PySMILESUtils – Enabling deep learning with the SMILES chemical language

Esben Jannik Bjerrum†,*, Tobias Rastemo†, Ross Irwin†, Christos Kannas†, Samuel Genheden†

†) Molecular AI, Discovery Sciences, R&D, AstraZeneca, Gothenburg, Sweden
*) esben.bjerrum@astrazeneca.com

Keywords: SMILES, Deep Learning, PyTorch, Framework, SMILES augmentation

Recent years have seen a large interest in using the Simplified Molecular Input Line Entry System (SMILES) chemical language as input for deep learning architectures solving chemical tasks. Many successful applications have been demonstrated within de novo molecular design, quantitative structure-activity relationship modelling, forward reaction prediction and single-step retrosynthetic planning as examples. PySMILESUtils aims to enable these tasks by providing ready-to-use and adaptable Python classes for tokenization, augmentation, dataset, and dataloader creation. Classes for handling datasets larger than memory and speeding up training by minimizing padding are also provided. The framework subclasses PyTorch dataset and dataloaders but should be adaptable for other deep learning frameworks. The project is open-sourced with a permissive license and made available at GitHub: https://github.com/MolecularAI/pysmilesutils

Introduction

Machine learning and deep learning have seen a research boom in the latest years. Deep learning in particular has significantly improved some long standing problems in machine learning, such as image recognition, speech recognition, and natural language processing (NLP). Deep learning can be understood as automatic feature extraction, rather than engineering features for shallow neural networks or other machine learning algorithms. In deep learning, the first layers extract and create useful features for the subsequent layers and final predictions. As such, deep learning can use formats that are more "raw" and yet reach similar or better performance than approaches that use engineered approaches. On the downside, deep learning needs enough data to learn the complex, often non-linear, transformations necessary to convert the raw format into useful features for predictive modelling.

In chemistry, molecules have previously been featurized with descriptors and fingerprints[1], but recently other formats such as images[2], strings[3] and graphs[4] have been used as input to deep neural networks which solve chemically related tasks such as property prediction. Moreover, the graphs and strings can also be sampled autoregressively and thus used to predict molecular structures[5], [6]. This particular generative or molecular read-out capability has enabled a wide range of advanced algorithms ranging from autoencoders[7] to molecular transformers for e.g. reaction informatics[8], representation learning[9] and molecular optimizations[10].

The Simplified Molecular Input Line Entry System (SMILES) format is a single-line molecular notation, which has been widely used for handling molecules in a convenient way. The SMILES strings can easily be organized in common spreadsheets and sent by e-mail. However, in combination with NLP network architectures, it provides an easy and convenient way to do deep learning on molecules. Techniques

such as data augmentation[3] can improve performance and bring the fidelity of the molecular creation close to 100%[11].

PySMILESUtils strives to make the use of SMILES for deep learning applications even easier by providing reusable and flexible objects for turning SMILES into tensors for deep learning, and sampled tensors back to SMILES again, as illustrated in Figure 1. SMILES strings can be tokenized in different ways, the most simple isto encode each character as a token. However, single atoms are often encoded on their own, so that e.g. chlorine "Cl" and bromine "Br" have their own token and can easily be distinguished from carbon "C" and boron "B". The tokenization class supports different tokenization schemes and can use regular expression patterns or explicit token patterns to search for tokens in the strings in the dataset. The token patterns are used to analyze the dataset and build the vocabulary that contains the translation table between tokens and integer indexes or one-hot encoding. For multicharacter tokens with potential overlap, ambiguity can exists, as example if both "ccc", "cc" and "c" token patterns exits. In that instance the order in the token list is important, as the tokens are extracted in order. Additionally. a class for data augmentation of the SMILES is also provided, as are PyTorch[12] compatible datasets and dataloaders.

The framework is available under the Apache 2.0 license on GitHUB: https://github.com/MolecularAI/pysmilesutils

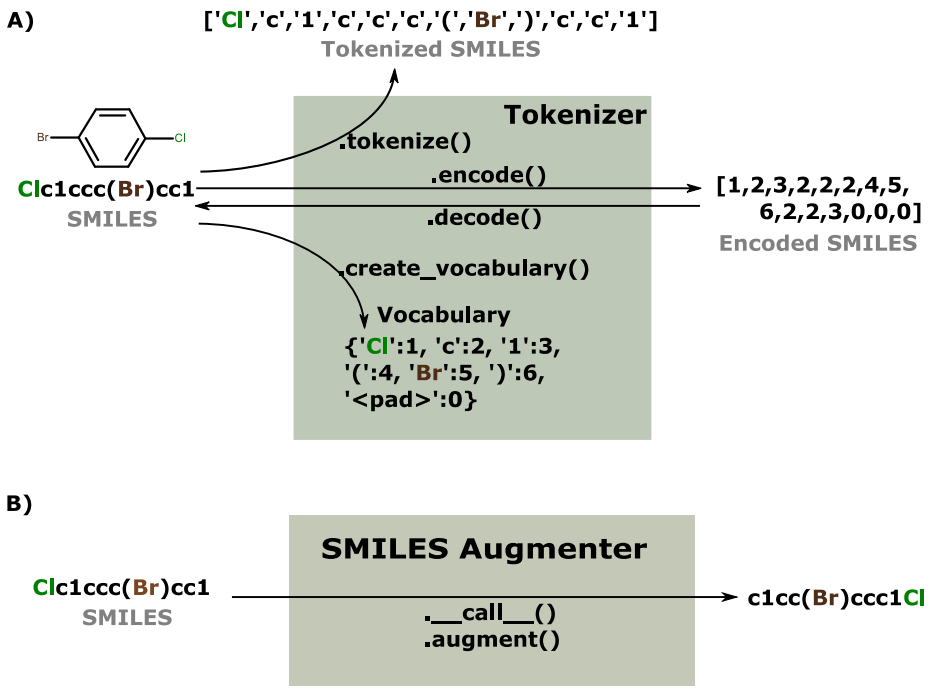

Figure 1 *The tokenization process.* ***A)*** *The tokenizer is central to converting SMILES strings into tensors for training. The datasets are analyzed with the given patterns to create the vocabulary that contains the translation table between tokens and integers. The tokenize function splits the given SMILES strings into lists of tokens. The encode function uses the tokenization and the vocabulary to produce an encoded SMILES in tensor form. The decode function reverse translates the encoded tensor back to a SMILES string.* ***B)*** *The SMILES augmentation object can be configured and called to augment SMILES string either via shuffling the atom-order and creating non-canonical SMILES strings*[3]*, or by randomizing the string during SMILES creation*[13]*.*

PySMILESUtils

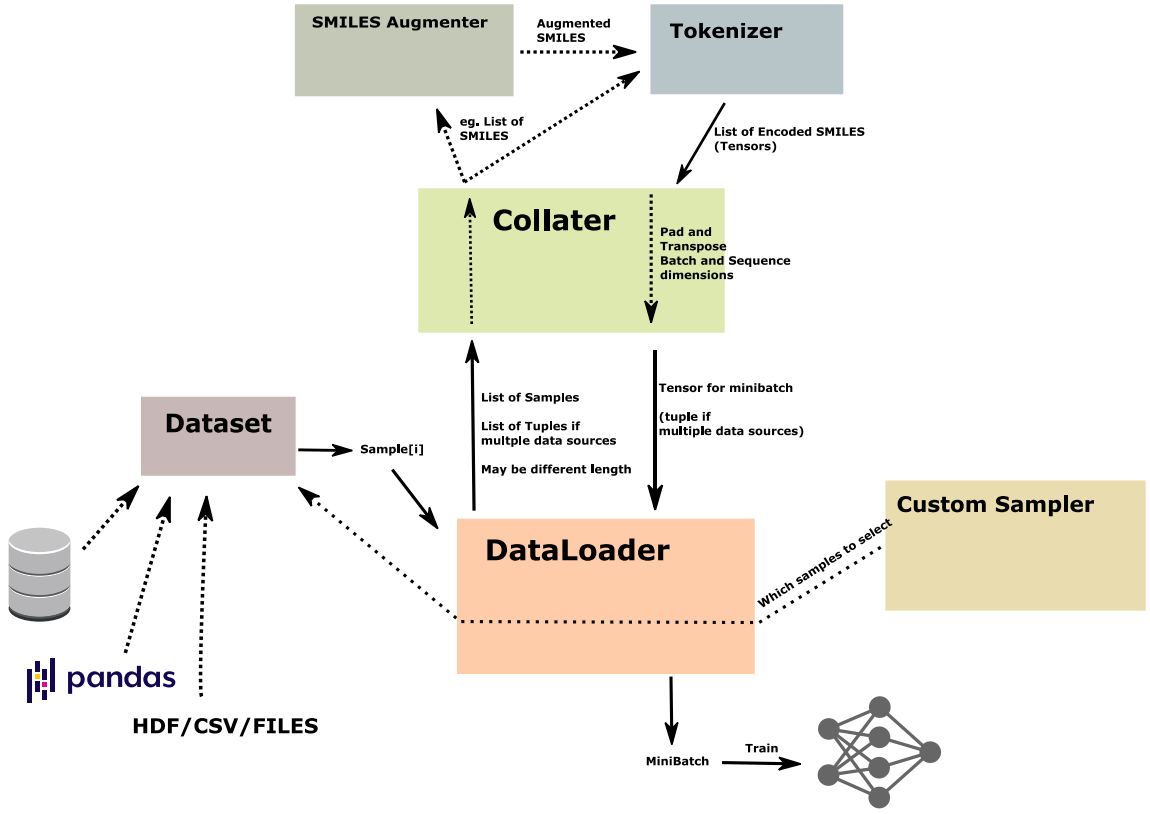

Figure 2 PySMILESUtils consists of reusable and customizable elements that work together to create mini-batches for training of artificial neural networks. Datasets provides a consistent interface that can be customized for different data types. The SMILES augmenter and tokenizer augments and converts SMILES strings into tensors, respectively. The DataLoader chooses samples from datasets and collates them into minibatch tensors ready for training.

PySMILESUtils has a range of different classes and sub-classes that serve different purposes during the creation of minibatches (Figure 2), and are similar to and compatible with the framework available within PyTorch, but has been customized to enable handling of the data in SMILES format.

Tokenizers

The tokenizers are central for working with text-based formats such as SMILES strings. The tokenizer in PySMILESUtils consists of an abstract class with methods that are considered common to tokenizers. A provided subclass, the SMILESAtomTokenizer, uses regular expression token patterns for tokenization. Regular token strings can be provided as well. Given token patterns and potential tokens, the dataset is analyzed and the found tokens are used to build the vocabulary that will be used to encode and decode the SMILES strings into tensors of indices ( ). Optionally one-hot encoded tensors can be created. Code Box 1 shows an example of how to work with the SMILESAtomTokenizer using the default regular expression patterns to create a vocabulary from the SMILES in the dataset. The tokens created are chemical symbols, where e.g. Chlorine is not split into “C” and “l”, but rather kept as “Cl”. After fitting the vocabulary it’s easy to convert a given SMILES string into a PyTorch tensor of indices that could go into an embedding layer of a deep learning model.

Code Box 1: Example of working with the SMILESAtomTokenizer. A vocabulary is created upon instantiation from the provided SMILES using the default regular expression token pattern, and the tokenizer is then called to tokenize a single SMILES.

```
import pandas as pd
from pysmilesutils.tokenize import SMILESAtomTokenizer

data = pd.read_csv("data/pande_dataset.csv") #Example dataset
tokenizer = SMILESAtomTokenizer(smiles=list(data.reactants + data.products))
print(tokenizer.vocabulary)

>>> {' ': 0, '^': 1, '&': 2, '?': 3, 'C': 4, 'S': 5, '(': 6, '=': 7, 'O': 8,
')': 9, '[': 10, '@': 11, 'H': 12, ']': 13, '1': 14, '.': 15, 'F': 16, 'c': 17,
'N': 18, '2': 19, 'n': 20, '3': 21, '4': 22, 'Cl': 23, '5': 24, 's': 25, 'B':
26, 'Br': 27, '+': 28, '-': 29, 'o': 30, '/': 31, 'Li': 32, '\\': 33, '#': 34,
'Si': 35, 'I': 36, 'P': 37, 'Mg': 38, 'Se': 39, '6': 40, '7': 41, 'Sn': 42, 'K':
43, 'Zn': 44, 'Cu': 45, 'se': 46, 'Pt': 47, '8': 48, '9': 49, 'Fe': 50, 'Pd':
51, 'p': 52, 'Pb': 53}

tokenizer("c1ccccc1N")

>>> [tensor([ 1, 17, 14, 17, 17, 17, 17, 17, 14, 18, 2])]
```

SMILES augmentation

SMILES augmentation is provided via the SMILESAugmenter object. After instantiation and configuration, the object itself can be called with a SMILES string or list of SMILES strings and will return a list of augmented SMILES strings. By default, the object will use atom order permutation[3], but full randomization[13] is also available. The augmenter object can be deactivated by setting the `.active` property to `False`, and it will then pass through the SMILES strings unaltered. This makes it possible to switch off the augmentation if needed, for example during model evaluation or inference.

Code Box 2: SMILES Augmentation example. The augmenter object provides easy SMILES augmentation with both full randomization and by default atom-order randomization (restricted). The augmenter accepts both single SMILES strings or lists of SMILES strings. The object can be temporarily deactivated, for example during evaluation.

```
from pysmilesutils.augment import SMILESAugmenter
augmenter = SMILESAugmenter()
augmenter(["Cc1ccccc1", "Oc1ccccc1"])
>>> ['c1(C)ccccc1', 'c1cccc(O)c1']

augmenter("Cc1ccccc1")
>>> ['c1(C)ccccc1']

augmenter.active = False
augmenter(["c1c(C)cccc1"]*5)
>>> ['c1c(C)cccc1', 'c1c(C)cccc1', 'c1c(C)cccc1', 'c1c(C)cccc1', 'c1c(C)cccc1']
```

Datasets

A couple of different datasets are provided with the PySMILESutils package. An example of a simple dataset is the SMILESDatasets, which uses one or more lists or list-like objects of SMILES strings, that it returns tuples from when indexed. Furthermore, the object has a property `.sorted_indices`, which contains the indexes of the samples sorted by the lengths of the elements of the first list, which is

necessary for the efficiency of the BucketBatchSampler (see section Combining the objects …). The usage of SMILESdatasets is illustrated in Code Box 3.

An example of a more advanced dataset is the MultiDataset, demonstrated at the bottom of Code Box 3. Here the MultiDataset is used to step through a list of data, which can then be used one after another. This is useful if there is a need to switch dataset in between epochs, one example could be to switch dataset because offline data augmentation such as Levenshtein augmentation[14] has been used. Subclassing the MultiDataset would then be necessary to load the data off the disk. An example of a MultiDataset loading from disk is available in the PickledMultiDataset, which uses a list of filenames of pickle files to load them one by one.

Code Box 3: Datasets

```
from pysmilesutils.datautils import SMILESDataset

mysmilesdataset = SMILESDataset(data.reactants, data.products)
mysmilesdataset[10]
>>>('CCCCCBr.O=Cc1ccc(Oc2ccc3ccccc3n2)cn1',
'CCCCCC(O)c1ccc(Oc2ccc3ccccc3n2)cn1')
shortest_smiles_index = mysmilesdataset.sorted_indices[0].item()
mysmilesdataset[shortest_smiles_index]
>>>('CI.CO', 'COC')
longest_smiles_index = mysmilesdataset.sorted_indices[-1].item()
mysmilesdataset[longest_smiles_index]
('C=C/C=C\\[C@H](C)[C@H](O)[C@@H](C)[C@@H](CC[C@H](C)C[C@H](C)[C@@H](O[Si](C)(C)
C(C)(C)C)[C@@H](C)/C=C\\[C@H](C[C@H](O[Si](C)(C)C(C)(C)C)[C@H](C)/C=C/C=C\\C(=O)
OC)O[Si](C)(C)C(C)(C)C)O[Si](C)(C)C(C)(C)C',
'C=C/C=C\\[C@H](C)[C@H](O)[C@@H](C)[C@@H](CC[C@H](C)C[C@H](C)[C@@H](O[Si](C)(C)C
(C)(C)C)[C@@H](C)/C=C\\[C@H](C[C@H](O[Si](C)(C)C(C)(C)C)[C@H](C)/C=C/C=C\\C(=O)O
)O[Si](C)(C)C(C)(C)C)O[Si](C)(C)C(C)(C)C')

from pysmilesutils.datautils import MultiDataset

# each list in the element represents one dataset
data_list = [list(range(5*idx, 5*(idx+1))) for idx in range(4)]

dataset = MultiDataset(data_list, repeats=False, shuffle=False)

for _ in range(dataset.num_steps):
    print(dataset[:])
    #Do training for an epoch with dataset
    dataset.step()
>>>[0, 1, 2, 3, 4]
 [5, 6, 7, 8, 9]
 [10, 11, 12, 13, 14]
 [15, 16, 17, 18, 19]
```

DataLoaders and variants

If a single epoch of the prepared dataset can't fit in memory, the BlockDataloader can provide efficient mitigation. The BlockDataloader loads data in chunks or blocks into memory from e.g. an HDF file that is too large for the machine memory. The chunck size can be chosen to fit the resource restraints, i.e. the amount of memory. The minibatches are then sampled randomly from the current in-memory

chunk. The dataloader discards the current block from memory and loads the next chunk when needed. If the chunks are sufficiently large, the random minibatches will probably be different between each epoch, while at the same time loading from disk into memory is efficient as the read is consecutive. The approach breaks with a full stochastic sampling of the dataset, but in practice we have not observed a difference in training and test performance if the dataset is pre-shuffled and the blocks are kept sufficiently large, as was done with the pre-processed graph tensors and action tensors in GraphInvent[15].

Combining the objects for an efficient whole

As shown in Figure 2, the individual elements must be combined to enable the creation of mini-batches for training. However, the objects are flexible and can be combined in several different ways. To investigate the most efficient way we constructed a small transformer model[16] and trained it with several different ways of organizing the code and training. First, we put the tokenization and collating of the mini-batches into the training loop itself. Secondly, we put the tokenization into the dataset and padded it with a customized collate function in the dataloader. Alternatively the dataset could produce padding of a fixed width. Lastly we added both the tokenization, padding and collating into a custom collate function used by the dataloader. The timings for training an epoch of test data (USPTO-50K[17] has 40.000 training samples) are shown in Table 1.

Table 1: Training efficiency comparison of different code organizations.

Code strategy	Epoch time (s)	Samples/s	Avg. batch length
In training loop	110	454	113
In Dataset	115	434	113
In Collate function	66	757	113
+ Sorted lengths	36	1390	60
+ BucketBatchSampling	39	1299	61

It is evident that the slowest option is to put the tokenization into the training loop itself or using the pre-tokenized dataset. Customizing the collate function is nearly twice as fast, presumably because this enables the dataloader to work on one or several workers in parallel to the GPU training. The bottom half of Table 1 shows the speedup that can be obtained by bundling together SMILES of similar length in the mini-batches. Sorting the samples by length is the optimal case for speed-up, but would likely be detrimental to training as the mini-batches would be the same in each epoch. The BucketBatchSampler object balances the need for similar size batches with the need for random batches, by dividing the data into a number of “buckets” based on the length of the SMILES strings. Mini-batches are then drawn randomly from each bucket, ensuring different mini-batches of similar length SMILES for each epoch. Padding is thus minimized and it is evident from Table 1 that the training performance in terms of speed is close to the optimally obtainable with the sorted lengths. The full example code for the different code organizations can be found as a %delimited examples_training.py script in the examples directory in the code package.

Comparable frameworks and projects

Several other projects and frameworks contain code for working with SMILES based deep learning for chemistry. One such project is SMILES-enumeration (https://github.com/EBjerrum/SMILES-enumeration)[3]. It is a more or less monolithic class for both tokenization and data augmentation aimed at providing tensors for Keras[18] training. It only supports single-character based tokenization,

one-hot encoding and fixed-length padding. It has been superseded by MolVecGen, which also contains SMILES based tensor generation as well as objects for creating chemception images[2] or tensors based on RDKit fingerprints[1] (https://github.com/EBjerrum/molvecgen). The well-established DeepChem[19] project has a SmilesToSeq class amongst the featurizers (https://deepchem.readthedocs.io/en/latest/api_reference/featurizers.html#smilestoseq), which supports character-based tokenization and fixed width padding, where it uses indices-based encodings. Augmentation utilities seem not to be available and would have to be coded independently.

OpenChem[20] is another project which can be used for SMILES based deep learning (https://github.com/Mariewelt/OpenChem/tree/master/openchem/data). The project seemingly aims to both provide models and various dataset classes, where one supports SMILES tokenization and augmentation. The code is adapted from the SMILES enumeration project mentioned above.

In comparison with the other frameworks, PySMILESUtils aims to only handle the SMILES conversions and datamodel in a flexible and extendible way and is fully model agnostic, so that it can be adapted to the needs of the task and deep learning algorithm. Many other projects aimed for single models or algorithms, also contain code for tokenization and handling, but are not aiming to be libraries or frameworks. As examples can be mentioned Reinvent[21] and Molecular Transformer[8], for de novo design and synthesis prediction. A full review of all projects using SMILES based deep learning is however out of scope of this application note.

Conclusion

PySMILESUtils is a framework for working with SMILES based deep learning architectures in PyTorch. It has classes for efficient vocabulary and tokenization, data augmentation, as well as optimized data loading for pre-augmented datasets, out-of-memory datasets. Significant speedups can be achieved by smart sampling of mini-batches based on SMILES length. The framework should provide a good and extensible starting point for building SMILES based deep learning models. By releasing it as open-source we hope to lower the barrier in for using SMILES based methods in deep learning for the molecular data science community and allow for greater collaboration and consistency across projects

Conflicts of Interests

The authors declare no conflicts of interests.

References

[1] "RDKIT: Open source cheminformatics." [Online]. Available: http://www.rdkit.org. [Accessed: 12-Sep-2019].

[2] G. B. Goh, C. Siegel, A. Vishnu, N. O. Hodas, and N. Baker, "Chemception: A Deep Neural Network with Minimal Chemistry Knowledge Matches the Performance of Expert-developed QSAR/QSPR Models," Jun. 2017.

[3] E. J. Bjerrum, "SMILES Enumeration as Data Augmentation for Neural Network Modeling of Molecules," Mar. 2017.

[4] D. Duvenaud *et al.*, "Convolutional networks on graphs for learning molecular fingerprints," *Adv. Neural Inf. Process. Syst.*, vol. 2015-Janua, pp. 2224–2232, Sep. 2015.

[5] M. H. S. Segler, T. Kogej, C. Tyrchan, and M. P. Waller, "Generating focused molecule libraries for drug discovery with recurrent neural networks," *ACS Cent. Sci.*, vol. 4, no. 1, pp. 120–131, Jan. 2018.

[6] E. J. Bjerrum and R. Threlfall, “Molecular generation with Recurrent Neural Networks (RNNs),” *arXiv*. 2017.

[7] R. Gómez-Bombarelli *et al.*, “Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules,” *ACS Cent. Sci.*, vol. 4, no. 2, pp. 268–276, Feb. 2018.

[8] P. Schwaller, T. Laino, T. Gaudin, P. Bolgar, C. Bekas, and A. A. Lee, “Molecular Transformer - A Model for Uncertainty-Calibrated Chemical Reaction Prediction,” 2018.

[9] E. J. Bjerrum and B. Sattarov, “Improving Chemical Autoencoder Latent Space and Molecular De Novo Generation Diversity with Heteroencoders,” *Biomolecules*, 2018.

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

[11] J. Arús-Pous *et al.*, “Randomized SMILES strings improve the quality of molecular generative models,” *J. Cheminform.*, vol. 11, no. 1, 2019.

[12] B. Steiner *et al.*, “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” *Adv. Neural Inf. Process. Syst.*, no. NeurIPS, 2019.

[13] T. B. Kimber, S. Engelke, I. V. Tetko, E. Bruno, and G. Godin, “Synergy Effect between Convolutional Neural Networks and the Multiplicity of SMILES for Improvement of Molecular Prediction,” Dec. 2018.

[14] D. Sumner, J. He, A. Thakkar, O. Engkvist, and E. J. Bjerrum, “Levenshtein Augmentation Improves Performance of SMILES Based Deep-Learning Synthesis Prediction,” 2020.

[15] R. Mercado *et al.*, “Graph networks for molecular design,” *Mach. Learn. Sci. Technol.*, 2021.

[16] A. Vaswani *et al.*, “Attention Is All You Need,” *Adv. Neural Inf. Process. Syst.*, no. Nips, pp. 5998–6008, 2017.

[17] B. Liu *et al.*, “Retrosynthetic Reaction Prediction Using Neural Sequence-to-Sequence Models,” *ACS Cent. Sci.*, vol. 3, no. 10, pp. 1103–1113, 2017.

[18] F. Chollet and others, “Keras.” GitHub, 2015.

[19] B. Ramsundar, P. Eastman, P. Walters, V. Pande, K. Leswing, and Z. Wu, *Deep Learning for the Life Sciences*. O’Reilly Media, 2019.

[20] M. Korshunova, B. Ginsburg, A. Tropsha, and O. Isayev, “OpenChem: A Deep Learning Toolkit for Computational Chemistry and Drug Design,” *J. Chem. Inf. Model.*, 2021.

[21] T. Blaschke *et al.*, “REINVENT 2.0: An AI Tool for de Novo Drug Design,” *J. Chem. Inf. Model.*, 2020.