Exploring the Huber Loss in MolDQN - A Reinforcement Learning model for Molecular Optimization

Candidate Number: 1043516

Word Count: 7161 [TeXcount]

A thesis submitted for the degree of

Master of Mathematics in Mathematics & Statistics

Trinity 2023

Abstract

In this dissertation, we explore MolDQN, a machine learning model for *de novo* molecular optimization first introduced by Zhenpeng Zhou et al. in their paper *Optimization of Molecules via Deep Reinforcement Learning* [49]. We give a detailed explanation of Q-Learning, the mathematical foundations of MolDQN, along with several advancements to Q-Learning that Zhou et al. used; these include Deep Q-Networks, Double Q-Learning and Bootstrapped DQN. In the Optimization of the Q-Network in MolDQN, the standard *Huber Loss* with delta value 1 is used. We explore the effects of altering the delta value and using the Pseudo-Huber Loss instead. Our experiments lead to the conclusion that the original set-up was optimal, however, we also propose further avenues of exploration for the Optimization of MolDQN.

Contents

Chapter 1

Introduction

Drug Discovery is a vital scientific endeavour that not only saves lives but also improves our quality of life. The first lab-made drug was developed by German pharmacist Friedrich Sertürner in 1804 [24]. From the opium, tarry poppy seed juice, Sertürner isolated morphine, a medicine used in hospitals all over the world to give pain relief. In 1928, Scottish Physician Alexander Fleming discovered penicillin [4], the first naturally-derived antibiotic and arguably medicine's greatest achievement. Gertrude B. Elion, an American biochemist, was awarded the 1988 Nobel Prize in *Physiology or Medicine* along with George H. Hitchings and Sir James Black for their discovery of Mercaptopurine, the first treatment used for the cancer, leukemia. Elion's work also included discovering AZT, (an anti-retroviral drug used in treating HIV), azathioprine, (the first immunosuppressive drug used to prevent rejection in organ transplants) and acyclovir, (a medicine used to treat herpes) [1]. In much more recent history, the COVID-19 pandemic and the quest for a vaccine has shown what the collaboration of the world's scientists can achieve. The COVID-19 vaccinations are predicted to have saved 19·8 million lives in 2021 alone [43].

However, to get a drug approved for market, it can take between 10-15 years and over 2 billion US dollars [5]. Another barrier in drug development is the vast space of possible candidates. The number of molecules with drug-like properties is estimated to be between 10^{23} and 10^{60} [32]. Computational strategies are being developed to save time and money in the early stages of drug discovery, allowing scientists to focus on the stages where human intuition and expertise is essential.

Machine Learning is becoming an increasingly powerful tool used to address society's challenges including in the field of Computer Aided Drug Design (CADD). Scientists are constantly discovering new applications of machine learning in the field of drug development and there have already been successful cases of molecules discovered that look like promising drug candidates such as in the papers *Deep learning* *enables rapid identification of potent DDR1 kinase inhibitors* [48] and *A Deep Learning Approach to Antibiotic Discovery* [36]. One widely-used method in CADD is Virtual Screening, which uses computational strategies to help explore large libraries of possible drugs to identify starting molecules for human scientists to develop [25]. There are two main categories of Virtual Screening, Structure-Based and Ligand-Based. In Structure-Based Virtual Screening the structure of the target protein is known, whereas in Ligand-Based Virtual Screening we only have knowledge of the ligand, the small molecule which binds to the target protein. In this paper, we will explore MolDQN [49], a reinforcement learning model for the Ligand-Based Virtual Screening approach of Molecular Optimization, which takes a starting drug candidate and explores the space of similar molecules to find more and possibly improved drugs.

1.1 Molecular Optimization

A drug is a molecule which binds to a target protein in the body, to produce a desired physiological response in living organisms. In Molecular Optimization, a molecule known to have positive properties is altered to increase its effectiveness as a drug. These alterations could be at the functional group level, where groups of atoms are added or removed from the molecule or, as is the case with MolDQN, at the atomic level. Another consideration in Molecular Optimization is what chemical property to optimize; in the paper by Zhou et al. they use the chemical measures penalized logP [14], QED [6] and the Tanimoto Similarity [37]. They also consider Single, Constrained and Multi-Objective optimization which we will explain in more detail, along with exactly how the chemical measures are optimized, in Chapter 3.

Penalized logP LogP is the logarithm of the Partition Coefficient, which is a measure of how well the molecule dissolves in water or octanol. More specifically, it is the ratio of the solute between octanol and water. LogP can take any value in $\mathbb{R}$, with a high logP value meaning the molecule is hydrophobic and dissolves well in octanol and a low value meaning the molecule is hydrophilic and dissolves well in water. Penalized logP is the logP value of a molecule minus its Synthetic Accessibility (how easy a molecule is to synthesize).

QED Quantitative Estimation of Drug-likeness (QED) is a summary of key physiochemical properties such as molecular mass, hydrophobicity, aromaticity and polarity

[6]. The value of QED ranges between 0 and 1, with a low QED meaning low Drug-Likeness and a QED closer to 1 meaning a high Drug-Likeness.

Tanimoto Similarity The Tanimoto Similarity, also known as the Jaccard Index, is the measure of how similar two sample sets are (Equation 1.1). In the case of molecular similarity it is the ratio of chemical features shared by two molecules to the total number of chemical features in those molecules and can be used on molecular fingerprints. The Tanimoto Similarity also ranges from 0 to 1 with a low value meaning low similarity and a high value meaning high similarity.

$$T(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{1.1}$$

1.2 Chemical Representation on Computers

To create computational chemistry models, scientists need a way to represent molecules on a computer. There are several available methods including those in 1D, 2D and 3D. 1D representations, called molecular fingerprints, include information about what atoms are present and the bonds between atoms. Examples include the popular Simplified Molecular-Input Line-Entry System (SMILES) [44] and Extended-Connectivity Fingerprint [33] also known as the Morgan Fingerprint. 2D representations comprise of graphs where in most cases the vertices are atoms and the edges are bonds. Matrices are used to represent the graphical information about atom adjacency, bond types, bond lengths and more. Popular 2D representations include Ctab [8] and Molfiles developed by MDL information Systems. 3D representations are useful if stereographic information is important as in Structure-Based Virtual Screening.

The MolDQN algorithm uses the Morgan Fingerprint, a vector of length 2048, filled with 0s and 1s. The Fingerprints are generated using the Morgan Algorithm which is outlined in Algorithm 1 and a visual representation of the Morgan Algorithm can be seen in Figure 1.1. However, the Morgan Fingerprint is not easily understood by humans so in actuality MolDQN takes in a SMILES string and converts it into a Morgan Fingerprint for the algorithm to train on. The outputs of MolDQN are numbers which are translated into SMILES strings for easier user interpretation. Figure 1.2 shows how a SMILES string is constructed.

Algorithm 1 Morgan Algorithm

```
1. All atoms in the molecule are assigned an integer
2. Each atom identifier is updated to include information about the atom's neighbours
3. Remove any duplicate identifiers caused by a fragment appearing more than once in the molecule
4. Repeat steps 2. and 3. for the preset number of iterations
5. The final identifiers are hashed into a 2048 length vector of 0s and 1s
```

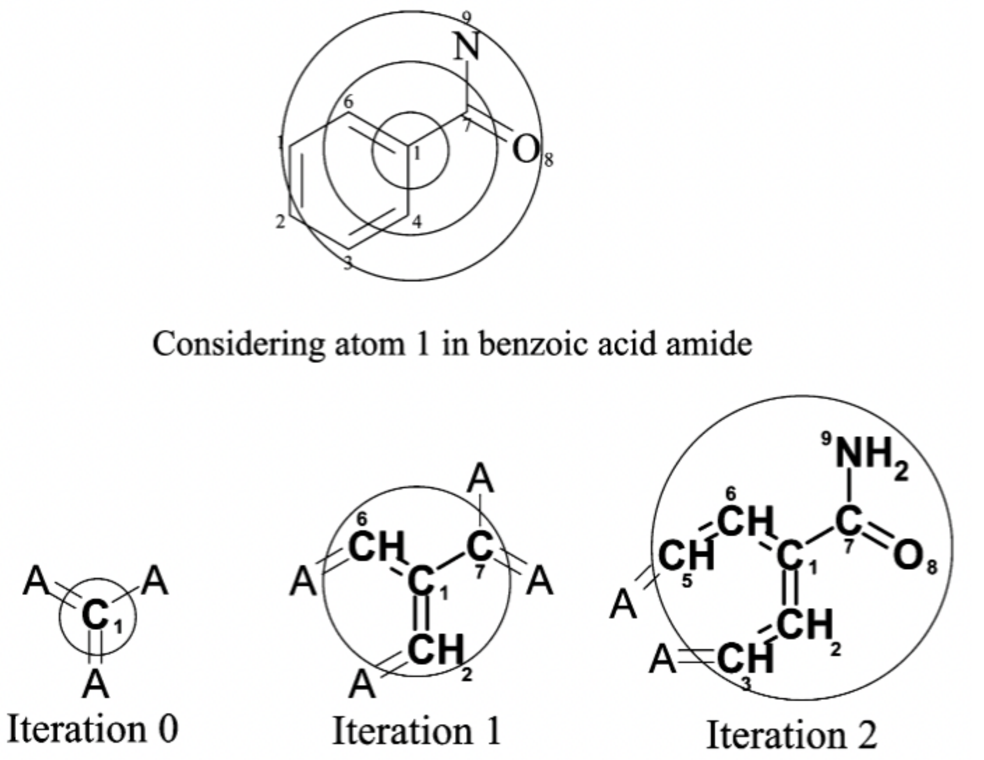


Figure 1.1: The Morgan Algorithm for Benzoic Acid Amide. (Taken from [33])

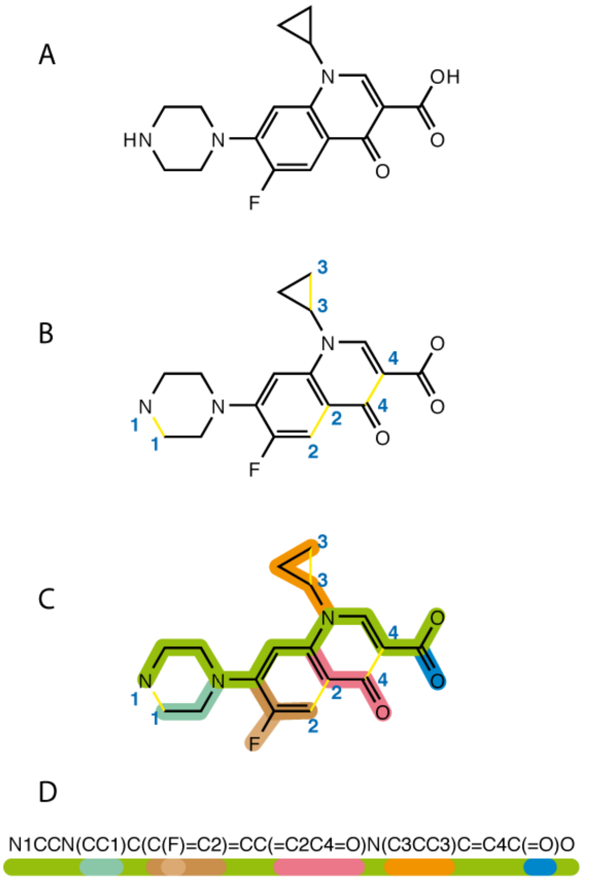


Figure 1.2: The construction of the SMILES string for Ciprofloxacin. (Taken from [9])

1.3 Related Work

There have been many molecular optimization models introduced, relying on a variety of machine learning architectures including Variational Autoencoders, Recurrent Neural Networks and Reinforcement Learning. A Variational Autoencoder (VA) is a probabilistic generative model that uses Neural Networks to encode and decode data to and from a latent space allowing for the generation of novel data. In the case of molecular optimization, the VA would be trained on a data-set of molecules in order to generate new molecules that could be possible drug candidates. Some VA models use 2D graph representations of molecules such as MDVAE, Monotoic Disentangled VAE [11], PCVAE, Property Controlable VAE [10] and JT-VAE, Junction Tree VAE [21]. Whereas others use 3D coordinates for its input such as The Coarse-Graining VAE [40].

Another machine learning method used in molecular optimization is the Recurrent Neural Network (RNN). An RNN can use previously outputted values as inputs allowing the NN to work on sequential data. One RNN model [35] uses Long Short-Term Memory [19] to generate new molecules from a data-set of SMILES strings. In the paper *Molecular Optimization by Capturing Chemist's Intuition Using Deep Neural Networks* [18] they experimented with two RNN models, one using the Seq2Seq framework and the other using a Transformer architecture.

Finally, the most popular machine learning framework used in molecular optimization is reinforcement learning. MolDQN, the model we will explore further, is one of many used in the field of drug design. GCPN, Graph Convolutional Policy Network [46], uses a Graph Neural Network along with a Markov Decision Process as the basis for its algorithm. RationaleRL [22] considers substructures of molecules responsible for different properties, which they call rationals. Molecules are optimized by adding/removing rationals using reinforcement learning. ORGAN, Objective-Reinforced Generative Adversarial Networks [16] relies on a Generative Adversarial Network and reinforcement learning to optimize molecules.

The models outlined above are just a few of the wide range of machine learning applications for molecular optimization, however, our project will focus on MolDQN. In Chapter 2, we give an in-depth explanation of the mathematical knowledge needed to understand how MolDQN works, including Q-Learning, Deep Q-Networks and Double Q-Learning. In Section 2.4 we explore how the model is optimized which will be the focus of our experiments. We follow this by a detailed description of the MolDQN model in Chapter 3 and in Chapter 4 we describe the set-up and results of our experiments with the choice of loss function. Finally, we conclude with an outline of further avenues of exploration to expand on our work in this dissertation and summarise our project. Whilst the theory and MolDQN model are the work of others, the experimental results and analysis is our own and, to the best of our knowledge, experimenting with the loss function for MolDQN has not appeared in the literature previously. Additionally, the figures (diagrams and charts) are our own unless otherwise stated.

Chapter 2

Mathematical Preliminaries

2.1 Reinforcement Learning

Reinforcement Learning is one of three paradigms of machine learning, the others being Supervised and Unsupervised Learning. The inspiration for the theory of Reinforcement Learning comes from two scientific fields, animal behavioural psychology and dynamic programming. In 1953 Psychologist B. F. Skinner introduced his theory that animals learn to behave in a way that minimises punishment and maximises reward. In 1957, Skinner proposed his *Reinforcement Theory of Motivation* [13] that states how humans act is dependent on our reactions to stimulus in our environment, good or bad, and that repeated rewards or punishments from the same stimulus will reinforce our behaviour.

At the same time that Skinner was developing his theories of animal behaviour, Richard Bellman in 1953 introduced Dynamic programming [3], an optimization technique that breaks down the main problem into sequential sub-problems and optimizes them individually. In 1957, Bellman developed his theory of dynamic programming and introduced Markovian Decision Processes (MDP) [2], a discrete-time stochastic control process used to model sequential decision making.

The modern concept of reinforcement learning, as a combination of animal behavioural theory and dynamic programming, comes largely due to the seminal thesis *Learning from Delayed Rewards* written by Christopher J.C.H. Watkins in 1989 [41]. Watkins took inspiration from Skinner's theory and built upon Bellman's Markovian Decision Process to derive what is now termed Q-Learning, a widely-used reinforcement learning algorithm. More formally, Reinforcement Learning aims to find the 'best' action an agent can take, through a series of trial and error in a sequentially updating environment, where 'best' means to optimize a prespecified reward. A decision process is called a policy which we will denote π.

As mentioned above, the mathematical foundation of a Reinforcement Learning algorithm is a Markovian (or Markov) Decision Process. A Markov Process is a series of states $S_0, \ldots, S_t,\ S_{t+1}$ where the probability of a state s occurring depends only on the previous state i.e $\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, S_2, \cdots, S_t]$. A MDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where

- $\mathcal{S}$ is the State space, the possible states the environment can take
- $\mathcal{A}$ is the Action space, the set of actions an agent can take
- $\mathcal{R} : \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function
- $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$.
- γ is the discount factor

In our case, we have a Deterministic MDP, meaning for any action and state the resulting state, s', is known or $\mathcal{P}^a_{ss'} = 1\ \forall\, a \in \mathcal{A},\ s \in \mathcal{S}$. Additionally, in MolDQN there are a fixed number of steps, T, the algorithm can take before it ends automatically.

Using the notation in [49], let $r_t(a, s) = \gamma^{T-t} \cdot R_t(a, s)$, where $R_t(a, s)$ is the reward of action a on state s at step t. The discount factor, γ, ensures that later rewards are weighted more heavily. For $a \in \mathcal{A}$ and $s \in \mathcal{S}$ let $Q^\pi(a, s)$, be the value of the action a on state s under the policy π, then $Q^\pi(a, s)$ is equal to the expected sum of future rewards:

$$Q^\pi(a, s) = \mathbb{E}_\pi \left[\sum_{n=t}^{T} r_n(a, s) \right]$$

The policy that results in the highest reward for each state is called the optimal policy and can be written as $\pi^*(s) = \arg\max_a Q^{\pi^*}(a, s)$. The aim of Reinforcement Learning is to find $\pi^*(s)$.

In some cases, we might want to optimize two or more rewards at the same time, this is termed Multi-Objective Reinforcement Learning. Suppose there are k properties we want to optimize, then for time t the reward in Multi-Objective RL is a vector of dimension k, $\overrightarrow{r_t} = [r_{1,t}, \ldots, r_{k,t}]^T \in \mathbb{R}^k$, where $r_{i,t}$ is the reward for property i at step t. In MolDQN they use a *scalarized reward framework* which produces a scalar summary of the multi-dimensional reward so it can be optimized. Given a set vector of weights $w = [w_1, w_2, \ldots, w_k]^T \in \mathbb{R}^k$, the scalarized reward equals:

$$r_{s,t} = w^T \overrightarrow{r_t} = \sum_{i=1}^{k} w_i r_{i,t}$$

2.2 Q-Learning

Q-Learning [41] is an off-policy reinforcement learning algorithm, which means that the target policy, $\pi^*(s)$ is different to the policy that is selecting the action during learning. The aim of Q-Learning is to find the optimal policy, $\pi^*(s)$, through finding the highest value of $Q^\pi(a,s)$ for every action and state combination. $Q^\pi(a,s)$ is called the Q-function and the Q-function evaluated for a particular action/state pair is called the Q-value. If we know all optimal Q-values then the optimal policy is to pick the action which results in the highest Q-value for any state.

Q-learning works by iteratively evaluating the Q-function for action/state pairs until it converges to the optimum Q-function, $Q^*(a,s) = \max_\pi Q^\pi(a,s)$. The evaluation is done using the Bellman Equation:

$$Q^*(a,s) = \mathbb{E}\left[R_{t+1}(a,s) + \gamma \cdot \max_{a'} Q^*(a',s')\right]$$

where a' is the best action to take at time $t+1$ and s' is the state from which a' can be taken. The Bellman Equation says that given an action a and state s, the optimal expected reward from time t ($Q^*(a,s)$) equals the expected reward at time $t+1$ ($\mathbb{E}\left[R_{t+1}(a,s)\right]$) plus the expected discounted reward, maximised over all possible actions and states at time $t+1$ ($\mathbb{E}\left[\gamma \cdot \max_{a'} Q^*(a',s')\right]$)

In traditional Q-Learning, the Q-values are kept in a Q-table which has actions for the columns and states for the rows, therefore each cell contains the Q-value for an action/state pair. As the Q-learning algorithm progresses, the Q-values are updated until they converge. In 1992, three years after submitting his thesis, Watkins and Peter Dayan proved that the Q-values would converge [42].

2.2.1 Deep Q-Learning

One problem with traditional Q-Learning is that as the Action and State spaces become larger it becomes computationally expensive to evaluate all Q-values. To address this issue, Volodymyr Mnih et al. introduced Deep Q learning [29] which uses a Deep Neural Network as an approximation for the Q-function instead of evaluating each action/state combination individually. This greatly reduces the computational cost, allowing the algorithm to learn a successful policy in high-dimensional environments.

A Deep Neural Network is a set of layers of interconnected nodes, termed neurons, that transmit information between each other. There is an input layer of nodes, a number of hidden layers, and an output layer, all of which can have any number of nodes. As a Deep Neural Network finds a map between inputs and outputs it can be used to approximate a function, which in our case is the Q-function. For a more detailed mathematical description of Deep Neural Networks, see Appendix A. In Deep Q-learning, the Neural Network, called the Deep Q-Network (DQN), has states as the inputs and the outputs are the Q-values for each action as shown in Figure 2.1.

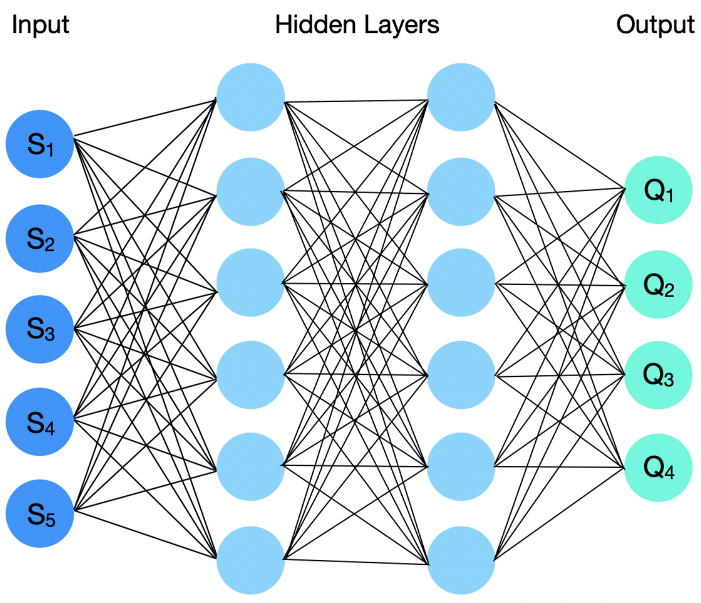


Figure 2.1: A Deep Q-Network where S_i = state i and Q_i = the Q-value of action i.

To evaluate the effectiveness of the DQN, a loss is used which compares the outputs of the network to the target outputs. Let $Q(a, s; \theta)$ be the Deep Q-Network, then the DQN is trained by minimising the expected loss:

$$l(\theta) = \mathbb{E}\left[f_t(y_t - Q(a, s; \theta))\right] \tag{2.1}$$

where y_t is the target output and f_t is a loss function. In traditional Q-learning the target output is:

$$y_t = R_t(a, s) + \max_a Q(a, s_{t+1}; \theta) \tag{2.2}$$

2.2.2 Double Q-Learning

The standard Q-learning algorithm is prone to overestimate action Q-values. Therefore, in 2010 Hado van Hasselt introduced an adaptation that he termed Double Q-Learning [17] which he, along with Arthur Guez and David Silver, later adapted to work with Deep Q-Networks [39] in 2015. In the Double DQN algorithm, to help prevent overestimation, the selection of an action is separated from the evaluation using the action. Rewriting Equation 2.2 we get:

$$y_t = R_t(a, s) + Q(\arg\max_a Q(a, s_{t+1}; \theta), s_{t+1}; \theta)$$

In Double DQN the target output is:

$$y_t = R_t(a, s) + Q(\arg\max_a Q(a, s_{t+1}; \theta), s_{t+1}; \boldsymbol{\theta'}) \tag{2.3}$$

where the action is selected using the most up-to-date information but the Q-value is evaluated using a second Q-Network which is only updated every C steps. The parameters for the second network, called the target Network, is denoted by $\boldsymbol{\theta'}$. In the 2015 paper, Hado van Hasselt et al. suggested that the overestimates were due to inaccurate approximations of the Q-values using a Neural Network and noise. They demonstrated, as shown in Figure 2.2 that Double DQN greatly reduced the overestimation. The true values are shown by the horizontal lines, as you can see the Double DQN estimate (blue) is closer to its true value than the standard DQN estimate (red).

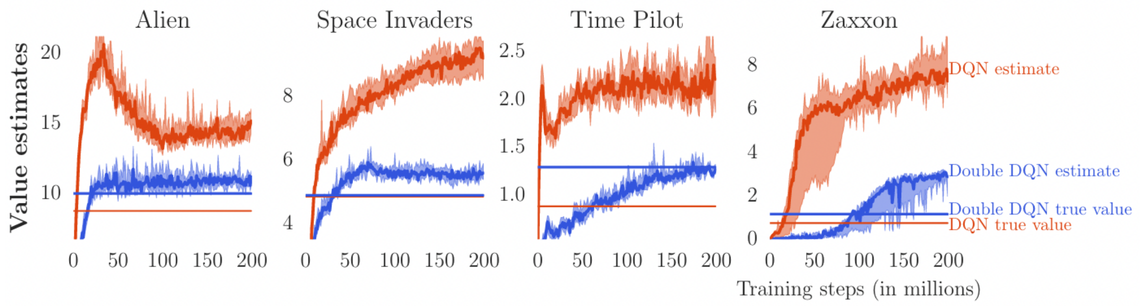


Figure 2.2: Value estimates for DQN (red) and Double DQN (blue) along with their true values (horizontal lines) on four Atari games, taken from the 2015 Double DQN paper [39].

2.2.3 Experience Replay

Another advancement to Q-Learning that is used in MolDQN is Experience Replay, a concept inspired by neuroscience and first applied to reinforcement learning by Long-Ji Lin in 1992 [26]. The intuition behind Experience Replay is to use the agent's prior experience to make learning more efficient. Let the model's experience at time t be represented by the tuple:

$$e_t = (a_t, s_t, r_{t+1}, s_{t+1})$$

where a_t and s_t is the action and state at time t, r_{t+1} is the reward for taking action a_t on state s_t, and s_{t+1} is the resulting state. In Experience Replay, the agent's experiences, e_t, are stored in a memory set, D, of size N, where N is a large finite number. Sample of experiences, called mini-batches, are randomly sampled from D and used to train the network. Initially when running a model, experience replay is not used as the memory set needs to be built first, then after a number of steps, samples can start being drawn from D. By using Experience Replay, we remove any correlations between consecutive steps and make use of past runs of the algorithm. This results in more efficient training.

2.3 Exploitation versus Exploration

To train the Q-Network we need a method to explore the action/state space. If we only investigate action/state pairs that have the highest Q-values at any given time then unexplored options that could result in higher rewards could be missed. Choosing to use known information and go with the highest reward is called Exploitation and choosing to randomly select the next, possibly unvisited, action/state pair is called Exploration. One of the simplest Exploitation/Exploration methods is the ϵ-Greedy algorithm (2) which, for a preset value $\epsilon \in (0, 1)$, exploits with probability $1 - \epsilon$ and explores with probability ϵ.
ϵ is usually small (≤ 0.1) to take advantage of known data. MolDQN uses an ϵ-Greedy algorithm, with linearly decreasing ϵ which means that as the time gets closer to the last step the algorithm exploits with greater probability. One disadvantage of the ϵ-Greedy algorithm is that it wastes time exploring options that are known to produce low Q-values. Therefore, MolDQN combines the ϵ-Greedy algorithm with Bootstrapped DQN.

Introduced by Ian Osband et al., Bootstrapped DQN [30] uses K Q-functions, each trained separately using different samples. MolDQN uses a multi-task Neural

Algorithm 2 ϵ-Greedy algorithm

```
Input: $\epsilon \in (0, 1)$
  for $t = 1, \ldots, T$ do
      $p \sim \mathcal{U}(0, 1)$
      if $p < \epsilon$ then
          Return $a_t \sim \mathcal{U}\{1, \cdots, |\mathcal{S}|\}$
      else
          Return $a_t = arg\, max_a Q^*(a, s)$
      end if
  end for
```

Network with K separate heads, $Q^{(k)}$, each of which is trained using a different set of data. For each episode of the algorithm, a head is chosen uniformly at random, $k \sim \mathcal{U}\{1, \cdots, K\}$ and is used normally as in a non Bootstrapped DQN.

2.4 Optimization of the Loss Function

To recap, the goal when training a Q-Network is to minimize the expected loss function as written in Equation 2.1, which we display below for convenience:

$$l(\theta) = \mathbb{E}\left[f_t(y_t - Q(a, s; \theta))\right] \tag{2.1}$$

There are two important considerations for optimization, which loss function, f_t, to use and which optimizer algorithm to use. Experimenting with changes to the loss function, in order to improve training, is the main aim of our dissertation.

2.4.1 Loss Function

There is a wide range of options for the loss function, f_t, in Equation 2.1. Two popular choices of loss function used in Neural Network training are the absolute loss, $f(x) = |x|$, and the squared error loss, $f(x) = x^2$. MolDQN uses the Huber Loss [20] which is a combination of these:

$$L_\delta(x) = \begin{cases} \frac{1}{2}x^2 & \text{for} |x| \leq \delta \\ \delta \cdot \left(|x| - \frac{1}{2}\delta\right), & \text{otherwise} \end{cases} \tag{2.4}$$

The Huber Loss is robust to outliers as it uses the absolute loss for large values of x, but is retains the sensitivity and differentiability of the squared error loss for small values of x. In MolDQN they set $\delta = 1$, we will explore how changing the value of δ affects the loss.

We also explore using the Pseudo-Huber Loss [7], a smooth approximation of the standard Huber Loss, which means the function is differentiable everywhere. The Pseudo-Huber Loss is written as:

$$L_\delta(x) = \delta^2 \left(\sqrt{1 + (x/\delta)^2} - 1 \right) \tag{2.5}$$

2.4.2 Optimizer Algorithm

When minimizing convex functions, we can differentiate and set to zero to find the optimal point. However, in Neural Network training, we often have highly non-convex loss landscapes to navigate. Therefore an optimizer algorithm is used to find the global minimum and to iteratively update the parameters of the neural network through backpropagation (see Appendix A for more details). In MolDQN they use the optimizer, Adam [23], which is an adaptation to stochastic gradient descent (SGD).

Traditional Gradient Descent uses all data available to calculate the gradient of the current point on the loss landscape and then moves in the opposite direction. On the other hand, SGD uses mini-batches of the data to calculate the gradient, resulting in a quicker algorithm. In SGD the parameter update at step t is:

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \alpha \nabla_{\boldsymbol{\theta}_{t-1}} f_i \left(\boldsymbol{\theta}_{t-1} \right)$$

where i denotes a random sample used to approximate the true gradient $f\left(\theta_t\right)$ and α is called the learning rate. For SGD, α is a fixed constant which needs to be manually set.

ADAptive Moment estimation, or Adam, is a combination of two progressions of SGD, AdaGrad [12] and RSMProp [38]. AdaGrad uses a different learning rate for each parameter and adjusts it automatically, allowing for lower or higher learning rates depending on if the feature is frequent or infrequent respectively. RSMProp, which was built upon AdaGrad, uses a decaying moving average of past gradients to calculate the learning rate. Adam uses both the average of past gradients (the first moment) and the uncentered variance (the second moment). The parameter update for Adam is shown in Equation 2.6 and the full algorithm can be seen in Algorithm 3. Note: κ is a small constant.

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \alpha \cdot \hat{\boldsymbol{m}}_t / \left(\sqrt{\hat{\boldsymbol{v}}_t} + \kappa \right) \tag{2.6}$$

One issue that might arise during the optimization process is exploding gradients. This is when the gradients calculated by the optimizer algorithm become very large,

Algorithm 3 Adam optimizer

```
Input: $\alpha$: Learning rate
Input: $\beta_1, \beta_2 \in [0, 1)$: Exponential Decay Rates for moment estimators
Input: $f(\theta)$: Stochastic objective function
Input: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$: Initialize $1^{st}$ moment estimator
  $v_0 \leftarrow 0$: Initialize $2^{nd}$ moment estimator
  $t \leftarrow 0$: Initialize timestep
  while $\theta_t$ not converged do
      $t \leftarrow t + 1$
      $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$ )
      $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
      $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
      $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
      $\widehat{v}_t \leftarrow v_t / \left(1 - \beta_2^L\right)$ (Compute bias-corrected second raw moment estimate)
      $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / \left(\sqrt{\widehat{v}_t} + \kappa\right)$ (Update parameters)
  end while
```

which, when accumulated, cause unstable training. MolDQN uses the solution gradient clipping which caps the possible size of the gradient to a pre-set value, G, before feeding it into the backpropogation algorithm. This prevents any sudden changes in the parameters of the network and results in more stable training.

Chapter 3

The MolDQN Model

MolDQN aims to perform molecular optimization through Deep Q-Networks. The State Space, $\mathcal{S}$, is the set of tuples, (m, t), where m is a molecule and t is the step number. $\mathcal{A}$ is the set of possible changes to the molecule that can be taken, in the case of MolDQN, only chemically valid actions can be taken resulting in only chemically valid molecules being produced. There are three possible actions:

1. **Atom Addition** Let $\mathcal{E}$ be the set of atoms in molecule $\mathcal{M}$, then there are two steps to atom addition. The first is to add an atom $a \in \mathcal{E}$ and the second is to add a bond from the new atom to $\mathcal{M}$. Hydrogen atoms are considered implicitly, meaning they are added along with a as well as removed from $\mathcal{M}$ to make way for the new atom. The atom can be joined via Single, Double or Triple bonds, each considered its own action.

2. **Bond Addition** The second possible action is to add a bond between any two atoms in $\mathcal{M}$ as long as it follows valency rules. The possible bond addition depends on if there is already a bond between two atoms and what type of bond it is:

   - No bond $\rightarrow$ {Single, Double, Triple} Bond
   - Single bond $\rightarrow$ {Double, Triple} Bond
   - Double bond $\rightarrow$ {Triple} Bond

   To prevent producing a molecule where there is a high amount of energy being contained (chemical strain) bonds are not allowed to form between two atoms that are already in a ring.

3. **Bond Removal** The final action that can be taken is to remove a bond between two atoms. The possible options are:

- Triple bond $\rightarrow$ {Double,Single,No} Bond
- Double bond $\rightarrow$ {Single,No} Bond
- Single bond $\rightarrow$ {No} Bond

Bonds can only be removed when it leaves no or one atom disconnected. Additionally, a bond cannot be removed if it breaks aromaticity.

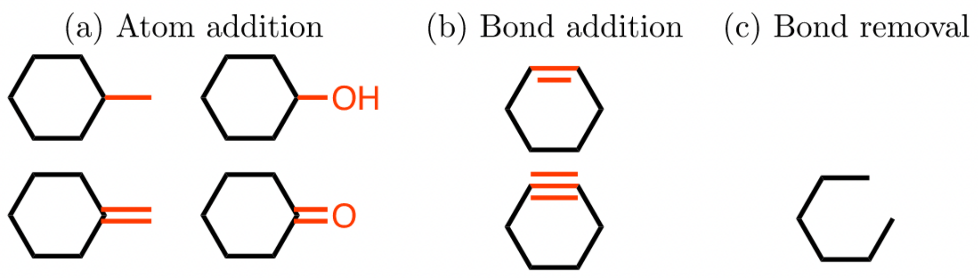


Figure 3.1: The valid actions the MolDQN agent can take on an example molecule, cyclohexane, taken from the MolDQN paper [49]. Changes are shown in red.

As mentioned in Section 2.1, MolDQN is founded on a deterministic MDP, meaning for a given molecule if we take an action, the resulting molecule is fixed, there is no element of randomness in what molecule will be produced. Also, there is a fixed number of steps, T. This was decided to prevent the molecule from becoming too dissimilar from the original molecule. The discount factor, γ, is set to 0.9.

The input values are vectors comprising of a Morgan Fingerprint (as described in Section 1.2) with radius 3, length 2048 and the time step, t, concatenated onto the end. In actuality the program takes in a SMILES string and converts it to a Morgan Fingerprint to use in the algorithm. The output of the model is a vector of length, K, with each of the values corresponding to one of the K heads in the bootstrapped DQN framework. The K numbers in the vector output have corresponding SMILES values which is what the user is given.

The fundamental base of MolDQN is a four layer fully-connected neural network with layer sizes [1024, 512, 128, 32] and ReLU activation function. The algorithm uses the advancements Double Q-Learning and Experience Replay. To control the exploitation/exploration balance during training, MolDQN uses the ϵ-Greedy algorithm along with Bootstrapped DQN as explained in Section 2.3. The model is trained using the Huber Loss (Section 2.4.1) and the optimizer Adam is used along with gradient clipping with a cap of 10 (Section 2.4.2).

In the MolDQN paper, the molecular properties they wanted to optimize were QED and penalized logP, taking into account the Tanimoto similarity of the starting molecule and the one produced (Section 1.1). They considered different choices for the reward function, depending on whether they were using Single Optimization, Constrained Optimization or Multi-objective Optimization.

Single Optimization In Single property Optimization the reward function is either penalized logP or QED. However, in the MolDQN paper Zhenpeng Zhou et al. found that optimizing penalised logP alone, without any constraint, resulted in MolDQN producing non drug-like molecules as can be seen in their shape in Figure 3.2. A drug molecule has a collection of chains, chemical rings, and functional groups, the molecules in Figure 3.2 clearly don't follow these rules. Therefore, using another metric to guide the optimization was deemed necessary.

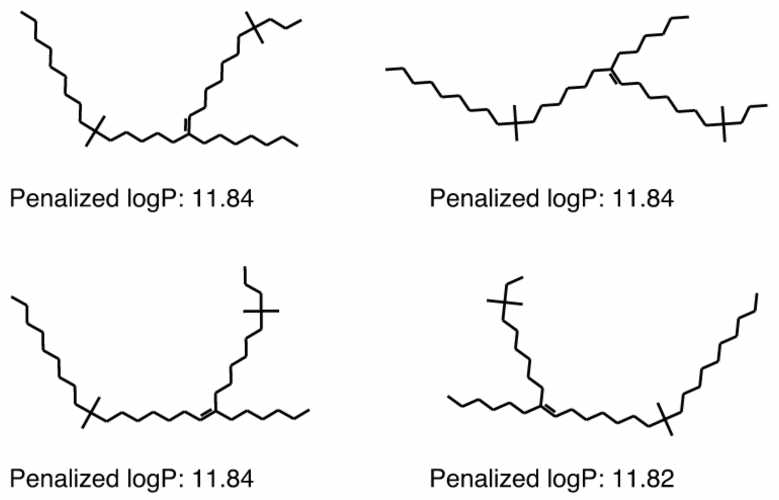


Figure 3.2: Molecules produced by MolDQN using Single Optimization of penalized logP. (Taken from [49])

Constrained Optimization In Constrained Optimization the goal is to produce a molecule with the highest penalized logP value while maintaining a Tanimoto similarity above a threshold, η. Let m_0 be the original molecule and m be the produced molecule then the reward function for constrained optimization is:

$$\mathcal{R}(s) = \begin{cases} \log \mathrm{P}(m) - \lambda \times (\eta - \mathrm{SIM}\,(m, m_0)) & \text{if } \mathrm{SIM}\,(m, m_0) < \eta \\ \log \mathrm{P}(m) & \text{otherwise} \end{cases}$$

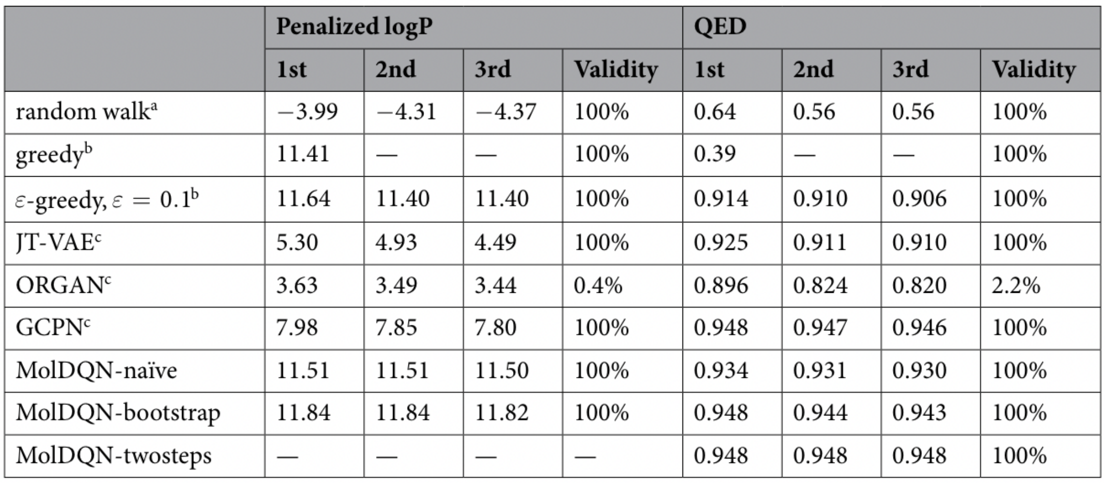
where logP(m) is the penalized logP of molecule m and λ is a constant used to control the weight of the similarity.

Multi-objective Optimization In the task of Multi-Objective Optimization the aim is to consider both the QED of a molecule and the Tanimoto similarity. The reward is a 2-dimensional vector $[\text{QED}(m),\ \text{SIM}(m, m_0)]$ and during optimization they use the scalarized multi-objective reward function:

$$\mathcal{R}(s) = w \times \text{SIM}(s) + (1 - w) \times \text{QED}(s)$$

where again w is a weight used to control the weight of the similarity and the QED.

In their paper, Zhenpeng Zhou et al. demonstrated that MolDQN outperforms several existing algorithms. Figure 3.3 shows that MolDQN produces the top 3 highest scoring molecules in both the penalized logP and QED measures.



	Penalized logP				QED			
	1st	2nd	3rd	Validity	1st	2nd	3rd	Validity
random walk[a]	−3.99	−4.31	−4.37	100%	0.64	0.56	0.56	100%
greedy[b]	11.41	—	—	100%	0.39	—	—	100%
ε-greedy, $\varepsilon = 0.1$[b]	11.64	11.40	11.40	100%	0.914	0.910	0.906	100%
JT-VAE[c]	5.30	4.93	4.49	100%	0.925	0.911	0.910	100%
ORGAN[c]	3.63	3.49	3.44	0.4%	0.896	0.824	0.820	2.2%
GCPN[c]	7.98	7.85	7.80	100%	0.948	0.947	0.946	100%
MolDQN-naïve	11.51	11.51	11.50	100%	0.934	0.931	0.930	100%
MolDQN-bootstrap	11.84	11.84	11.82	100%	0.948	0.944	0.943	100%
MolDQN-twosteps	—	—	—	—	0.948	0.948	0.948	100%

Figure 3.3: Top three unique molecule property scores found by each method. a - "random walk" is a baseline that chooses a random action for each step. b - "greedy" is a baseline that chooses the action that leads to the molecule with the highest reward for each step. "ϵ-greedy" follows the "random" policy with probability ϵ, and "greedy" policy with probability 1-ϵ. In contrast, the ϵ-greedy MolDQN models choose actions based on predicted Q-values rather than rewards (Taken from [49])

Chapter 4

Experiment

In this project, we explore how changes to the Huber Loss function affect the output of the MolDQN model. We use Multi-Objective Optimizaton on the molecule troglitazone (4.1) and use the change in training error and the score of the output molecules to measure the effect of our alterations where the score of a molecule, m, is:

$$\text{Score}(m) = \frac{1}{2} \cdot \text{Sim}(m) + \frac{1}{2} \cdot \text{QED}(m)$$

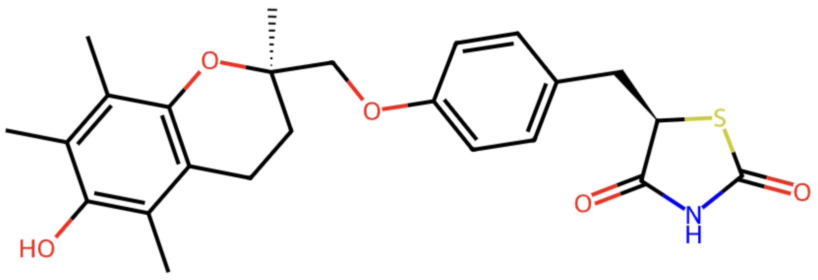


Figure 4.1: A 2D drawing of troglitazone using RDKit.

4.1 Set-up

Our MolDQN set-up follows the model described in Chapter 3. The similarity weight for the multi-objective optimization is set to 0.5, resulting in the reward being the average of the Tanimoto Similarity and the QED. The score was chosen to be equal to the reward. For a full listing of the model parameters see Appendix B.

We used code written by Kamen Petrov [31], a Part II MChem Student, who adapted the paper's original code [15] for their thesis project. See Appendix C for the main body of the code.

Each run of the program has 300 episodes and each episode contains 10 steps, so there is a total of 3000 steps. The training error is calculated every four steps once the algorithm has passed step 308. Therefore the training error is calculated 672 (=(3000-312)/4) times. This threshold was chosen to remove erratic data from the earlier episodes. The training error is shown in Equation 4.1.

$$Q_t(a, s) - R_t(a, s) - \xi \cdot Q_{t+1}(a', s') \tag{4.1}$$

where $Q_t(a, s)$ and $R_t(a, s)$ are the Q-function and reward at time t respectively, ξ is a constant and $Q_{t+1}(a', s')$ is the Q-function for time $t + 1$. In our experiment, ξ is equal to one. The Tanimoto Similarity and QED are evaluated for each step, therefore we can calculate the scores for all molecules produced over the 3000 step reinforcement learning algorithm.

For the ϵ-Greedy algorithm, the value of ϵ starts at 1 then is linearly decreased over the first 150 steps to 0.1. For the next 150 steps, the value of ϵ is linearly decreased again from 0.1 to 0.01.

As described in Section 2.4.1, The Huber Loss is a combination of the absolute loss and the squared error loss. The larger the delta value, the larger the range of values where the SE loss is used. As the squared error loss is differentiable we wanted to test whether increasing the delta value of the Huber Loss would result in better optimization. Therefore, we used delta values ranging from 1 to 2 in increments of 0.05 and ran the model five times for each value. Each run took approximately half an hour on a 10-core CPU (Apple M1 Pro). The adaptations were made by changing the delta value, denoted by `d` in the code snippet below (lines 63-65 in the code given in Appendix C). Note: in the code, the training error is represented by `td_error`.

```
errors = tf.where(
    tf.abs(td_error) < d, tf.square(td_error) * 0.5,
    d * (tf.abs(td_error) - 0.5 * d))
```

We also explored whether using the Pseudo-Huber Loss (Equation 2.5) would improve the model's outputs due to its smoothness. We ran the code five times when using the Pseudo-Huber Loss, where we set the delta value equal to 1 as shown below.

```
errors = tf.sqrt(1 + tf.square(td_error)) - 1
```

4.2 Results

In this section, we present the results of our experiments. Note: all values are given to three significant figures.

4.2.1 Training Error

We first looked at how changing the delta value of the Huber Loss would affect the training error of the model. To see how the training error varied over the steps, we took an average of the five runs for the standard Huber Loss with delta value equal to one and plotted the trend. The `td_error` data was extremely noisy, even when removing earlier stages, therefore we also include a moving average for the trend. The moving average for each point was calculated by taking an average of the 19 closest values, the nine before and the nine following. The evolution of the `td_error` and moving average can be seen in Fig 4.2.

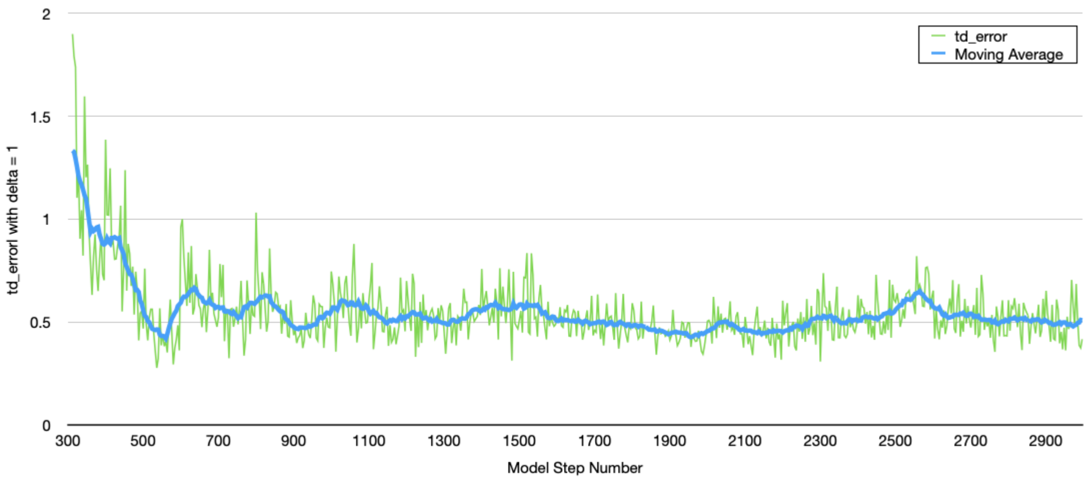


Figure 4.2: A plot with the `td_error` on the y-axis and the number of model steps on the x-axis. The green line represents the average td_error taken over the five runs and the blue line represents a moving average for the green line.

For each value of delta between 1 and 2 (0.05 increments) we calculated an average of the `td_error` over all five runs and all 672 steps where the training error is calculated. Resulting in an average taken over 3360 data points. We also calculated a 95% Confidence Interval (CI) for each average to better understand if any differences in the average were statistically significant or due to noise. The averages along with the upper and lower 95% Confidence Intervals are shown in Figure 4.3. The average `td_error` for the standard Huber Loss with delta = 1 is 0.550 with 95% CI [0.541, 0.559]. As you can see in 4.3 two values of delta give an average training error below the lower CI for the standard set-up, these being delta = 1.05 (which has an upper CI lower than 0.541) and 1.20. In Section 4.2.2 we explore the molecules being produced by the models with these delta values.

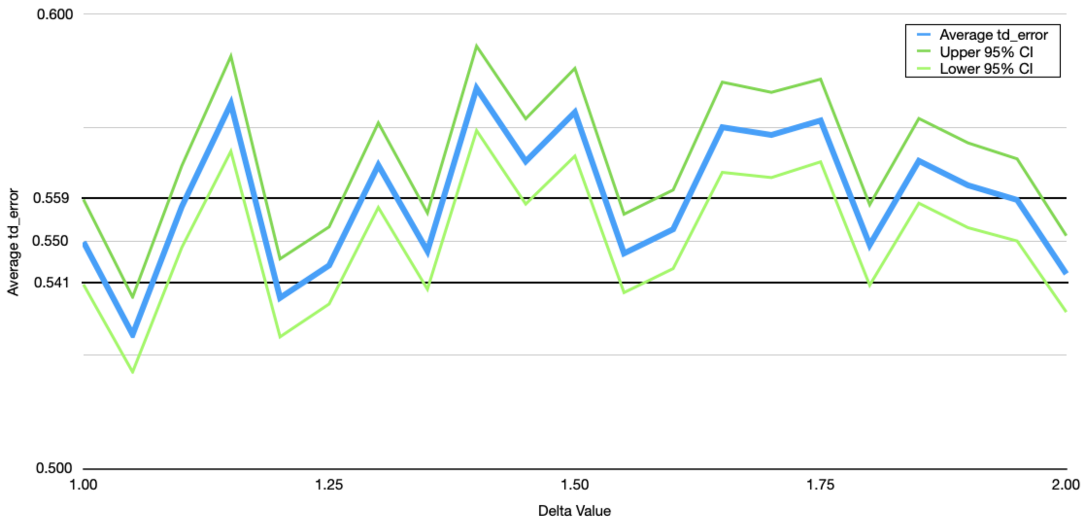


Figure 4.3: A plot with the average `td_error` and 95% CI on the y-axis and the delta value on the x-axis. The delta values ranged from 1 to 2 in 0.05 increments. The average `td_error` is represented by the blue line and the CIs by the green lines. The two black horizontal lines reference the Upper and Lower CI for the average `td_error` with delta value equal to one.

When we looked at the Pseudo-Huber Loss we got an average training error of 0.531 with 95% CI [0.522,0.540]. Compare this with the standard Huber loss, you see that the upper CI for the Pseudo-Huber Loss is below the lower CI for the standard Huber Loss set-up (0.541). Therefore, in Section 4.2.2 we also go on to explore the molecules being produced by the Pseudo-Huber Loss model.

4.2.2 Molecule Score

We next looked at the molecule scores for the models with delta value equal to 1, 1.05, 1.2 and the model with the Pseudo-Huber Loss. Note: the higher the score the better. We took the molecules that gave the top ten scores over all steps and over the five runs (a total of 15,000 molecules for each parameter) and created a heat map as can be seen in Table 4.2.2. The heat map shows that the molecules being produced have similar scores and in fact produce the same molecules. The model set-ups with $\delta = 1.05$ and 1.2 have the same 10 highest molecule scores and do not vary until

Table 4.1: Top 10 Molecule Scores for varying set-ups

$\delta = 1$	$\delta = 1.05$	$\delta = 1.2$	Pseudo-Huber
0.762	0.749	0.749	0.749
0.749	0.744	0.744	0.742
0.742	0.742	0.742	0.738
0.741	0.738	0.738	0.737
0.738	0.737	0.737	0.735
0.737	0.735	0.735	0.734
0.735	0.734	0.734	0.73
0.734	0.73	0.73	0.728
0.73	0.728	0.728	0.726
0.728	0.726	0.726	0.725

the 14th highest molecule score. Ultimately, the set-up that produced the highest molecule scores was the standard Huber Loss with delta value 1.

The molecules produced by the MolDQN experiments had small changes to the original troglitazone molecule. In Figure 4.4 we show the five highest scoring molecules over all runs of the code along with the original molecule. The molecule images were created using the Python package RDKit.

When performing experiments, there is always the challenge of finding the right balance between depth and breadth. There wasn't a lot of variation between the five runs we performed for each set-up so we feel the depth was satisfactory. However, if we had more time we would have liked to explore more delta values, both for the standard and Pseudo Huber Loss. Another area we would have liked to explore is changes to the optimizer. There have been several improvements to the Adam algorithm introduced, we give a brief outline of three of these in the next Section.

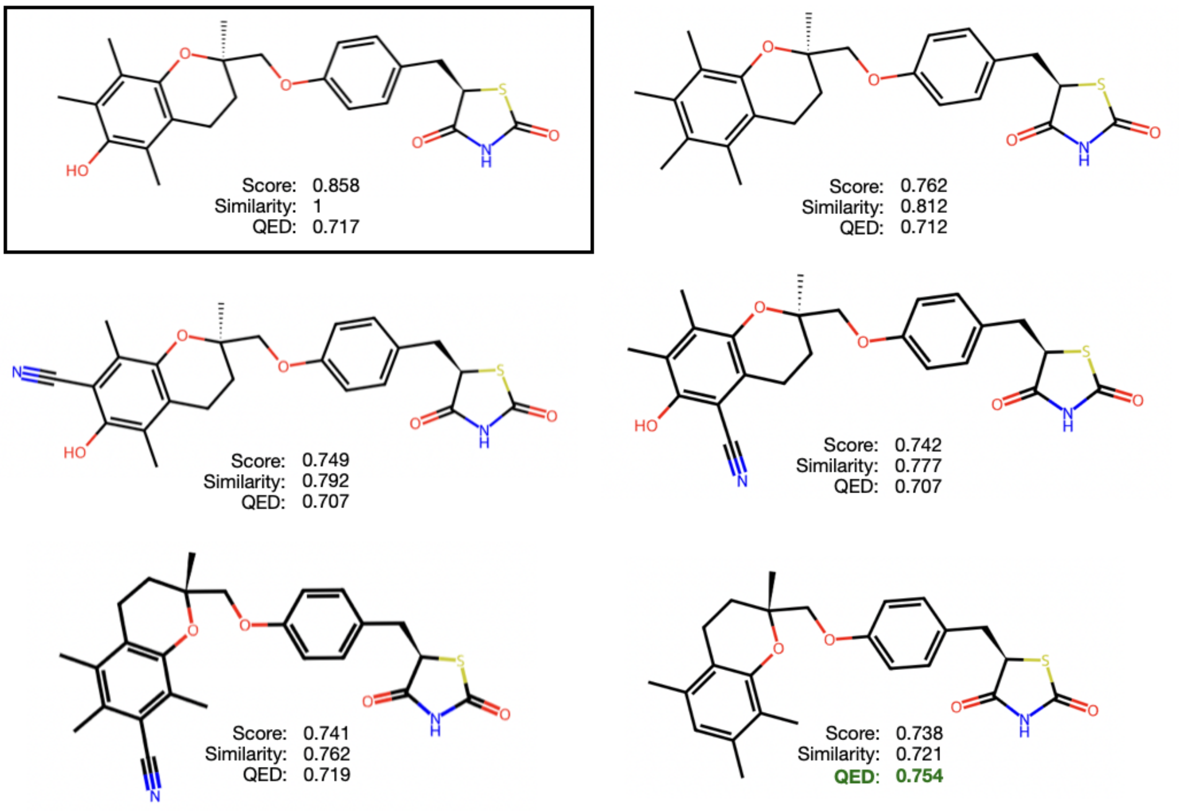


Figure 4.4: The six highest scoring molecules, along with their Similarity and QED values. The original molecule, troglitazone, is shown with a black border and the molecule with the highest QED has its value in bold green text.

Chapter 5

Discussion

5.1 Further Exploration

If we had more time we would have liked to explore whether changes to the Optimizer algorithm would improve the optimization of the loss function. MolDQN uses the traditional Adam Optimizer, however, there have been several adaptations made to Adam that are worth testing, a few of which we outline below. The MolDQN code was written using a now outdated Python package, Tensorflow 1. With more time we could update the code to work on Tensorflow 2 which does support some of the Adam advancements.

AdamW In 2017 Ilya Loshchilov and Frank Hutter introduced the widely-adopted AdamW algorithm [27], which stands for Adam with decoupled Weight decay. They note, as first suggested by Wilson et.el. [45], that Adaptive Gradient Optimizers such as Adam do not always generalize as well as SGD with Momentum and that L2 regularization is not nearly as effective in Adam as it is in SGD. AdamW improves on Adam by "decoupling the weight decay from the gradient-based update". Repeating from Section 2.4.2, Equation 2.6 is the traditional Adam parameter update and Equation 5.1 is the parameter update for the AdamW algorithm.

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \alpha \cdot \hat{\boldsymbol{m}}_t / \left(\sqrt{\hat{\boldsymbol{v}}_t} + \kappa\right) \tag{2.6}$$

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left(\alpha \cdot \hat{\boldsymbol{m}}_t / \left(\sqrt{\hat{\boldsymbol{v}}_t} + \kappa\right) + \lambda \boldsymbol{\theta}_{t-1}\right) \tag{5.1}$$

ND-Adam Another improvement, proposed by Zijun Zhang et al. [47], is ND-Adam, Normalized Direction-preserving Adam. Zhang et al. also noted that Adam didn't always generalize well and stated that ND-Adam allows for "more precise control of the direction and step size for updating weight vectors". In the ND-Adam algorithm the learning rate is adapted to each weight vector, instead of each individual weight, and the direction of the gradient is preserved. Also, the magnitude of each weight vector is kept constant, only the direction varies allowing for more control.

QHAdam Jerry Ma and Denis Yarats [28] introduced QHAdam, Quasi-Hyperbolic Adam. QHAdam replaces the moment estimators used in the parameter updates with quasi-hyperbolic terms. Ma et al. suggests that these changes "decrease the variance of the momentum buffer". The parameter update is set as the weighted average of the momentum and the current gradient; it is shown in Equation 5.2.

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \alpha \left[\frac{(1-\nu_1) \cdot \boldsymbol{g}_{t-1} + \nu_1 \cdot \hat{\boldsymbol{m}}_t}{\sqrt{(1-\nu_2)\left(\boldsymbol{g}_{t-1}\right)^2 + \nu_2 \cdot \hat{\boldsymbol{v}}_t} + \kappa} \right] \tag{5.2}$$

5.2 Conclusion

Finding new machine learning applications can save us both time and money in drug discovery. In this dissertation we explored one reinforcement learning model, MolDQN introduced by Zhou et al. We described the ligand-based virtual screening method Molecular Optimization, the aim of MolDQN and outlined how molecules are represented in the algorithm. We also gave detailed descriptions of the three chemical measures used in MolDQN, penalized logP, QED and the Tanimoto Similarity. In the Preliminary Chapter, we gave detailed mathematical explanations for all machine learning theory needed to understand how MolDQN works. These theories included the basics of Reinforcement Learning followed by a closer look at Q-Learning, the foundations of MolDQN. We then described the Q-Learning advancements used by Zhou et al. which were Deep Q-Networks, Double Q-Learning and Experience Replay. We also explained the Exploitation/Exploration method used, ϵ-Greedy with Bootstrapped DQN. Finally, in the preliminary chapter we described the optimization process in MolDQN, which involved the Huber Loss, the focus of our experiments and the Adam Optimization Algorithm.

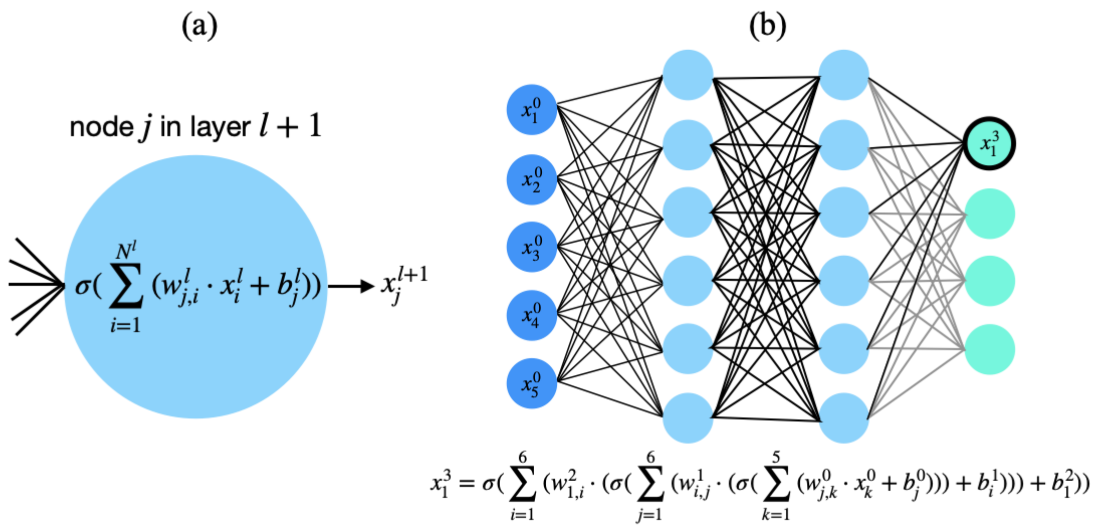
In the next Chapter, we gave a detailed description of MolDQN, including the possible actions the agent can take, the input of the algorithm and the Deep Neural Network that is the foundation of the model. We also outlined the three optimization techniques used, Single, Constrained and Multi-Objective. In the final chapter, we explored two changes to the Huber Loss, using different delta values and using the

Psuedo-Huber Loss. We began by outlining the important details of the experimental set-up, leaving the full model parameters for Appendix B. We looked at the training error of the model and the scores of the molecules being produced. Whilst some alternative delta values resulted in lower average training errors, when looking at the molecule scores, the average of the Tanimoto Similarity and QED, we found that changing the delta values or using the Psuedo-Huber Loss resulted in no improvements and in fact produced the same molecules. We conclude that the set-up used by Zhou et al. was the best out of those explored. Finally, we noted that there have been several improvements introduced to the Adam Optimizer used in MolDQN and therefore exploring these advancements has worth.

Appendix A

Explanation of Neural Networks

As mentioned in Section 2.2.1 a Neural Network is a set of layers of interconnected nodes. Each node holds a piece of data which we will label x_i^l where l denotes the layer and i denotes which node within the layer. Let L be the number of layers and N^l the number of nodes within layer l then the input and output nodes are labeled x_i^0 and x_i^{L-1} respectively. Each node takes in a weighted sum of all inputs from the previous layer with a bias term added. Then a non-linear function σ, called the activation function, is applied (see Figure A.1(a)). For a visualization of how an output of a four layer neural network relates to the inputs see Figure A.1(b).



(a)
(b)
node j in layer $l+1$
$\sigma(\sum_{i=1}^{N^l}(w_{j,i}^l \cdot x_i^l + b_j^l)) \rightarrow x_j^{l+1}$
x_1^0
x_2^0
x_3^0
x_4^0
x_5^0
x_1^3
$x_1^3 = \sigma(\sum_{i=1}^{6}(w_{1,i}^2 \cdot (\sigma(\sum_{j=1}^{6}(w_{i,j}^1 \cdot (\sigma(\sum_{k=1}^{5}(w_{j,k}^0 \cdot x_k^0 + b_j^0))) + b_i^1))) + b_1^2))$

Figure A.1: (a) The calculations for the output of node j in layer $l+1$, where $i = 1, \ldots, N^l$ (b) The value of an output for a four layer NN in terms of the inputs.

When training Neural Networks, backpropagation [34] is used to update the weight parameters. Let C be the objective function, what we want to optimize during training. In the case of MolDQN this is the expected loss function. To update the network parameters, $w^l_{j,i}$ and b^l_j, we need to find the partial derivative of C with respect to these parameters. Let

$$z^l_j = \sum_{i=1}^{N^l} (w^l_{j,i} \cdot x^l_i + b^l_j)$$

We can use the chain rule (Equation A.1) and the partial derivative of z^l_j with respect to $w^l_{j,i}$ (Equation A.2) to find $\frac{\partial C}{\partial w^l_{j,i}}$ (Equation A.3).

$$\frac{\partial C}{\partial w^l_{j,i}} = \frac{\partial C}{\partial z^l_j} \frac{\partial z^l_j}{\partial w^l_{j,i}} \tag{A.1}$$

$$\frac{\partial z^l_j}{\partial w^l_{j,i}} = x^l_i \tag{A.2}$$

$$\frac{\partial C}{\partial w^l_{j,i}} = \frac{\partial C}{\partial z^l_j} x^l_i \tag{A.3}$$

Similarly, to find $\frac{\partial C}{\partial b^l_j}$ we have:

$$\frac{\partial C}{\partial b^l_j} = \frac{\partial C}{\partial z^l_j} \frac{\partial z^l_j}{\partial b^l_j}$$

$$\frac{\partial z^l_j}{\partial b^l_j} = 1$$

$$\frac{\partial C}{\partial b^l_j} = \frac{\partial C}{\partial z^l_j}$$

Then, using the traditional backpropagation algorithm, the parameters of a Neural Network are updated using the equations:

$$w := w - \alpha \frac{\partial C}{\partial w}$$

$$b := b - \alpha \frac{\partial C}{\partial b}$$

Appendix B

Model Parameters

In Table B.1 I lay out the full model parameters I used throughout my experiments. I use * as a shorthand for "denoted by"

Parameter	Value
Atom Types that can be added	Carbon, Nitrogen, Oxygen
Molecule Ring Sizes allowed	5, 6
Morgan Fingerprint (input) Radius	3
Morgan Fingerprint (input) Length	2048
Layer Sizes for Fully Connected Neural Network	1024, 512, 128, 32
Activation Function for Neural Network	ReLU
№ of Episodes	300
№ of steps per episode, $*T$ in 2.1	10
№ of steps between each training operation	4
№ of steps between updates for target Q-Network, $*C$ in 2.2.2	20
Epsilon (for ϵ-Greedy 2.3) Start Value	1
№ of Heads for Bootstrapped DQN, *K in 2.3	12
Experience Replay Buffer Size, $*N$ in 2.2.3	5000
Experience Replay Mini-Batch Size	128
Xi value in reward, $*\xi$ in Eq 4.1	1
Reward Discount Factor, $*\gamma$ in 2.1	0.9
Optimizer	Adam
Learning rate of Optimizer, $*\alpha$ in Eq 2.6	0.0001
Gradient Clipping Value, $*G$ in 2.4.2	10

Table B.1: Model Parameters for my MolDQN implementation

Appendix C

Code

The most important section of code, I give here. The code appertaining to the loss function is in lines 63-65.

```
1  class DeepQNetwork(object):
2      def build(self):
3      """Builds the computational graph and training operations"""
4          self._build_graph()
5          self._build_training_ops()
6          self._build_summary_ops()
7
8      def _build_single_q_network(self, observations, head, state_t,
       ↪  state_tp1, done_mask, reward_t, error_weight):
9      """Builds the computational graph for a single Q network.
10     Briefly, this part is calculating the following two quantities:
11     1. q_value = q_fn(observations)
12     2. td_error = q_fn(state_t)-reward_t-gamma*q_fn(state_tp1)
13     The optimization target is to minimize the td_error.
14     """
15         with tf.variable_scope('q_fn'):
16         # q_value have shape [batch_size, 1].
17             q_values = tf.gather(self.q_fn(observations), head,
               ↪  axis=-1)
18
19         # calculating q_fn(state_t) The Q network shares parameters
           ↪  with the action graph.
20         with tf.variable_scope('q_fn', reuse=True):
21             q_t = self.q_fn(state_t, reuse=True)
22         q_fn_vars =
           ↪  tf.trainable_variables(scope=tf.get_variable_scope().name
           ↪  +'/q_fn')
23
```

```
        # calculating q_fn(state_tp1)
        with tf.variable_scope('q_tp1', reuse=tf.AUTO_REUSE):
            q_tp1 = [self.q_fn(s_tp1, reuse=tf.AUTO_REUSE) for s_tp1 in state_tp1]
        q_tp1_vars = tf.trainable_variables(scope=tf.get_variable_scope().name + '/q_tp1')

        if self.double_q:
            with tf.variable_scope('q_fn', reuse=True):
                q_tp1_online = [self.q_fn(s_tp1, reuse=True) for s_tp1 in state_tp1]
            if self.num_bootstrap_heads:
                num_heads = self.num_bootstrap_heads
            else:
                num_heads = 1
            # determine the action to choose based on online Q estimator.
            q_tp1_online_idx = [
                tf.stack(
                    [tf.argmax(q, axis=0),
                     tf.range(num_heads, dtype=tf.int64)],
                    axis=1) for q in q_tp1_online
            ]
            # use the index from max online q_values to compute the value function
            v_tp1 = tf.stack(
                [tf.gather_nd(q, idx) for q, idx in zip(q_tp1, q_tp1_online_idx)], axis=0)
        else:
            v_tp1 = tf.stack([tf.reduce_max(q) for q in q_tp1], axis=0)

        # if s_{t+1} is the terminal state, we do not evaluate the Q value of the state.
        q_tp1_masked = (1.0 - done_mask) * v_tp1

        q_t_target = reward_t + self.gamma * q_tp1_masked

        # stop gradient from flowing to the computating graph which computes the Q value of s_{t+1}. td_error has shape [batch_size, 1]
        td_error = q_t - tf.stop_gradient(q_t_target)

        # If use bootstrap, each head is trained with a different
         ↪  subset of the training sample. Like the idea of
         ↪  dropout.
        if self.num_bootstrap_heads:
            head_mask = tf.keras.backend.random_binomial(
                shape=(1, self.num_bootstrap_heads), p=0.6)
            td_error = tf.reduce_mean(td_error * head_mask, axis=1)
        # The loss function:
        errors = tf.where(
            tf.abs(td_error) < 1.95, tf.square(td_error) * 0.5,
            1.95 * (tf.abs(td_error) - 0.5*1.95))
        weighted_error = tf.reduce_mean(error_weight * errors)
        return q_values, td_error, weighted_error, q_fn_vars,
         ↪  q_tp1_vars

    def _build_input_placeholder(self):
    """Creates the input placeholders."""
        batch_size, fingerprint_length = self.input_shape

        with tf.variable_scope(self.scope, reuse=self.reuse):
            # Build the action graph to choose an action. The
             ↪  observations, which are the inputs of the Q
             ↪  function.
            self.observations = tf.placeholder(
                tf.float32, [None, fingerprint_length],
                 ↪  name='observations')
            # head is the index of the head we want to choose for
             ↪  decison.
            self.head = tf.placeholder(tf.int32, [], name='head')

            # When sample from memory, the batch_size can be fixed,
             ↪  as it is possible to sample any number of samples
             ↪  from memory. state_t is the state at time step t
            self.state_t = tf.placeholder(
                tf.float32, self.input_shape, name='state_t')
            # state_tp1 is the state at time step t + 1, tp1 is
             ↪  short for t plus 1.
            self.state_tp1 = [
                tf.placeholder(
                    tf.float32, [None, fingerprint_length],
                     ↪  name='state_tp1_%i' % i)
                for i in range(batch_size)
            ]
            # done_mask is a {0, 1} tensor indicating whether
             ↪  state_tp1 is the terminal state.
            self.done_mask = tf.placeholder(
                tf.float32, (batch_size, 1), name='done_mask')

            self.error_weight = tf.placeholder(
                tf.float32, (batch_size, 1), name='error_weight')

    def _build_training_ops(self):
    """Creates the training operations."""
        with tf.variable_scope(self.scope, reuse=self.reuse):
            self.optimization_op = tf.contrib.layers.optimize_loss(
                loss=self.weighted_error,
                global_step=tf.train.get_or_create_global_step(),
                learning_rate=self.learning_rate,
                optimizer=self.optimizer,
                clip_gradients=self.grad_clipping,
                learning_rate_decay_fn=functools.partial(
                    tf.train.exponential_decay,
                    decay_steps=self.learning_rate_decay_steps,
                    decay_rate=self.learning_rate_decay_rate),
                variables=self.q_fn_vars)

            self.update_op = []
            for var, target in zip(
                    sorted(self.q_fn_vars, key=lambda v: v.name),
                    sorted(self.q_tp1_vars, key=lambda v: v.name)):
                self.update_op.append(target.assign(var))
            self.update_op = tf.group(*self.update_op)

    def get_action(self, observations, stochastic=True, head=0,
     ↪  update_epsilon=None):
    """Function that chooses an action given the observations."""
        if update_epsilon is not None:
            self.epsilon = update_epsilon
        print('Epsilon greedy value is ', self.epsilon)

        if stochastic and np.random.uniform() < self.epsilon:
            print('The action was randomly chosen!')
            return np.random.randint(0, observations.shape[0])
        else:
            print('The action was the current optimal!')
            return self._run_action_op(observations, head)

    def train(self, states, rewards, next_states, done, weight,
     ↪  summary=True):
```

```
    """Function that takes a transition (s,a,r,s') and optimizes
    ↪  Bellman error."""
        if summary:
            ops = [self.td_error, self.error_summary,
             ↪  self.optimization_op]
        else:
            ops = [self.td_error, self.optimization_op]
        feed_dict = {
            self.state_t: states,
            self.reward_t: rewards,
            self.done_mask: done,
            self.error_weight: weight
        }
        for i, next_state in enumerate(next_states):
            feed_dict[self.state_tp1[i]] = next_state
        return tf.get_default_session().run(ops, feed_dict=feed_dict)

class MultiObjectiveDeepQNetwork(DeepQNetwork):
    def __init__(self, objective_weight, **kwargs):
    """Creates the model function."""
        # Normalize the sum to 1.
        self.objective_weight = objective_weight /
         ↪  np.sum(objective_weight)
        self.num_objectives = objective_weight.shape[0]
        super(MultiObjectiveDeepQNetwork, self).__init__(**kwargs)

    def _build_graph(self):
    """Builds the computational graph."""
        batch_size, _ = self.input_shape
        with tf.variable_scope(self.scope, reuse=self.reuse):
            self._build_input_placeholder()
            self.reward_t = tf.placeholder(
                tf.float32, (batch_size, self.num_objectives),
                 ↪  name='reward_t')
            # reward = sum (objective_weight_i * objective_i)
            self.objective_weight_input = tf.placeholder(
                tf.float32, [self.num_objectives, 1],
                 ↪  name='objective_weight')

            # split reward for each q network
            rewards_list = tf.split(self.reward_t,
             ↪  self.num_objectives, axis=1)
            q_values_list = []
            self.td_error = []
            self.weighted_error = 0
            self.q_fn_vars = []
            self.q_tp1_vars = []

            # build a Q network for each objective
            for obj_idx in range(self.num_objectives):
                with tf.variable_scope('objective_%i' % obj_idx):
                    (q_values, td_error, weighted_error,
                     q_fn_vars, q_tp1_vars) =
                      ↪  self._build_single_q_network(
                        self.observations, self.head, self.state_t,
                         ↪  self.state_tp1,
                        self.done_mask, rewards_list[obj_idx],
                         ↪  self.error_weight)
                    q_values_list.append(tf.expand_dims(q_values, 1))
                    # td error is for summary only.
                    # weighted error is the optimization goal.
                    self.td_error.append(td_error)
                    self.weighted_error += weighted_error /
                     ↪  self.num_objectives
                    self.q_fn_vars += q_fn_vars
                    self.q_tp1_vars += q_tp1_vars
            q_values = tf.concat(q_values_list, axis=1)
            # action is the one that leads to the maximum weighted
            ↪  reward.
            self.action = tf.argmax(
                tf.matmul(q_values, self.objective_weight_input),
                 ↪  axis=0)

    def _build_summary_ops(self):
    """Creates the summary operations."""
        with tf.variable_scope(self.scope, reuse=self.reuse):
            with tf.name_scope('summaries'):
                # The td_error here is the difference between q_t
                 ↪  and q_t_target. Without abs(), the summary of
                 ↪  td_error is actually underestimated.
                error_summaries = [
                    tf.summary.scalar('td_error_%i' % i,
                     ↪  tf.reduce_mean(tf.abs(self.td_error[i])))
                    for i in range(self.num_objectives)
                ]
                self.error_summary =
                 ↪  tf.summary.merge(error_summaries)
                self.smiles = tf.placeholder(tf.string, [],
                 ↪  'summary_smiles')
                self.rewards = [tf.placeholder(tf.float32, [],
                 ↪  'summary_reward_obj_%i' % i)
                    for i in range(self.num_objectives)
                ]
                # Weighted sum of the rewards.
                self.weighted_reward = tf.placeholder(tf.float32, [],
                 ↪  'summary_reward_sum')
                smiles_summary = tf.summary.text('SMILES',
                 ↪  self.smiles)
                reward_summaries = [
                    tf.summary.scalar('reward_obj_%i' % i,
                     ↪  self.rewards[i])
                    for i in range(self.num_objectives)
                ]
                reward_summaries.append(
                    tf.summary.scalar('sum_reward',
                     ↪  self.rewards[-1]))

                self.episode_summary =
                 ↪  tf.summary.merge([smiles_summary] +
                    reward_summaries)

    def log_result(self, smiles, reward):
    """Summarizes the SMILES string and reward at the end of an
     ↪  episode."""
        feed_dict = {self.smiles: smiles,}
        for i, reward_value in enumerate(reward):
            feed_dict[self.rewards[i]] = reward_value
        # calculated the weighted sum of the rewards.
        feed_dict[self.weighted_reward] = np.asscalar(
            np.array([reward]).dot(self.objective_weight))
        return tf.get_default_session().run(
            self.episode_summary, feed_dict=feed_dict)

    def _run_action_op(self, observations, head):
    """Function that runs the op calculating an action given the
     ↪  observations."""
        return np.asscalar(tf.get_default_session().run(
            self.action,
            feed_dict={
                self.observations: observations,
                self.objective_weight_input: self.objective_weight,
                self.head: head
            }))
```

```python

def multi_layer_model(inputs, hparams, reuse=None):
    """Multi-layer model for q learning"""
    output = inputs
    for i, units in enumerate(hparams.dense_layers):
        output = tf.layers.dense(output, units, name='dense_%i' % i,
         ↪  reuse=reuse)
        output = getattr(tf.nn, hparams.activation)(output)
        if hparams.batch_norm:
            output = tf.layers.batch_normalization(
                output, fused=True, name='bn_%i' % i, reuse=reuse)
    if hparams.num_bootstrap_heads:
        output_dim = hparams.num_bootstrap_heads
    else:
        output_dim = 1
    output = tf.layers.dense(output, output_dim, name='final',
     ↪  reuse=reuse)
    return output

def get_fingerprint(smiles, hparams):
    """Get Morgan Fingerprint of a specific SMILES string."""
    if smiles is None:
        return np.zeros((hparams.fingerprint_length,))
    # print(smiles)
    molecule = Chem.MolFromSmiles(smiles)
    if molecule is None:
        return np.zeros((hparams.fingerprint_length,))
    fingerprint = AllChem.GetMorganFingerprintAsBitVect(
        molecule, hparams.fingerprint_radius,
         ↪  hparams.fingerprint_length)
    arr = np.zeros((1,))
    # ConvertToNumpyArray takes ~ 0.19 ms, while
    # np.asarray takes ~ 4.69 ms
    DataStructs.ConvertToNumpyArray(fingerprint, arr)
    return arr

def get_fingerprint_with_steps_left(smiles, steps_left, hparams):
    """Get Morgan Fingerprint of a SMILES string with number of
     ↪  steps left"""
    fingerprint = get_fingerprint(smiles, hparams)
    return np.append(fingerprint, steps_left)
```

Bibliography

[1] M. E. Avery. “Gertrude Belle Elion. 23 January 1918 — 21 February 1999”. In: *Biographical Memoirs of Fellows of the Royal Society* 54 (2008), pp. 161–168. DOI: 10.1098/rsbm.2007.0051.

[2] R. Bellman. “A Markovian Decision Process”. In: *Journal of Mathematics and Mechanics* 6.5 (1957), pp. 679–684. ISSN: 0095-9057. JSTOR: 24900506.

[3] R. E. Bellman. *An Introduction to the Theory of Dynamic Programming*. Tech. rep. RAND Corporation, 1953.

[4] J. W. Bennett and K.-T. Chung. “Alexander Fleming and the Discovery of Penicillin”. In: *Advances in Applied Microbiology*. Vol. 49. Academic Press, 2001, pp. 163–184. DOI: 10.1016/S0065-2164(01)49013-7.

[5] N. Berdigaliyev and M. Aljofan. “An Overview of Drug Discovery and Development”. In: *Future Medicinal Chemistry* 12.10 (2020), pp. 939–947. ISSN: 1756-8919. DOI: 10.4155/fmc-2019-0307.

[6] G. R. Bickerton et al. “Quantifying the Chemical Beauty of Drugs”. In: *Nature Chemistry* 4.2 (2012), pp. 90–98. ISSN: 1755-4349. DOI: 10.1038/nchem.1243.

[7] P. Charbonnier et al. “Deterministic Edge-Preserving Regularization in Computed Imaging”. In: *IEEE Transactions on Image Processing* 6.2 (1997), pp. 298–311. ISSN: 1941-0042. DOI: 10.1109/83.551699.

[8] A. Dalby et al. “Description of Several Chemical Structure File Formats Used by Computer Programs Developed at Molecular Design Limited”. In: *Journal of Chemical Information and Computer Sciences* 32.3 (1992), pp. 244–255. ISSN: 0095-2338. DOI: 10.1021/ci00007a012.

[9] O. b. F. DMacks slight edit by. *Deriving the SMILES Representation of a Chemical Molecule, Shown Example: Ciprofloxacin, a Fluoroquinolone Antibiotic.* 2007-08-13, updated 22:57, 27 June 2010 (UTC).

[10] Y. Du et al. “Deep Latent-Variable Models for Controllable Molecule Generation”. In: *2021 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 2021, pp. 372–375. DOI: 10.1109/BIBM52615.2021.9669692.

[11] Y. Du et al. *Interpretable Molecular Graph Generation via Monotonic Constraints*. 2022. arXiv: arXiv:2203.00412.

[12] J. Duchi, E. Hazan, and Y. Singer. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. ISSN: 1533-7928.

[13] C. B. Ferster and B. F. Skinner. *Schedules of Reinforcement / by C.B. Ferster and B.F. Skinner*. Century Psychology Series. New York: Appleton-Century-Crofts, 1957.

[14] R. Gómez-Bombarelli et al. “Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules”. In: *ACS Central Science* 4.2 (2016), pp. 268–276. ISSN: 2374-7943, 2374-7951. DOI: 10.1021/acscentsci.7b00572. arXiv: 1610.02415.

[15] *Google-Research/Mol_dqn at Master · Google-Research/Google-Research*. 2018.

[16] G. L. Guimaraes et al. *Objective-Reinforced Generative Adversarial Networks (ORGAN) for Sequence Generation Models*. 2017. arXiv: arXiv:1705.10843.

[17] H. Hasselt. “Double Q-learning”. In: *Advances in Neural Information Processing Systems*. Vol. 23. Curran Associates, Inc., 2010.

[18] J. He et al. *Molecular Optimization by Capturing Chemist’s Intuition Using Deep Neural Networks*. 2020.

[19] S. Hochreiter and J. Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

[20] P. J. Huber. “Robust Estimation of a Location Parameter”. In: *The Annals of Mathematical Statistics* 35.1 (1964), pp. 73–101. ISSN: 0003-4851, 2168-8990. DOI: 10.1214/aoms/1177703732.

[21] W. Jin, R. Barzilay, and T. Jaakkola. *Junction Tree Variational Autoencoder for Molecular Graph Generation*. 2018. arXiv: arXiv:1802.04364.

[22] W. Jin, R. Barzilay, and T. Jaakkola. *Multi-Objective Molecule Generation Using Interpretable Substructures*. 2020. arXiv: arXiv:2002.03244.

[23] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: arXiv:1412.6980.

[24] C. Krishnamurti and S. C. Rao. “The Isolation of Morphine by Serturner”. In: *Indian Journal of Anaesthesia* 60.11 (2016), pp. 861–862. ISSN: 0019-5049. DOI: 10.4103/0019-5049.193696.

[25] A. Lavecchia and C. D. Giovanni. “Virtual Screening Strategies in Drug Discovery: A Critical Review”. In: *Current Medicinal Chemistry* 20.23 (2013), pp. 2839–2860.

[26] L.-J. Lin. “Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching”. In: *Machine Learning* 8.3 (1992), pp. 293–321. ISSN: 1573-0565. DOI: 10.1007/BF00992699.

[27] I. Loshchilov and F. Hutter. *Decoupled Weight Decay Regularization*. 2017. arXiv: arXiv:1711.05101.

[28] J. Ma and D. Yarats. *Quasi-Hyperbolic Momentum and Adam for Deep Learning*. 2018. arXiv: arXiv:1810.06801.

[29] V. Mnih et al. “Human-Level Control through Deep Reinforcement Learning”. In: *Nature* 518.7540 (2015), pp. 529–533. ISSN: 1476-4687. DOI: 10.1038/nature14236.

[30] I. Osband et al. *Deep Exploration via Bootstrapped DQN*. 2016. arXiv: arXiv:1602.04621.

[31] K. Petrov. *MolDQN-adapted for MChem Thesis ”De Novo Molecular Design of Antivirals Robust to Resistance Using Novel Reward Functions in Structure-Based Deep Reinforcement Learning”*. 2023.

[32] P. G. Polishchuk, T. I. Madzhidov, and A. Varnek. “Estimation of the Size of Drug-like Chemical Space Based on GDB-17 Data”. In: *Journal of Computer-Aided Molecular Design* 27.8 (2013), pp. 675–679. ISSN: 1573-4951. DOI: 10.1007/s10822-013-9672-4.

[33] D. Rogers and M. Hahn. “Extended-Connectivity Fingerprints”. In: *Journal of Chemical Information and Modeling* 50.5 (2010), pp. 742–754. ISSN: 1549-9596, 1549-960X. DOI: 10.1021/ci100050t.

[34] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. “Learning Representations by Back-Propagating Errors”. In: *Nature* 323.6088 (1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0.

[35] M. H. S. Segler et al. “Generating Focused Molecule Libraries for Drug Discovery with Recurrent Neural Networks”. In: *ACS Central Science* 4.1 (2018), pp. 120–131. ISSN: 2374-7943. DOI: 10.1021/acscentsci.7b00512.

[36] J. M. Stokes et al. “A Deep Learning Approach to Antibiotic Discovery”. In: *Cell* 180.4 (2020), 688–702.e13. ISSN: 0092-8674. DOI: 10.1016/j.cell.2020.01.021.

[37] T. T. Tanimoto. “An Elementary Mathematical Theory of Classification and Prediction”. In: *International Business Machine Corporation* (1958).

[38] T. Tieleman and G. Hinton. *Lecture 6a: RMSProp*. Tech. rep. 2012.

[39] H. van Hasselt, A. Guez, and D. Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: arXiv:1509.06461.

[40] W. Wang et al. *Generative Coarse-Graining of Molecular Conformations*. 2022. arXiv: arXiv:2201.12176.

[41] C. Watkins. “Learning from Delayed Rewards.Pdf”. PhD thesis. University of Cambridge, 1989.

[42] C. J. C. H. Watkins and P. Dayan. “Q-Learning”. In: *Machine Learning* 8.3 (1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698.

[43] O. J. Watson et al. “Global Impact of the First Year of COVID-19 Vaccination: A Mathematical Modelling Study”. In: *The Lancet Infectious Diseases* 22.9 (2022), pp. 1293–1302. ISSN: 1473-3099. DOI: 10.1016/S1473-3099(22)00320-6.

[44] D. Weininger. “SMILES, a Chemical Language and Information System. 1. Introduction to Methodology and Encoding Rules”. In: *Journal of Chemical Information and Computer Sciences* 28.1 (1988), pp. 31–36. ISSN: 0095-2338. DOI: 10.1021/ci00057a005.

[45] A. C. Wilson et al. *The Marginal Value of Adaptive Gradient Methods in Machine Learning*. 2017. arXiv: arXiv:1705.08292.

[46] J. You et al. *Graph Convolutional Policy Network for Goal-Directed Molecular Graph Generation*. 2018. arXiv: arXiv:1806.02473.

[47] Z. Zhang et al. *Normalized Direction-preserving Adam*. 2017. arXiv: arXiv:1709.04546.

[48] A. Zhavoronkov et al. “Deep Learning Enables Rapid Identification of Potent DDR1 Kinase Inhibitors”. In: *Nature Biotechnology* 37.9 (2019), pp. 1038–1040. ISSN: 1546-1696. DOI: 10.1038/s41587-019-0224-x.

[49] Z. Zhou et al. “Optimization of Molecules via Deep Reinforcement Learning”. In: *Scientific Reports* 9.1 (2019), p. 10752. ISSN: 2045-2322. DOI: 10.1038/s41598-019-47148-x.