', s)$ the transition probabilities that gives the conditional probability of each state given the taken action.

2.4.5.1 Example: Using the Markov Property to show why LSTM and GRU outperform regular RNN

Consider the case of a regular RNN that is tasked with this generation task of molecules. It will use the input state to create the probability distribution for the next token. If for a trained model, the last generated token was the double bond '=' (see 2.5.2), then the probability of the next token being 'C', 'S' or 'O' becomes very high, while the probability of 'F' or another '=' approaches 0. However, since the state passed to the network is the last generated token, the RNN will create its prediction based on that token irrespective of the token generated two steps back, which led to the current state. This has a weakness that can be conceptualized with e.g. the last generated token being 'C'. Since it is the most common atom in

any organic molecules, there are many tokens with a fair chance of being generated. Consider now that the case of the token before 'C' being a triple bond '#', which would make '=' and another '#' impossible due to Carbon (with few exceptions) having more than four bonds. Since the regular RNN can't remember the history of the states, the probability distribution would remain the same given all input states 'C', which leads to a significant amount of invalid SMILES strings generated. This example serves as a motivation for the usage of LSTM and GRU, whose hidden states contain more information (see 2.2.2.1).

2.4.6 Statistical properties of sampling the optimal model

Suppose the goal of the network is to sample the entire chemical space of a multinomial model; this can be considered as sampling with replacement. Let $n \in \mathbb{N}$ be the size of the space and $x \in \mathbb{N}$ the size of the sample. Then, if all the compounds in the space is enumerated and define the random variable $X := \{X_1, \ldots, X_n\}$ where $X_i = 1$ if compound i is present in the current sample and 0 otherwise. Let the probabilities p_i denote the probability of sampling each corresponding compound. Then for a sample of size x, the probability that compound i is absent is $(1 - p_i)^x$ and thus the probability of a sample including i is $1 - (1 - p_i)^x$ and by the linearity of expectation the expected number of unique compounds

$$E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} (1 - (1 - p_i)^x). \tag{2.32}$$

It was proven by [27] that the expected value is maximal when the distribution is uniform (without bias), which yields the expectation for an ideal model

$$E[X] = n(1 - (1 - \frac{1}{n})^x) = n(1 - (\frac{n-1}{n})^x). \tag{2.33}$$

To compute the variance for the uniform distribution we observe that

$$E[X^2] = E[(\sum_{i=1}^{n} X_i)^2] = E[nX_1^2 + n(n-1)X_1X_2] = nE[X_1] + n(n-1)E[X_1X_2], \tag{2.34}$$

where $E[X_1X_2]$ is the probability that a particular pair of compounds is present, which by the inclusion-exclusion principle is expressed as

$$E[X_1X_2] = 1 - 2(\frac{n-1}{n})^x + (\frac{n-2}{n})^x. \tag{2.35}$$

The variance is then computed as

$$\begin{aligned} Var(X) &= E[X^2] - E[X]^2 \\ &= n(1 - (1 - \frac{1}{n})^x) + n(n-1)(1 - 2(\frac{n-1}{n})^x + (\frac{n-2}{n})^x) - n^2(1 - (1 - \frac{1}{n})^x)^2. \end{aligned} \tag{2.36}$$

Note that, by definition variance is non-negative, $E[X^2] \geq E[X]^2$ and that since $X \geq 1$ (at least one distinct compound is generated for any $x > 1$) both $E[X^2]$

and $E[X]^2$ are increasing functions w.r.t $E[X]$ ($E[X]^2$ holds trivially, and note that since $X \in \{0, 1\}$, $E[X^2] = nE[X] + \sum_{i \neq j} E[X_i X_j]$, where the second term is non-negative). Furthermore observe that since variance in this sampling problem can be non-zero (which can easily be proven numerically by choosing $n = x = 2$, you get the two possible scenarios that $\sum_i X_i = 1$ or $\sum_i X_i = 2$), it implies that since $E[X]$ is maximal when the distribution is uniform, so is the variance.

2.4.7 Principal Component Analysis: how to extract and display information

During this project there is a need to display the results in an informative way. Since it is hard to infer information from SMILES for a non-chemist (or perhaps, anyone in general), more sophisticated methods are used. The most common numerical representations are called fingerprints, which are vectors of descriptors using numbers. This has the drawback of having so many dimensions to the data that it is hard to do any comparisons. In addition, some variables in the vector are likely correlated, which implies they contain the same information. *Principal component analysis* (PCA) is an attempt to compress the vital information into a lower dimension representation by linear transformation. This is a projection that aims to maximize the explained variance i.e. how much of the total variation in the data that can be expressed in one dimension. Usually for a 2-dimensional PCA plot, the explained variance is just given as a sum of the explained variance in the displayed dimensions.

2.5 Representation of molecules

2.5.1 MQN Fingerprints

The molecular quantum numbers (MQN) are 42-dimensional vectors of descriptors that count different properties of a molecule such as atoms, bonds and polar groups [28]. It has been shown to correlate with bioactivity. This representation is useful when studying the properties of a compound, but as it is not unique for each molecule, it can't be directly used for training the REINVENT model, and is instead used for post-processing.

2.5.2 SMILES

There are several ways to represent molecules in programming, but within the field of machine learning the most common one is the SMILES notation [4]. This is a string-based language which represents a molecule by listing each heavy atom with a corresponding *token*. In this report we will not give full details of the complete grammar of SMILES, but for a basic summary to ease the reading, we list a few of the common rules. There are more than one system for building a smiles string from a molecule, but we here use canonical smiles. These are readable by default in the **RDkit** package, and are unique for each individual compound.

- Tokens next to each other in the string are generally bonded to each other, but brackets denote side-chains.
- The tokens = and # denote double and triple bonds, respectively.
- An integer token denotes the start of a cyclic ring, which is closed by repeating the same integer and as such all valid rings contain integers in pairs. An example is a benzene ring, which would be written as "c1ccccc1", the two '1's closes the ring, denoting which atoms are connected.
- Capital letters denote regular bond type atoms (except elements which have two letters, e.g. Cl, Br) while lowercase letters are atoms in an aromatic bond. Using a simple ring as an example, "c1ccccc1" is benzene, while "C1CCCCC1" is cyclohexane.
- While stereochemistry can be represented using / and \for directions and while there is also a representation using @ and @@ for denoting the chirality in the counter-clockwise direction, these are omitted from this project, as the data sets used for our benchmark (see 3.3.2) do not contain them.
- For organic molecules hydrogen atoms are implicit.

An example of how to construct a SMILES from a molecule can be found in Figure 2.9.

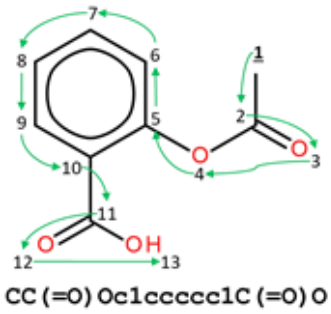

Figure 2.9: Example of creating the canonical SMILES using graph traversal of aspirin [29].

While there are more comprehensive representations that can capture more features than the SMILES notation, they require magnitudes more processing time. As an example, a molecule can be represented as nodes and edges in a connected graph, where each node is an atom with a feature vector containing all relevant chemical information. This gives a more complete description of a molecule but at the time of writing no graph-based model has managed to beat the state-of-the-art SMILES-based models, presumably due to technical limitations.

3
Materials and Method

3.1 Packages used

3.1.1 PyTorch

PyTorch is a scientific computing package that is based on python. Using **PyTorch** a researcher can harness the power of graphic processing units to train machine learning models. It is known for providing two of the most high-level features; namely, tensor computations with strong GPU acceleration support and building deep neural networks on a tape-based autograd system. **TensorFlow** is a big competitor of the **PyTorch** library and has been on the market much longer. However, **PyTorch** has several features that may make it a preferred choice for some researchers.

3.1.1.1 Dynamic Computational graphs

The framework maintains a computational graph that defines an order of computations that needs to be performed. It allows **PyTorch** to set up an execution mechanism that is different from the programming language interpreter in order to optimize operations and execute in parallel on the GPU. A static computational graph defines all computations explicitly before the program execution. When the program compiles, a static computational graph reserves all the memory it needs for the execution and can no longer be altered, but a dynamic computational graph reserves the memory when the program requests it. Additionally there is also a room for low level code optimization during the program execution. It is very convenient for the researcher not to explicitly define the computational graph – this is done by the **PyTorch** during the execution which makes the framework much easier to use compared to its competitors.

3.1.1.2 Different back end support

PyTorch splits the backend for the CPU and GPU. The core functional class of the **PyTorch** framework is a tensor. In its essence tensor is a matrix that contains data of the same type. If gradient flag on the tensor object is turned on, then the object will store a complete history of computations performed with it. This feature is used to construct a computation graph and perform backpropagation on it. If one wants to use a CPU to perform computations on, then a CPU tensor object needs to be created. The same holds true for a GPU. This makes it very easy for the researcher to use **PyTorch** on constrained platforms.

3.1.2 RDKit

RDKit is a python package containing a collection of chemoinformatics and machine learning software, it is used to translate molecules into SMILES and vice versa. The molecule object can also be processed in **RDKit** to get numerous chemical properties e.g. fingerprints and similarity measure values between two or more molecules [30].

3.1.3 Scikit-learn

Scikit-learn is a package designed for machine learning that can perform most common tasks in data processing [31]. This toolkit has been used here to do randomized splitting of data into training and validation. It is also used when performing PCA (see 2.4.7).

3.2 Hardware used

The computational resources included both NVIDIA Tesla K80s and NVIDIA Tesla V100s, accessible through the SLURM workload manager [32]. In order to run multiple tests in parallel, we opted to perform tests for training time on the K80 graphic cards, due to higher availability.

3.3 Datasets

For all machine learning, it is important to be aware of what kind of data that one is training a model on. This can decide what features and patterns that the model learns and also assist in the inference from the results.

3.3.1 ChEMBL

The ChEMBL database contains information about 1.8 million biomedically active compounds [33]. Subsets of ChEMBL is commonly used in chemoinformatical research. For this project, the database has been filtered to remove rare chemical properties and further preprocessed to remove SMILES tokens that appeared too infrequently. After filtering, the dataset contained 1.2 million SMILES strings, which was still too large for running multiple models for hyperparameter study. For faster computational performance, a randomly selected subset of the filtered ChEMBL with 400,000 SMILES was used for training. To verify that the subset does not have a different distribution than the full ChEMBL, we computed the statistics for some **RDKit** properties in table 3.1.

	Mean	Variance
Full ChEMBL		
Number of heavy atoms	26.998	49.81
Number of rings	3.419	1.559
Sequence Lengths	45.29	168.36
Number of molecules with at least 4 rings	527009	-
Number of molecules with at least 35 heavy atoms	168930	-
Training Set		
Number of heavy atoms	27.004	49.78
Number of rings	3.419	1.559
Sequence Lengths	45.29	168.28
Number of molecules with at least 4 rings	179024	-
Number of molecules with at least 35 heavy atoms	57350	-

Table 3.1: Comparison between the full ChEMBL and the training subset.

Another subset of the filtered ChEMBL of size 10,000 was used for validation. The sets contained 29 different tokens, excluding the start and end tokens.

3.3.2 GDB-13

The GDB-13 database is a publicly available collection of most drug-like molecules with up to 13 heavy atoms of types C, N, O, S and Cl (hydrogen atoms are implicit). The full database consists of 977,468,314 structures [34]. The training set is a randomly sampled subset of size 1 million, with a separate validation set of size 10,000. This, unlike ChEMBL, did not require any preprocessing, as the shorter sequence lengths allowed for reasonable training speed. The number of token per SMILE is approximately normally distributed with mean 22.19 and variance 4.754. This dataset contains 24 different tokens excluding the start and end tokens. The distribution of the validation set is similar with mean 22.16 and variance 4.740.

3.4 The REINVENT model

The original REINVENT model is based on reinforcement learning and consists of two phases: the pre-training of the prior and the reinforcement learning of the agent. This section will detail the structure of the prior (and by extent, the agent) and its training procedure. This is followed by a description of how the model generates new molecules as an output. Finally, this section will address how to use reinforcement learning to tailor the output towards a desired feature.

3.4.1 The input format

We input data to the model using files with a line-by-line list of SMILES which can then be loaded in any order (see 2.1.4.1). Each SMILES string is then split into an ordered sequence of its individual tokens. A vocabulary is initialized that enumerates each unique token. The sequence is then translated into a vector representation of

integers using an encoding based on a one-hot encoded table of the same size as the vocabulary i.e. a token corresponding to element 14 in the one-hot encoded table, will be represented by the integer 14 in the vector. The vector representation is then later used as model input. To the vector a start token (also an integer from the encoded table) is attached as the first item in sequence. The model processes all batch sequences simultaneously in a vectorized form, and the rest of the section will describe what the model does on an individual batch item, but the actions can be generalized to any batch size (within available memory constraints). All vectors in the batch are padded to the same size as the largest sequence length of the batch using the '0' (end) token.

3.4.2 Training the model

The model reads the start token as the first input item to be passed through a forward-pass of the network. The forward pass of any input (not necessarily just the first) is fed through a linear embedding layer and then (together with the current hidden state, (see 2.2) through the hidden (GRU) layers. The hidden output is passed through a linear layer that compresses the output size down to the vocabulary size. This network output goes through the logarithmic softmax function to generate the probabilities of the next item for each token in the vocabulary. The likelihood of picking the correct token is translated to a NLL (see 2.4.3). The output token to be fed into the next forward pass is generated from the multinomial distribution of the probabilities computed. Due to the properties of logarithms, the NLLs can be summed up to give a cost for the entire sequence and the forward passes continue until the longest sequence in the batch has passed all its items. From this cost the backpropagation is performed and optimization is done using the ADAM optimizer.

3.4.3 Generating Molecules - Sampling the model

Generating molecules works very similar to the training. The network is given the start token into the forward pass to compute a probability distribution from which a random token is generated. This is fed back into the network and the loop continues until an end token is generated. This is illustrated in Figure 3.1. This generation of molecules is considered a Markov decision process (See 2.4.5).

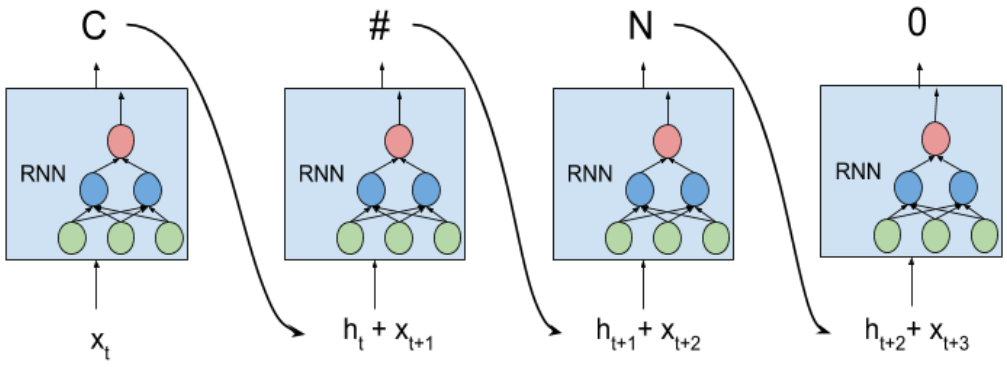

Figure 3.1: Schematic illustration of the sampling process of the RNN model.

3.4.4 Reinforcement Learning – how to specialize the model

After the model is trained it will generate molecules that are similar to the training set. If one then wants to shift the distribution of the generated molecules towards a particular feature, the reinforcement learning phase is introduced. The current model is called the *prior* model and a duplicate model, the *agent* is created. This agent will be performing the training in the following phase. The idea of using a prior is that while the intention is to learn the policies of the new feature the model still needs to learn the structures of general SMILES (to see what happens to the REINFORCE policy if there is no prior we refer to [7]). The learning process in each epoch starts with sampling a batch of molecules from the agent, along with the likelihood of generating the molecules. Then, the NLL of the molecules using the prior model is computed. Here, some scoring function $S(A) \in [0, 1]$ is defined where A is the output SMILES string generated by the model. $S(A)$ will take the value 1 if A perfectly matches the desired feature and 0 if there is no match at all. $S(A)$ can be a continuous function instead of a binary, if it is desired. An augmented likelihood function that combines the scoring function and the negative loss of the prior is constructed and the new loss function is defined as the distance between the agent likelihood and the augmented likelihood. The training is performed on the agent model based on this loss.

3.5 Implementing the DNC

Implementing a DNC is a central task of the project. There is a publicly available implementation of the DNC [21]. However, since it is written using the **TensorFlow** framework and since the current stable version of REINVENT is written using the **PyTorch** framework, it was not used for this project. Instead an implementation by [35] was taken as a core. A serious issue was documented in this implementation, which we successfully fixed (see 3.5.1).

3.5.1 Numeric instability

It was noted that values with the minus sign appeared in the memory's write weights, when by definition that they should be in $[0, 1]$. This issue kept reappearing on different datasets and tasks. By eq. 2.12, the memory in the corresponding positions will get amplified in forwards time instead of dampened, leading to a numerical explosion towards infinity. To prevent this, clamping on the write weights to $[0, 1]$ was performed.

This has solved all observed stability issues. Our hypothesis for the cause of numeric instability is the way computers handle calculation on float numbers, which are used for numeric approximation. If several successive approximations using float numbers happens around zero, there is a chance that the final result may be accidentally negative.

Float numbers are designed to handle infinity, and floats becoming infinity is not alone enough to cause a crash. The issue arises when infinite numbers are passed through the softmax function of **PyTorch** (see 2.22). There is no way to

numerically compare two infinite numbers, and as such the output of this function would thus become NaN (not a number). These NaNs then propagate into the rest of the sequence as any numerical operation on a NaN gives a NaN output. While theoretically there is a way to handle a single infinity to the softmax function, it would significantly slow down the operation for general use. Since the appearance of an infinite number is both an edge case and deemed to have a high likelihood of resulting in more infinite numbers after forward passes, the **PyTorch** team decided to not implement this feature in their softmax operation [36].

3.5.2 Validating the model performance

To insure that the model was implemented correctly it was tested on the copy task, which is described in [20]. The idea behind a copy task is to check how well the model can memorize and later reproduce a sequence. The input is a random binary sequence with a delimiter flag. The target is simply an input sequence without a delimiter flag.

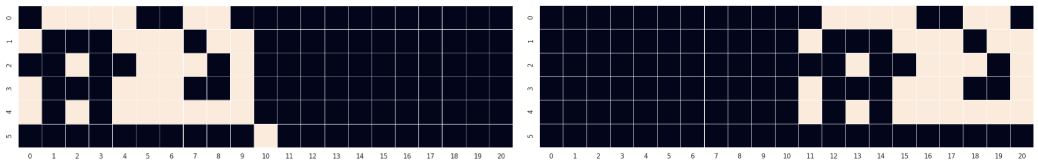

Figure 3.2: Input and target output in the copy task

Storage and access to the information over long period of time has been historically difficult for RNN and other dynamic architectures [20]. The models were trained on randomized sequences of length from 1 to 20. The models that were compared are DNC with the GRU controller, the RNN with LSTM cells and the RNN with GRU cells. The results of training are summarized below.

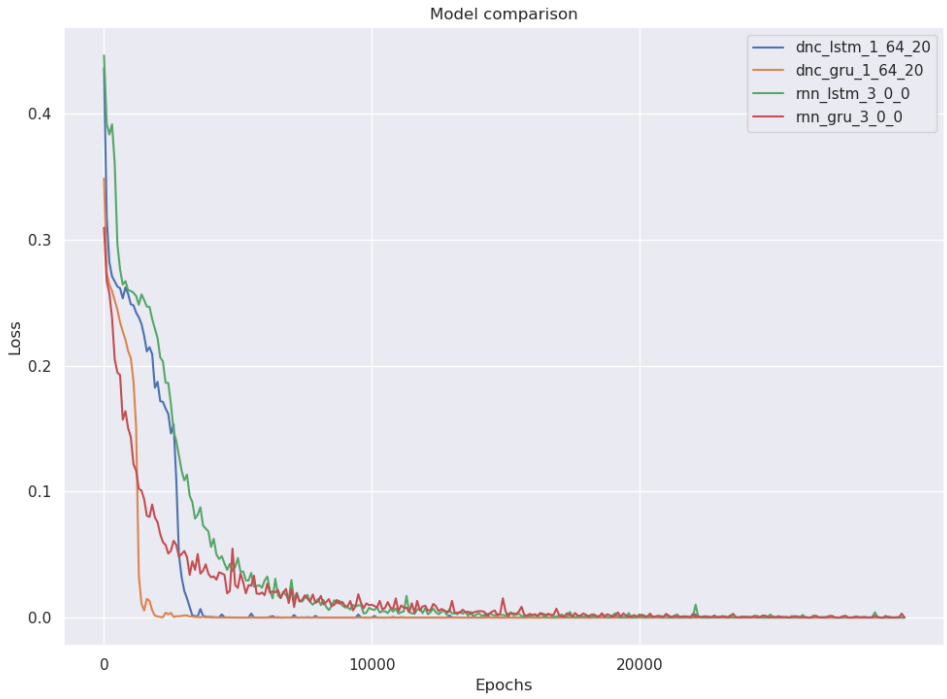

Figure 3.3: Comparing different models on the copy task

The naming format is the following: {model type}_{cell type}_{amount of layers}_{memory cells}_{cell size}. The DNC models start learning slower but converge faster compared to the GRU and LSTM models. This is probably happening because of all the additional memory weights which have to be trained. In the end both models are able to learn copy task and perform with loss at nearly zero. The results are comparable to the ones achieved by [20]. Examples of performing the copy task on sequence lengths of same size as the training set can be found in Figures A.1 to A.4.

The important result in [20] is that NTMs can generalize much better than RNNs. That means after being trained on sequences of length 20, NTMs are able to perform reasonable well on sequences 50% longer and even double the size of the training set. Given that DNC is the improved version of NTM it was assumed that it also would be able to generalize. As can be seen in Figures 3.4 and 3.5, the regular LSTM and GRU models begin to make a considerable amount of mistakes. Still, there is an early indication that the LSTM models do perform better than their GRU counterparts on this task.

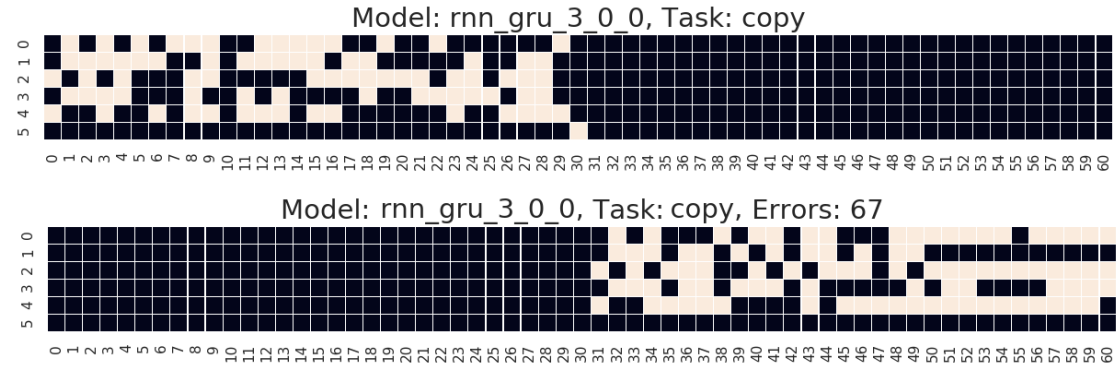

Figure 3.4: GRU model generalization on test sets 50% bigger than during training. The model begins to make mistakes already at column 5.

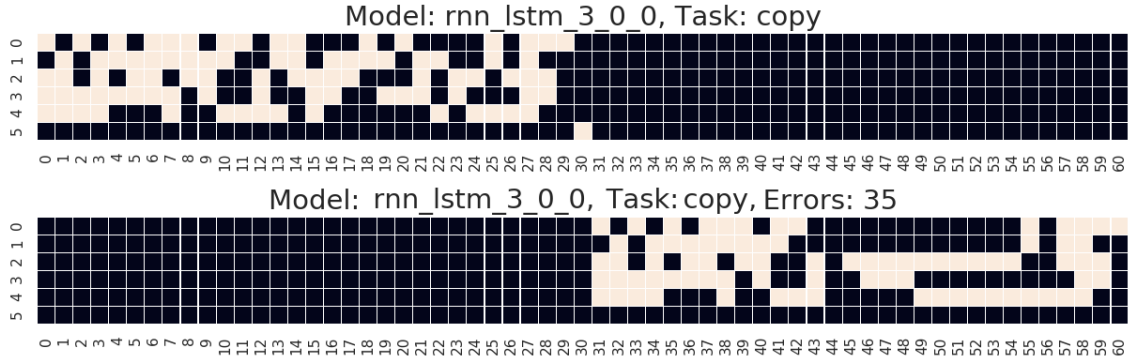

Figure 3.5: LSTM model generalization on test sets 50% bigger than during training. The first 12 columns are perfectly reproduced, before the model begins to make mistakes.

In Figures 3.6 and 3.7, we can observe that the DNC models generalize very well on sequences this length, where the regular LSTM and GRU models completely failed.

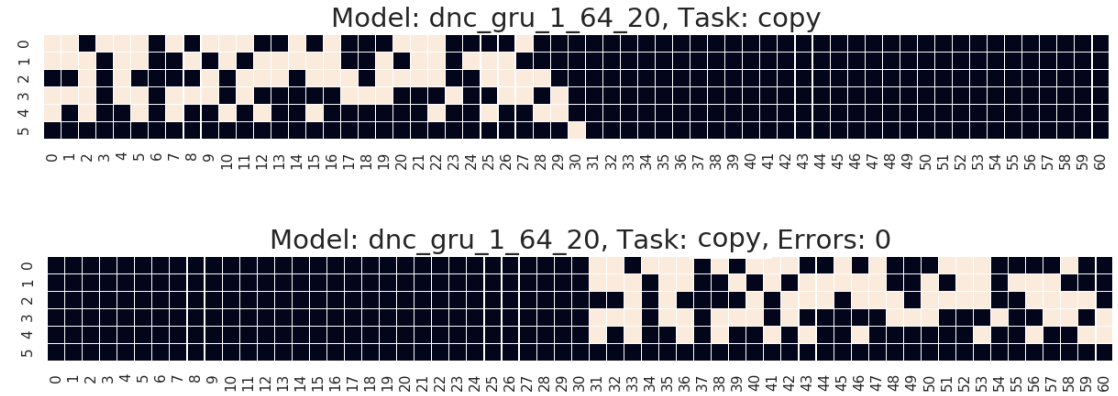

Figure 3.6: DNC with GRU controller generalization on test sets 50% bigger than during training.

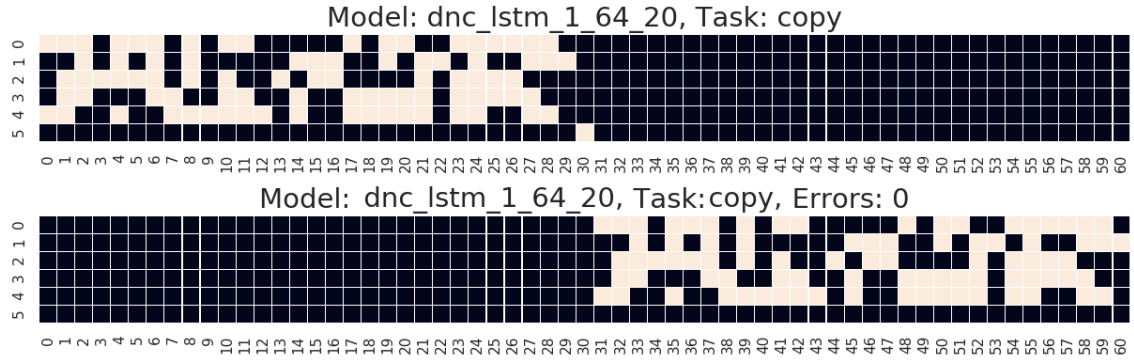

Figure 3.7: DNC with LSTM controller generalization on test sets 50% bigger than during training.

When increasing the curriculum to sequences of twice the length of the training set, the GRU DNC begins to perform significantly worse, whilst the LSTM DNC maintains high accuracy. This is shown in Figures 3.8 and 3.9. To see how the regular GRU and LSTMs performed on this sequence length, see Figures A.5 and A.6.

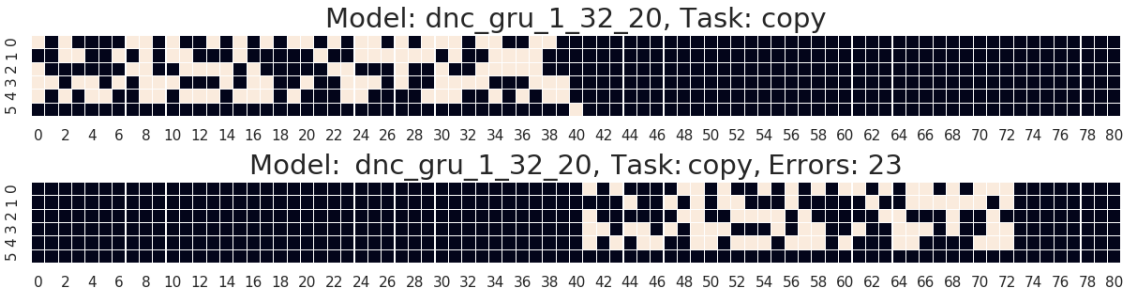

Figure 3.8: DNC with GRU controller generalization on test sets 100% bigger than during training.

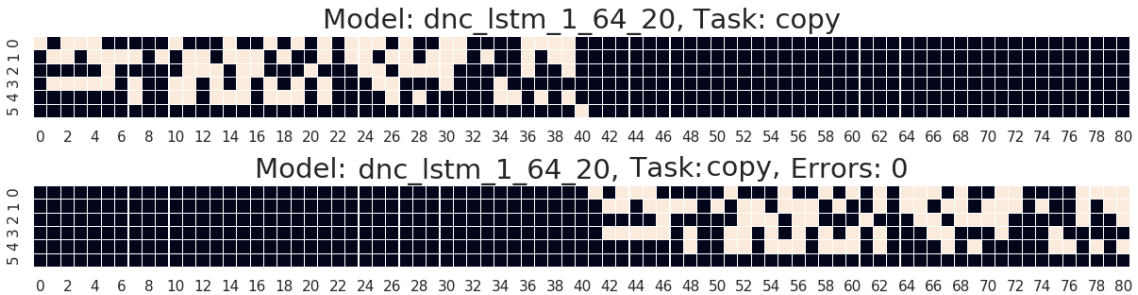

Figure 3.9: DNC with LSTM controller generalization on test sets 100% bigger than during training.

These results are inline with [20], and so we are assuming that the implementation of the DNC is accurate.

3.5.3 Combining with REINVENT

The DNC implementation onto the main REINVENT framework is rather simple: the input from the embedding layer is concatenated with the read vector from the previous sequence item (a zero-vector if it is the start of a sequence). The controller is defined to expect a larger input than in the base model and the output of the hidden layers is then fed to the memory in order to perform the write/read operation. The output read vector is concatenated with the controller output as defined in 2.2.3.1.

Note that between each batch of sequences the memory, like the controller hidden state, is reset. The memory stored between batches is only in the trainable weights. An useful motivation would be that the model would otherwise make decisions based on earlier observed sequences instead of only the current, which would prevent the sampling from being independent random variables.

3.5.4 Hyperparameters and design choices

The embedding layer of the reference REINVENT model consists of 128 units, followed by 3 hidden layers using the GRU architecture with 512 units each. A reference LSTM model was created as baseline using the same settings. The DNC controller settings used were the same for both the GRU and the LSTM cells.

Following the idea of [12], the DNC has greater performance if the reliance and training gradients are put on the memory weights rather then the controller weights (one can imagine this as a trained network extracting information from the total generated sequence rather than the current input). To reduce the influence of the controller, the amount of regularization was maximized: the dropout rate was put to 0.5, layer normalization was used after the controller layer output before the information was passed through the memory. We used a gradient norm scaling with the norm 1. The choice of memory size parameters were performed using a baseline of 32 memory cells, memory size 20 and 8 read heads.

3.6 Benchmarking

Currently, in *de novo* molecular generation there is not one common benchmark that is used by all research groups, though there has been attempts. *MOSES* is one collection of metrics that scores models based on generated SMILES strings after training on provided standardized datasets [37]. It measures the fraction of valid SMILES strings and the fraction of unique strings in a sample of size 10,000. *MOSES* then also provides scores based on similarity measures to a reference data set and the internal diversity in the sample.

3.6.1 Benchmark on GDB-13

In [27], the ability for a model to learn the grammar of SMILES is assessed based on the total coverage of GDB-13 (see 3.3.2) that the model can generate from a sample of size $2 \cdot 10^9$ using a training set of 1,000,000 SMILES strings. The generative strength of the model is assessed by comparing the coverage to that of a theoretical

ideal model (see 2.4.6), which is unbiased and samples all molecules in GDB-13 with an uniform distribution.

3.6.1.1 On the lowest significant difference in coverage

The sample size ($2 \cdot 10^9$) used in the benchmark described in [27] is too computationally expensive to perform repeated trials on the same model, and as such the distribution of data coverage can only be estimated. Suppose that we use a confidence interval to describe the coverage C of any model with $P(\mu_C - Z_\alpha \cdot \frac{\sigma_C}{\sqrt{n}} \leq C \leq \mu_C + Z_\alpha \cdot \frac{\sigma_C}{\sqrt{n}}) = 1 - \alpha$, where α is the confidence level and $n = 1$ the number of samples. Consider now the a conservative scenario for two models with different coverages $C_1 < C_2$. If both coverages are within their respective confidence intervals in the worst possible way, C_1 is in the lower end of its distribution and C_2 in the upper end. To guarantee that, at some significance level α, the confidence intervals don't overlap, there must be a difference of $C_2 - C_1 \geq 2Z_\alpha\sigma_1 + 2Z_\alpha\sigma_2$. However, since the true standard deviations σ_1, σ_2 are unknown, we use the upper estimation σ^* from that of the uniform model (see 2.4.6 for proof that this is indeed an upper estimation) and as such the inequality that must hold is thus

$$C_2 - C_1 \geq 4Z_\alpha\sigma^*. \tag{3.1}$$

This is applied when analysing the results in 4.1.3.

3.7 Tests Performed

The comparison between REINVENT and DNC+REINVENT was performed by monitoring the results of the trained priors yielded by having both models train on the GDB-13 and ChEMBL datasets. The statistics of each epoch were saved down to get an initial view of the differences in the models. For the GDB-13, the same benchmark as in [27] was performed to check the generative chemical space of the model as well as to check for the bias using PCA plots (To be explained in 3.6). A study was conducted on perturbation of the parameters by multiples of 4, sans a run with 2 read heads.

For the GDB-13 dataset, the number of memory cells in the DNC examined were 8, 16, 32 and 64 for both the GRU and LSTM models. For the ChEMBL dataset, only the numbers 8 and 32 were tested, due to CUDA memory constraints for the longer sequences. While it would have been possible to run models of interest with 64 and 128 memory cells using a smaller batch size, this would impact the results and prevent us from relating the GDB-13 results to the ChEMBL results.

For the GDB-13 training 165 epochs were conducted and for ChEMBL 330 epochs. The training on both datasets used a batch size of 512. An exponential learning rate was used with the initial learning rate 0.005 and halved every 15 epochs on the GDB-13 models, every 30 on the ChEMBL models.

4
Results

The following section is sorted by the dataset which yielded the corresponding results and has the structure of training behaviour first, followed by benchmark results. For reader convenience, we chose to only show the validation losses. We consider this the most important loss metric, since the generative task of the network needs to be able to sample novel compounds. The training and sampled losses can be found in Figures A.7 and A.8. These are found to continue decreasing after the validation loss is stagnating (or increasing) on the observed training epochs, which indicates that the model is overfitting to the training set. One can refer to Figure A.9, to see the expected JSD between training and sampling distribution.

4.1 Results for training priors on GDB-13

For training on the GDB-13 dataset, the LSTM models required only around 15 epochs before showing a clear advantage and as the training continued the convergence (defined as the global minimum of the JSD shown in Figure 4.1) was observable in both LSTM models within 50 epochs, whereas the GRU models required around 100 (epoch varies on the chosen level of smoothing). It is from this epoch in each model that the metrics are compared at. It is noteworthy that the regular RNN LSTM outperforms the DNC GRU (see 4.1.3). We are choosing to only display the best memory setting of each model so that the trends are distinguishable from each other.

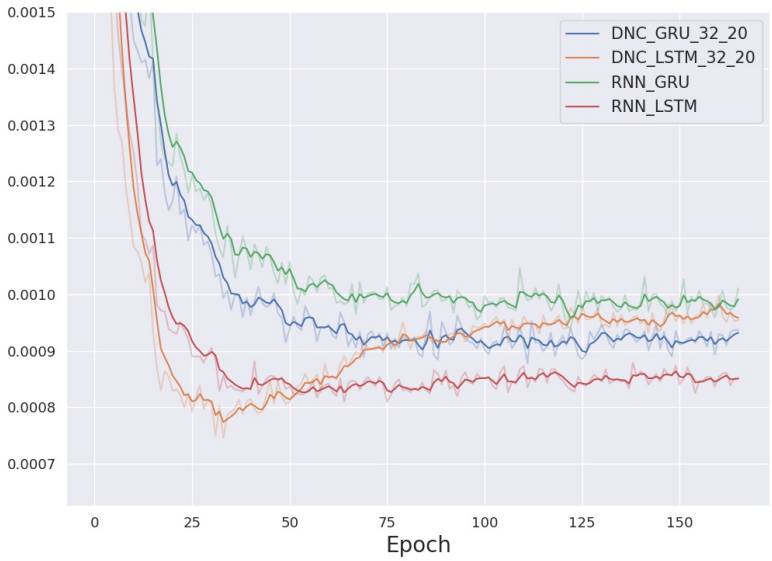

Figure 4.1: Comparison of Jensen-Shannon Divergences for the best model of each type on GDB-13. A lower value is better.

Next, we look at the validation NLLs (Figure 4.2 for the different models. This gives a measure for the likelihood of sampling other molecules that are inside GDB-13. We observe that while the LSTM model has a lower minimum, it also diverges faster during the overfitting (seen in both figs 4.1,4.2). This result indicates that LSTM models learns more features from the training set than GRUs. While the values of the metric still favors the DNC models over their RNN counterparts, the difference is hardly (at the respective best epochs of the models) by any significant amount.

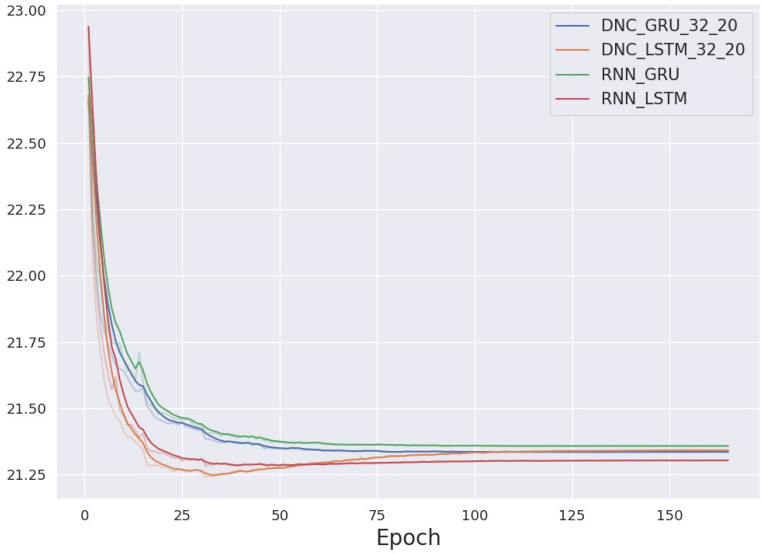

Figure 4.2: Average validation NLLs on GDB-13

However, as can be seen in Figure 4.3, the difference in variance between the LSTM and GRU models is notably smaller, especially compared to that of the reference GRU. The lower validation variance indicates that there is less bias towards any particular molecule in the validation set (and by extension the the full GDB-13).

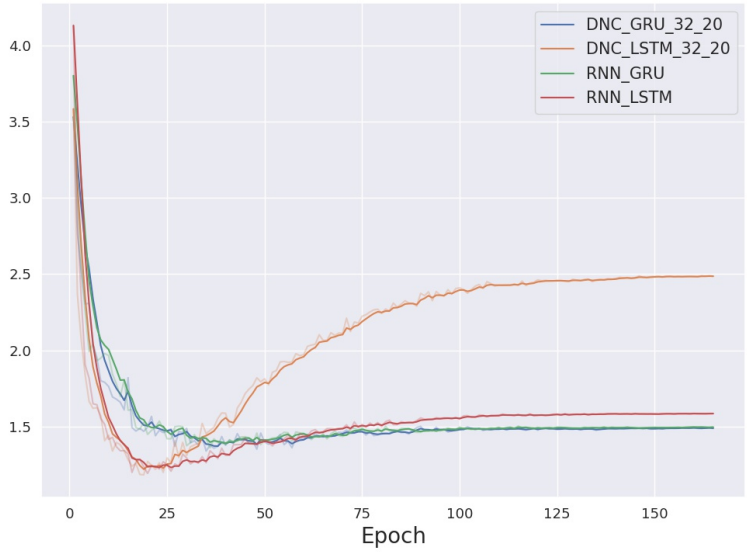

Figure 4.3: Variance of validation NLLs on GDB-13

For the ratio of valid sampled smiles generated, the same hierarchy of model performance was found. Since all observed models have a ratio above 99%, we do not see the difference in this metric as a significant difference between them.

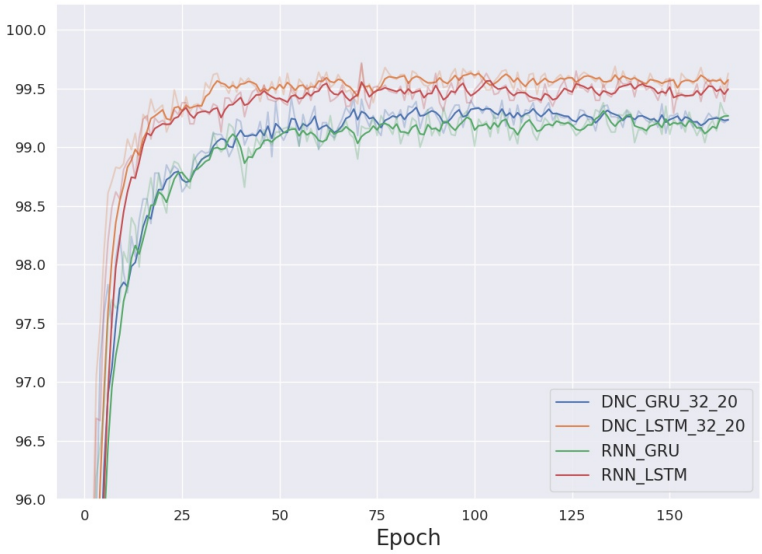

Figure 4.4: Percentage Valid SMILES sampled GDB-13. A higher value is better.

4.1.1 Comparison of training time for GDB-13

The trained models were compared based on the time to complete the 100 first epochs, to get a representative average time per epoch. The times here displays the time required for all calculations conducted every epoch including sampling and validation losses.

Model	Time per 100 epochs
RNN GRU	6h 42min 22s
RNN LSTM	7h 15min 18s
DNC GRU	19h 30min 54s
DNC LSTM	19h 29m 17s

Table 4.1: Time elapsed for 100 epochs for each model on GDB-13

4.1.2 Testing hyperparameters using GDB-13

The general pattern for the read heads seem to be that a higher amount of read heads is better, but the model with 8 read heads managed to reach the same minimum as its 16 read head counterpart when smoothing was removed, which is why it was used in the other tests. The order of parameters described in the following figures are **number of memory cells**, **size of memory cells** and **number of read heads**. All results are based on the LSTM cell controllers.

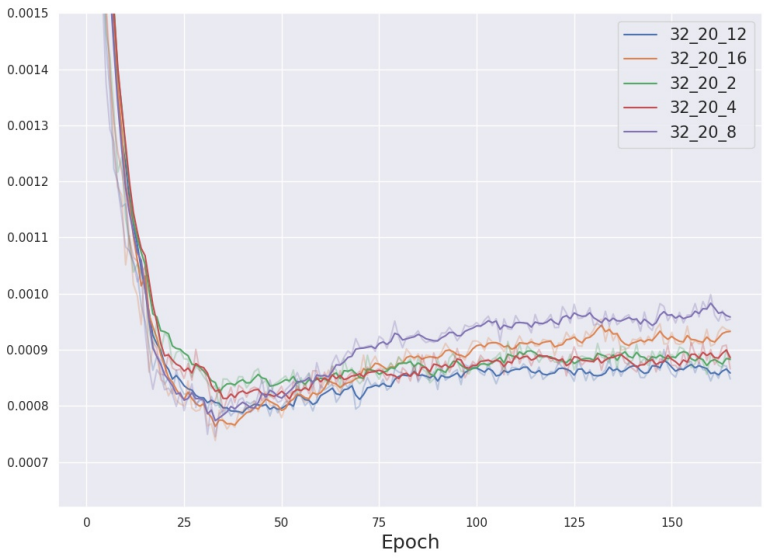

Figure 4.5: Jensen-Shannon Divergences comparison using different number of read heads.

For the memory sizes, the models behaved very closely around their respective global minimums, while they did diverge at different rates. We can not explain why the model with 20 read heads acted more extremely with a lower minimum and higher overfitting, but we deem this to be a parameter that does not need optimization, due to its low impact.

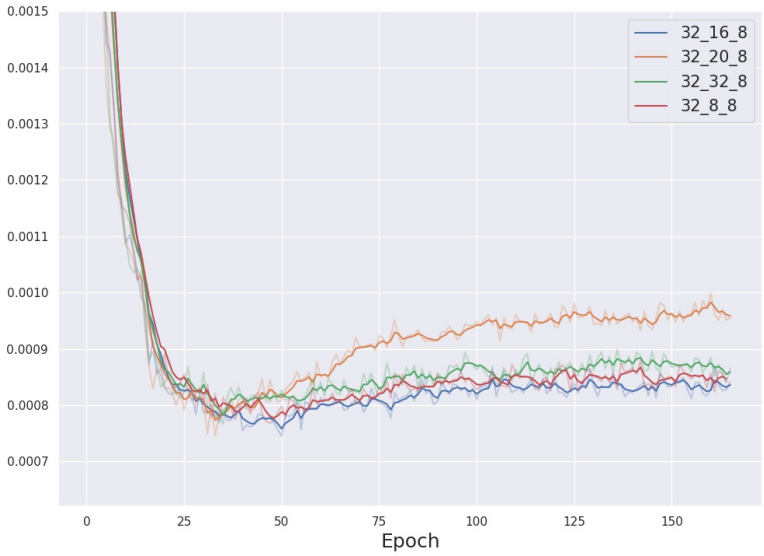

Figure 4.6: Jensen-Shannon Divergences comparison of memory sizes

The number of memory cells is the most important parameter to tune since it scales the worst with respect to computational time and memory consumption. There was no clear pattern on how the results depended on the number, as for the GDB-13 shown in Figure Figure 4.7, the performance seemed to improve until a model of twice the maximum sequence length (which was 32 tokens) showed a considerably worse results. As would later appear on the ChEMBL training though (see Figure 4.9), a choice of 8 memory cells outperformed the previous baseline of 32, for the LSTM architecture.

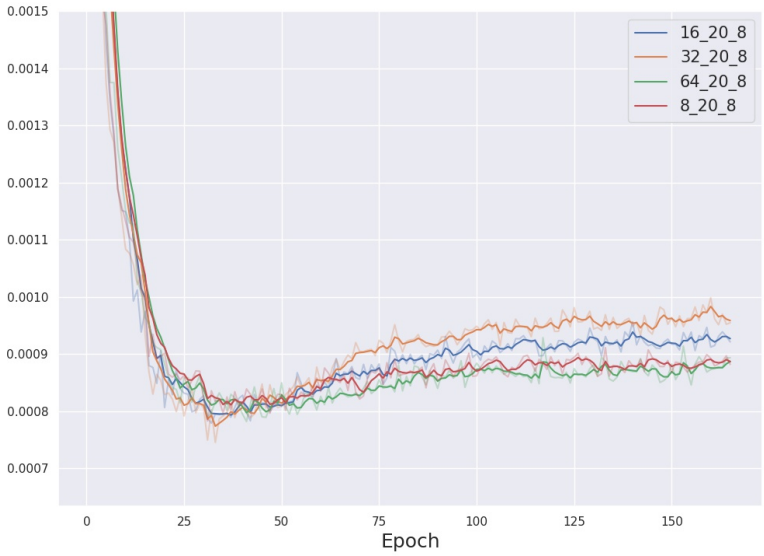

Figure 4.7: Comparison of Jensen-Shannon Divergences between the different number of memory cells

The behaviour of the 32_20_8 model was also examined for dropout values of 0.25 and no dropout. As the values of dropout was lowered, the performance not only worsened, but also became numerically unstable when dropout was removed, as can be seen by the peaks in Figure 4.8. This confirms that the observations of [12] is transferable to this task, and that the DNC trains better when the model has to rely more on the memory.

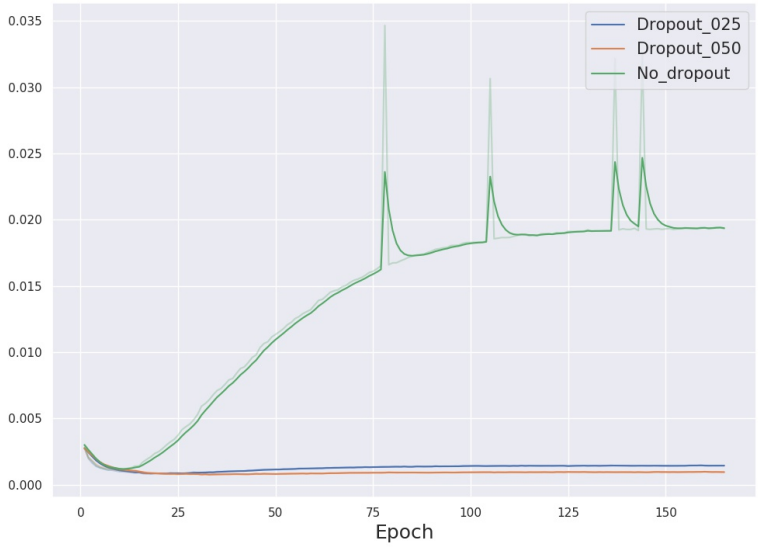

Figure 4.8: Jensen-Shannon Divergence comparison for different values of dropout

The impact of memory settings for the global minimum seems to be fairly small, and as such only one memory setting per DNC model type was sampled for the benchmark as each DNC model required around 100 GPU hours per sample.

4.1.3 GDB-13 Benchmark

The coverage of GDB-13 for each model is presented in table 4.2. This lists the properties of a sample of size $2 \cdot 10^9$ for the best models of each type. The uniform model coverage is computed using eq. 2.32, where the number samples k is equal to the number of sampled SMILES that were inside GBD-13. This is done since the SMILES outside GDB-13 are unaccounted for in the sense of sampling with replacement, as they are outside the considered set. The normalized coverage is computed as the fraction

$$coverage_{norm} = \frac{coverage_{real}}{coverage_{uniform}}, \tag{4.1}$$

to give a representation of the distance to the ideal model of sample size k.

Model	Valid %	# of generated in GDB-13	Total coverage % of GDB-13	uniform model coverage %	normalized coverage %
$GRU_{baseline}$	99.1795	1650364531	72.5499	81.5186	88.9980
$LSTM_{baseline}$	99.4431	1711818789	73.4307	82.6448	88.8510
GRU_{DNC}	99.2622	1664504526	72.7460	81.7841	88.9488
$LSTM_{DNC}$	99.4643	1713440000	74.4578	82.6736	90.0624

Table 4.2: Summary of the coverage benchmark

Note that the normalized coverage is a poor metric to compare between models, as it is shown with $GRU_{baseline}$. This model has a lower amount of molecules overall inside of GDB-13, yet a higher normalized coverage than both its DNC and LSTM counterparts. This can be amended by computing the statistics on a standard amount of sampled SMILES inside the database. This can be done in multiple ways, such doing the calculations for coverages while considering only a random subset of sampled SMILES from each model equal to the # in GDB-13 of the worst model. We advise against this normalization, as this penalizes the models with a higher % inside GDB-13, unless one guarantees that a proportional amount of unique valid molecules is removed for the calculation (computing the number of unique molecules is computationally expensive, as one has to parse all SMILES through e.g. **RDKit** to get the canonical form to find the uniques and not just count all unique strings). Our proposed alternative is to sample each model until a standard amount, e.g. $2 \cdot 10^9$ is sampled, before doing calculations. It would theoretically be possible for us to sample another $500 \cdot 10^6$ molecules per model for this to give a proper normalization, but we omitted this study due to constrained computational power. While the difference in coverage between the model might seem small at first, we can prove that the models differ significantly using the method for a conservative confidence interval described by eq. 3.1. Let C_1 be the coverage of the baseline GRU model. Then to at confidence level α ensure that we have a significantly better model

at covering GDB-13, the computed coverage has to be at least $4Z_\alpha\sigma^*$ higher, where σ^* is computed using the variance of the uniform model for $k = 1650364531$. Using eq. 2.36, we get that $\sigma^* = 9.75 \cdot 10^{-6}$. A conservative choice of $\alpha = 0.00001$ yields $Z_\alpha = 4.265$. Inserting into eq. 3.1, we get that a difference in coverage greater than $C_2 - C_1 = 0.0166391\%$ is significant with a 99.999% confidence interval.

4.2 Results on ChEMBL

The ChEMBL dataset was orders of magnitude harder to train on than GDB-13 for all models. As can be seen in 4.9, the JSD is around 30 times higher, which implies that this distribution is considerably harder for the models to learn. Still, the relative behaviour between the models remains the same, although the best epochs appear later, around epoch 70 for LSTMs and 150 for GRUs. For this dataset we chose to use the models with the lowest global minimum in JSD in the following figures, as the different memory settings showed close to visually indistinguishable results,

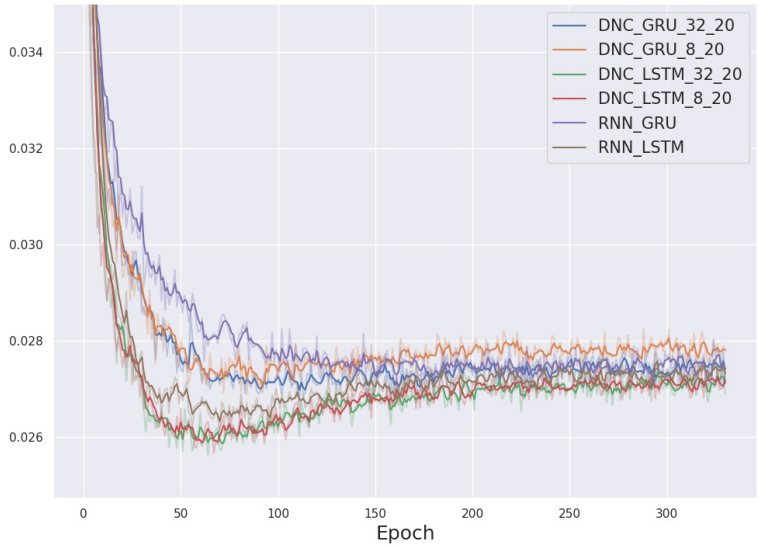

Figure 4.9: Jensen-Shannon Divergence comparison for the different models on ChEMBL

The higher values of the JSD are supported by the higher magnitude of the average validation NLLs shown in Figure 4.10, where on average, the chances of a molecule in the validation set being generated by the LSTM DNC is twice as high as that of the reference regular GRU (at their respective best epochs). Still, there is no significant difference between the LSTM DNC and the regular LSTM.

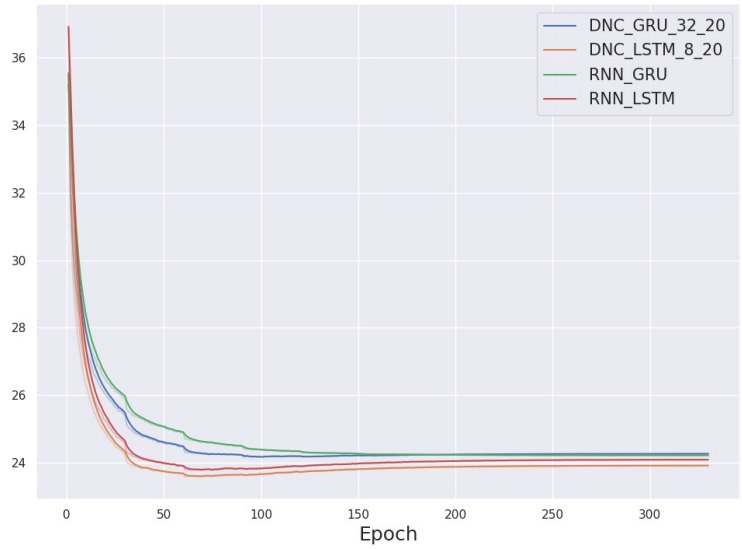

Figure 4.10: Average of validation NLLs for ChEMBL compounds

From observing the variance of validation NLLs (Figure 4.11), the training supports the GDB-13 observation that the models that models that learns the most from the training set also overfits the most as training continues.

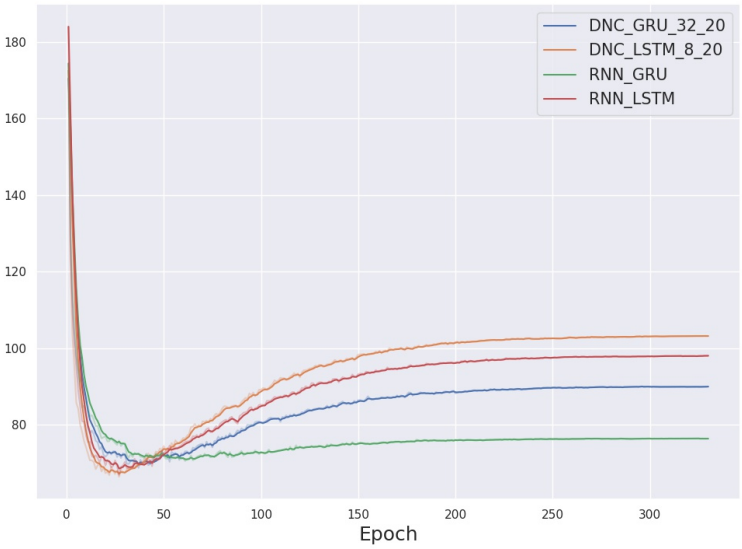

Figure 4.11: Variance of validation NLLs for ChEMBL compounds

The ratio of valid smiles for training on ChEMBL is lower than that of the GDB-13 training as expected, since the variation of the data set is greater. When the model trains on a batch this variation results in lower gradients on the weights since

the optimizer will try to move in too many directions, which results in a middle ground with less useful information learned.

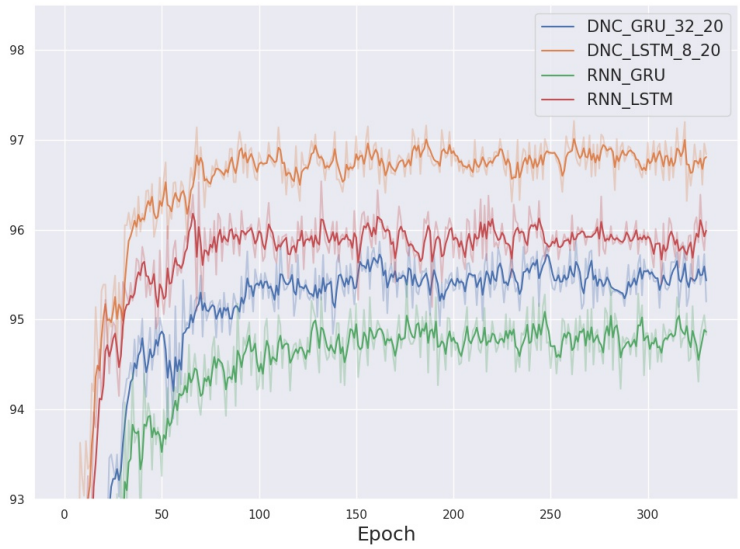

Figure 4.12: Percentage of Valid sampled SMILES for models trained on ChEMBL

4.2.1 Comparison of training time for ChEMBL

In a similar fashion to table 4.1, the training times are based on a 100 epochs to reduce any eventual noise. These models required more time despite having a training set only 40% the size of the GDB-13 training set size. The reason behind this is in the amount of outliers in ChEMBL in terms of sequence lengths. When a batch is processed in forward passes, the function will continue to do forward passes until the longest sequence has reached its end token, which results in every batch doing a considerable amount of forward passes where only a couple of sequences in each batch are still processed. Even if the average sequence length in ChEMBL is around 45 tokens, the number of forward passes per batch could easily reach above 90.

Model	Time per 100 epochs
RNN GRU	6h 51min 42s
RNN LSTM	7h 25min 26s
DNC GRU	24h 42min 12s
DNC LSTM	25h 3m 48s

Table 4.3: Time elapsed for 100 epochs for each model on ChEMBL

4.2.2 Sampling on ChEMBL

Since the ChEMBL database is not explicitly designed towards virtual screening, there is not one common way to benchmark the performance. The benchmark for

coverage on GDB-13 works because GDB-13 is a compact discrete chemical space i.e. there is a strict set of rules that decide whether a compound is in GDB-13 or not. ChEMBL, and by extension the chemical space of all drug-like molecules, is in a theoretical sense compact, but not with any practical or numerical means. There is no existing method to determine if a molecule should belong to ChEMBL by only observing its SMILES string without any chemical experiments, unless it is a string already inside the database, we can thus not compute a theoretical coverage of the ChEMBL chemical space. We sampled 400,000 SMILES from each model at their respective lowest JSD epoch, and compared the sample sets with the training set. The table of results can be found in A.4. We could not find any distinct differences in terms of sequence lengths or number of rings sampled in each model, but one LSTM DNC model with 8 memory cells, 8 read heads and a memory length of 20 did have the closest resemblance to the training set (see table. 3.1). The samples of the best LSTM DNC model, the regular LSTM and the training set are compared in a PCA plot based on their MQN (see 2.5.1) fingerprints in Figure 4.13.

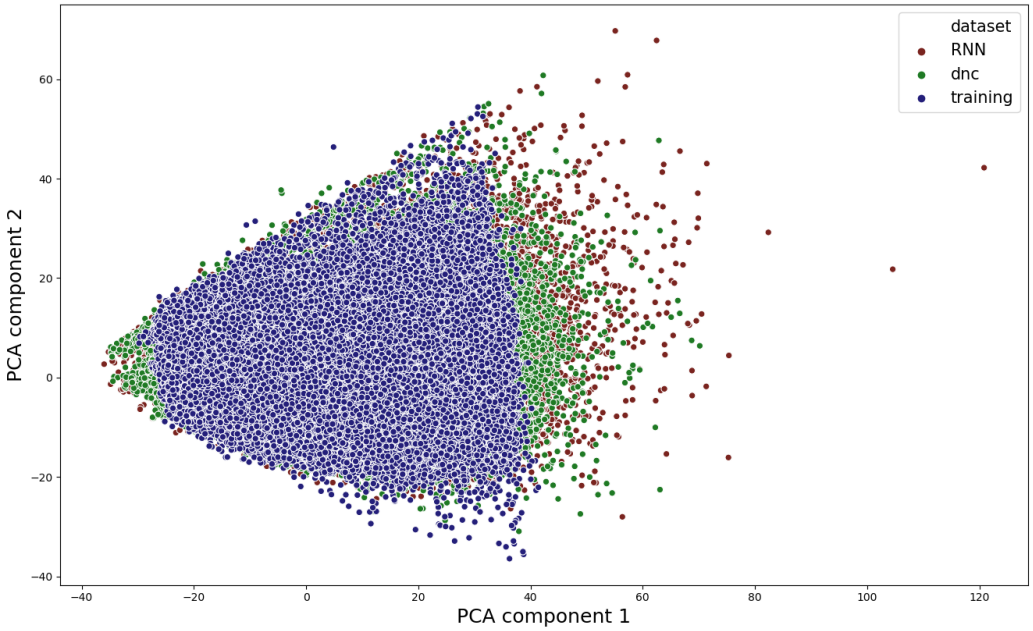

Figure 4.13: PCA plot of the MQN fingerprints of the ChEMBL training set, the LSTM sampled molecules and the best LSTM DNC sampled molecules. The triangular shape of the chemical space the datasets occupy likely implies that the pca-one axis is closely correlated to molecular size, as the variance increases linearly in this direction. Explained variance is 74.62%

The occupied chemical space supported by the MQN fingerprints does not seem to differ between the LSTM and DNC architectures, but they both seem to have a higher variance than the training set, which is very dense.

4.2.3 Model comparison using MOSES

The metrics computed for MOSES were computed for the models as a measure of how well they represented the training set at their respective epochs of lowest JSD. This represent the moment in time where the models are as general as possible, which means that while the models could get a higher score on the benchmark it would be due to overfitting. The sampled SMILES used in table 4.4 are the same as in Figure 4.13.

Model	Valid %	% of unique	FCD	Fragment Similarity	Nearest Neighbour Similarity	Scaffold Similarity
GRU	94.71	98.05	0.1362	**0.9998**	0.5501	0.9222
GRU DNC	95.09	**98.11**	0.1549	0.9995	0.5456	0.9195
LSTM	95.84	96.09	0.1421	0.9995	0.5680	0.9261
LSTM DNC	**96.50**	97.15	**0.0884**	0.9996	**0.5777**	**0.9316**

Table 4.4: MOSES metrics on the ChEMBL dataset. The models displayed in the table are the best models found of each model type. For the FCD metric, lower value is better. For the other metrics a higher value is better. Whereas MOSES computes the unique SMILES at 1,000 and 10,000 samples, we compute just a value for all 400,000 samples.

The ratio of unique SMILES string among the valid strings favouring the GRU architectures is an interesting result. This result is presumably because it learns *less* than the LSTM, and thus does not narrow its sample space as much when it trains. This is supported by the lower values of nearest neighbour and scaffold similarities.

5

Discussion

5.1 Observations about memory parameters

The results of the DNC architecture seems to be robust with respect to all memory parameters, while the same can not be said about the computational speed.

The amount of memory cells is the hyperparameter that most affects the training time and results according to our research. Since DNC performs one write per token in the sequence, the maximum amount of writes that the model can do is equal to the amount of tokens in the sequence. Thus the upper bound for the amount of memory cells should be the size of the longest sequence in the dataset W. We think when the sequence size is relatively small setting memory size to W is optimal. However, if W is big, it might result in a longer computation time and slower convergence. For big values of W the optimal value of memory cells according to our results is likely considerably smaller than the average sequence length. We believe that this is because of the behaviour of the DNC when it runs out of memory. The DNC then attempts to free the memory by overwriting the existing content of the cells. If the DNC does not get enough sequence with length higher than the amount of cells, the model will not learn how to efficiently reset memory contents. The ability to free and reuse memory is one of the the key differences between DNC and the NTM that makes model so robust. The increase in training time when using more cells is caused by higher size of the temporal link matrix that stores the order in which the model wrote to the memory.

The second hyperparameter that we investigated is the memory cell size. We found out that cell size does not have a strong effect on the training but an increase will also result in a dramatic increase of training time. In the literature we saw examples of setting the cell size to the amount of unique tokens in the training set [2]. This makes sense if one expects the model to learn that each cell location may be used to store information about a particular token. However, in practise we never saw it giving an advantage to a model compared to its counterparts with higher or lower cell size. In the end we used cell size 20 in the majority of our experiments because we did not see a significant improvement beyond that.

Read heads is one of the memory hyperparameters that we studied the least. It has the smallest impact on training time and memory consumption. From our experiments we saw a slight improvement when using higher number of read heads. We think that the optimal value for the read heads varies based on the task, but we used 8 read heads for every experiment we performed, since it didn't seem to differ all too much from 16 read heads. It is also likely, due to the nature of the interaction

between the number of memory cells and the number of read heads, that part of the results observed can be attributed to a synergistic effect depending on the ratio of the two, but an extensive grid search of this interaction was omitted due to resource constraints.

While the DNC for this study did not show a large enough improvement to compensate for the training time consumption on the ChEMBL data set, we do not discard the possibility that there is a more ideal setting of hyperparameters that utilizes the memory in a more optimal way. From the experiments conducted on the memory hyperparameters we deem it more likely that changing the controller settings have a greater impact on the result.

5.2 Is the improvement from DNC equal to actual better quality of generation?

The improvement seen on GDB-13 is very hard to translate into any practical measure. Whilst the greater coverage combined with the higher validity indicates that the DNC is less biased against some outliers of the dataset, we do not have the same metric available when we move onto the general chemical space. The results for all attempts made on ChEMBL show that the DNC performed better on the average SMILES string both in terms of distribution for the training set or the similarities of the sampled SMILES, but we have yet to find a generation task the DNC can do that the regular LSTM can not. If both training and sampling are faster than the DNC, the LSTM with the same computational resources would sample more than enough excess SMILES to make up for its less powerful learning capabilities.

The DNC model could possibly have an impact in reinforcement learning task. When limiting the chemical space by a certain objective criteria, the property of the model to have a bigger sampling space combined with higher validity could have resulted in sampling some of the compounds that are very unlikely to be sampled for the regular LSTM model. However, we have not conducted experiments on the reinforcement learning part of the generation process, and thus refrain from making any claims on the ability to target specific compounds.

5.3 Is a stronger prior always good?

One of the metrics that we used to evaluate the models is the NLL on the validation set. If NLL is low is means that the model is more likely to sample the molecule from the validation set. That in turn means, that the model has learned the training set well and is able to sample compounds that are similar to it. However, one downside is that since the sample space of the model is smaller and more focused around training set the novelty percentage of structures and scaffolds is lower. In the context of the generative chemical problems, generating molecules similar to the training set could have one disadvantage.

In order to obtain a patent the compound has to be significantly different compared to existing ones. Thus it is great if model can learn the chemical properties

that are similar among all compounds in the training set and be innovative in finding new, different structures that have those properties. That is a hard task and a success depends as much on the model as on the dataset. One also has to keep in mind that models only trains on a string representation of the molecules. It would be accurate to say the model learns SMILES language rather than chemistry and the quality of results wholly depends on how well can SMILES capture chemical meaning of the compound.

We think that having a very low score on the validation set means that model will only generate compounds similar to a training set and be less innovative. However, as shown by [7], the models still have to learn enough from the training set to still give SMILES strings that resembles the training set structures during reinforcement learning.

5.4 Future work

There is still further analysis that can be performed on the memory of DNCs. By observing the step-by-step operations the reading and writing during a sequence, it might be possible to find patterns that can be related to the chemical structures and thus make inferences on what parts of a SMILES string that DNC considers important as it makes its predictions.

We have not made any experiments varying the number of write heads. This was based on a lack of precedence of experiments in other studies. Furthermore, we did not know how this would impact the optimal amount of memory cells, which as previously mentioned is the most sensitive parameter with respect to both results and computational time. It is possible that this could affect the results significantly, for better or worse, which we refrain from commenting on in this project.

Given the possibility that there exist a point where the trade between computational power and actual result is undesirable, we propose that computational time should be considered in future benchmark studies of generative models in general. The DNCs do offer an improvement compared to the regular RNN methods, but there is a limit to how much longer a model should be allowed to train, that is not covered by a limit on number of epochs.

One possible way to reduce the required computational time, especially for the ChEMBL dataset, would be to perform bucketing of the training data. This is a common method to reduce training time and improve efficiency on sequential data by preprocessing. The data is split into groups based on their sequence length and then, for each batch loaded into the training, the data in the batch is all fetched from one group per batch, which reduces the variance in length. The only downside to this method is that it might run the risk of biasing the training in a different way, for better or worse. We chose not to implement data bucketing as it would add a confounding effect on the results of the benchmark that we would be unable to track, but it is a method worth considering for general usage.

Finally, we do acknowledge that the original DNC architecture has received further extensions, such as the Sparse Differential Neural Computer (SDNC) [38]. This scales significantly better in time and required RAM size with no resulting loss in performance due to the bigger memory size of the DNC. We did not implement it,

because we assumed that the memory required per SMILES string would be fairly small. Implementing the sparse memory might be the speed increase the DNC architecture needs to make it competitive for practical applications.

6
Conclusion

We can with high confidence, supported by the benchmarks on both GDB-13 and ChEMBL, state that the DNC is a more powerful architecture than the regular RNN networks for the task of molecular generation. It has a higher capacity to learn the "grammar" of the SMILES language. It does so, however, at the expense of several times more computational demand and, if time is an important factor for the task, the regular RNN can make up for its weaker learning by generating a higher quantity. We have also compared the RNN architectures of GRU and LSTM against each other, and found that for the task of generative modeling of molecules, the LSTM consistently outperformed GRU networks of the same size.

Bibliography

[1] Dimasi, J. A., Grabowski, H. G., & Hansen, R. W. (2016). Innovation in the pharmaceutical industry: New estimates of R&D costs. *Journal of Health Economics*, 47, 20–33. DOI: 10.1016/j.jhealeco.2016.01.012

[2] Putin, E., Asadulaev, A., Ivanenkov, Yan., Aladinskiy, V., Sanchez-Lengeling, B., Aspuru-Guzik, A., & Zhavoronkov, A., Reinforced Adversarial Neural Computer for de Novo Molecular Design *Journal of Chemical Information and Modeling* 2018 58 (6), 1194-1204 DOI: 10.1021/acs.jcim.7b00690

[3] Chen, H., Engkvist, O., Wang, Y., Olivecrona, M., & Blaschke, T. (2018). The rise of deep learning in drug discovery. *Drug Discovery Today*, 23(6), 1241–1250. DOI: 10.1016/j.drudis.2018.01.039

[4] Andersson, E., Veith, G.D., Weininger, D., SMILES: A line notation and computerized interpreter for chemical structures, *U.S. Environmental Protection Agency*, Environmental Research Laboratory, 1987.

[5] Chung. J, Gulcehre. C, Cho. K, Bengio. Y., Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, University of Montreal *arXiv* arXiv:1412.3555v1

[6] Wiess, G., Goldberg, Y., Yahav, E., On the Practical Computational Power of Finite Precision RNNs for Language Recognition, *arXiv* arXiv:1805.04908

[7] Olivecrona, M., Blaschke, T., Engkvist, O., & Chen, H. (2017). Molecular de-novo design through deep reinforcement learning. *Journal of Cheminformatics*, 9(1). DOI: 10.1186/s13321-017-0235-x

[8] Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., … Hassabis, D. (2016). Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626), 471–476. DOI: 10.1038/nature20101

[9] Sanchez-Lengeling, B., Outeiral, C., Guimaraes, G. L., Aspuru-Guzik, A. (2017, August 17). Optimizing distributions over molecular space. An Objective-Reinforced Generative Adversarial Network for Inverse-design Chemistry (ORGANIC) (Version 3), *ChemRxiv*. DOI:10.26434/chemrxiv.5309668.v3

[10] Putin, E., Asadulaev, A., Vanhaelen, Q., Ivanenkov, Y., Aladinskaya, A. V., Aliper, A., & Zhavoronkov, A. (2018). Adversarial Threshold Neural Computer for Molecular de Novo Design. *Molecular Pharmaceutics*, 15(10), 4386–4397. DOI: 10.1021/acs.molpharmaceut.7b01137

[11] Benhenda, M. (2017). ChemGAN challenge for drug discovery: can AI reproduce natural chemical diversity? *arXiv* arXiv:1708.08227.

[12] Franke, J., Nieheue, J., Waibel, A., Robust and Scaleable Differentiable Neural Computer for Question Answering, *arXiv* arXiv:1807.02658

[13] Srivastava. Dropout: a simple way to prevent neural networks from overfitting. Retrieved April 31, 2019, from http://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf

[14] Goodfellow, I., Bengio, Y., & Courville, A. (2016). "Deep Learning". MIT Press, Retrieved April 31, 2019, from http://www.deeplearningbook.org,2016

[15] Rosenblatt, Frank. The perceptron, a perceiving and recognizing automaton Project Para. Cornell Aeronautical Laboratory, Psychological Review Vol. 65, No. 6, 1958

[16] UC Business Analytics R Programming Guide (2018). "Feedforward Deep Learning Models", Retrieved April 31, 2019 from http://uc-r.github.io/feedforward_DNN

[17] Understanding LSTM Networks. Retrieved April 31, 2019, from https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[18] Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. Neural Computation, 9(8), 1735–1780. DOI: 10.1162/neco.1997.9.8.1735

[19] Cho, K., Merrienboer, B. V., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). DOI: 10.3115/v1/d14-1179

[20] Graves, A., Wayne, G., Danihelka, I,. (2014). Neural turing machines. *arXiv* arXiv:1410.5401

[21] DeepMind, Differentiable Neural Computer, (2016), *GitHub repository*, Retrieved January 31, 2019, from https://github.com/deepmind/dnc

[22] Sutton R.S., Barto A.G. (2017). Reinforcement Learning: An Introduction, *The MIT Press* London, England, second edition

[23] Google Developers, Multi-Class Neural Networks: Softmax, *excerpt from machine learning course* Retrieved January 31, 2019, from https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/softmax

[24] Weisstein, E.W., Sigmoid Function. *MathWorld*. Retrieved January 31, 2019 from http://mathworld.wolfram.com/SigmoidFunction.html

[25] Weisstein, E.W., Hyperbolic Tangent. *MathWorld*. Retrieved January 31, 2019 from http://mathworld.wolfram.com/HyperbolicTangent.html

[26] Lin, J., (1991). Divergence measures based on the Shannon Entropy, *IEE Transactions on information theory*. vol. 37, NO. I, Retrieved January 31, 2019 from https://www.cise.ufl.edu/~anand/sp06/jensen-shannon.pdf

[27] Arús-Pous, J., Blaschke, T., Reymond, J.-L., Chen, H., & Engkvist, O. (2019). Exploring the GDB-13 Chemical Space Using Deep Generative Models. *Journal of Chemoinformatics* DOI: 10.1186/s13321-019-0341-z

[28] Nguyen, K. T., Blum, L. C., van Deursen, R., & Reymond, J.-L. (2009). Classification of Organic Molecules by Molecular Quantum Numbers. *ChemMedChem*, 4(11), 1803–1805. DOI: 10.1002/cmdc.200900317

[29] Arús-Pous, J., unpublished work.

[30] RDKit, Retrieved April 31, 2019, from https://www.rdkit.org/

[31] Scikit-learn, Retrieved April 31, 2019, from `https://scikit-learn.org/stable/index.html`

[32] SLURM Workload Manager, Retrieved June 4, 2019, from `https://slurm.schedmd.com/documentation.html`

[33] Gaulton, A., Bellis, L. J., Bento, A. P., Chambers, J., Davies, M., Hersey, A., ... Overington, J. P. (2011). ChEMBL: a large-scale bioactivity database for drug discovery. *Nucleic Acids Research*, 40(D1). DOI: 10.1093/nar/gkr777.

[34] Blum, L.C., Reymond, J,. (2009). 970 Million Druglike Small Molecules for Virtual Screening in the Chemical Universe Database GDB-13. *Journal of the American Chemical Society* DOI: 10.1021/ja902302h

[35] Chatterjee, R., Diffentiable Neural Computers for Pytorch, (2017), *GitHub repository*, `https://github.com/ixaxaar/pytorch-dnc`

[36] Nan in Pytorch softmax. Retrieved April 31, 2019, from `https://github.com/pytorch/pytorch/issues/6864`

[37] Polykovskiy, D., Zhebrak, A., Sanchez-Lengeling, B., Golovanov, S., ... Zhavoronokov, A., (2018). Molecular Sets (MOSES): A Benchmarking Platform for Molecular Generation Models, *arXiv* arXiv:1811.12823

[38] Rae., J.W., Hunt, J.J., Harley, T., Danihelka, I., Senior, A., Wayne, G., Graves, A., Lillicrap, T., (2016). Scaling Memory-Augmented Neural Networks with Sparse Reads and Writes, *arXiv*, arXiv: 1610.09027

A

Appendix 1

A.1 Copy Task auxiliary figures

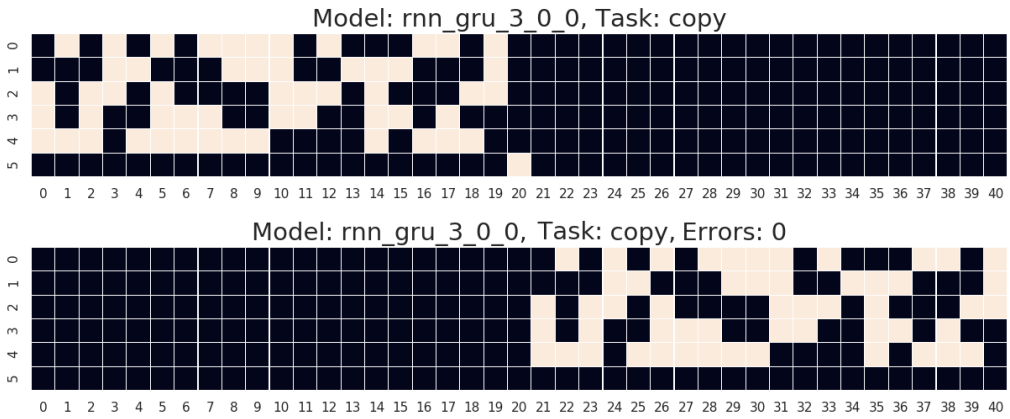

Figure A.1: Regular GRU on copy task of training set size

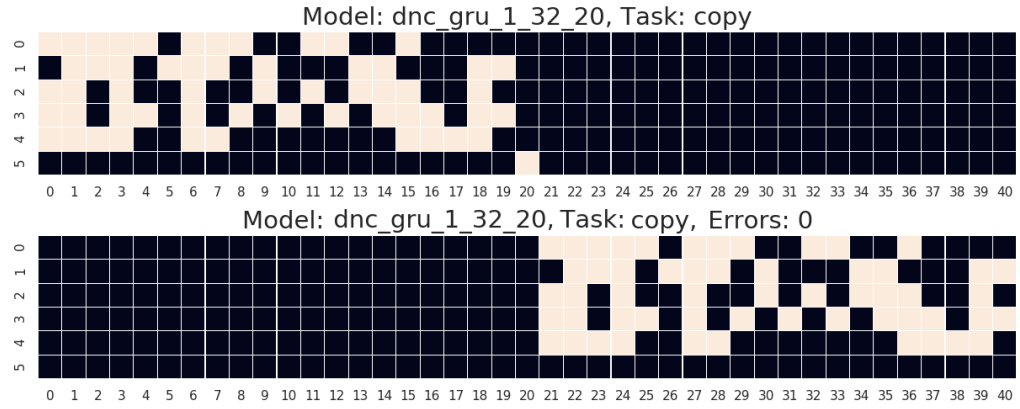

Figure A.2: GRU DNC on copy task of training set size

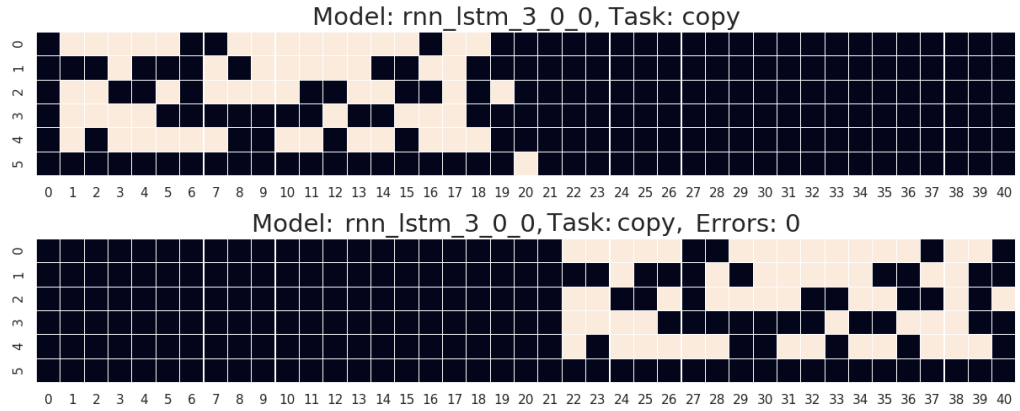

Figure A.3: Regular LSTM on copy task of training set size

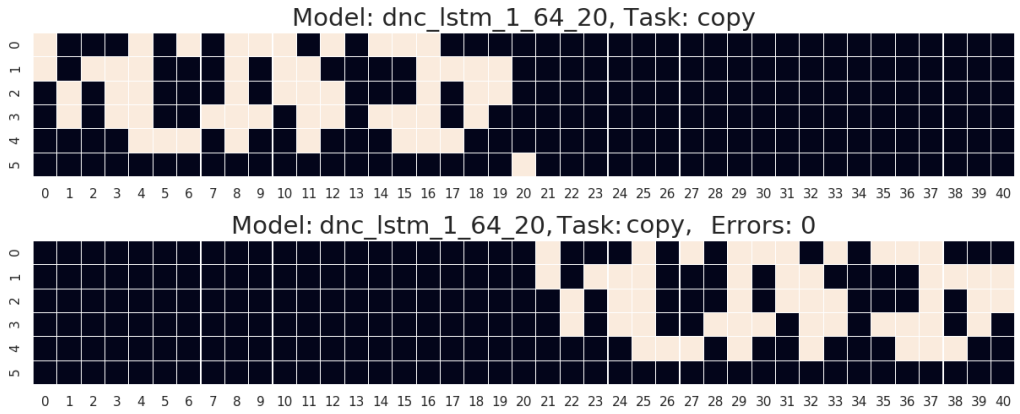

Figure A.4: LSTM DNC on copy task of training set size

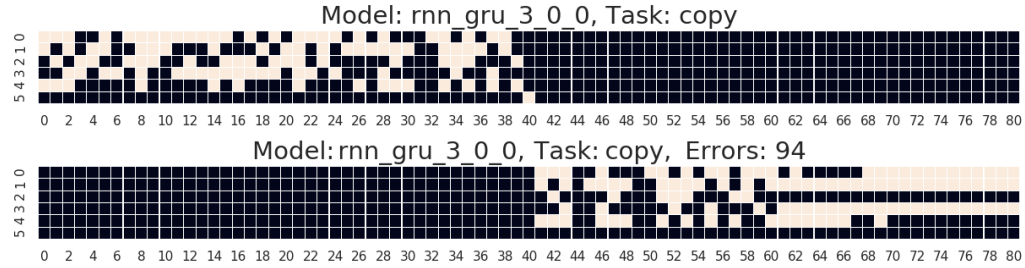

Figure A.5: Regular GRU on copy task on training sets 100% bigger than during training.

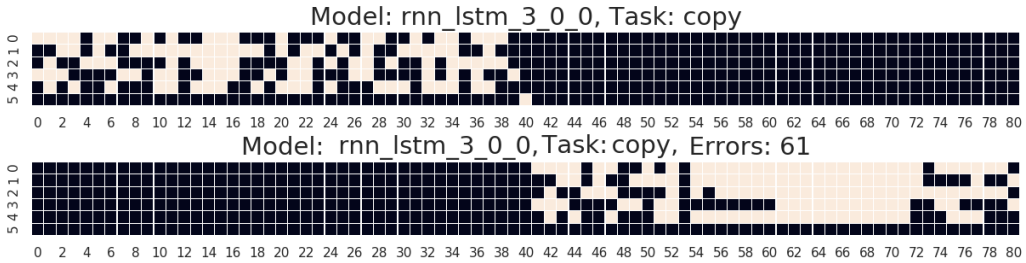

Figure A.6: Regular LSTM on copy task on training sets 100% bigger than during training.

A.2 Figures of training and sampling losses

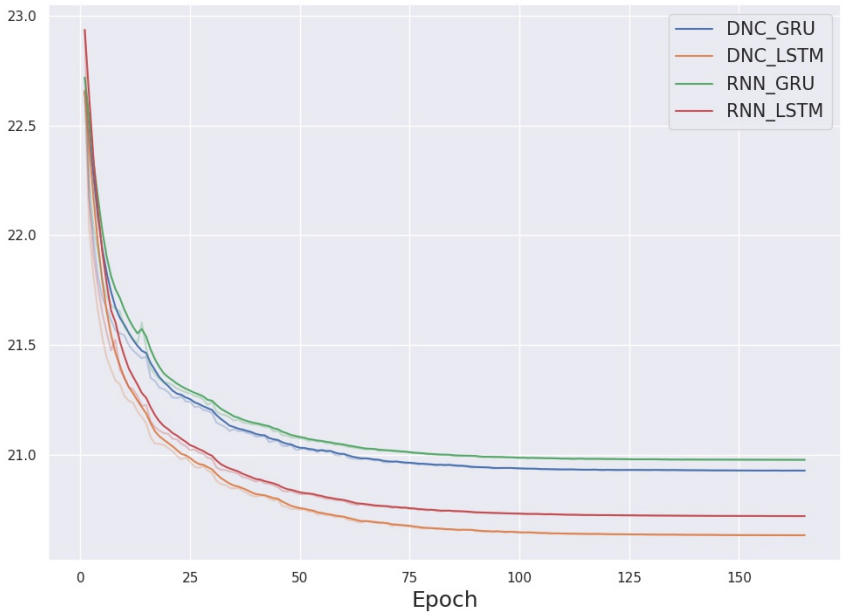

Figure A.7: Average of training NLLs for GDB-13 compounds

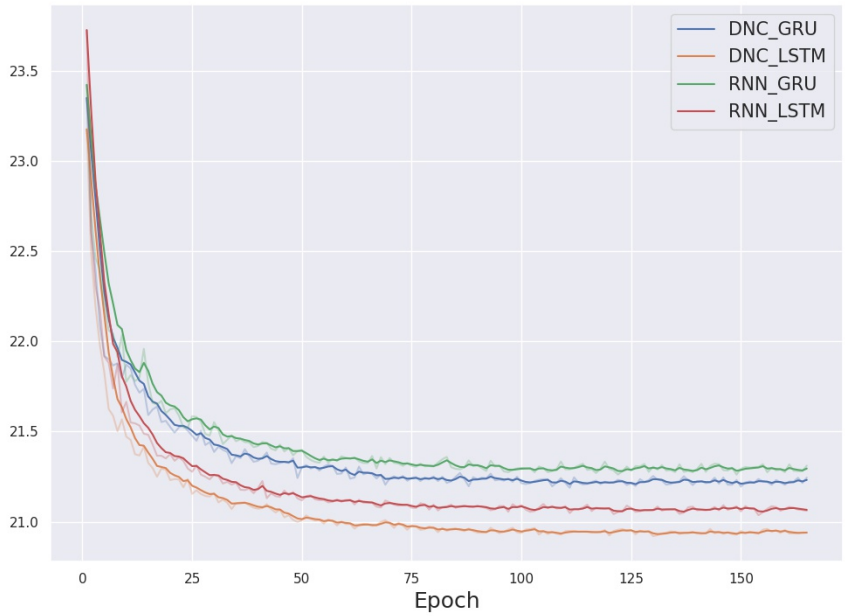

Figure A.8: Average of sampled NLLs for GDB-13 compounds

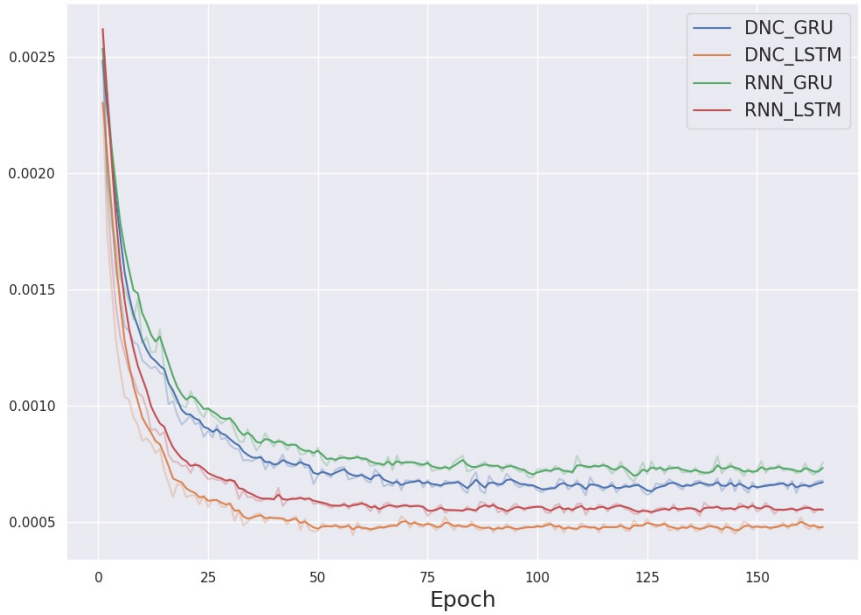

Figure A.9: Jensen-Shannon Divergence between the training and sampled distributions on the GDB-13 training

A.3 Sampling results

	Mean	Variance
Training Set		
Number of heavy atoms	12.854	0.1691
Number of rings	2.123	1.460
Sequence Lengths	22.19	4.754
Number of molecules with at least 4 rings	124790	-
Number of molecules with at most 12 heavy atoms	126584	-
GRU Baseline RNN		
% Valid SMILES	99.20	-
Number of heavy atoms	12.850	0.1788
Number of rings	2.151	1.506
Sequence Lengths	22.215	4.804
Number of molecules with at least 4 rings	129548	-
Number of molecules with at most 12 heavy atoms	130648	-
GRU DNC 32_20_8		
% Valid SMILES	99.26	-
Number of heavy atoms	12.849	0.1826
Number of rings	2.135	1.496
Sequence Lengths	22.221	4.796
Number of molecules with at least 4 rings	126440	-
Number of molecules with at most 12 heavy atoms	131572	
LSTM Baseline RNN		
% Valid SMILES	99.44	-
Number of heavy atoms	12.853	0.1780
Number of rings	2.147	1.503
Sequence Lengths	22.237	4.818
Number of molecules with at least 4 rings	129331	-
Number of molecules with at most 12 heavy atoms	127572	-
LSTM DNC 32_20_8		
% Valid SMILES	99.47	-
Number of heavy atoms	12.844	0.1887
Number of rings	2.138	1.474
Sequence Lengths	22.225	4.755
Number of molecules with at least 4 rings	126534	-
Number of molecules with at most 12 heavy atoms	135364	-

Table A.1: Summary of the sampling. Note that the last two metrics are given in number instead of fraction. This is since it can be misleading to normalize the value to only valid SMILES

	Mean	Variance
Validation Set		
Number of heavy atoms	12.855	0.1682
Number of rings	2.124	1.459
Sequence Lengths	22.19	4.758
Number of molecules with at least 4 rings	125236	-
Number of molecules with at most 12 heavy atoms	125548	-
GRU DNC 8_20_8		
% Valid SMILES	99.31	-
Number of heavy atoms	12.854	0.1753
Number of rings	2.125	1.471
Sequence Lengths	22.199	4.777
Number of molecules with at least 4 rings	123195	-
Number of molecules with at most 12 heavy atoms	127781	-
GRU DNC 16_20_8		
% Valid SMILES	99.33	-
Number of heavy atoms	12.851	0.1801
Number of rings	2.134	1.4963
Sequence Lengths	22.224	4.820
Number of molecules with at least 4 rings	126263	-
Number of molecules with at most 12 heavy atoms	129914	-
GRU DNC 64_20_8		
% Valid SMILES	99.33	-
Number of heavy atoms	12.851	0.1781
Number of rings	2.146	1.5044
Sequence Lengths	22.225	4.810
Number of molecules with at least 4 rings	129231	-
Number of molecules with at most 12 heavy atoms	130048	-

Table A.2: Summary of the Sampling for GDB-13 (part 2)

	Mean	Variance
LSTM DNC 8_20_8		
% Valid SMILES	99.34	-
Number of heavy atoms	12.859	0.1703
Number of rings	2.167	1.515
Sequence Lengths	22.223	4.841
Number of molecules with at least 4 rings	132699	-
Number of molecules with at most 12 heavy atoms	123899	-
LSTM DNC 16_20_8		
% Valid SMILES	99.40	-
Number of heavy atoms	12.865	0.1635
Number of rings	2.146	1.5088
Sequence Lengths	22.205	4.784
Number of molecules with at least 4 rings	129321	-
Number of molecules with at most 12 heavy atoms	119693	-
LSTM DNC 64_20_8		
% Valid SMILES	99.38	-
Number of heavy atoms	12.847	0.1822
Number of rings	2.173	1.4964
Sequence Lengths	22.250	4.777
Number of molecules with at least 4 rings	132233	-
Number of molecules with at most 12 heavy atoms	133833	-

Table A.3: Summary of the Sampling for GDB-13 (part 3)

	Mean	Variance
GRU Baseline RNN		
% Valid SMILES	94.71	-
Number of heavy atoms	26.550	47.05
Number of rings	3.338	1.419
Sequence Lengths	44.436	156.83
Number of molecules with at least 4 rings	159521	-
Number of molecules with at least 35 heavy atoms	45403	-
GRU DNC 32_20_8		
% Valid SMILES	95.10	-
Number of heavy atoms	26.611	47.45
Number of rings	3.330	1.408
Sequence Lengths	44.59	157.29
Number of molecules with at least 4 rings	158751	-
Number of molecules with at least 35 heavy atoms	47339	-
LSTM Baseline RNN		
% Valid SMILES	95.84	-
Number of heavy atoms	26.699	46.55
Number of rings	3.341	1.417
Sequence Lengths	44.762	156.45
Number of molecules with at least 4 rings	161820	-
Number of molecules with at least 35 heavy atoms	47177	-
LSTM DNC 8_20_8		
% Valid SMILES	96.50	-
Number of heavy atoms	26.853	47.29
Number of rings	3.377	1.463
Sequence Lengths	45.00	159.25
Number of molecules with at least 4 rings	167618	-
Number of molecules with at least 35 heavy atoms	50401	-
GRU DNC 8_20_8		
% Valid SMILES	0.94712	
Number of heavy atoms	26.269	44.56
Number of rings	3.298	1.395
Sequence Lengths	44.04	148.34
Number of molecules with at least 4 rings	154828	
Number of molecules with at least 35 heavy atoms	41250	
LSTM DNC 32_20_8		
% Valid SMILES	0.96528	
Number of heavy atoms	26.567	46.00
Number of rings	3.360	1.433
Sequence Lengths	44.57	155.46
Number of molecules with at least 4 rings	164511	
Number of molecules with at least 35 heavy atoms	46323	

Table A.4: Summary of the sampling for ChEMBL