Lecture 14

Retrosynthesis with Sequence-to-Sequence Modeling

Contents

when introducing reaction kinetics with RNNs as alternative ways to represent reactions numerically.

1 Reaction Representation

Chemical reactions involve multiple chemical species and therefore constitute a more complex system than single-component molecules. In this section, we discuss several standard strategies for representing chemical reactions in text-based formalisms.

1.1 Reaction SMILES

A reaction SMILES string follows the general format

```
reactants>agents>products
```

where the > symbol separates different sections of the reaction.

Multiple species within the same section are separated using the dot symbol `.`:

```
reactant1.reactant2
```

The *agents* field refers to species that are present in the reaction system but do not contribute atoms to the products, such as solvents and catalysts. Including agents is optional; therefore, a reaction SMILES may also take the form

```
reactants> >products
```

Example: Reaction SMILES

Consider the following reaction:

$$CH_3COOH + HOCH_2CH_3 \xrightarrow[HOCH_2CH_3]{H^+,\ Cl^-} CH_3COOCH_2CH_3$$

The corresponding reaction SMILES is

```
CC(=O)O.OCC>[H+].[Cl-].OCC>CC(=O)OCC
```

Summary

- Use the > symbol to separate reactants, agents, and products.
- Use the `.` symbol to separate multiple species within the same section.
- Species can be represented as SMILES (molecules) or SMARTS (substructure patterns).
- The agents field is optional.

1.1.1 SMIRKS

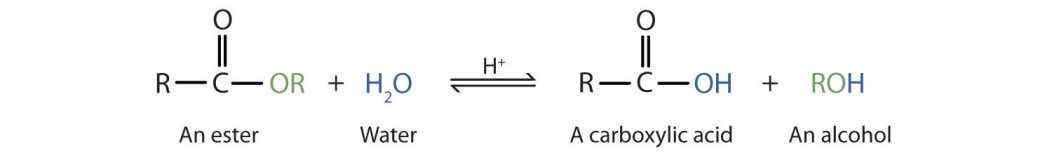
In some cases, we do not wish to represent a single, specific reaction, but rather a **class of reactions**, such as bond-order changes, atom deletion or creation, or changes in formal charge. This is achieved using *SMIRKS*, a rule-based reaction representation.

Example: SMIRKS Examples

Example 1: Carbonyl reduction (a C=O bond is transformed into a C–O bond)

```
[C:1]=[O:2]>>[C:1][O:2]
```

Example 2: Ester hydrolysis




```
[C:1](=[O:2])[O:3][C:4]>>[C:1](=[O:2])[O-:3].[C:4][OH]
```

The rules governing a valid SMIRKS string are as follows:

- Atoms are specified using the syntax `[SMARTS:Index]`, where `Index` is a unique identifier indicating atom identity, not ordering.

- Atom mapping is mandatory. Every atom that persists across the transformation must appear on both sides with the same mapping index.
- If a mapped atom appears only on the left-hand side, it is deleted; if it appears only on the right-hand side, it is newly created.
- SMIRKS encodes only structural transformations; no agents, solvents, or reaction conditions are included.

1.1.2 RXN SMILES to Atom Mapping

One can map the RXN SMILES into the SMIRKS form with automatic toolkits. A widely used Python Library is `indigo`, installed by

```
pip install epam.indigo
```

Below is a simple example

```
from indigo import Indigo

my_indigo = Indigo()
rxn = my_indigo.loadReaction("CCBr.O>>CCO")
rxn.automap("discard") # recompute
mapped_smiles = rxn.smiles()
print(mapped_smiles)
```

One can also use RDKit to visualize and transform SMILES into SMIRKS with slightly different syntax.

1.2 Reaction Fingerprints

Beyond string-based representations, chemical reactions are often encoded as **numerical fingerprints** for use in machine learning models. Two widely used reaction fingerprints are *DRFP* (Differential Reaction Fingerprint) [1] and *RXNFP* (Reaction Fingerprint via Language Models) [2]. Similar to the fingerprints for molecules discussed in the previous lectures, these two reaction fingerprints aim to represent reactions as fixed-length vectors.

Unlike popular molecular fingerprints, these two fingerprints do not have direct implementations in Python libraries such as `RDKit`. However, you can install individual libraries:

```
pip install drfp
pip install rxnfp
```

An example of using `dfrp` is

```python
from drfp import DrfpEncoder

rxn = "CCO.O=[O]>>CC=O"
encoder = DrfpEncoder()
fp = encoder.encode(rxn)
print(fp)  # numpy array of 0s and 1s
```

2 Sequence-to-Sequence (Seq2Seq) Models

So far, we have focused on predicting the next value in a single time series using RNNs. Sometimes, we need to *map one entire time series (a sequence) to another*.

Example: Sequence to Sequence Mapping in Chemical Reaction

Consider the following chemical reaction:

$$CH_3CH_2OH + CH_3COOH \longrightarrow CH_3COOCH_2CH_3 + H_2O$$

We would like to map the reactants to the products, represented by SMILES. We can treat a SMILES string as a sequence of characters (tokens). Then to map the reactants to the products, we are mapping one sequence to another:

- Input sequence: CCO.CC(=O)O $\Rightarrow$ C, C, O, ., C, C, (, =, O,), O
- Output sequence: CCOC(=O)C $\Rightarrow$ C, C, O, C, (, =, O,), C

Note that the dot symbol is used to separate two molecules and we omitted non-organic molecules (i.e. the water molecule in this example).
Therefore, a forward reaction prediction can be seen as a sequence to sequence task. If we swap the input and output sequences, i.e., product SMILES $\rightarrow$ reactant SMILES, then the task becomes a **retrosynthesis** task.

In the above example, the input and output sequences have different lengths, which is common in chemical tasks. In addition to mapping between the same type of sequences, e.g., SMILES to SMILES, we sometimes also need to map between different types of sequences, e.g., SMILES to fingerprints. Seq2Seq model is the tool that learns mapping between sequences with flexible lengths and types, which is build upon the RNN models we introduced earlier.

2.1 Architecture

A typical Seq2Seq model consists of an **encoder** and a **decoder**: the encoder takes the input sequence and compresses it into a hidden representation called the *context vector*, and the decoder takes this context vector to generate the output sequence, as illustrated in Fig. 1.

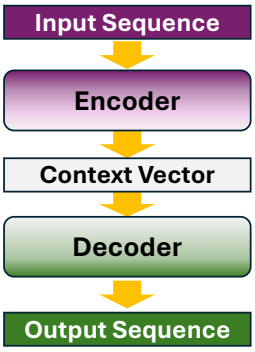


Figure 1: Seq2Seq Architecture.

2.1.1 *Encoder*

The encoder converts a sequence into a vector that is both machine-readable and contains information about the sequence. As an example, consider the reactant SMILES sequence `CCO.CC(=O)O`:

$$\texttt{C} \rightarrow \texttt{C} \rightarrow \texttt{O} \rightarrow \texttt{.} \rightarrow \texttt{C} \rightarrow \texttt{C} \rightarrow \texttt{(} \rightarrow \texttt{=} \rightarrow \texttt{O} \rightarrow \texttt{)} \rightarrow \texttt{O}$$
$$x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4 \rightarrow x_5 \rightarrow x_6 \rightarrow x_7 \rightarrow x_8 \rightarrow x_9 \rightarrow x_{10} \rightarrow x_{11}$$

In previous lectures, we called each character a **token** in the sequence, and introduced *one-hot encoding* and *learned embeddings* to represent tokens. We also discussed *pooling* strategies to aggregate the tokens into a single vector representing the entire sequence.

However, simple pooling is *order-invariant* and can be too coarse for molecular structures, especially when the training dataset is small. To retain sequential and structural information, we use RNNs and the variants (e.g, LSTM, GRU).

An RNN maintains a hidden state $\mathbf{h}_t$ at each time step t, which is updated based on the current token x_t and the previous hidden state $\mathbf{h}_{t-1}$. The hidden state $\mathbf{h}_t$ encodes the *memory* of the sequence up to step t, from which the sequence information can be reconstructed.

Since there is a hidden state $\mathbf{h}_t$ at every step, which one should we use as the context vector? The simplest choice is the last hidden state $\mathbf{h}_T$, which accumulates all previous memory. We denote this final encoded vector as:

$$\text{Encoded Vector: } \mathbf{z} = \mathbf{h}_T \tag{1}$$

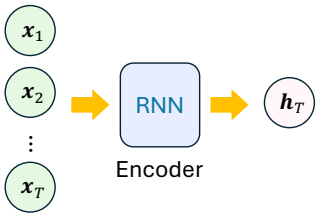


Figure 2: The encoder in Seq2Seq model.

2.1.2 *Decoder*

Now we turn the encoded vector $\mathbf{z}$ into the output sequence $(y_1, y_2, \cdots, y_{T'})$ token by token.

We can start from this encoded vector, often called the **context vector**, to generate the output sequence.

Let's define the hidden state for the output sequence as $\mathbf{g}_t$, then $\mathbf{g}_0 = \mathbf{z}$.

To start the decoding process, we define a special start-of-sequence (SOS) token y_0, which is not

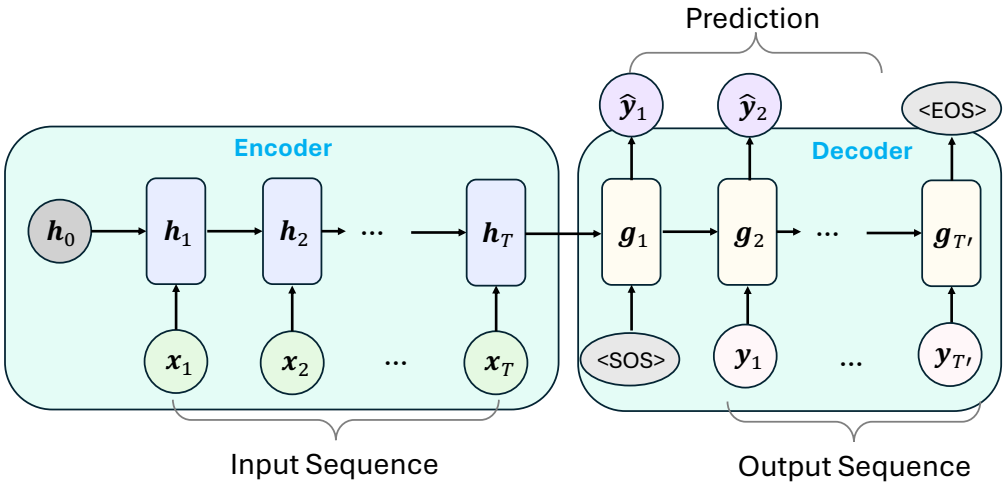


Figure 3: Seq2Seq Model with RNN backend: training.

a real SMILES token. We represent it as <SOS>:

$$y_0 = \texttt{<SOS>}, \quad \mathbf{g}_0 = \mathbf{z}. \tag{2}$$

At each subsequent step t, the decoder uses the previous token y_{t-1} and the hidden state $\mathbf{g}_{t-1}$ to compute the next hidden state $\mathbf{g}_t$ and predict y_t.

Token Prediction with Softmax

Since we are predicting discrete tokens, we use a softmax to produce a probability distribution over the output vocabulary:

$$p(\hat{y}_t) = \text{Softmax}(W_y \mathbf{g}_t + b_y), \tag{3}$$

and the predicted token is chosen as

$$\hat{y}_t = \arg\max p(\hat{y}_t). \tag{4}$$

Stop Criteria

Just like we have <SOS> to start the sequence, we define an <EOS> token to signal the end of generation. When $\hat{y}_t$ = <EOS>, the decoder stops. We also set a maximum sequence length to avoid infinite outputs.

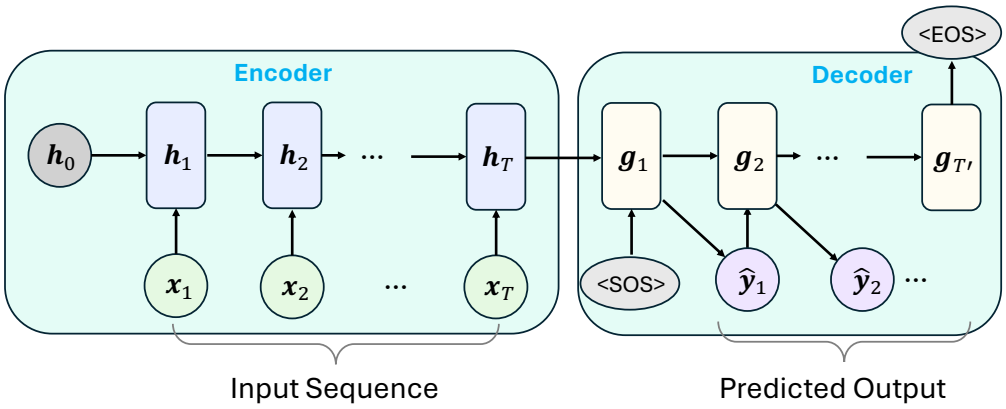


Figure 4: Seq2Seq Model: inferencing.

2.1.3 Loss Function

Since the decoder predicts a sequence of discrete tokens, the training objective is a classification problem at each step. We use the cross-entropy loss:

$$\mathcal{L} = \sum_{t=1}^{T'} \text{CrossEntropy}\big(p(\hat{y}_t), y_t\big), \tag{5}$$

where y_t is the true token at step t.

The architecture of Seq2Seq with RNN encoder and decoder is shown in Fig. 3.

2.1.4 Inferencing

After the model is trained, we can inference output sequences from the input sequence. The procedure is shown in Fig. 4.

2.2 Attention (Advanced Reading)

Relying solely on the final encoder hidden state $\mathbf{h}_T$ can lose important information, particularly for long input sequences. Attention addresses this limitation by allowing the decoder to consider all encoder hidden states $(\mathbf{h}_1, \ldots, \mathbf{h}_T)$ at each decoding step. This mechanism, called **attention**, enables the model to dynamically focus on the most relevant parts of the input sequence when generating each output token.

At decoder step t, we compute a relevance *score* between the decoder hidden state $\mathbf{g}_t$ and each encoder hidden state $\mathbf{h}_j$. A simple dot-product score is:

$$\text{score}(\mathbf{g}_t, \mathbf{h}_j) = \mathbf{g}_t^\top \mathbf{h}_j. \tag{6}$$

A more flexible option introduces a learnable weight matrix W_{att}:

$$\text{score}(\mathbf{g}_t, \mathbf{h}_j) = \mathbf{g}_t^\top W_{\text{att}} \mathbf{h}_j. \tag{7}$$

We normalize these scores with a softmax to obtain attention weights:

$$\alpha_{t,j} = \frac{\exp(\text{score}(\mathbf{g}_t, \mathbf{h}_j))}{\sum_{i=1}^{T} \exp(\text{score}(\mathbf{g}_t, \mathbf{h}_i))}. \tag{8}$$

The context vector at step t is a weighted sum of encoder states:

$$\mathbf{c}_t = \sum_{j=1}^{T} \alpha_{t,j} \mathbf{h}_j. \tag{9}$$

Finally, we combine the decoder hidden state and context vector to predict the next token:

$$p(\hat{y}_t) = \text{Softmax}\Big(W_y^{\text{att}} [\mathbf{g}_t, \mathbf{c}_t] + b_y\Big). \tag{10}$$

3 Retrosynthesis

Retrosynthesis deconstruct a target molecule into simpler precursors, ultimately identifying *commercially available or easily synthesizable* starting materials. Traditional retrosynthetic analysis relies on human expertise and rule-based systems, but modern approaches increasingly leverage machine learning, particularly the Seq2Seq models.

Sequence-to-sequence models mapping a *product molecule* to one or more *reactants*. Both input and output are commonly represented as **SMILES strings**.

Product SMILES → Reactants/Precursor SMILES

Dataset Seq2seq retrosynthesis models are trained on large reaction datasets, such as:

- USPTO reaction dataset (website for data).
- Reaxys or proprietary reaction databases (Website).

Tokenize Strings Tokenization of a rxn SMIRKS is a little complicated, as we need to group parts in the bracket together.

Example: Tokenization of RXN SMIRKS

`[CH3:1][CH2:2]Br.[OH2:4]` $\xrightarrow{\text{tokenize}}$ `([CH3:1], [CH2:2], Br, ., [OH2:4])`

One has to explicitly define rules to extract segments beyond a single character.

Token Vocabulary A vocabulary collects all distinct tokens and assign an index to each token.

Example: Vocabulary of RXN strings

Suppose we have an rxn string

`CCO.O>>CC=O`

Then we can build a vocabulary for reactant and product with look up indices:

Reactant Vocab: `{"C": 0, "O": 1, ".":2}`

Product Vocab: `{"C": 0, "=": 1, "O":2}`

Then we can map the reaction to a sequence mapping using the indices:

`CCO.O>>CC=O` becomes

`[0, 0, 1, 2, 1]` → `[0, 0, 1, 2]`

Once we have a digital representation of the sequences, we can use feature engineering techniques such as one-hot encoding and learned embedding to turn the sequences into ML features.

4 Lab

Link:
`https://colab.research.google.com/drive/1zA7qO-tzORtCg6w1AZf2nFLtystE4KgE?usp=sharing`

References

[1] Daniel Probst, Philippe Schwaller, and Jean-Louis Reymond. Reaction classification and yield prediction using the differential reaction fingerprint drfp. *Digital discovery*, 1(2):91–97, 2022.

[2] Philippe Schwaller, Daniel Probst, Alain C Vaucher, Vishnu H Nair, David Kreutter, Teodoro Laino, and Jean-Louis Reymond. Mapping the space of chemical reactions using attention-based

neural networks. *Nature Machine Intelligence*, 3(2):144–152, 2021.