Wesleyan University

Target Specific Drug Design with Deep Reinforcement Learning

by

Theodore Beck Sternlieb
Class of 2022

A thesis submitted to the
faculty of Wesleyan University
in partial fulfillment of the requirements for the
Degree of Master of Arts

Middletown, Connecticut April, 2022

Target Specific Drug Design with Deep RL

Abstract

In this paper we present a deep learning approach to diversifying early leads in drug discovery. By learning a generative model over a space of know druglike small molecules we are able to take know leads and explore the molecular space around them to find chemically similar molecules. We do this by treating a small molecule as a graph where nodes represent different atoms and edges, bonds. Initially, we fit an autoregressive model $P_\theta(\cdot)$ over a dataset of small molecules which learns to build molecular graph atom by atom. We then bias $P_\theta(\cdot)$ using reinforcement learning to place higher probability densities on portions of the molecular space which score well on a set of reward functions. We choose reward functions which value druglike molecules as well as a reward for molecules which have a high binding affinity to a specific protein of interest. Finally after training the model, we sample molecules $x \sim P_\theta(\cdot|M_g \subseteq_g x)$ where M_g, an early lead molecule, is a subgraph of x, to generate novel drug candidates.

If you are the dealer
I'm out of the game
If you are the healer
It means I'm broken and lame
If thine is the glory then
Mine must be the shame
You want it darker
We kill the flame

Leonard Cohen, *You Want It Darker*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The last decade has seen an explosion in the variety and power of machine learning algorithms. Ushered in by advances in theory and hardware, deep neural networks have shown remarkable success over a broad range of tasks and domains from image recognition to protein folding. Of particular note is the field of deep reinforcement learning, or deep RL which augments the techniques of traditional RL with the expressive power of deep neural networks. Companies like DeepMind have proven that these techniques can produce agents capable of competing at, or above, human level in games like Go or Dota 2. One of the key factors in the AI systems is access to perfect simulation. In contrast to humans, or even animals, deep RL is extremely sample inefficient meaning that agents need far more experience than their biological counterparts to achieve comparable performance for a given task. For example, AlphaGo Zero, a Go ai developed by Deepmind played a staggering 29 million games to achieve its super-human level of performance[2]. As Francois Chollet, creator of the popular deep learning framework Keras, put it when retweeting a video of a rat learning to drive a car, "rats generalize better than deep RL, and are more sample efficient."[3] Luckily, similar hardware advances to the ones underpinning deep learning, have allowed for incredibly quick simulation of protein and

molecule interaction. These technologies in conjunction present an exciting opportunity in the realm of drug design. What if instead of building models to discover successful strategies in Go, we instead built models to discover small molecule cancer therapeutics. In this thesis, we seek to apply the tools of deep RL in conjunction with high throughput molecular dynamics to the drug design problem. Specifically we examine using Policy Gradients to find alternative, yet functionally similar molecules to pk-11000 for rescuing P53 Y220C.

Chapter 2

Background

2.1 Notation

Table 2.1: Notation

$G = (V, E)$	$\triangleq$	Graph G with nodes V and edges $E \subseteq V \times V$
$G^+ = (V, E, V_f, E_f)$	$\triangleq$	Graph G as above, where each node has an associated feature vector in V_f and edges an associated type in E_f
$G_m = G^+$	$\triangleq$	Where G^+ was generated by featurizing molecule m
Leaf Atom of Molecule	$\triangleq$	An atom with only one bond which connects is to the rest of the molecule
Leaf Ring of Molecule	$\triangleq$	A ring with only one bond which connects is to the rest of the molecule
y	$\triangleq$	a non bold variable refers to a scalar value
$\mathbf{x}$	$\triangleq$	a bold variable refers to a non scalar value, usually a vector
$\mathcal{G}$	$\triangleq$	a Group
KL	$\triangleq$	Kullback Leibler divergence between two distributions
$D_{KL}(p, q) = \int_x p(x) \log \frac{p(x)}{q(x)} dx$	$\triangleq$	equation for KL divergence
$H(X)$	$\triangleq$	entropy of a distribution
ρ	$\triangleq$	$\rho : G \times X \to X$ a group action with group $\mathcal{G}$ on set $\mathcal{X}$
$Ber(p)$	$\triangleq$	is the Bernoulli distribution with $Ber(x\|p) = \begin{cases} p, & \text{if } x = 1 \\ 1 - p, & \text{if } x = 0 \end{cases}$
$x \sim P(\cdot)$	$\triangleq$	A sample x from distribution $P(\cdot)$
Softmax($\mathbf{x}$)	$\triangleq$	$\left[\frac{e^{x_1}}{\sum_{i=1}^n e^{x_i}} \cdots \frac{e^{x_n}}{\sum_{i=1}^n e^{x_i}}\right]$ The Softmax transforms a vector into a categorical probability distribution

2.2 Lewis Structure of Molecules and Aromatic Rings

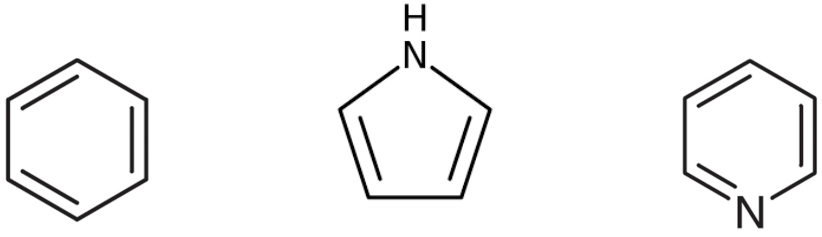
We offer a brief overview of the limited chemical knowledge necessary for understanding this paper. This will essentially boil down to how many bonds an atom can make and the different kinds bonds involved. To avoid Chemistry as much as possible we just include a table of elements to number of bonds.

Hydrogen	1
Nitrogen	3
Carbon	4
Oxygen	2
Sulfur	2
Fluoride	1
Chlorine	1
Sodium	1
Phosphorous	3
Bromine	1
Silicon	4
Boron	3
Selenium	2
Potassium	1

Table 2.2: Number of Bonds an atom can make

We also briefly mention the concept of aromatic rings. Aromatic rings are rather tricky, so we just provide a list of possible aromatic rings and comment that they have desirable properties for drug design as a sort of structural skeleton on which to build.

Figure 2.1: Aromatic Rings: Benzine, Pyrolle, Pyridine



2.3 Small Molecules

Small Molecules are a class of molecule which weight somewhere in the realm of 100 to 900 daltons which include common drugs like aspirin or caffeine. These molecules are often able to bind to proteins in our body as well as being more bioavailable (able to enter blood stream and have an active effect) than larger molecules. As such they are often a good class of molecules to look at when seeking new leads in drug discovery. The relatively small size of these molecules also vastly shrinks what is a humongous combinatorial realm of all drug like molecules, simplifying the task of fitting drug generative models. In our case, we consider small molecules that weigh between 200 and 300 daltons, or, around 15-30 atoms.

2.4 p53 and Y220C

The protein p53 plays a crucial role in the regulation of the life and death of cells in mammals. Due to its important role, mutations in the genes which encode for p53 can have disastrous effects on the body's susceptibility to cancer. In fact some form of mutant p53 is found in over 50 percent of all human cancers. We focus here on the specific Y220C mutation which although is only the 9th most common mutations of p53 is the most common one to occur outside the DNA binding domain[4]. As the mutation occurs far from the DNA binding site, the mutant protein is an appealing candidate for treatment with small molecule drugs. Although the overall approach detailed in this paper is relatively agnostic to choice of target, we use p53 with the Y220C mutation as a case study for its effectiveness

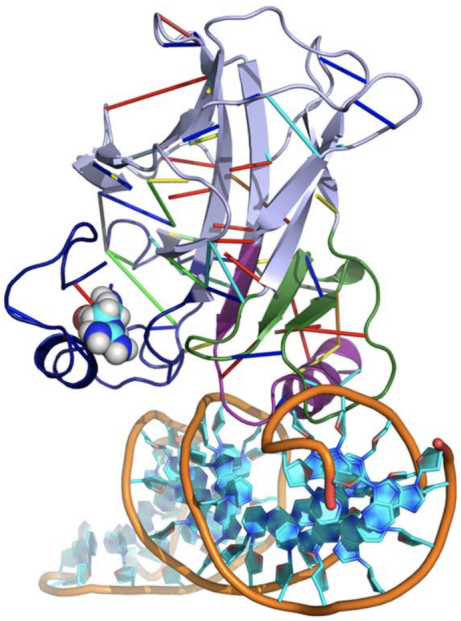


Figure 2.2: p53 bound with DNA complex [1]

2.5 Docking

Molecular Docking is a simulation technique developed for quickly predicting binding affinities between two molecules, often a macro-molecule such as a protein and a small drug molecule. In a typical docking program, first a score function will be defined, a sort of proxy for the potential energy of the system, which maps the system to some scalar value. Next the docking system moves around either molecule to try and find minima of the scoring function. Molecules with low associated scores are likely to bind well to the receptor molecule. Docking tools are used widely throughout industry when in the early stages of drug development. For example, if one was to search for potential small molecules to cure some deficient protein, they might run a docking procedure across small molecule datasets with millions of entries to identify a few potential leads.

Chapter 3

Neural Networks

3.1 Deep Neural Networks

Within the realm of deep learning, neural networks have become the standard algorithm for most problems. Although there is an expansive environment of niche architectures for any specific domain or task, most are at their core, reformulations and extensions of Multilayered Perceptrons (MLPs) trained with some sort of first order gradient based optimization. We now define the standard components, or layers, of an MLP.

We define a dense layer, $f_{dense} : \mathbb{R}^m \to \mathbb{R}^n$, of an MLP as an affine transform (a linear transformation followed by translation) as

$$f_{dense}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$

where $\mathbf{W} \in \mathbb{R}^{n \times m}$ and $\mathbf{b} \in \mathbb{R}^n$ are referred to as the learnable weights or parameters of the model. We next define a non-linear layer f_{nl} as the pointwise application of some non-linear function to a vector.

typical non-linearities	
Logistic Sigmoid	$f(x) = \frac{1}{1+e^x}$
ReLU	$f(x) = \max(0, x)$
LeakyReLU	$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ -ax, & \text{otherwise} \end{cases}$ for $0 < a < .3$

Formally we define the class of basic MLPs as

$$F_{\text{MLP}} = \{f_\theta^1 \circ f_{nl}^1 \circ f_\theta^2 ... f_\theta^n | n \in \mathbb{N}\}$$

By interweaving dense and non linear layers we are able to generate a rich class of functions that are able to approximate any function of interest given sufficient size [5]. We refer to the dimension of the co-domain of some layer f_θ^n as the width of the layer and refer to the number of layers in the network as the model's depth. Further, we might refer to the different indices of $f_\theta^n(\mathbf{h})$ as the different neurons in the layer and the value at an index i as the activation of the i-th neuron. These terms are more natural when thinking of MLP's as networks crudely fashioned in the image of brains. We try to avoid this representation of MLPs as we find it restrictive as well as adding unnecessary confusion.

3.1.1 Training Deep Neural Networks

Having defined an expressive class of parameterized functions we now move to their uses. Consider a dataset $D = \{\{\mathbf{x}_i, y_i\}_{i=1}^N\}$. We are often tasked with finding parameters θ to minimize or maximize some function of the sort $\mathcal{L}(D, \theta)$. By convention, we usually minimize these objectives and refer to them as loss functions. We also may assume the D term and merely write

$\mathcal{L}(\theta)$. Thus, most problems we use Neural Networks for are of the form

$$\boldsymbol{\theta} = \arg\min_{\theta} \mathcal{L}(D, \theta)$$

Particularly, we are often interested in finding the parameters which assigns the highest probability to our dataset. This approach is called Maximum Likelihood Estimation and is defined as

$$\boldsymbol{\theta}_{mle} = \arg\max_{\theta} p(D|\theta) \tag{3.1}$$

We see that we can rewrite, assuming that samples from D were generated independently of one another, $p(D|\theta)$ to get

$$\boldsymbol{\theta}_{mle} = \arg\max_{\theta} p(D|\theta) = \arg\max_{\theta} \prod_{i=1}^{N} p(y_i|\mathbf{x}_i, \theta)$$

As function optima are preserved under logarithms we can further rewrite to get

$$\boldsymbol{\theta}_{mle} = \arg\max_{\theta} \sum_{i=1}^{N} \log p(y_i|\mathbf{x}_i, \theta)$$

Negating this objective, we produce the negative log likelihood objective (NLL)

$$\text{NLL}(\theta) = -\sum_{i=1}^{N} \log p(y_i|\mathbf{x}_i, \theta)$$

and recover

$$\theta_{mle} = \arg\min_{\theta} -\sum_{i=1}^{N} \log p(y_i|\mathbf{x}_i, \theta)$$

We may then choose different parameterizations of $p(y_i|x_i, \theta)$ to create suitable loss functions for different goals. For example in the case of modelling a two class classification problem, we parameterize $p(y|x, \theta)$ as

$$p(y|x, \theta) = \text{Ber}(y|\sigma(f_\theta(x))$$

from which we can derive the binary cross entropy objective.

To minimize the NLL objective, or any other fully diferentiable objective function $\mathcal{L}(\theta)$ we take the gradient of our objective function with respect to θ, $\nabla\mathcal{L}(\theta) = \begin{bmatrix} \frac{\partial\mathcal{L}(\theta)}{\partial\theta_1} \\ \vdots \\ \frac{\partial\mathcal{L}(\theta)}{\partial\theta_n} \end{bmatrix}$ to find the change of parameters which minimize our loss function. We then update the parameters by moving the parameters in the direction of the gradient.

Algorithm 1: ***Gradient Descent***

Input: D: Dataset; θ: Paramaters of f_θ; P: Distribution over θ; γ: Learning Rate; *epochs*: number of steps
Output: θ optimized on D
$\theta \sim P$
for *t = 1,2, ... epochs* **do**
  Evaluate $\mathbf{g}_t = \nabla\mathcal{L}(\theta)$
  $\theta_{t+1} \leftarrow \theta_t + \gamma\mathbf{g}_t$
return θ_t

We can also estimate the gradient by sampling the objective function across portions of the dataset called batches. We then repeat this process for a number of epochs. We define

$$\mathcal{L}_{[m:n]}(\theta) = \mathcal{L}(D_{[m:n]}, \theta)$$

as the loss over a portion of the dataset to get Minibatch Stochastic Gradient Descent. Combined with more advanced update rules like Adam[6] or RMSProp, Minibatch Stochastic Gradient Descent makes up most optimization techniques for Deep Learning.

3.2 Layers

Here we discuss alternatives to dense and non-linear layers which allow for more efficient training, more powerful models and architectures built

Algorithm 2: ***Minibatch Stochastic Gradient Descent***

Input: D: Dataset; θ: Paramaters of f_θ; P: Distribution over θ; γ: Learning Rate; *epochs*: number of step; b: batch size
Output: θ optimized on D
$\theta \sim P$
$step = 0$
for t = *1,2, ... epochs* **do**
  for j = *1,2, ...* $\lfloor \frac{len(d)}{b} \rfloor$ **do**
    Evaluate $\mathbf{g}_{step} = \nabla L_{[j \cdot b:(j+1) \cdot b]}(\theta)$
    $\theta_{step+1} \leftarrow \theta_{step} + \gamma \mathbf{g}_{step}$
    $step \leftarrow step + 1$
  Shuffle(D)
return θ_{step}

specifically for inputs with added geometric structure.

3.2.1 Gated Recurrent Units

Gated Recurrent Unites are a simplification of the typical Long Short-Term Memory layers popular in deep learning for sequential data [7]. The layer works as follows for hidden state h_{t-1} and input x

$$z_t = \sigma(W^z(x_t) + U^z(h_{t-1})$$

$$r_T = \sigma(W^r(x_t) + U^r(h_{t-1})$$

$$\hat{h}_t = tanh(x_t * z_t + W_g(r_t \odot h_{t-1}))$$

$$h = z_t \odot h_t + (1 - z_t) \odot \hat{h}_t$$

3.2.2 Normalization

In training deep neural networks, it is often useful to normalize neuron activations in between hidden layers [8]. We consider two main normalization strategies in our search over architectures, spectral normalization [9] and layer normalization [10]. We do not consider batch normalization as its

assumption of a stationary dataset is not guaranteed in the reinforcement learning setting. In fact, if the RL agent behaviour is improving at all, we expect the underlying distribution over states from episode to episode to change. Additionally, batch norm becomes brittle for inference when only single states are considered at a time.

3.2.2.1 Layer Normalization

In a layer normalization layer, mean and variance are calculated across neuron activations, given by

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad\qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2}$$

The normalized activations becomes

$$\mathbf{a}' = \frac{\boldsymbol{g}}{\sigma} \odot (\boldsymbol{a} - \mu) + \boldsymbol{b}$$

Where **g** and **b** are learned parameters. These parameters help avoid the issue Sigmoid functions look linear close to 0.

3.2.2.2 Spectral Normalization

Spectral normalization is a normalization technique from GAN literature which has shown to be useful in stabilizing policy gradient methods [11] [12]. The spectral norm of a weight matrix is calculated as

$$\sigma(\mathbf{A}) = \max_{\|h\|_2 \leq 1} \|\mathbf{A}h\|_2$$

$$W_{sn} = \frac{W}{\sigma(W)}$$

and helps to control the lipschitz constant, or smoothness, of the function.

3.2.2.3 Dropout

Dropout is a commonly used method to reduce overfitting in the parameters of a neural network in which a random selection of entries in the vector output of a layer are temporarily zeroed during each pass through the network [13]. We define a dropout layer with dropout rate r as

$$f_{\text{dropout}}(\mathbf{x}) = \mathbf{x} \odot \begin{bmatrix} r_0 \sim \text{Ber}(r) \\ | \\ r_n \sim \text{Ber}(r) \end{bmatrix}$$

Where r parameterizes the Bernoulli distribution.

At inference time, we turn dropout off resulting in a model that can be loosely thought of as an ensemble off all the sub-networks in the model. We must also consider the outputs of dropout layers during inference time. We see the expected activation for a neuron z_i following a dropout layer is

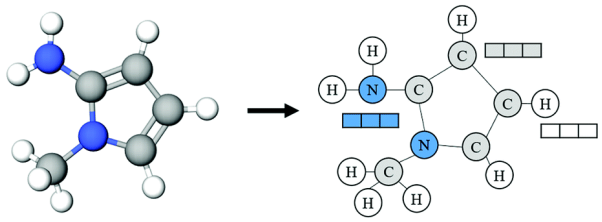
$$\mathbb{E}_{Ber(p)}[z_i] = z_i(1-p) + 0(p) = z_i(1-p)$$

Hence, to maintain our expectation during inference time, we rescale the activations of each neuron by $1-p$. We note that for reinforcement learning, the noisiness dropout introduces has a negative impact on training stability and so all dropout layers are switched to inference mode during RL fine tuning.

3.3 Graph Neural Networks

Stemming from the success of NN for tabular data, vector inputs with no geometric structure, it is natural to try and apply these techniques to

data with richer geometry. One such class of data is graphs. Many real world phenomena (molecules, social networks, ...) can be modeled with graphs and as such there has been extensive interest in deep learning on graphs. However, this is not without its difficulties. Let us examine the shortcomings of using a traditional MLP architecture on graphs to motivate our discussion of graph specific architectures. As an example we consider a small molecule.



Molecular or crystal structure Molecular graph

We see a graph of a molecule where each node/atom is assigned a feature vector which represents something about the node, say the element, and edges represent bonds.

Let's imagine how we might feed this information into a standard MLP. As a first thought we could stack all the feature vectors and use this as an input vector, however we would need to fix the number of nodes in the graphs we look at to ensure input vectors with constant shapes and we would have also lost all structural information. To handle variable input size, we could use recurrent neural networks, (RNNs) an MLP variant built for variable length sequence inputs, pick an ordering of nodes, and feed the network one feature vector at a time to get an embedding of the graph. Although this solves our variable length input issue we have to rely on an arbitrary order of nodes in which to feed out feature vectors in and we still

haven't included any information about the edges in our graph.

3.3.1 Groups, Symmetry and Graphs

Ultimately we would like whatever network we choose to be loyal the underlying structure of graphs as well as use a strong enough inductive bias for graph structured data that learning is possible. To derive our architecture, let us introduce the idea of a symmetry group over some domain. A symmetry group $\mathcal{G}$ is a set of transformations on some geometric domain $\mathcal{X}$ which preserve some aspect of the input. We also define a group action $\rho : \mathcal{G} \times \mathcal{X} \to \mathcal{X}$ such that for $g \in \mathcal{G}, \rho(g)(x)$ is the the group action applied to x. In our case, this represents applying one of our symmetry operations to x. We now define the family of $\mathcal{G}$-invariant functions on geometric domain $\mathcal{X}$, $\mathcal{F}$ as

$$\mathcal{F}_i^{\mathcal{Y}} = \{f : \mathcal{X} \to \mathcal{Y};\ \forall g \in \mathcal{G},\ \forall x \in \mathcal{X}, f(x) = f(\rho(g)(x))\}$$

Essentially this is the set of functions whose outputs do not change if an element of the symmetry group is applied to the input. Similarly we define the family of $\mathcal{G}$-equivariant functions

$$\mathcal{F}_e^{\mathcal{Y}} = \{f : \mathcal{X} \to \mathcal{Y};\ \forall g \in \mathcal{G},\ \forall x \in \mathcal{X}, f(\rho(g)(x)) = \rho(g)f(x)\}$$

We now move to define these families in the context of graphs. Let $\mathcal{X}$ be the set of signals over graphs and $\mathcal{G}$ to be a symmetry group over $\mathcal{X}$ whose members are permutations of node ordering. Examples of $f \in \mathcal{F}_i^{\mathcal{Y}}$ might be a sum or average over nodes. For useful examples of $f \in \mathcal{F}_e^{\mathcal{Y}}$ some more work is required. Essentially, we would like to extend the notion of filters and locality from Convolutional Neural Networks to graphs.

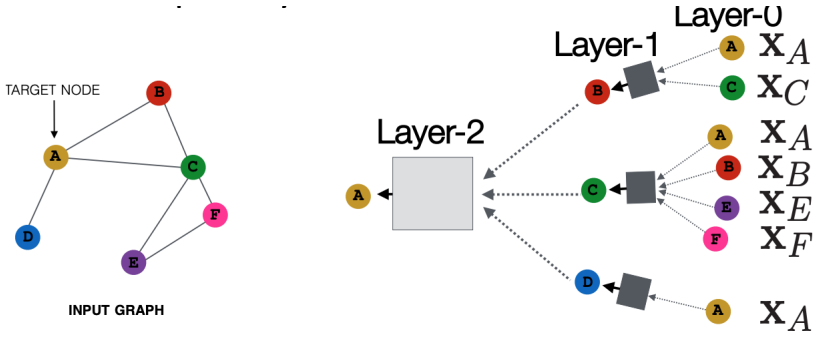


Figure 3.1: Multiple Layers of Message Passing [15]

3.3.2 Message Passing

We now introduce the idea of message passing which helps unify the class of Graph Neural Networks [14]. In the message passing framework, the hidden state of nodes are updated as a function of a nodes current hidden state as well as the hidden states of its neighborhood. More formally, let $h_i \in \mathcal{R}^n$ be the hidden state of node v_i. We define the set of messages to v_i as

$$M_i = \{f_{message}(h_i, h_u, e_{iu}) | v_u \in \mathcal{N}(v_i)\}$$

h_v is then updated as

$$h'_v = U(h_v, g(M_i))$$

where g is some order invariant function like an average or sum. Let f_{mpnn} be functions on graphs which apply these steps to each node. We see that as neighborhoods are equivariant under node permutations $f_{mpnn} \in \mathcal{F}_e^{\mathcal{Y}}$. By paramaterizing $f_{message}$ and U we create a fully diferentiable Graph Neural Network Layer. As we pass a graph through multiple layers of message passing, individual node features become functions of larger and large neighborhoods. After a number of message passing layers, we can also optionally use a permutation invariant readout layer, like a sum or average across node features, to obtain a graph level representations. When we parameterize $f_{message}$ with a single layer perceptron, we get Graph Convo-

lution Networks.

3.3.3 Relational Graph Convolution Networks

Relational Graph Convolution layers are a simple extension of graph convolutional neural networks to the multi edge setting. We consider this version of graph convolution both for its simplicity and for how it informs adapting other more complicated graph architectures to multi edge graphs. We define the update rule for node i as

$$h_i' = \sigma\left(\sum_{r\in R}\sum_{j\in\mathcal{N}_r(i)}\frac{1}{c_{i,r}}\mathbf{W}_r h_j + \mathbf{W}_0 h_i\right) \text{[16]}$$

where $c_{i,r}$ is some normalization factor. We can think of this in the language of message passing where for node i, $j \in \mathcal{N}(i)$ and $e_{i,j}$ having type r, we have

$$f_{message}(h_i, h_j, e_i j) = \frac{1}{c_{i,r}}\mathbf{W}_r h_j$$

and

$$U(h_i, g(\mathcal{M}_i)) = \sigma\left(\mathbf{W}_0 h_i + \sum_{m\in\mathcal{M}_i} m\right)$$

In general we see that we can simply adapt any graph architecture to work with multi edges as

$$\mathcal{M}_i = \bigcup_{r\in R}\mathcal{M}_i^r$$

where $\mathcal{M}_i^r$ is calculated as it would be in the original architecture while considering a subgraph of the original containing only edges of type r.

3.3.4 Gated Graph Sequence Neural Networks

Gated Graph Sequence Neural Network, or, GGNNs combine GRUs with the GNN message passing paradigm [17]. Where as the standard message

passing function maps a hidden representation and edge type to a vector $f_{message}(h_i, e_m) = \mathbf{W}_m h_i + \mathbf{b}_m$, GGNNs augment the message passing function with a GRU such that

$$a_i^t = \sum_{j \in \mathcal{N}} \mathbf{W}_m h_j^t + \mathbf{b}_m$$

$$h_i^{t+1} = \mathbf{GRU}_m(a_i^t, h_i^t)$$

To handle the multiple edge types in our graph, we follow the procedure outlined as above.

Chapter 4

Reinforcement Learning

4.1 Traditional Reinforcement Learning

Reinforcement Learning (RL) is a subfield of Machine Learning in which one tries to find optimal solutions for non-differentiable functions by interacting with an environment. To highlight the distinction between reinforcement learning and other machine learning tasks, lets consider a supervised regression problem. Our model and loss takes the form

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|w_0 = \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}, \sigma^2) \qquad \mathcal{L}(\boldsymbol{\theta}) = \sum_{n=1}^{N}(y_n - \boldsymbol{w}_{\boldsymbol{\theta}}^{\boldsymbol{T}}\boldsymbol{x}_{\boldsymbol{n}})^2 \tag{4.1}$$

which we fit by computing

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \frac{1}{N}\sum_{n=1}^{N}(y_n - \boldsymbol{w}_{\boldsymbol{\theta}}^{\boldsymbol{T}}\boldsymbol{x}_{\boldsymbol{n}})^2$$

over some dataset. In most approaches to fitting this equation, we make use of some variant of gradient descent. However for many problems, we are met with non-differentiable objectives which we would like to optimize. These types of challenges often come up within the context of simulated

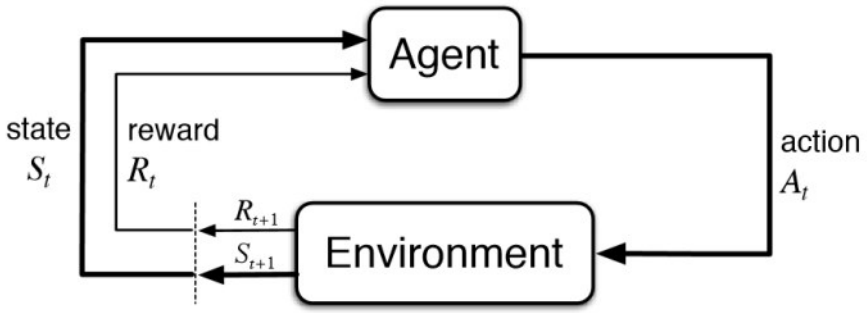


Figure 4.1: Usual RL setup

environments. In the typical RL setting, an agent within some simulated environment will take actions to interact with the environment and receive some sort of reward based on the quality of said action. By interacting with the environment over a long enough period of time, the agent learns to optimize the reward it receives.

Let A be the set of actions which our agent can take, S be the set of states in the environment, $P(s', r|s, a)$ be the transition function which defines a probability distribution over next state and reward produced by taking action a in state s, and $\pi(\cdot|s)$ be a probability distribution over all actions conditioned on state $s \in S$. π is usually referred to as the policy, or the agent, and it controls which actions our agent takes. When interacting with the environment at timestep t_i, an action is sampled from the distribution produced by π. The next state, as well as the reward, is then generated by the transition function. With this, we can generate a trajectory $\tau = (s_0, a_0, r_0, ..., s_n, a_n, r_n)$ where repeatedly the agent will take an action and receive a reward, and the environment will update accordingly. We call this process of generating a, or many, trajectories a rollout.

Typically we would like our agent to learn to maximize some idea of cumulative reward. For some time step t_i of trajectory τ, we define return

as

$$G_t = \sum_{i=t}^{T} r_i$$

We may also make use of a discounted return which converges as long as r_i behaves appropriately.

$$G_t = \sum_{i=t}^{T} \gamma^i r_i$$

where $0 < \gamma \leq 1$. This discounting of future reward makes additional intuitive sense. A reward at a given time step was probably produced by more recent actions than those farther back in time.

We define some more helpful notation.

$$v^{\pi}(s) = E_{\pi}[G_t | s_t = s] = \sum_{s',r} p(s', r'|s, a)(r' + v^{\pi}(s'))$$

$$q^{\pi}(s, a) = E_{\pi}[G_t | s_t = s, a_t = a]$$

We define a reward function r as well as the total reward of a trajectory as such

$$r(s_i, a_i) = r \qquad\qquad R(\tau) = \sum_{i=1}^{N} r(s_i, a_i) \tag{4.2}$$

We see that the probability of a given trajectory under policy π is

$$P(\tau|\pi) = \prod_{i=1}^{N} \pi(a_i|s_i)$$

We define a utility function $J()$ which measures the performance of our policy as the expected reward under a given policy π as

$$J(\pi) = E_{\tau \sim \pi_\theta}[R(\tau)] = \int_{\tau} P(\tau|\pi) R(\tau)$$

Our goal thus becomes finding

$$\arg\max_{\boldsymbol{\pi}} J(\pi)$$

4.2 Deep Reinforcement Learning

Here we move our attention to deep reinforcement learning. Many of the strategies used in deep RL involve augmenting more old school RL methods with the power of neural networks. For example, deep Q-learning merely replaces a tabulated approach to finding $Q(s, a)$ with function approximation using deep neural networks. In this section we look at Policy Gradient methods, a technique in which a policy π parameterized by a neural network with parameters θ is directly fit using experiences generated from interacting with the environment

4.2.1 Policy Gradient Methods

We now turn our attention to policy gradient methods. Although these methods are by no means ubiquitous within the realm of Deep RL, the RL algorithm used in the paper is of this family. The overarching idea of Policy Gradient methods is to parameterize our policy with a neural network and then directly optimize our policy with gradient based methods. We use π_θ to refer to a policy parameterized by θ. However, this is not a straight forwards task. We see that performance of our agent depends not only depends on our action selection, but additionally on the stationary distribution over states induced by π_θ which is often unknown.

We recall our utility function J which we would like to maximize

$$J(\theta) = \int_\tau P(\tau|\pi)R(\tau)$$

Can we maximize this objective directly using gradient descent? We note the following useful result of the chain rule from calculus as well as the Leibniz integral rule

$$\nabla f(x) = f(x)\nabla \log f(x) \qquad \nabla_\theta \int_\tau f(\theta, \tau) d\tau = \int_\tau \nabla_\theta f(\theta, \tau) d\tau \tag{4.3}$$

We proceed with calculating $\nabla_\theta J(\theta)$

$$\nabla_\theta J(\theta) = \nabla_\theta \int_\tau [R(\tau) p_\theta(\tau)] \, d\tau$$

Moving ∇ underneath the integral and using our log trick we get

$$\nabla_\theta \int_\tau [R(\tau) p_\theta(\tau)] \, d\tau = \int_\tau [R(\tau) p_\theta(\tau) \nabla_\theta \log p_\theta(\tau)] \, d\tau$$

We now consider the $\nabla_\theta \log p_\theta(\tau)$ term

$$\nabla_\theta \log p_\theta(\tau) = \nabla_\theta \log \left[p(s_1) \prod_t \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t) \right]$$

Expanding the product under the log we get

$$\nabla_\theta \log p(s_1) + \sum_t [\nabla_\theta \log \pi_\theta(a_t|s_t) + \nabla_\theta \log p(s_{t+1}|s_t, a_t)]$$

We see that $\nabla_\theta \log p(s_1) = 0$ and $\nabla_\theta p(s_{t+1}|s_t, a_t) = 0$ as neither are functions of θ and thus get

$$\nabla_\theta \log p_\theta(\tau) = \nabla_\theta \log \sum_t \pi_\theta(a_t|s_t)$$

Hence

$$\nabla_\theta J(\theta) = \int_\tau \left[R(\tau) p_\theta(\tau) \nabla_\theta \log \sum_t \pi_\theta(a_t|s_t) \right] d\tau$$

We now recover an expectation

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[R(\tau) \sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)) \right]$$

We can thus approximate $\nabla_\theta J(\theta)$ by sampling trajectories under policy π_θ and use them to estimate the gradient.

One of the simplest algorithms to make use of policy gradients is REINFORCE, where one samples trajectories under the policy π_θ approximates the gradients and updates θ using gradient descent. The $\nabla_\theta \log \pi_\theta(a_t|s_t))$

Algorithm 3: *REINFORCE*

Input: π_θ: a policy parameterized by θ,
env: an environment in which the agent acts,
rollout: a function which produces a set of trajectories
Output: π_θ optimized on environment *env*
for *t = 1,2, ... epochs* **do**
  experiences $\leftarrow$ *rollout*(π_θ, *env*)
  Evaluate $\mathbf{g}_t = \sum_\tau R(\tau) \sum_t \nabla_\theta \log \pi_\theta(a_t|s_t))$
  $\theta_{t+1} \leftarrow \theta_t + \gamma \mathbf{g}_t$
return π_θ

has a nice interpretation as a version of max likelihood estimation 3.1 where we weight each example with the total reward for that run. Importantly a negative reward for a run would minimize the likelihood of all actions in that run. Still, this method suffers from a few drawbacks. First, it is incredibly sample inefficient as you can only update θ once per rollout. Second, is what is commonly referred to as the credit assignment problem. In the current formulation we increase the likelihood every action taken across a positive trajectory uniformly. We would like to be able to pinpoint what exact move or set of moves taken in a trajectory was responsible for what part of the reward. One good tweak is to replace $R(\tau)$ with $Q^\pi(s, a)$ as any step taken after receiving a reward did not contribute to the past reward

as well as subtracting a baseline,

$$b = \mathbb{E}_{\tau \sim \pi_\theta} [R(\tau)]$$

from the reward to get

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)(Q^{\pi_\theta}(s_t, a_t) - b) \right]$$

which gives a lower variance gradient estimate without adding bias. However there are still improvements to be made. To solve these issues, we introduce two new methods in the next section, Proximal Policy Optimization[18] and Actor Critic Methods.

4.2.2 Actor Critic Methods

Actor Critic methods are a natural extension to the baseline trick talked about above where we formulate our notion of reward with value function approximation. Consider the advantage function

$$A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s, a)$$

We see that A^{π_θ} gives us the difference between how well the agent did having take a given action and how well the agent would typically do. Intuitively, this value represents how much better or worse the actor did by taking action a. Thus we can reformulate our gradient as

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t) \right]$$

To estimate A^{π_θ} we introduce a second network called the critic which we fit using the generated trajectories.

4.2.3 Proximal Policy Optimization

Proximal Policy Optimization Algorithms [18] seek to overcome the sample inefficiency of other policy gradient methods by using a surrogate loss to allow repeated policy updates using the same set of experiences. Let us first consider what would happen if we tried to update our policy using mini-batches or multiple epochs from a single rollout. Consider that after one update to the policy π_θ, generating π'_θ, $p_\theta(\cdot) \neq p'_\theta(\cdot)$.Hence, We can see that we can no longer use the trajectories which we have generated to properly estimate the gradient. However, we can make use of a tool from importance sampling to allow us to repeatedly update using samples generated from the old policy. First, we see how to transform expectations of a function from one distribution to another

$$\mathbb{E}_{x\sim p(\cdot)}\left[f(x)\right] = \mathbb{E}_{x\sim q(\cdot)}\left[\frac{q(x)}{p(x)}f(x)\right] \tag{4.4}$$

Thus we see of

$$J(\theta) = \mathbb{E}_{\tau\sim p_\theta(\cdot)}\left[R(\tau)\right] = \mathbb{E}_{\tau\sim p_{\theta_{old}}(\cdot)}\left[\frac{p_\theta(\tau)}{p_{\theta_{old}}(\tau)}R(\tau)\right]$$

Using a similar derivation as the original policy gradient we get

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_t \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t)\right] \text{[19]}$$

As the distance between the policies which π_θ and $\pi_{\theta_{old}}$ parameterize grows, the importance sampling based estimation grows noisier and noisier. To ensure that the policy can't be too greedy in its updates during training, the objective is reshaped by enforcing that the policy doesn't change too drastically. For a single policy action we get a new objective

$$L^{CLIP} = \min(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}A^{\pi_\theta}(s_t, a_t), \text{clip}(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon)A^{\pi_\theta}(s_t, a_t))[18]$$

the standard Proximal Policy Optimization objective, where the clip operator constrains the first argument to fall in the range of the second and third.

Using PPO in conjunction with Actor Critic methods produces an algorithm that performs well in a variety of environments and it is the RL algorithm which we choose to use. Although this version of PPO works well, we additionally include further optimization which we discuss in our methods.

Chapter 5

Related Work

5.1 Deep Learning Drug Generation

Here we provide a brief overview of deep generative methods for drug design and diversification, as well as more broadly, deep graph generative models.

One of the older models for generating novel drugs is druGAN, an adversarial auto encoder (AAE) which operate on molecular fingerprints (vector representation of graphs custom made for molecules) [20]. The AAE learns in a similar way to a vanilla autoencoder, where an encoder network compresses inputs to a latent variable which is then decoded by the decoder network to produced a vector similar to the input. An additional adversarial objective is added in which a discriminator tries to distinguish between encoded latent variables and samples drawn from a normal distribution of the same dimension.

As the paper points out, there are a few drawbacks of this method. First, molecular fingerprints do not capture the molecular structure as well as other representations might, namely graphs or SMILES (text encodings of molecules). Additionally there is no real way to bias the molecules under some desired property

MolGan is a Generative Adversarial Network [21] in which a Generator Network takes a noise vector and transforms into a dense $n \times n$ probability distribution over possible adjacency matrices and an $n \times t$ probability distribution over atom types. Both these distributions are sampled from to generate a real adjacency matrix and list of atom types. These two representations are then used to generate an actual graph with which to train the convolutional graph neural network discriminator. Additionally a reward network learns to estimate a variety of rewards for molecular properties. As the reward network is fully differentiable, the generator can be trained with the gradient of the reward network to maximize reward. Although able to optimize some molecular rewarding function, one has to constrain the size of considered molecules.

GraphRNN allows for non fixed graph size by employing an autoregressive generative strategy which iteratively grows a graph over a set of steps [22]. At each timestep t the network produces an adjacency vector which encodes which previous nodes are connected to a new node n_t. The network is trained using MLE on a set of graph fragments obtained through a breadth first search decomposition scheme. This decomposition scheme also produced a weakly canonicalized ordering scheme of graph building steps.

Most relevant to our work are Graph Convolutional Policy Networks for Goal Directed Graph Generation[23]. This work uses a similar RL scheme for graph generation which seeks to maximize a set of rewards in an environment and builds graphs in an iterative fashion. They also make use of a discriminator network trained on generated molecules as well as know drugs to bias the RL agent towards druglike molecules.

Chapter 6

Methods

6.1 Graph Building

6.1.1 Generation Scheme

We choose to model the drug generation task as a sequence prediction problem. Specifically we cast a molecule as a sequence of graph building steps which iteratively add new structural elements to an initial starting molecule until a final molecule has been generated. Let $\mathbf{M}$ be the set of all small molecules, Σ be a set of graph building steps, and Σ^* correspond to the set of all sequences over Σ. We define a function

$$h : \mathbf{M} \rightarrow P(\Sigma^*)$$

such that for $m \in \mathbf{M}$, we have

$$h(G_m) = \{(\sigma_1^1, ...\sigma_n^1), (\sigma_1^2, ...\sigma_m^2), ..., (\sigma_1^x, ...\sigma_l^x)\}$$

where for $\tau \in h(G)$, τ corresponds to a graph building sequence resulting in graph G. We note that the co-domain of h is the powerset of sequences as there could be multiple graph building sequences for any one molecule.

6.1.2 Graph Building Actions

We define the our most general formulation of the Graph Building Action space below, but we will further restrict this to help our model. Given a graph $G = (V, E)$, under some ordering π, we define legal actions to be a set of node or cycle additions as well as edge additions between v_i, v_j for $v_i, v_j \in V$. Keeping the set of node/cycle additions fixed across all graphs, we see that for the set of legal edge additions $\mathbf{E}$, $|\mathbf{E}| = \frac{|V|(|V|-1)}{2}$. Thus our action space becomes a function of the size of our graph. To specify our action space, we define some $f_{action} : G' \to \mathcal{P}(\mathcal{A})$ where G' is the set of all molecular graphs and $\mathcal{A}$ is the set of all possible actions for all graphs.

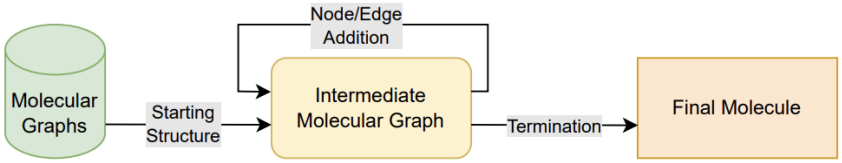


Figure 6.1: Generative procedure for molecular graphs.

To define f_{action}, we let $Nodes$ be a set of atoms or rings to add, $Edges$ be the set of actions which correspond to adding a bond between the last atom added to the molecule and another atom in the graph, and $Terminate$ be the action which ends the graph generation process. We limit the $Edges$ set in order to avoid the action space blowing up exponentially with the size of the graph.

6.1.3 Architecture and Data Representation

Let us define the representation of small molecules which we feed into the model. We represent the molecular graph with n atoms and m edges, under some node ordering as the tuple, $G^+ = (V, E, N_f, E_f)$ where $N_f =$

$\begin{bmatrix} | & & | \\ v_1 & ... & v_n \\ | & & | \end{bmatrix}$ where v_i are node features, and $E_f = \begin{bmatrix} | & & | \\ e_1 & ... & e_m \\ | & & | \end{bmatrix}$ where e_i are edge features. To generate the node features for a given atom, we stack one-hot encodings of its element, hybridization, aromaticity, explicit valence, implicit valence as well as atomic mass. For edges, we assign a categorical variable based on whether the edge is a single, double, or aromatic bond in the original molecule. We augment this graph representation with a binary variable encoding whether the graph is connected as well as $N_f[-1]$, i.e. the features of the most recent node added. Thus our model operates on tuples of the form $((A, N_f, E_f), N_f[-1], \mathit{connected})$

We now define a class of models which operate on this tuple. Let NN_{graph} be a GNN, and NN_{node} and NN_{edge} be standard MLPs. For input

$$\mathbf{x} = ((A, N_f, E_f), N_f[-1], \mathit{connected})$$

we let

$$N_f^{'} = NN_{graph}((A, N_f, E_f))$$

$$[\text{Node Addition}] = NN_{node}(\text{Sum}(N_f^{'}))$$

$$[\text{Edge Prediction}] = NN_{edge}(\begin{bmatrix} repeat(N_f^{'}[-1]) \\ N_f^{'} \end{bmatrix})$$

$$\text{Output} = \text{Softmax}([\text{Node Addition}, \quad \text{Edge Addition}])$$

6.2 Self Supervised Pretraining

In this stage of training, our goal is to fit a generative model to the space of small drug molecules. let M be the set of molecule and let

$$M^s = \{\tau \mid \tau \in h(m) \text{ for } m \in M\}$$

be the set of all building sequences which produce a molecule $m \in M$. Similar to autoregressive text generation, we model the joint distribution over atoms and bonds in a graph as

$$p(G_m) = \prod_{i=1}^{n} p(\sigma_i | \sigma_1, ..., \sigma_{i-1})$$

where G_m is a graph molecule and $\sigma_1, ..., \sigma_n$ are graph building steps which produce it. It should be noted that for some molecule M, and $(\sigma_1^1, ..., \sigma_n^1), (\sigma_1^2, ..., \sigma_m^2) \in h(M)$, we do not ensure that

$$\prod_{i=1}^{n} p(\sigma_i^1 | \sigma_1^1, ..., \sigma_{i-1}^1) = \prod_{i=1}^{m} p(\sigma_i^2 | \sigma_1^2, ..., \sigma_{i-1}^2)$$

holds. Hence, for molecule M our model may assign different likelihoods to two different sequences $\tau_1, \tau_2 \in h(M)$.

Although we could generate a training dataset which contains all possible node orderings, with $n!$ possible orderings for a graph of size n, this becomes computationally intractable. Despite this, we find that the model learns to assign reasonable probabilities.

We also note that we do not explicitly feed the molecule a series of graph building steps $(\sigma_1, ...\sigma_n)$, but instead the graph that these actions would generate.

6.2.1 Graph Decomposition

In order for our model to learn an appropriate policy, we found it helpful to expose the model to a large dataset of bioactive molecules with drug like properties. In our experiments we chose the freely available ChEMBL database [24]. Examples of typical molecules can be found in A.1 and A.2. In order to allow our model to learn on molecules from these datasets it was necessary to devise a training procedure amenable to the graph building procedure. To accomplish this, we go through each molecule and choose a graph corruption action which loosely corresponds to a graph building step. We ensure that corruption action is one that could be repaired in a single graph building step. We define the decomposition algorithm as such

Algorithm 4: ***Graph Decomposition***

```
Input: G_m: Graph Molecule
Output: set of graph, action pairs
s ← [ ]
while True do
    if Lone Atom Exists then
        Remove lone atom n_i from G_m
        s.append((G_m, Element(n_i)))
    else if Unconnected Aromatic Ring Exists then
        Remove lone aromatic ring r_i from G_m
        s.append((G_m, Element(n_i)))
    else if Leaf Atom Exists then
        Remove leaf's edge from G_m
        s.append((G_m, edge))
    else if Leaf Aromatic Ring Exists then
        Remove Ring Atom's leaf edge from G_m
        s.append((G_m, edge))
    else if Non Cut Edge Exists then
        Remove non cut edge from G_m
        s.append((G_m, edge))
    else
        break
return s
```

We thus generate a dataset $D = \{\{\mathbf{g}_i, y_i\}_{i=1}^{n}\}$ where $\mathbf{g}_i$ are intermediate molecular graphs and y_i are corresponding graph building steps.

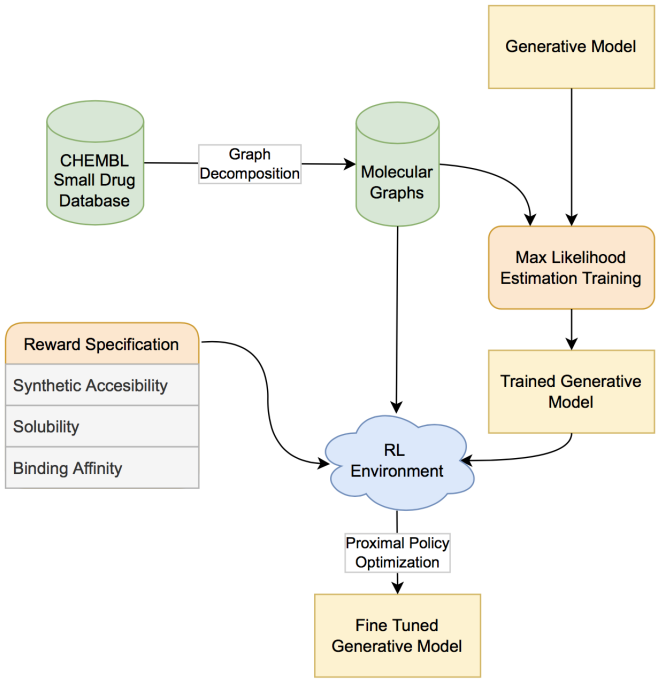


Figure 6.2: Full Training Procedure for Model

6.3 Reinforcement Learning Fine Tuning

6.3.1 PPO Implementation

We found our implementation of PPO based directly on the algorithm described in the original paper to be exceptionally brittle. Training would often be derailed by a single bad update and never recover. These results are relatively consistent with the literature which suggests that a number of tricks are needed to get reliable results from PPO [25]. To improve results, we implement a number of techniques not mentioned in the original paper.

The first method we use is to update the loss function of the critic using a similar surrogate objective as the actor. For a single target value V_{targ} and predicted value V_θ, instead of

$$L_{critic}(V_{targ}, V_{\theta_t}) = (V_{\theta_t} - V_{targ})^2$$

$$L_{critic}(V_{targ}, V_{\theta_t}) = \min\left[(V_{\theta_t} - V_{targ})^2, (\text{clip}(V_{\theta_t}, V_{\theta_{t-1}}, V_{\theta_{t+1}}) - V_{targ})^2\right]$$

Which works empirically to ensure that the critic network doesn't suffer from untrustworthy updates. We also experiment with a reward scaling scheme in which we normalize by a discounted running total of the mean and standard deviation. We additionally use orthogonal initialization, learning rate annealing using the Adam optimizer and global gradient clipping.

Finally, we implement a common early stopping method based on an approximate KL divergence between successive updates. We estimate our KL with samples from collected trajectories. If a given value for KL is reached, the agent stops optimizing over the current trajectory and collects a new set of experiences. Additionally, we perform optimization over minibatches taken from shuffled trajectories. This last optimization seems to have been the most important. With these tricks implemented, training became far more robust, rarely suffering from irrecoverable policy divergence while at the same time improving faster.

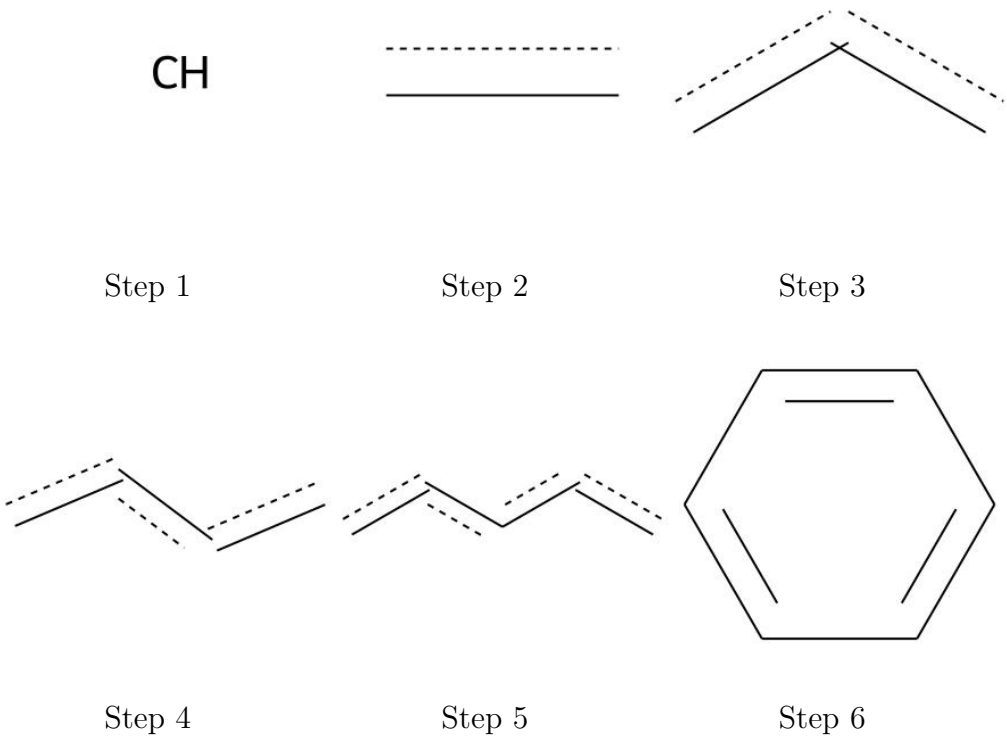
6.3.2 Chemical Environment

To provide a world in which our model can act, we implement a chemical environment in python using RDKit, an opensource chemoinformatics tool [26]. The environment consists of a single RDKit molecule on which the model was able to work. After an action is proposed by the model, a set of chemical checks are applied before any action is carried out. These checks consist predominantly of ensuring that the number of bonds made by a given atom does not exceed the maximum for that element and making sure that the molecule is connected if the action proposed was an bond addition. If the molecule resulting from taking the proposed action is chemically sound the action is then taken and the environment is updated with the

new molecule. Otherwise, the environment is unchanged. Additionally, on reset, a random molecular fragment is selected from the self supervised training set to be the current working structure. We do this both to avoid catastrophic forgetting of learned material from pretraining [27] as well as avoid degenerate policies.

We now talk about our choices of actions as well as the intricacies of chemically invalid intermediate atoms. Aromatic rings are crucial tools in small molecule drug design, and as such, it is important that our model is capable of expressing these molecular motifs when necessary. However, this presents an issue for our graph building protocol. Consider building a benzine ring atom by atom. RDKit represents bonds in an aromatic ring as aromatic bonds and so we have to add an aromatic bond each time.

Table 6.1: Building Benzine Ring



However, outside of an aromatic ring these chemical meaning of these bonds is ambiguous, further, in the event that the model used aromatic

bonds but failed to make a ring, it is unclear how we might evaluate the produced molecule. To avoid this we fix a set of aromatic rings (benzine, pyrrole, pyridine) which the model may add as well as limiting the model to adding only single and double bonds.

Another common occurrence was ambiguous aromaticization after adding certain types of aromatic rings. to fix this, we make use of the Auto Martini which matches aromatically ambiguous structures to common aromatic patterns in order to produce sound molecules[28].

6.3.3 Rewards

We experiment with a number of rewards to bias our molecular generation. First we give a small reward of $\pm.1$ at each step of graph generation correspond to whether or not the agent proposes a chemically legitimate action. Doing this ensures the model learns the chemical laws of the environment.

Reward Functions	
Size	The number of atoms in the molecule
Synthetic Accessibility	Synthesizability estimate
LogP	Solubility estimate
QED	Quantitative drug likeliness estimate
Synth	A Synthetic Accessibility modulated with size reward
Docking	Ligand Protein binding estimate

The rest of the rewards are given after the model chooses to terminate the graph, or after the max number of steps takeable in the environment. We use a size reward to avoid degenerate solutions as well as Lipinski's rule of 5, which lays out a set of basic molecular properties associated with drug-likeness [29]. A function which approximates a molecules $\log p$ (a measure

of the solubility of a molecule) helps the agent learn to produce bioavailable drugs[30]. We test a quantitative estimation of drug-likeness (QED) score which takes into account many of the other rewards mentioned above[31]. We also use deep learning models which try to predict how synthesizable drugs are. This avoids the agent generating unbuildable drugs.

Our final reward is a docking protocol using AutoDock Vina[32][33], which computed the binding affinity between generated small molecules and a drugable pocket in p53 Y220C. After each episode, we use open babel, a bioinformatics format conversion tool, to generate 3-D coordinates for the molecule using a rule based approach. We choose not to use energy minimization of proposed molecule confirmations due to constraints on computational resources.

We normalize the initial rewards by subtracting and dividing by the mean and standard deviation of the reward applied to all molecules in the training set. As the model is initially trained to learn a density over the unsupervised training set, we believe normalizing the rewards based on training set to be reasonable. The final reward given to the molecule is a linear combination of all the property rewards.

Chapter 7

Experimental Results

7.1 Self-Supervised Pretraining

7.1.1 Hyperparameter Optimization

We initially configure the hyper parameter search space as below.

Model Architecture		
Hyperparameters	Values	Distribution
Dense Activation	(ReLU, Leaky ReLU, ELU)	$\mathcal{U}$
Dense Normalization	(Layer Norm, Spectral Norm)	$\mathcal{U}$
Dense Layers	(3,4,5)	$\mathcal{U}$
Dense Hidden Dim	[250,550]	$\mathcal{U}_{[250,550]}$
Dropout	[.3,.7]	$\mathcal{U}_{[.3,.7]}$
Graph Activation	(ReLU, Leaky ReLU, ELU)	$\mathcal{U}$
Graph Normalization	(Layer Norm, Spectral Norm)	$\mathcal{U}$
Graph Layers	(3,4,5)	$\mathcal{U}$
Graph Model	(GCRN, GGNN)	$\mathcal{U}$

Self-Supervised Training		
Hyperparameters	Values	Distribution
Learning Rate	$[3 \times 10^{-5}, 3 \times 10^{-3}]$	$\mathcal{U}$
LR Decay	[.9,1]	$\mathcal{U}$
Batch Size	(32,64,128,256,512)	$\mathcal{U}$
Epochs	[4,6,8,12]	$\mathcal{U}$

We found that the most dramatic increases in accuracy came from a proper learning rate schedule as well as high dropout probability to reduce over-fitting. Additionally, extending the number of training steps taken improved the skill of the model. Although increasing model size was slightly beneficial for pretraining, deeper models can often be unwieldy when trained with policy gradients so we tried to keep the network relatively small.

After fine tuning hyperparameters, our model achieves a validation set accuracy of .66, which represents an improvement of roughly 10 points compared to initial guesses for hyperparameter. We found validation accuracy to be heavily influenced by dropout rate and number of epochs, as well as proper learning rate schedules.

We note that accuracy is a poor metric by which to judge the overall success of our model, but presents itself as a useful proxy to examine successful training. For any given molecular input, especially molecules with a low number of atoms, there might be a number of different 'correct' actions as there are a number of different molecules that can be made from an initial structure. Ultimately, the quality of our model lies in how well a predicted probability distribution over actions captures the underlying distribution of possible graph building actions. As our dataset is, in some sense, under determined, there is an upper limit to any estimators accuracy on the set.

Although we achieved good accuracy, we found that the model became too confident in its predictions for node addition which led to a policy which behaved too deterministically. To combat this, we introduced a secondary regularization term which penalized the model for low entropy distributions over the training set. Ensuring a higher entropy across suitable actions leads to a better exploration phase from the model and avoids convergence to early sub-optimal strategies[34].

Additionally, we found that owing to the large imbalance in edge addition actions over the training set, it was important to use a weighting in the Cross Entropy loss function to over incentivize double bond predictions. Our final loss function becomes

$$L(\theta) = \text{CrossEntropy}_{\text{weight}}(f_\theta(x)|x, y|x) + kH(\text{NodeAddition})$$

where k is a tunable hyperparameter.

7.1.2 Examining Distribution over simple molecules

To probe the generative capabilities of the model, we perform beam search using predicted probabilities of different graph building steps as our heuristic. We first consider a benzine ring as our starting structure and expand each node with the top two most probable actions suggested by the model. Unsurprisingly, the model defaults to adding carbon and oxygen atoms. To generate more novel structures we can choose to sample from the distribution instead of taking the max, as well as consider more branches.

Next we consider a branching number of 5 and pick some more interesting molecules.

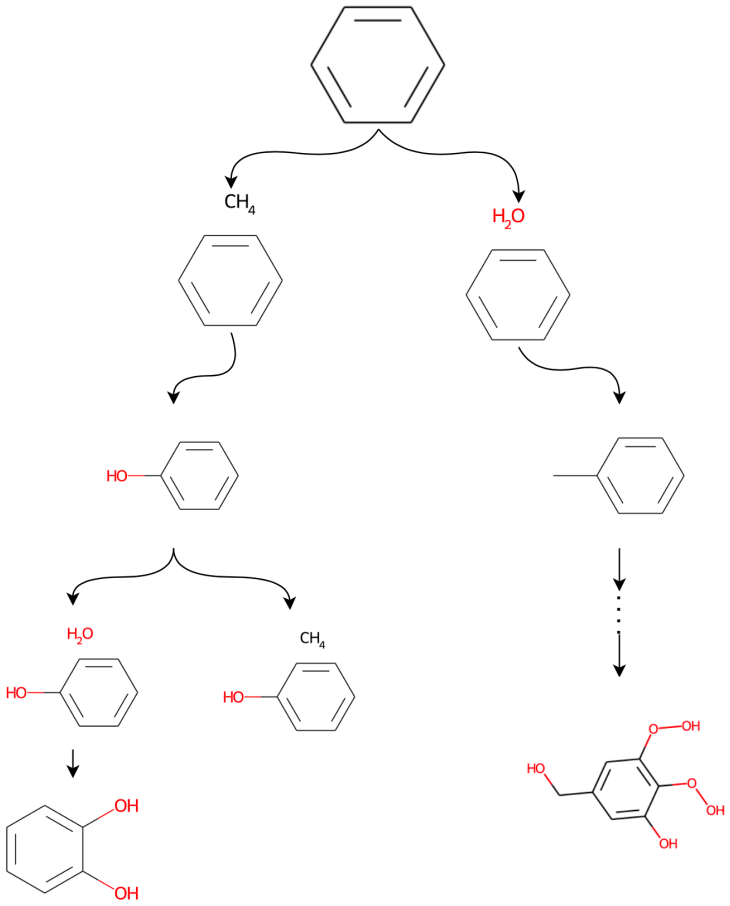


Figure 7.1: Benzine Start with Top Two Branches

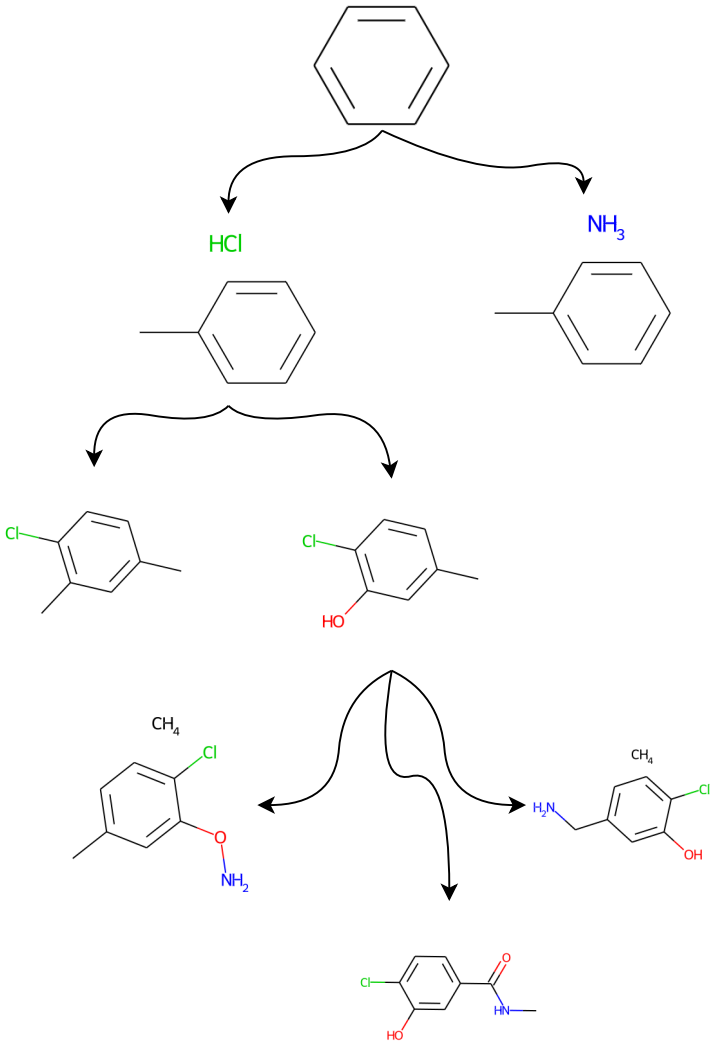
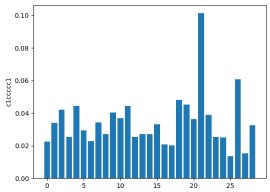


Figure 7.2: Benzine Start with Top Five Branches and Pruning

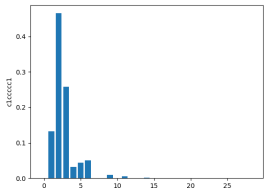
7.1.3 Entropy Regularization

To test the results of different values for the entropy regularization coefficients k, we graph predicted action distributions over Benzine rings change during training.

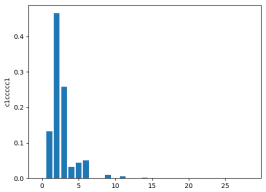
Table 7.1: Impact of k



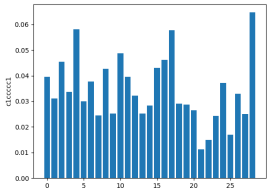
Epoch = 1, $k = .1$



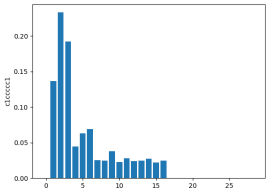
Epoch = 5, $k = .1$



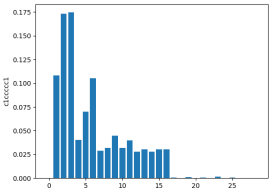
Epoch = 10, $k = .1$



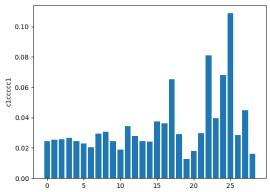
Epoch = 1, $k = 1$



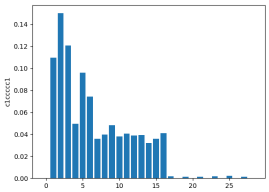
Epoch = 5, $k = 1$



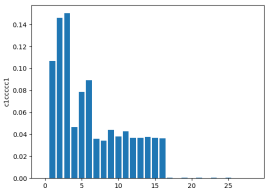
Epoch = 10, $k = 1$



Epoch = 1, $k = 1.5$



Epoch = 5, k=1.5



Epoch = 10, $k = 1.5$

Where actions 1 through 17 represent node additions and those after represent edge additions. We see that for $k = .1$ the probability mass is still too concentrated on only a few actions. While the distribution for $k = 1$ has better support over all actions, we choose $k = 1.5$ for further experiments.

7.2 Pure RL + Reward Shapining

To probe the necessity of pretraining as well at the efficacy of a pure Rl approach, we remove all initial self supervision from the training protocol and examine the fully RL generated policy. Specifically we train in a pure RL environment for 100000 steps with a final learning rate of $3e-5$ which we linearly anneal from zero during a warm-up period of 8 rollouts. We shape rewards to explore whether or not a generative model for realistic looking drugs can be learned in a purely RL manner.

For each rollout, we save out molecules which receive the highest synthetic accessibility score. We also always include a step reward unless stated otherwise. Additionally, We note that our model for synthetic accessibility is a neural network trained on a dataset of preexisting drug-like small molecule, there is no guarantee that it will give reasonable predictions for the policy produced molecules which are almost certainly out of distribution.

Figure 7.3: Early RL Molecules

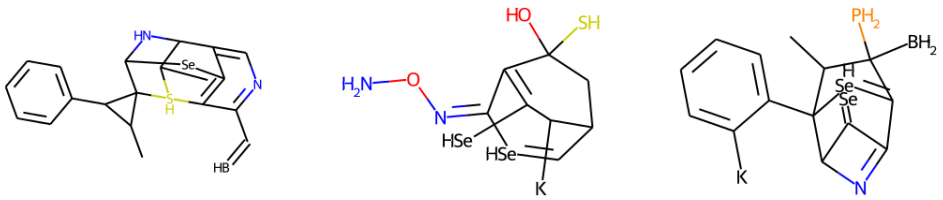


Figure 7.4: Molecules produced by RL agent maximizing Synthetic Accessibility

We see that these molecules are notably different from those in the training set. Most obviously, they are far more dense.

7.2.1 Lipinski Rule of 5 + Size

To obviate the concerns of out of distribution predictions for Synthetic Accessibility predictions, we replace this reward with a scaled indicator of whether a proposed molecule adheres to Lipinski's rule of 5. To prevent degenerate solutions, we include a size reward in addition. However, the

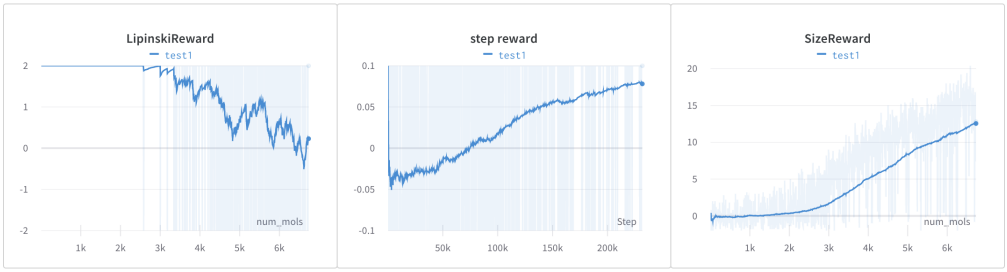


Figure 7.5: Lipinski + Unnormalized Size

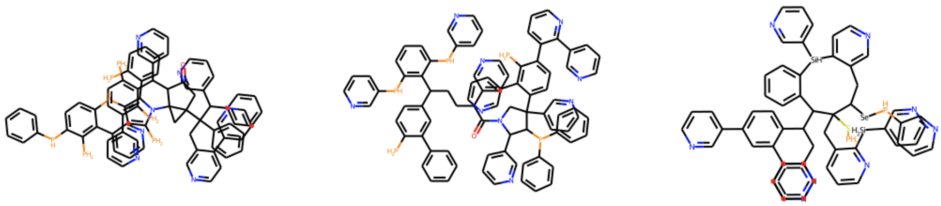


Figure 7.6: Top molecules after 200000 steps of learning

model was able to ignore the Lipinski reward in favor of optimizing size which led to degenerate solutions. To prevent this, we pass the size reward through a *tanh* function and multiply it by 3 to map it to $[-3, 3]$. Thus the model receives diminishing returns for continuing to grow the molecules.

7.3 RL with Pretraining

7.3.1 Initial quality of Pretrained Model

To examine the quality of the model produced by pretraining, we randomly sample start a set of start states from our environment to show the model.

We then track the percentage of chemically legitimate steps taken by the model against the number of considered actions.

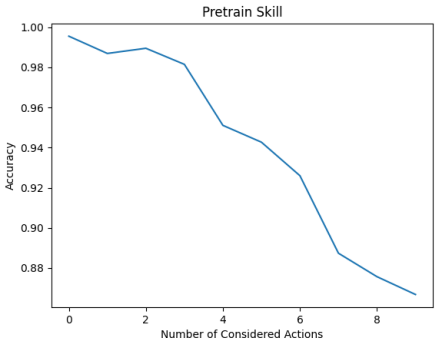


Figure 7.7: Percentage of chemically legitimate steps proposed by the model

Additionally we see how quickly either model learns to the chemical rules of the environment.

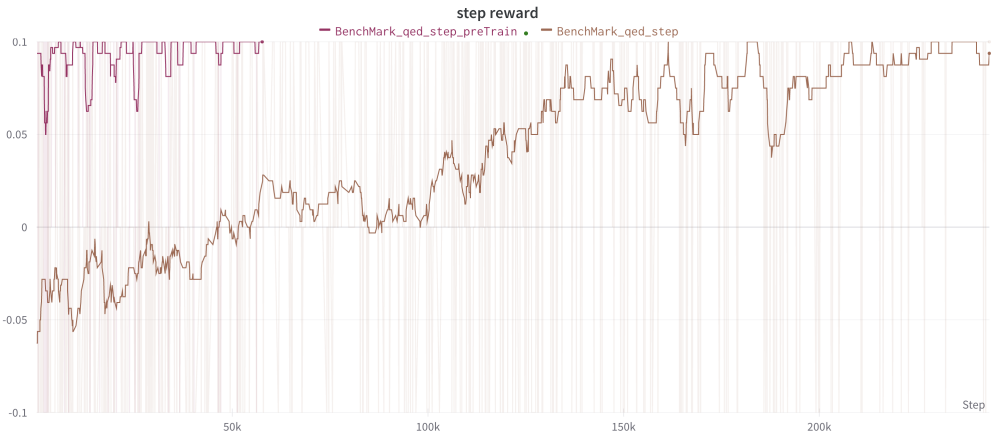


Figure 7.8: Comparison of the time taken to solve step reward between a pretrained model(magenta) and a raw model(brown)

From this we gather that self-supervised pretraining produces an agent which already understands on some level chemical valency as well as what makes for chemically legitimate actions. Interestingly enough this does not appear sufficient to learn all the rules of the chemical environment as we see step reward still increases over time. We imagine that this represents

some sort of mismatch between observations during training versus observations in the chemical environment as well as the laws of the chemical environment. Most likely it is due to the way aromatic rings are treated in either training regime

To investigate the chemical quality of the molecules, plot the distribution of rewards over molecules sampled from the ChEMBL dataset and those generated by out pretrained model

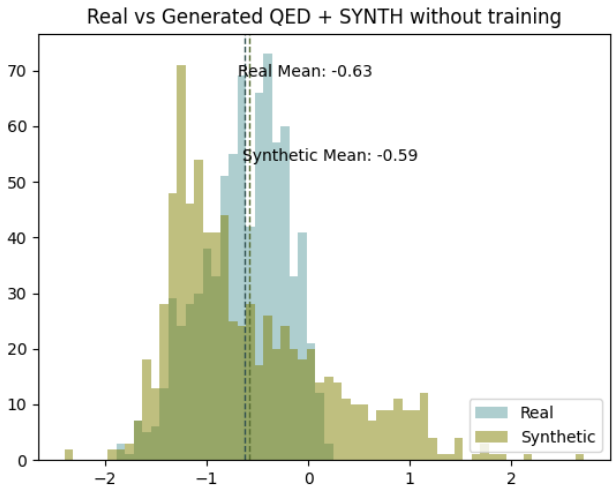


Figure 7.9: Comparison between real and generated molecules on a Synthetic Accessibility scoring function.

We can see that the mean reward of the pretrained model is roughly equivalent to that of random samples from the CHembl database. This suggests that self supervised pretraining is capable of learning a generative model over the space of drug molecules, at least in terms of the expected reward under the real distribution of molecules and that of the distribution learned by our model. However we also see that the models distribution is right skewed whereas the real distribution appears mostly symmetric.

7.3.2 Transition from Self Supervised Pretraining to Reinforcement Learning

When the model transitioned from the self supervised to RL training regime, we found that it often suffered from large early updates resulting in unstable learning and degradation in the quality of the policy. We considered a few possibilities for this phenomenon and implement a policy constraint which we anneal over training to combat these issues. Our first thought is that there is already a mismatch between Actor and Critic which leads to the critic network lagging in quality behind the actor. Improper advantage estimates would thus lead to unstable updates to the actor. Secondly, we consider the possibility that there is a small shift in distribution between the examples seen in pretraining and those seen in the RL environment which leads to the model needing to relearn parts of its representations. In order to address these issues, we use a warm-up schedule for the optimizer of our actor which increases the learning rate as such according to

$$lr_{actor} = \min(\frac{lr_{goal}}{speed} \cdot step, lr_{goal})$$

where lr_{goal} is the final desire learning rate, *speed* is how fast we would like to achieve the desired learning rate and *step* is the number of times we have performed a rollout.

7.4 Final Model

We look at two variants of the full model. The first is focused on simply biasing generated Graphs towards being generally good drug candidates. For this version of the model, we include a Synthetic Accessibility score weighted by molecule size (this avoids degenerate solutions) as well as QED estimation of drug likeness. The second we solely optimize on docking

reward. We choose to only include a docking reward as this reward is far sparser than the others, thus we wanted to provide the clearest signal for the model to learn on. Additionally, the docking reward increases training time by days, so detailed reward shaping was difficult.

We test the final performance of both models, by looking at the distribution of rewards over generated molecules versus random samples from the CHembl dataset. Additionally we examine if the models are able to iterate on fragments to improve the quality of molecules over the original. We start both models RL training using a pretrained model with regularization coefficient $k = 1.3$

PPO Hyperparameters	
Hyperparameters	Values
Actor LR	$1e-4$
Critic LR	$3e-4$
γ	.99
Clip	.1
Batch Size	256
Updates per Iteration	6
Horizon	3000
Episode Length	40
Number of Steps	400000

We first look at the model trained using the QED and Synth reward. Training converged after only 25k molecules were produced, or roughly 800k graph building steps were taken.

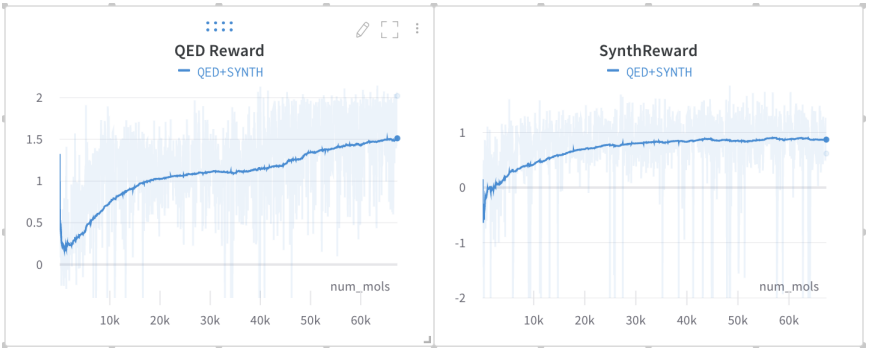


Figure 7.10: Training Curves for QED and Synthetic Accessibility training

As we will see, convergence speed for this environment is far faster than the more complex docking environment. The synthetic accessibility reward begins to saturate at 35k steps while the quantitative druge-likeness estimate reward plateaus at 30k steps before starting to climb again. We next examine the general quality of the model against random Chembl samples when evaluated under the environmental reward. We sample a set of 800 molecules from the Chembl Database as well as generating a set of 800 from randomized initial molecular fragments and plot both.

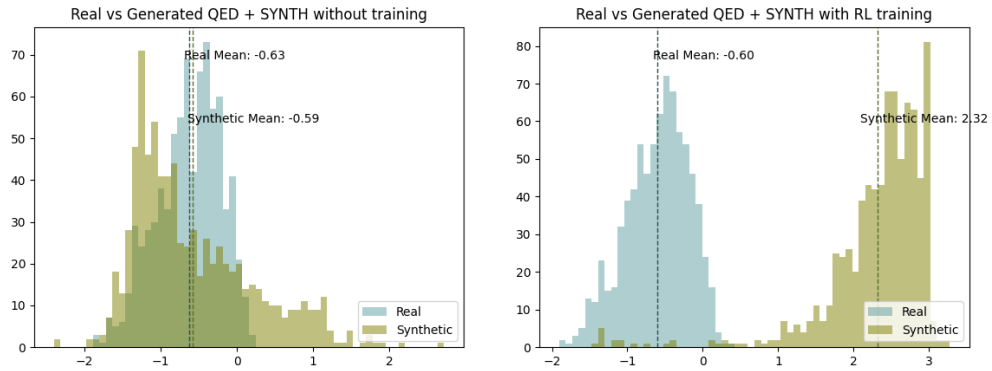


Figure 7.11: Comparison between real and generated molecules on a Synthetic Accessibility scoring function. The left figure is a pretrained model and the right is the result of RL fine tuning

We see that our sampling from out model instead of the ChEMBL database produces molecules of far higher quality. We achieve a 2.92 average point improvement over the ChEMBL baseline. Additionally we see a similar variance in reward between real and generated molecules, implying that we have similar variety across generated molecules. See A.4 for examples of generated molecules

We next examine model success within the docking environment. We see similar reward distributions as before with the pretrained model, except for a large amount of mass at -2. In shaping the reward, we clipped negative rewards such that they were also above -2 as we saw that often for rewards less than -2, we would get wildly negative results which we assumed were

anomalies of the docking procedure as opposed to being representative of the actual binding affinity. We thus imagine that the large amount of mass on -2 is the result of the model generating more of these anomalous molecules.

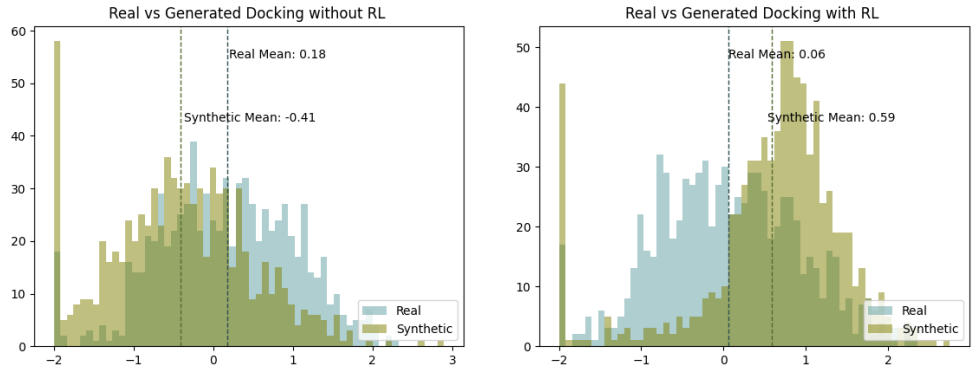


Figure 7.12: Comparison between real and generated molecules on a Binding Affinity scoring function. The left figure is a pretrained model and the right is the result of RL fine tuning

Our RL agent achieves an increase of .53 points over random samples drawn from Chembl in average binding score. Compared to the previous QED Synth environment, we see that the agent has a much harder time learning the environment. The model used in this experiment took two

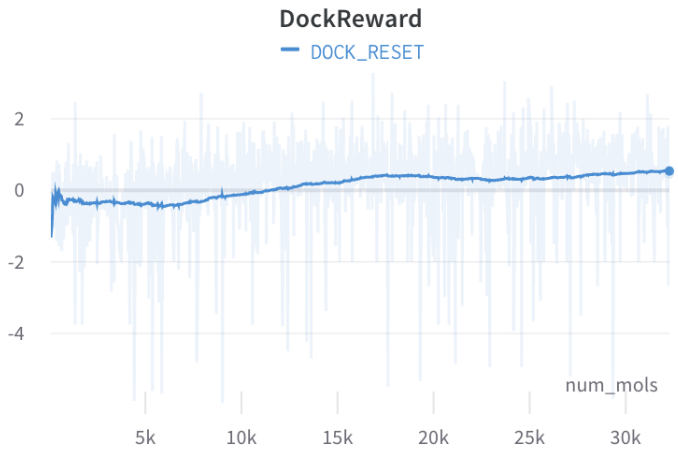


Figure 7.13: Training curve for Docking environment

million steps to achieve its performance compared to 800k. Additionally,

training took around 100 hours due to the time spent performing docking simulations compared to less than a day for the other environments. One more challenge was that the docking protocol would often fail silently, shutting down the training script without raising a catchable error. With that in mind, a few training runs looked to be improving after a long plateau so we believe that increasing training time would lead to further improvements. See A.5 for examples of generated molecules

Chapter 8

Discussion and Conclusions

We find that combining self supervised pretraining on large drug databases coupled with Reinforcement Learning produces generally capable Chemical Generative models. Molecules

Despite some success the model suffers from a few limitations of the algorithm as well as the more general realities of deep RL. These shortcomings likely restricts the application of Deep Drug Diversification to the very early stages of the drug discovery pipeline. For one, the docking procedure we use is incredibly crude when compared with the actual dynamics of the systems it seeks to replicate. This is no fault of docking, but simply the cost of a high throughput, low thought, screening process. We believe that using a deep learning model trained on Molecular Dynamic simulation to score our proposed molecule would give less noisy and more reliable results. To that end, we are interested in the possibility of exposing an RL agent to an actual molecular dynamic physics simulator and have it build molecules there. Although this would be a tremendously difficult engineering feat, we suspect that this is the way the field will trend and are excited for its arrival.

For further refining the Algorithm, we pinpoint a few relatively straight-

forwards optimizations which would help model performance. We first note the bottleneck introduced by rollouts in our environment, especially that of our docking environment. In our current implementation, we are constrained by the model predicting actions for a single state and then updating the environment. Instead, we could maintain a number of environments concurrently and predict actions for each environment in parallel. Through this, we could vastly speed up the time is takes to generate a full rollout. This would be especially important when dealing with more complex environments, like ones running docking simulation. Additionally augmenting our featurized molecular graphs with positional embeddings would greatly increase the expressivity of our models [35]. We also believe further reward shaping to be necessary. For example, the step reward might be spreading useless noise throughout the training which prevents the model from picking up on the more important rewards. Finally,a more principled approach to reward functions is necessary before the model would be useful in any capacity. Jointly optimizing a number of different tasks is hard for an RL agent. How would it know that making a molecule which binds well to a protein at the cost of a small degradation in synthetic accessibility is a worth wile trade off? We hope to solve these problems down the roads.

Appendix A

Appendix

Code for project available at github.com/tsteternlieb/DrugDesignThesis

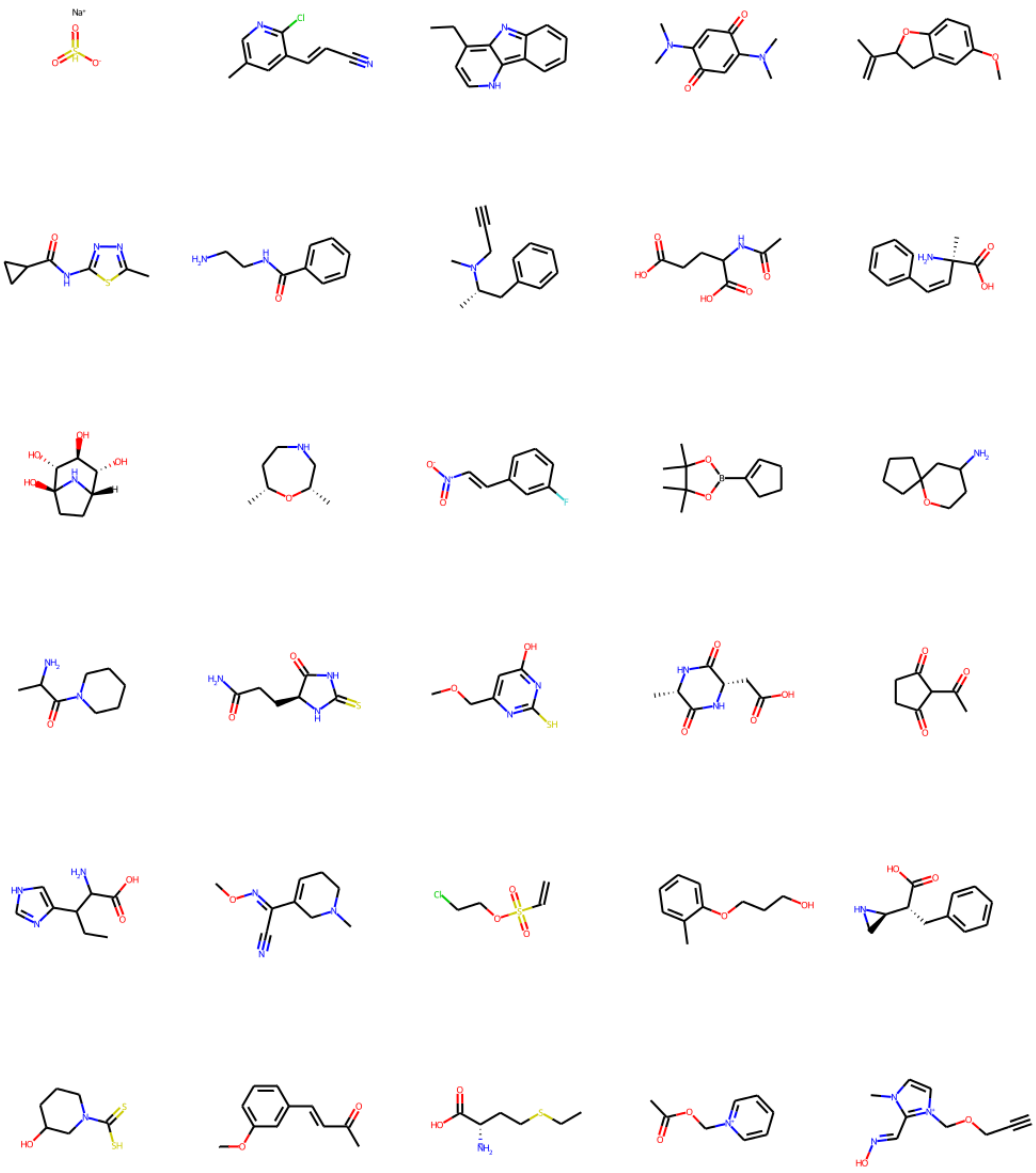


Figure A.1: Molecules from Chembl

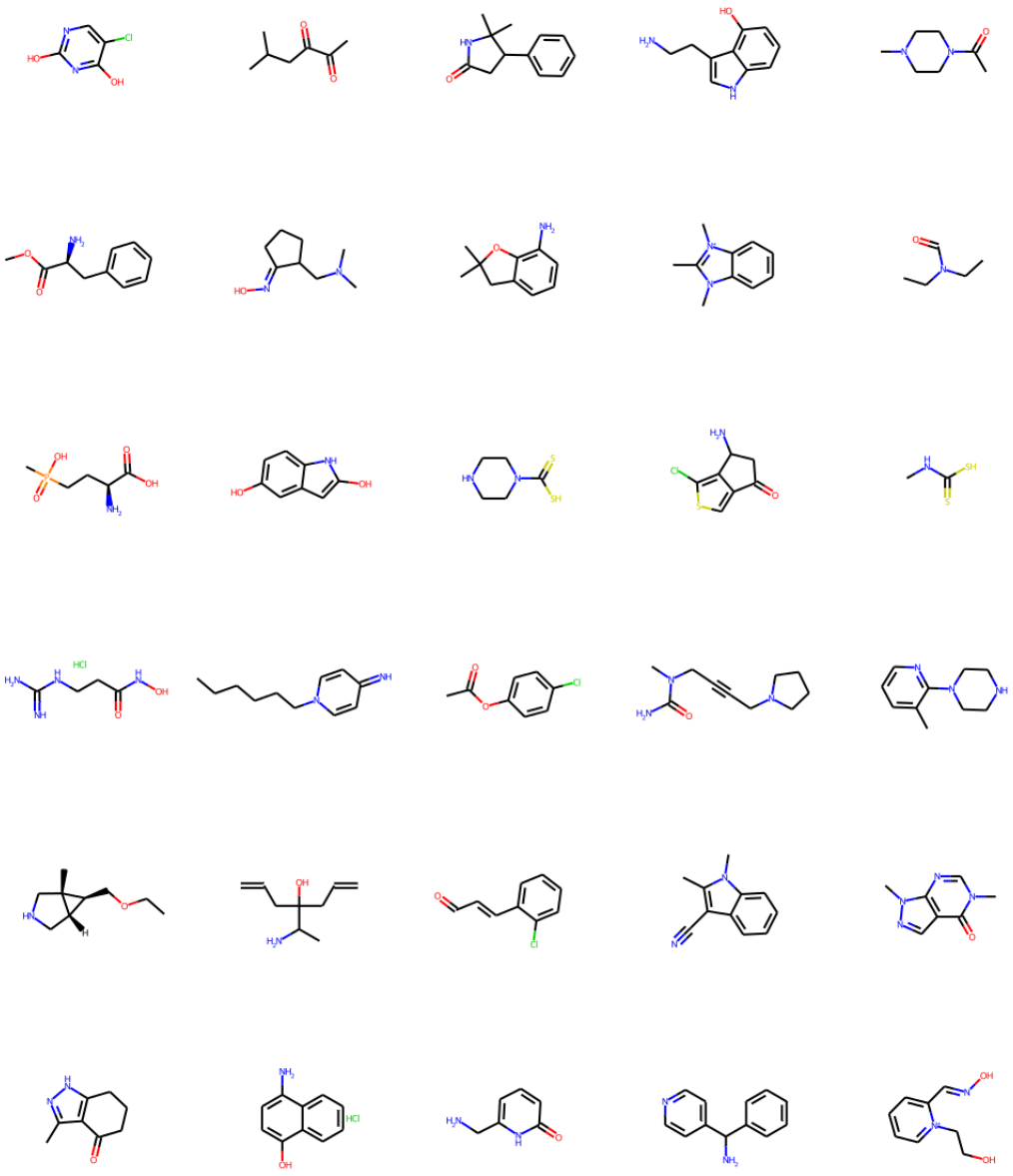


Figure A.2: More Molecules from Chembl

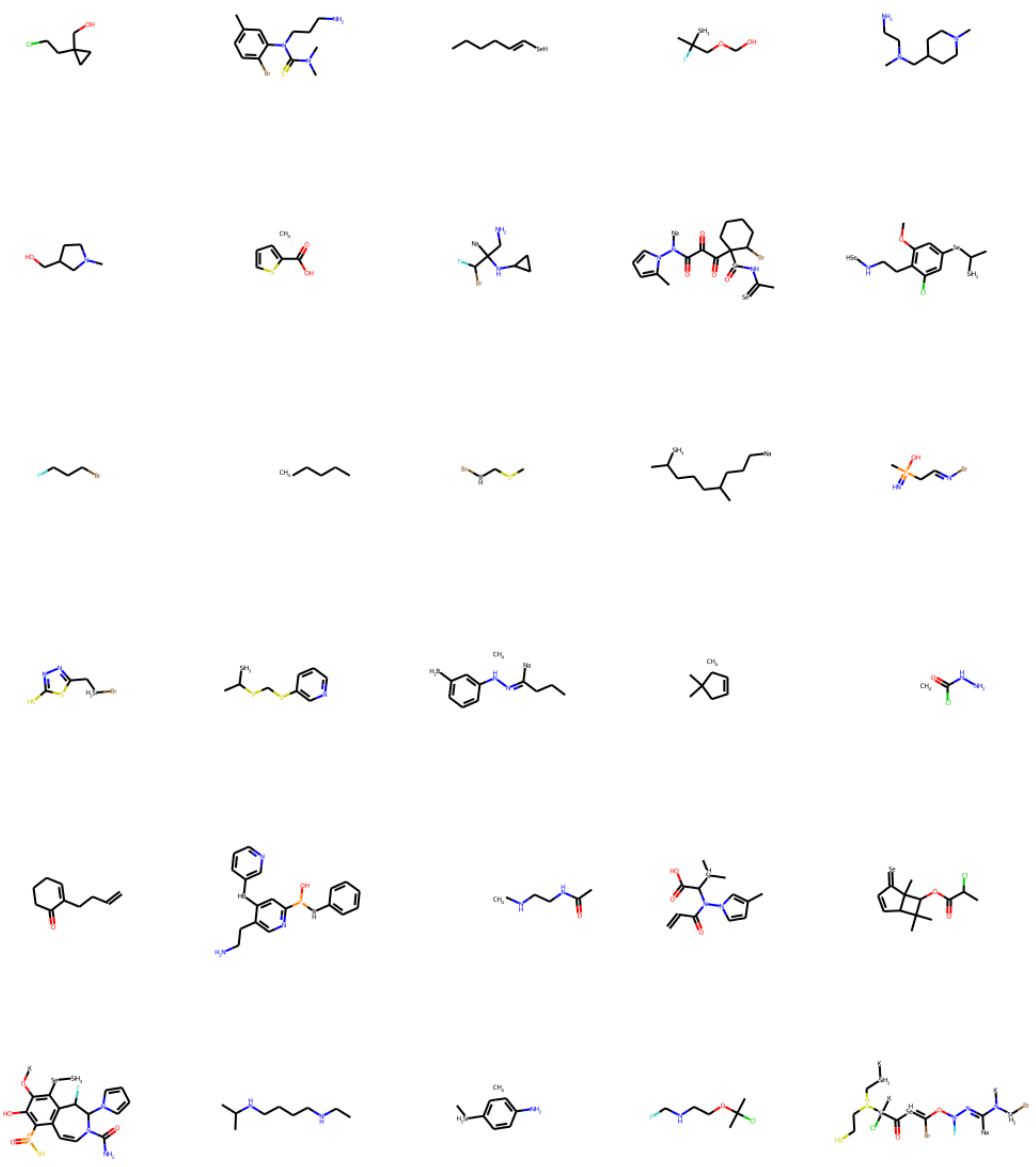


Figure A.3: Molecules generated by self supervised model with $k = 1.3$

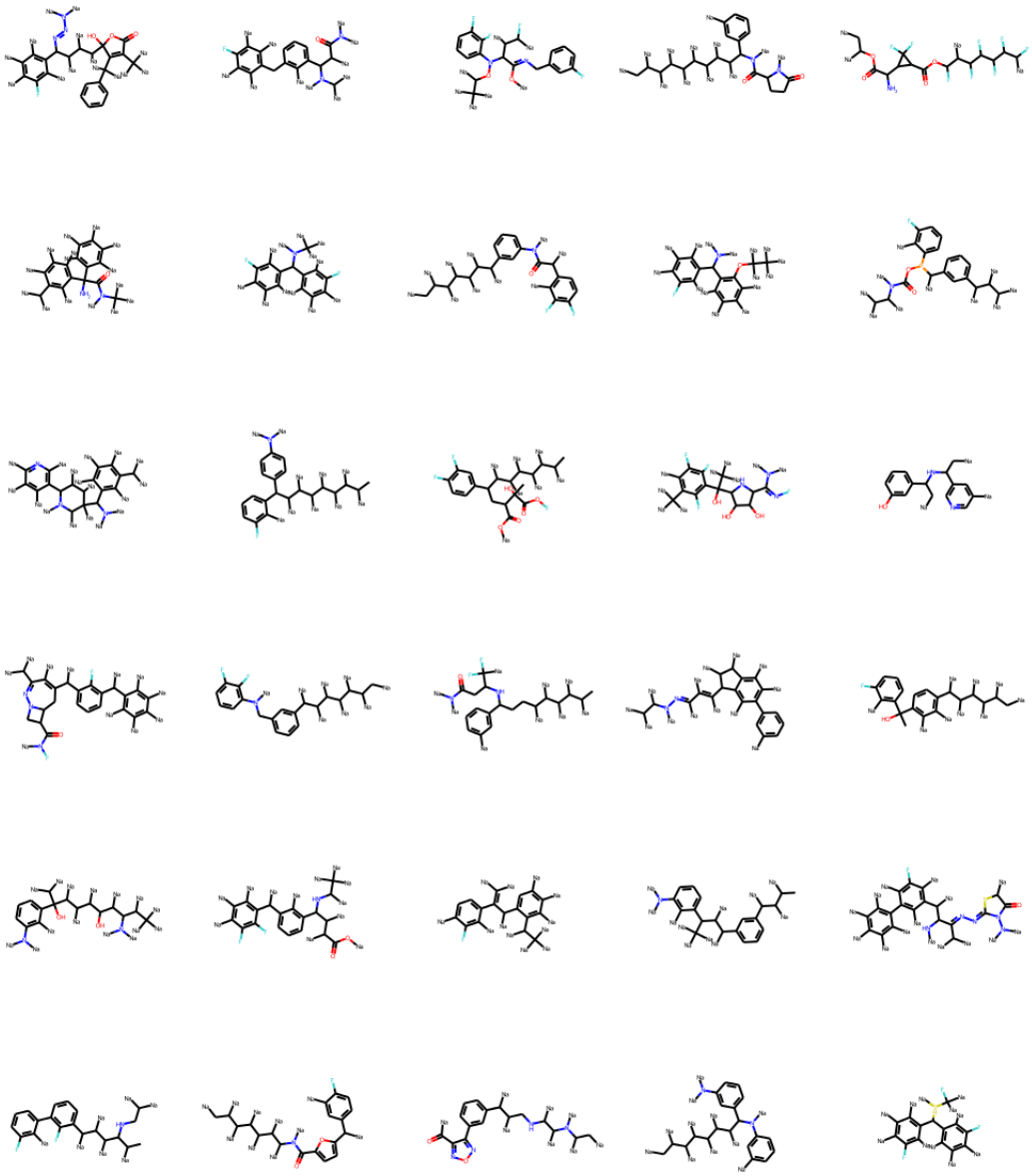


Figure A.4: Molecules generated by an early checkpoint from Synth QED training

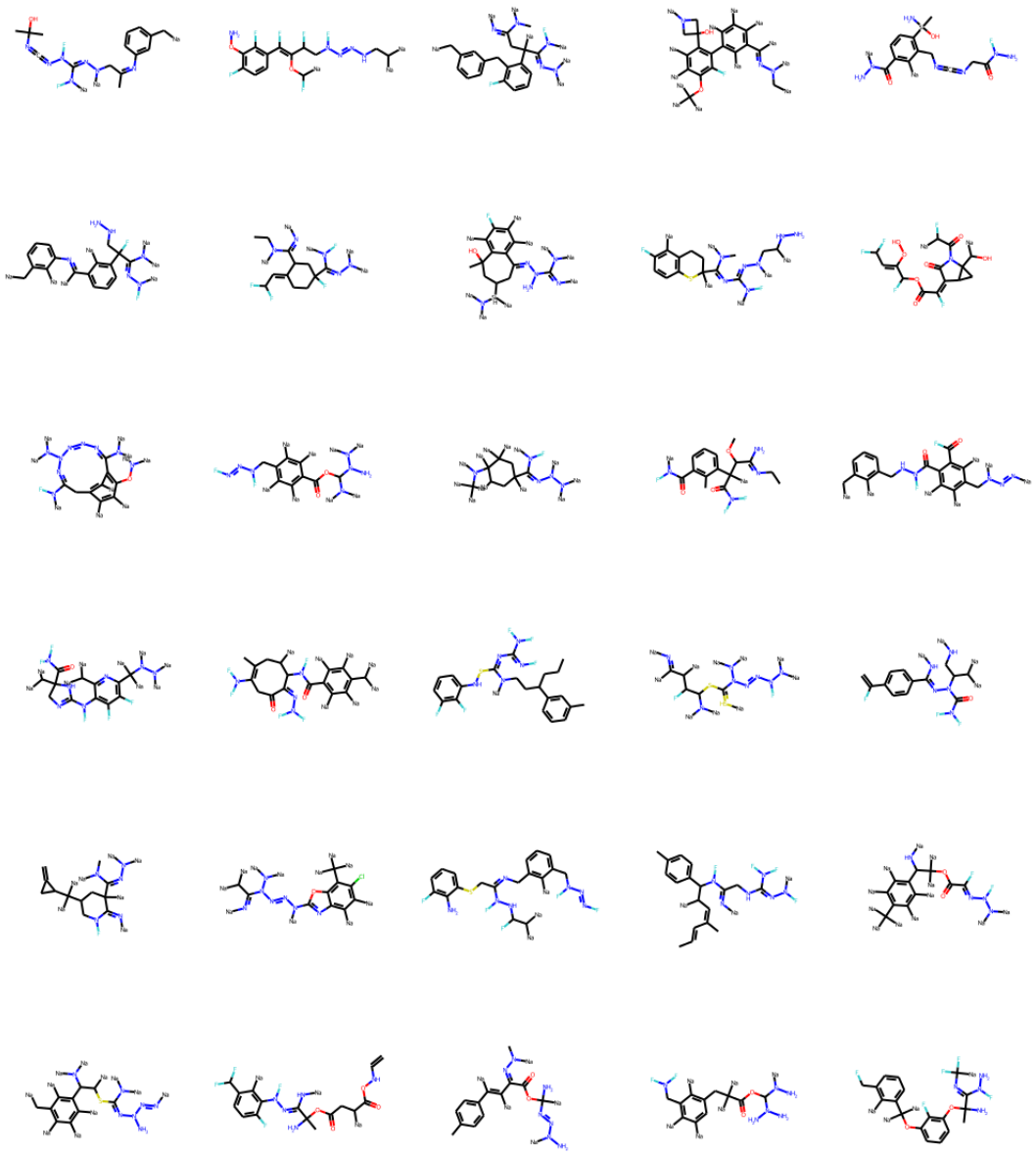


Figure A.5: Molecules generated by model fine tuned with docking training

Bibliography

[1] Thayer, Kelly M and Quinn, Taylor R. p53 r175h hydrophobic patch and h-bond reorganization observed by md simulation. *Biopolymers*, 105(3):176–185, 2016.

[2] Silver, David, Schrittwieser, Julian, Simonyan, Karen, Antonoglou, Ioannis, Huang, Aja, Guez, Arthur, Hubert, Thomas, Baker, Lucas, Lai, Matthew, Bolton, Adrian, Chen, Yutian, Lillicrap, Timothy, Hui, Fan, Sifre, Laurent, van den Driessche, George, Graepel, Thore, and Hassabis, Demis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017. URL `http://dx.doi.org/10.1038/nature24270`.

[3] Chollet, Francois. Rats generalize better than deep rl, and are more sample efficient. URL `https://twitter.com/fchollet/status/1187024029871304704`.

[4] Rauf, Shah Md, Endou, Akira, Takaba, Hiromitsu, Miyamoto, Akira, et al. Effect of y220c mutation on p53 and its rescue mechanism: a computer chemistry approach. *The protein journal*, 32(1):68–74, 2013.

[5] Hornik, Kurt, Stinchcombe, Maxwell, and White, Halbert. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(89)90020-8. URL `https://www.sciencedirect.com/science/article/pii/0893608089900208`.

[6] Kingma, Diederik P. and Ba, Jimmy. Adam: A method for stochastic optimization. In Bengio, Yoshua and LeCun, Yann, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

[7] Cho, Kyunghyun, van Merrienboer, Bart, Bahdanau, Dzmitry, and Bengio, Yoshua. On the properties of neural machine translation: Encoder-decoder approaches, 2014. URL https://arxiv.org/abs/1409.1259.

[8] Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

[9] Bjorck, Johan, Gomes, Carla P., and Weinberger, Kilian Q. Towards deeper deep reinforcement learning. *CoRR*, abs/2106.01151, 2021. URL https://arxiv.org/abs/2106.01151.

[10] Ba, Jimmy, Kiros, Jamie Ryan, and Hinton, Geoffrey E. Layer normalization. *ArXiv*, abs/1607.06450, 2016.

[11] Miyato, Takeru, Kataoka, Toshiki, Koyama, Masanori, and Yoshida, Yuichi. Spectral normalization for generative adversarial networks, 2018. URL https://arxiv.org/abs/1802.05957.

[12] Gogianu, Florin, Berariu, Tudor, Rosca, Mihaela, Clopath, Claudia, Busoniu, Lucian, and Pascanu, Razvan. Spectral normalisation for deep reinforcement learning: an optimisation perspective, 2021. URL https://arxiv.org/abs/2105.05246.

[13] Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan. Dropout: A simple way to prevent neural networks from overfitting. *Journal of*

Machine Learning Research, 15(56):1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

[14] Gilmer, Justin, Schoenholz, Samuel S., Riley, Patrick F., Vinyals, Oriol, and Dahl, George E. Neural message passing for quantum chemistry. *CoRR*, abs/1704.01212, 2017. URL http://arxiv.org/abs/1704.01212.

[15] Syllogismos, Anil. 8.graph neural networks, Dec 2020. URL https://wandb.ai/syllogismos/machine-learning-with-graphs/reports/8-Graph-Ne

[16] Schlichtkrull, Michael, Kipf, Thomas N., Bloem, Peter, Berg, Rianne van den, Titov, Ivan, and Welling, Max. Modeling relational data with graph convolutional networks, 2017. URL https://arxiv.org/abs/1703.06103.

[17] Li, Yujia, Zemel, Richard, Brockschmidt, Marc, and Tarlow, Daniel. Gated graph sequence neural networks. In *Proceedings of ICLR'16*, April 2016. URL https://www.microsoft.com/en-us/research/publication/gated-graph-sequence-ne

[18] Schulman, John, Wolski, Filip, Dhariwal, Prafulla, Radford, Alec, and Klimov, Oleg. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

[19] Schulman, John, Levine, Sergey, Moritz, Philipp, Jordan, Michael I., and Abbeel, Pieter. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015. URL http://arxiv.org/abs/1502.05477.

[20] Kadurin, Artur, Nikolenko, Sergey, Khrabrov, Kuzma, Aliper, Alex, and Zhavoronkov, Alex. drugan: an advanced generative adversarial autoencoder model for de novo generation of new molecules with desired molecular properties in silico. *Molecular pharmaceutics*, 14(9): 3098–3104, 2017.

[21] Goodfellow, Ian J., Pouget-Abadie, Jean, Mirza, Mehdi, Xu, Bing, Warde-Farley, David, Ozair, Sherjil, Courville, Aaron, and Bengio, Yoshua. Generative adversarial networks, 2014. URL https://arxiv.org/abs/1406.2661.

[22] You, Jiaxuan, Ying, Rex, Ren, Xiang, Hamilton, William L., and Leskovec, Jure. Graphrnn: A deep generative model for graphs. *CoRR*, abs/1802.08773, 2018. URL http://arxiv.org/abs/1802.08773.

[23] You, Jiaxuan, Liu, Bowen, Ying, Rex, Pande, Vijay, and Leskovec, Jure. Graph convolutional policy network for goal-directed molecular graph generation, 2018. URL https://arxiv.org/abs/1806.02473.

[24] Gaulton, Anna, Hersey, Anne, Nowotka, Michał, Bento, A Patricia, Chambers, Jon, Mendez, David, Mutowo, Prudence, Atkinson, Francis, Bellis, Louisa J, Cibrián-Uhalte, Elena, et al. The chembl database in 2017. *Nucleic acids research*, 45(D1):D945–D954, 2017.

[25] Engstrom, Logan, Ilyas, Andrew, Santurkar, Shibani, Tsipras, Dimitris, Janoos, Firdaus, Rudolph, Larry, and Madry, Aleksander. Implementation matters in deep policy gradients: A case study on PPO and TRPO. *CoRR*, abs/2005.12729, 2020. URL https://arxiv.org/abs/2005.12729.

[26] Landrum, Greg. Rdkit: Open-source cheminformatics. URL http://www.rdkit.org.

[27] Kirkpatrick, James, Pascanu, Razvan, Rabinowitz, Neil, Veness, Joel, Desjardins, Guillaume, Rusu, Andrei A., Milan, Kieran, Quan, John, Ramalho, Tiago, Grabska-Barwinska, Agnieszka, Hassabis, Demis, Clopath, Claudia, Kumaran, Dharshan, and Hadsell, Raia. Overcoming catastrophic forgetting in neural networks, 2016. URL https://arxiv.org/abs/1612.00796.

[28] Bereau, Tristan and Kremer, Kurt. Automated parametrization of the coarse-grained martini force field for small organic molecules. *J Chem Theory Comput*, 11(6):2783–2791, 2015. doi: 10.1021/acs.jctc.5b00056.

[29] Lipinski, Christopher A, Lombardo, Franco, Dominy, Beryl W, and Feeney, Paul J. Experimental and computational approaches to estimate solubility and permeability in drug discovery and development settings. *Advanced drug delivery reviews*, 23(1-3):3–25, 1997.

[30] Wildman, Scott A and Crippen, Gordon M. Prediction of physicochemical parameters by atomic contributions. *Journal of chemical information and computer sciences*, 39(5):868–873, 1999.

[31] Bickerton, G Richard, Paolini, Gaia V, Besnard, Jérémy, Muresan, Sorel, and Hopkins, Andrew L. Quantifying the chemical beauty of drugs. *Nature chemistry*, 4(2):90–98, 2012.

[32] Eberhardt, Jerome, Santos-Martins, Diogo, Tillack, Andreas F, and Forli, Stefano. Autodock vina 1.2. 0: New docking methods, expanded force field, and python bindings. *Journal of Chemical Information and Modeling*, 61(8):3891–3898, 2021.

[33] Trott, Oleg and Olson, Arthur J. Autodock vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. *Journal of computational chemistry*, 31 (2):455–461, 2010.

[34] Mnih, Volodymyr, Badia, Adrià Puigdomènech, Mirza, Mehdi, Graves, Alex, Lillicrap, Timothy P., Harley, Tim, Silver, David, and Kavukcuoglu, Koray. Asynchronous methods for deep reinforcement learning. 2016. doi: 10.48550/ARXIV.1602.01783. URL `https://arxiv.org/abs/1602.01783`.

[35] Dwivedi, Vijay Prakash, Joshi, Chaitanya K., Laurent, Thomas, Bengio, Yoshua, and Bresson, Xavier. Benchmarking graph neural networks. *CoRR*, abs/2003.00982, 2020. URL `https://arxiv.org/abs/2003.00982`.