ARISTOTLE UNIVERSITY OF THESSALONIKI
DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

SMILES Generation Using Reinforcement Learning

Afroditi Latskou

May 2025

Contents

1 Introduction

According to many sources designing drugs with specific desired properties is a big challenge in drug discovery. Computational methods have been proposed as a solution to this, in order to search larger areas of chemical space for novel, appropriate, drug-like molecules. One such method is the use of deep learning as well as reinforcement learning for molecular generation.

Specifically, many methods use the Simplified Molecular Input Line Entry System or SMILES to represent molecules. It can be used to describe the structure of a chemical species using short ASCII strings.

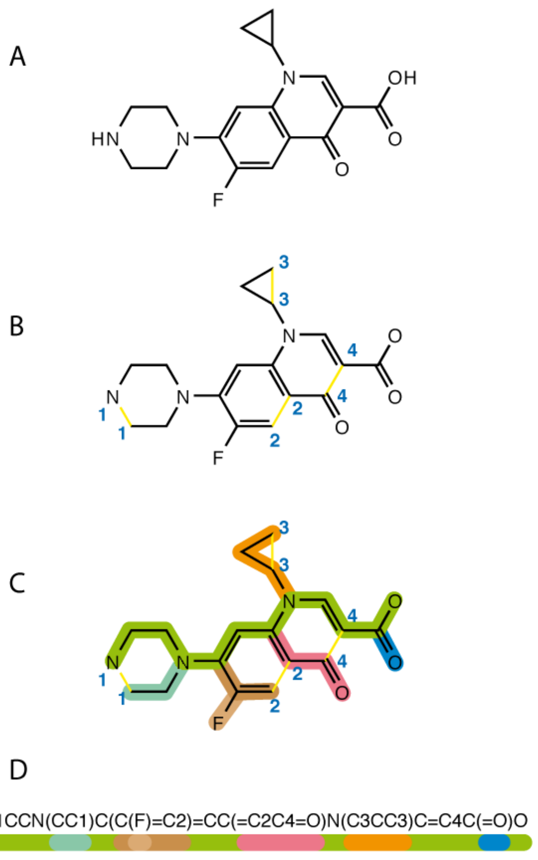

Figure 1: Molecule represented as SMILES (D)

Several approaches have been applied to tackle this challenge using a variety of different deep learning algorithms, such as autoencoders, RNNs as well as on-policy reinforcement learning models.

2 Methodology

The methodology used in this project can be described as follows:

1. Pretraining a Generator capable of producing syntactically valid SMILES
2. Training a Predictor to predict a target molecular property from a SMILES sequence
3. Implementing reinforcement learning fine-tuning to improve the Generator's performance

2.1 SMILES Representation

For the purposes of this project, a portion of the ZINC SMILES Database was used. The database contains the SMILES representation of discovered molecules along with some of their properties, like the partition coefficient $LogP$, which measures a compound's lipophilicity (the affinity for fat or water) and the quantitative estimate of drug-likeness QED, which scores how drug-like a molecule is based on 8 physicochemical properties.

For in order to turn the SMILES vocabulary into something an algorithm can learn, tokenization is used. Specifically, the following tokens are defined:

```
TOKENS = [’<’, ’>’, ’#’, ’%’, ’)’, ’(’, ’+’, ’-’, ’/’, ’.’,
    ’1’, ’0’, ’3’, ’2’, ’5’, ’4’, ’7’, ’6’, ’9’, ’8’, ’=’,
    ’A’, ’@’, ’C’, ’B’, ’F’, ’I’, ’H’, ’O’, ’N’, ’P’, ’S’,
    ’[’, ’]’, ’\\’, ’c’, ’e’, ’i’, ’l’, ’o’, ’n’, ’p’, ’s’,
    ’r’, ’\n’, ’<PAD>’]
```

We also create a SMILES Dataset after ensuring that all the SMILES contained in the original file are valid.

2.2 Model Architecture and Training

2.2.1 Generator

For the generator, an RNN is used; more specifically, a Gated Recurrent Unit. Therefore, we have the following three main components:

- **Embedding Layer**: Converts token indices to dense vectors
- **GRU Layer**: The recurrent unit that processes the sequence
- **Linear Layer**: Projects the hidden state to vocabulary space

The sampling follows the following steps:

- Starts generation with a special start token (`<`)
- Uses multinomial sampling from the probability distribution
- Stops when reaching either:
  - Maximum length (`max_len`)
  - End-of-sequence token (`\n`)
- Returns sequence of token indices

And for our experiments we use the following hyperparameters:

Parameter	
Embedding dimension	128
Hidden dimension	512
Maximum length	100

Table 1: Model hyperparameters

The generator is trained with an Adam optimizer and using Cross Entropy for loss.

2.2.2 Predictor

Our predictor implements an LSTM-based regression model for molecular property prediction from SMILES strings. The architecture consists of:

- Embedding layer with padding handling; maps discrete tokens to continuous vectors
- Bidirectional LSTM for sequence processing
- Two-layer feedforward network for property prediction with ReLU activation function to add non-linearity

The predictor is trained with an Adam optimizer and Mean Squared Error as a loss function

2.3 Reinforcement Learning

Like many other relative works, the main algorithm behind the RL implementation is REINFORCE, a policy gradient method.

Initially, the pretrained generator is loaded and set to training mode. Pretraining ensures that the model can generate valid SMILES syntax before applying goal-directed reinforcement. A custom environment is also set up (as we will discuss later).

A mask is created to disable invalid tokens during generation (i.e. $< PAD >$)

```
    mask = torch.full((vocab_size,), float('-inf'),
       device=args.device)
    mask[torch.tensor(list(token_to_idx.values()),
       dtype=torch.long)] = 0
```

and the first token is explicitly constrained to chemically plausible atoms (C, 0, etc), reflecting SMILES grammar.

```
    if t == 0:
                logits[:] = -1e9
                for tok in ['C', 'c', 'O', 'N']:
                    logits[0, token_to_idx[tok]] = 1.0
```

For each step in our RL implementation, a sequence is rolled out using the policy (generator), an action is sampled from $Categorical(probs)$ distribution and the log-probability and entropy are loaded for later gradient computation. The gradients are propagated through the generator model using an Adam optimizer.

Finally, the loss includes:

1. Policy loss: Standard REINFORCE term
2. Entropy regulization (with a weight): to encourage stochasticity

```
    loss = policy_loss - 0.05 * entropy_bonus
```

Every 100 steps the best performing generator is saved as a checkpoint.

3 Environment and Reward Design

3.1 Environment definition

For the needs of this project, a custom Gymnasium environment $SMILESEnv$. The agent -for us the generator- sequentially selects tokens, aiming to form a valid and drug-like molecule and receives the corresponding reward.

During the initialization, the predictor model is taken as input in order to estimate the molecular properties. In our case, the action space can be defined as the discrete set of SMILES tokens and the observation space as the fixed-length vector of token indices.

Our step function decodes the token index into a SMILES character and appends it to the current sequence. The episode will end if the maximum length is reached or if the end token is chosen. At that point the complete SMILES string is assembled and using the RDKit library it tries to convert the string into a valid molecule. If that is then valid, a reward is computed with the predicted properties.

3.2 Reward Function

The reward design is pretty simple. Short or empty SMILES string lengths are lightly punished while invalid strings are more severely punished. For the valid strings, the reward is computed using both the *logp* and *qed* values. For this project, the selected range of appropriate such values doesn't have any actual chemical significance, mainly it is to see if we can teach the model to produce molecules in any such range. Therefore, in our implementation positive values are encouraged.

More specifically, the reward function is defined as:

```
    def reward_fn(smiles, pred=None):
        smi = smiles.replace('<', '').replace('\n',
            '').strip()
        if not smi or len(smi) < 5:
            return -1.0
        mol = Chem.MolFromSmiles(smi)
        if mol is None:
            return -2.0
        try:
            logp = Crippen.MolLogP(mol)
            qed = QED.qed(mol)
            return 0.5 * (logp / 5.0) + 0.5 * qed
        except:
            return -1.0
```

4 Results and Discussion

4.1 Results

Various experiments were conducted using different hyperparameters and making modifications in the reward function. The best performing one is presented bellow. The hyperparameters used are the ones included in the code. The algorithm ran for a total of 2000 steps.

In the end we obtain the following metrics:

```
    Validity: 723/1000 = 72.3%
    Mean predicted property: -2.163
    Mean QED: 0.340
```

As we can see, the model performs with better validity than the pretrained generator, which is shown bellow for comparison:

```
    Validity: 233/1000 = 23.3%
    Mean predicted property: 1.730
    Mean QED: 0.713
```

However, we can already tell the results will not be satisfactory since the variance of the properties is lacking on the RL results. Indeed, if we plot these distributions:

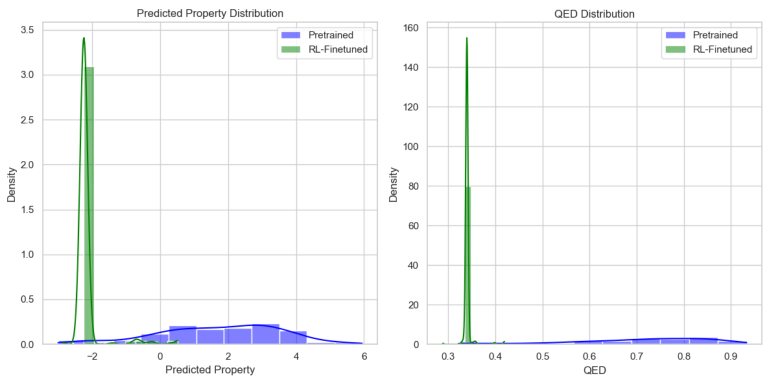

Figure 2: Comparison of predicted property distribution

We can tell that the distribution is tightly concentrated around one value, meaning that the same SMILES strings are constantly being preferred. If we also plot the token distribution for the pretrained and RL model:

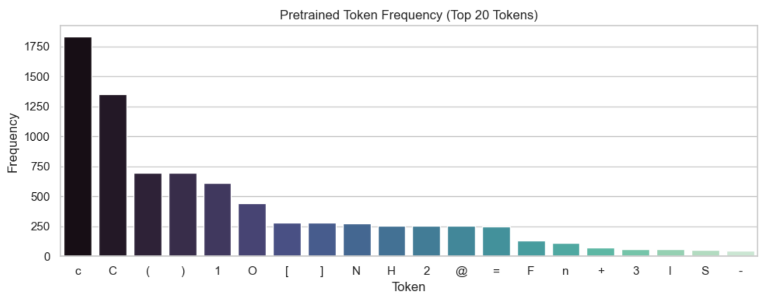

Figure 3: Pretrained Token Distribution

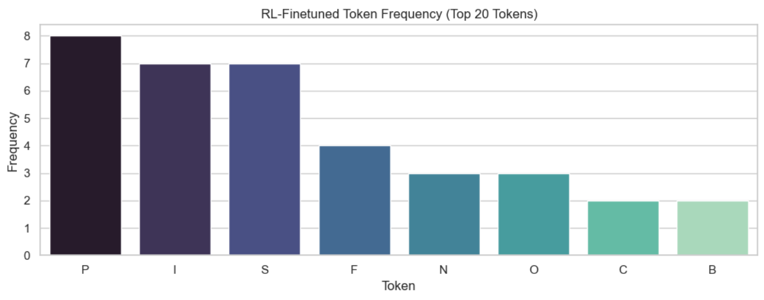

Figure 4: RL Token Distribution

Our assumptions are confirmed. The same tokens are more preferred and appear the most; specifically tokens that represent whole elements found in molecules, since they themselves can be considered a molecule and are therefore always valid. This hints at the algorithm falling back on the same 'safer' solutions since it fails when trying to create larger SMILES strings.

4.2 Discussion

After the implementation and evaluation of this project, the following comments can be made:

1. For time efficiency, the initial data size for training the generator was chosen as 10.000. However, that hyperparameter was never changed and all consequent experimentations were completed using that saved checkpoint. Therefore the low performance of the generator can be mainly explained with the insufficient data. And as consequence, since the generator did not learn the SMILES syntax effectively, it limited the RL model's ability to produce valid and satisfactory results.

2. As proposed by previous literature, the tokens were defined as such. However, tokens like '%' or '.' are very rare to encounter in actual SMILES strings and serve more as noise for the generator and RL model.

3. Not enough attention was paid to ensuring there was a balance between exploration and exploitation. As such the model often resorts to the same or similar solutions.

4. As noticed by previous literature as well, the RL model tends to prefer larger strings, therefore limiting the possibility that they are entirely valid.

These are all solvable and are therefore possible future improvements in order to obtain better results. Moreover, as an addition it would be beneficial to redefine the reward function so that it is chemically and biologically significant in terms of the $logP$ and QED values. More testing on the generator and predictor separately should also be done to ensure each unit is working adequately.

References

[1] E. Mazuz, G. Shtar, B. Shapira, and L. Rokach, "Molecule generation using transformers and policy gradient reinforcement learning," *Scientific Reports*, vol. 13, no. 1, p. 8799, May 2023, doi: 10.1038/s41598-023-35648-w.

[2] M. Mokaya, F. Imrie, W. P. van Hoorn, A. Kalisz, A. R. Bradley, and C. M. Deane, "Testing the limits of SMILES-based de novo molecular generation with curriculum and deep reinforcement learning," *Nature Machine Intelligence*, vol. 5, no. 4, pp. 386–394, Apr. 2023, doi: 10.1038/s42256-023-00636-2.

[3] M. Popova, O. Isayev, and A. Tropsha, "Deep reinforcement learning for de novo drug design," *Science Advances*, vol. 4, no. 7, p. eaap7885, Jul. 2018, doi: 10.1126/sciadv.aap7885.

[4] "ZINC," *zinc.docking.org.* [Online]. Available: `https://zinc.docking.org/`.

[5] "RDKit," *www.rdkit.org.* [Online]. Available: `https://www.rdkit.org/`.

[6] "Farama-Foundation/Gymnasium," *GitHub*, Apr. 16, 2023. [Online]. Available: `https://github.com/Farama-Foundation/Gymnasium`.