Lecture 8

Chemical Feature Engineering II

Contents

Feature engineering transforms the system of interest (raw data) into mathematical inputs that can be directly processed by machine learning models, as illustrated in Fig. 1. It involves (1) representing the raw data and (2) converting the *representation* into tensors (vectors, matrices, etc.), a process known as *vectorization*.

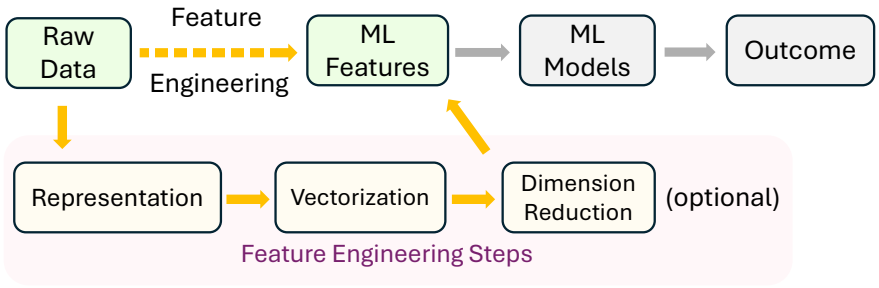

Figure 1: Feature Engineering Process.

Regarding chemical representation and feature engineering, we have already introduced structure-based representations (1D strings, 2D graphs, and 3D coordinates) as well as descriptor-based feature engineering (e.g., fingerprints and chemical properties in Lecture 3).

However, descriptor-based feature engineering relies on human-selected information, which can limit generalization to cases not anticipated by human intuition. In this lecture, we introduce a more general approach to chemical feature engineering: **structure-based feature engineering**.

1 One-Hot Encoding

In this section, we introduce a fundamental encoding technique for **categorical** (discrete-valued) features: *one-hot encoding*. Two key concepts are required: **tokens** and a **vocabulary**.

Tokens are symbols used to represent discrete entities. Given a dataset, all distinct tokens appearing in the data are collected into a *vocabulary*.

Example: Tokens and vocabulary

Tokens can represent different types of discrete entities, for example:

- Letters: `c`, `C`, `O`, `S`.
- Chemical elements: `Br`, `Cl`, `Mg`.
- Symbols: `=`, `#`, `[`.
- Numbers (when combined with other tokens): `0`–`9`.
- Words: as used in large language models.
- Anything that is discrete!

Consider a dataset consisting of two SMILES strings:

$$\{\texttt{CCO}, \texttt{C=CN}\}.$$

One possible vocabulary extracted from this dataset is

$$\mathcal{V} = \{\texttt{C}, \texttt{O}, \texttt{=}, \texttt{N}\}.$$

Once a vocabulary is defined, one-hot encoding maps each token to a **binary vector** whose length equals the vocabulary size.

For a vocabulary containing m distinct tokens,

$$\mathcal{V} = \{s_1, s_2, \ldots, s_m\},$$

each token s_i is encoded as a vector whose i-th entry is 1 and all other entries are 0. This corresponds to the natural basis vectors:

$$\text{token } \; s_i \longrightarrow \text{vector } \; \mathbf{e}_i = [0, 0, \cdots, \underbrace{1}_{i\text{-th position}}, \cdots, 0]. \tag{1}$$

A sequence of L tokens can then be represented as a **matrix** of size $L \times m$, where each row corresponds to the one-hot vector of a token in the string.

Why not use 1, 2, 3, . . . to represent categories?

One-hot encoding is preferred over using integer labels such as 1, 2, 3, . . . for categorical features because integers imply an artificial ordering and magnitude that do not exist in the data. For example, representing atomic numbers or bond types as 1, 2, 3 suggests a numerical relationship

between categories, which may mislead machine learning algorithms into treating them as ordinal or continuous values. In contrast, one-hot encoding represents each category independently as a binary vector, ensuring that all categories are treated equally and avoiding unintended biases in the model.

Limitations. One-hot encoding preserves the exact symbolic identity of each token and introduces no chemical bias beyond the chosen vocabulary. However, it produces high-dimensional, sparse representations and does not explicitly capture chemical similarity between tokens. Learned embeddings, which will be introduced in a later section, address this limitation.

In the following, we review the three common levels of chemical representation—1D, 2D, and 3D—and illustrate how one-hot encoding can be used to vectorize their features.

1.1 1D Strings

One-dimensional string representations of chemical systems use a **sequence of tokens** to encode structural information. Common examples include SMILES and InChI strings. We use an example to denote how to encode a string representation with one-hot encoding.

Example: One-hot encoding for SMILES

We define a vocabulary

$$\mathcal{V} = \{\texttt{Cl}, \texttt{Br}, \texttt{C}, \texttt{O}, =\}.$$

The one-hot encoding vectors for this five-token vocabulary are

$$\begin{aligned} &\texttt{Cl}: \ [1,0,0,0,0], \qquad \texttt{Br}: \ [0,1,0,0,0], \qquad \texttt{C}: \ [0,0,1,0,0], \\ &\texttt{O}: \ [0,0,0,1,0], \qquad =: \ [0,0,0,0,1]. \end{aligned}$$

The SMILES string of chloromethanol, `ClCO`, is tokenized as [`Cl`, `C`, `O`] and encoded as the matrix

$$\texttt{ClCO} \Rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

1.2 2D Graph Representation

A graph is composed of **nodes** (vertices) and **edges** that connect nodes. Mathematically, a graph is

$$G = (V, E), \tag{2}$$

where V denotes the set of nodes and E denotes the set of edges.

A graph representation of a molecule takes atoms as **nodes** and chemical bonds as **edges**:

- **Nodes (V):** Atoms and their properties (e.g., element type, charge).
- **Edges (E):** Bonds and their properties (e.g., bond order, conjugation).

The vectorization of a molecular graph includes three matrices: (1) the **feature matrix**, (2) the **edge index**, and (3) the **edge attribute matrix**.

1.2.1 Feature Matrix

The feature matrix X is used to represent the nodes, i.e., the atoms. Several structural features can be attached to each atom, e.g., atomic number, hybridization, and formal charge.

For a molecule with N atoms, the feature matrix X has N rows.

Example: Feature matrix for CO_2

We include two features for each node:

1. **Atomic number:** 8 for O, 6 for C.
2. **Hybridization:** 1 = sp, 2 = sp^2, 3 = sp^3.

Then the feature matrix for CO_2 is

$$\text{Atoms: } \begin{bmatrix} \text{O} \\ \text{C} \\ \text{O} \end{bmatrix} \longrightarrow X = \begin{bmatrix} 8 & 2 \\ 6 & 1 \\ 8 & 2 \end{bmatrix}$$

One-hot encoding. Atomic numbers are not an ideal representation, because although $8 > 6$, the numerical magnitude itself is not meaningful in this context and may mislead the learning algorithm.

Since we have introduced one-hot encoding, we adopt it here. We define two simple

vocabularies:

$$\mathcal{V}_1 = \{\mathrm{C}, \mathrm{O}\}, \quad \mathcal{V}_2 = \{sp, sp^2, sp^3\}.$$

Then X becomes

$$X_{\text{one-hot}} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

where the blue columns denote atomic types and the yellow columns denote hybridization types.

1.2.2 Edge Index

An edge is identified by the two nodes it connects. An edge index matrix therefore stores the *source node* and *target node*. Suppose there are n_e edges; then the edge index matrix has size $(2 \times 2n_e)$, where the first row denotes the indices of the source nodes and the second row denotes the indices of the target nodes.

We have $2n_e$ columns instead of n_e columns because we do not distinguish sources and targets in an **undirected graph**. To preserve symmetry, each edge is stored in both directions.

Example: Edge index matrix

Suppose we have a graph with four nodes and four edges, shown in Fig. 2.

The four edges are identified by the node pairs they connect:

$$(0, 1),\ (0, 3),\ (1, 3),\ (2, 3).$$

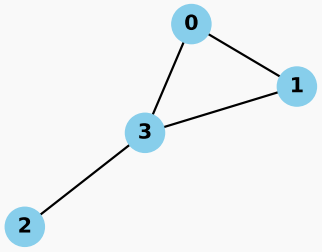

Figure 2: Example graph.

The edge index matrix is

$$\text{Edge Index:} \quad \begin{matrix} \text{source:} \\ \text{target:} \end{matrix} \begin{bmatrix} 0 & 0 & 1 & 2 & 1 & 3 & 3 & 3 \\ 1 & 3 & 3 & 3 & 0 & 0 & 1 & 2 \end{bmatrix}$$

The reason we have eight columns instead of four is that we include both directions, e.g., $(0, 1)$ and $(1, 0)$.

Here, the flipped edges are placed at the end for clarity. A more common convention is to

place the two directions of the same edge next to each other:

$$\begin{matrix}\text{source:}\\ \text{target:}\end{matrix}\begin{bmatrix}0 & 1 & 0 & 3 & 1 & 3 & 2 & 3\\ 1 & 0 & 3 & 0 & 3 & 1 & 3 & 2\end{bmatrix}$$

1.2.3 Edge Attributes

Chemical bonds have their own features. For example, a bond can be a single, double, or triple bond. Bonds may also be part of a conjugated system or a ring system, and may include stereochemical information such as cis/trans (E/Z) configurations.

We use the edge attribute matrix to encode features for each edge. The edge attribute matrix has $2n_e$ rows, where n_e is the number of unique edges (bonds). The factor of 2 again arises from the undirected nature of the graph. The number of columns depends on how the bond features are encoded.

Example: Acetic acid `CC(=O)O`

Here, we include only the bond order as the edge feature. Using one-hot encoding, instead of representing bond orders by 1, 2, and 3, we use

Single bond: $[1, 0, 0]$, Double bond: $[0, 1, 0]$,
Triple bond: $[0, 0, 1]$.

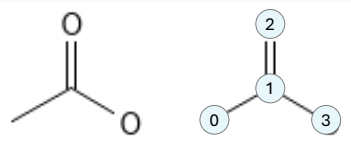

Figure 3: Graph representation of acetic acid.

The (undirected) edge index matrix and edge attribute matrix for the acetic acid graph are

$$\text{Edge Index:}\quad \begin{bmatrix}0 & 1 & 1 & 2 & 1 & 3\\ 1 & 0 & 2 & 1 & 3 & 1\end{bmatrix},\quad \text{Edge Attributes:}\quad \begin{bmatrix}1 & 0 & 0\\ 1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 1 & 0\\ 1 & 0 & 0\\ 1 & 0 & 0\end{bmatrix}$$

Remarks: One-hot encoding is widely used for *categorical* features, such as atomic types and bond orders. However, we do not apply one-hot encoding to the edge index matrix, because the index values explicitly identify graph connectivity and therefore carry intrinsic meaning.

1.3 3D Shapes

3D representations capture the spatial features of a chemical system more accurately than 1D or 2D representations. In this section, we introduce some 3D representation strategies combined with one-hot encoding.

1.3.1 Atom-Based Point Clouds

The 3D information of a molecule is defined by the atomic type and its 3D coordinates:

$$\{a_i, (x_i, y_i, z_i)\}$$

where a_i is the atomic symbol of the ith atom, and (x_i, y_i, z_i) is its 3D coordinate.

Since (x_i, y_i, z_i) are continuous vectors, we retain them as-is. We use one-hot encoding to represent the atomic types.

Example: 3D atom-based point cloud for H_2O

The 3D representation of H_2O is

Atom	Element	x (Å)	y (Å)	z (Å)
0	O	0.000	0.000	0.000
1	H	0.958	0.000	0.000
2	H	-0.239	0.927	0.000

Let the vocabulary of elements be $\mathcal{V} = \{\text{H}, \text{O}\}$. Using one-hot encoding, the atom features become:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}$$

The 3D coordinates are stored in a separate matrix:

$$R = \begin{bmatrix} 0.000 & 0.000 & 0.000 \\ 0.958 & 0.000 & 0.000 \\ -0.239 & 0.927 & 0.000 \end{bmatrix}$$

The final atom-based point cloud representation is the tuple (X, R):

$$(X, R) = \left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0.000 & 0.000 & 0.000 \\ 0.958 & 0.000 & 0.000 \\ -0.239 & 0.927 & 0.000 \end{bmatrix} \right)$$

This representation preserves the atomic type information as one-hot vectors while maintaining the 3D geometry for ML models.

1.3.2 Distances as Invariant Features

While the 3D point cloud captures the geometry, it is *rotation- and translation-dependent*. For many learning tasks, such as predicting potential energy surfaces (PES), the relevant information is the **interatomic distances** rather than the absolute coordinates. Therefore, we can represent a molecule using the pairwise distance matrix.

For a molecule with N atoms, define

$$D_{ij} = \|\mathbf{r}_i - \mathbf{r}_j\| \tag{3}$$

where D is an $N \times N$ symmetric matrix with zeros on the diagonal. Only the *upper triangle* ($j > i$) contains unique information, reducing the number of numbers from N^2 to $N(N-1)/2$.

Example: Vanilla Distance-Based Representation for H_2O

From the H_2O point cloud, the pairwise distance matrix is:

$$D = \begin{bmatrix} 0.000 & 0.958 & 0.958 \\ 0.958 & 0.000 & 1.513 \\ 0.958 & 1.513 & 0.000 \end{bmatrix}$$

Flattening the upper triangle gives the feature vector:

$$\mathbf{f}_{\text{dist}} = [D_{01}, D_{02}, D_{12}] = [0.958, 0.958, 1.513]$$

For ML models, one can combine the atomic types (one-hot encoding) with distances:

$$(X, \mathbf{f}_{\text{dist}}) = \left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0.958 & 0.958 & 1.513 \end{bmatrix} \right)$$

This representation preserves both the chemical identity of each atom and the 3D geometry, while being invariant to rotations and translations.

In the next lecture, we will explore learning potential energy surfaces (PES), where we introduce more advanced representations built on top of these distance-based features.

1.4 Case Study: Ultrafast Shape Recognition (USR)

Ultrafast Shape Recognition (USR) is a method for representing molecules based on their **3D geometry**. The goal is to efficiently capture the overall shape of a molecule and enable rapid comparison between molecules.

For each molecule, a set of **reference points**—such as the centroid, the atom farthest from the centroid, etc.—is selected. Then, we compute geometric descriptors based on the *distances* of all atoms to these reference points. Instead of using all distances directly, we summarize them using **statistical descriptors** such as mean, variance, and skewness. The result is a fixed-length vector representing the molecular shape.

$$\text{Mean: } \langle \|\mathbf{r}_i - \mathbf{r}_{\text{ref}}\| \rangle, \quad \text{Variance: } \langle (\mathbf{r}_i - \mathbf{r}_{\text{ref}})^2 \rangle, \quad \text{Skewness: } \langle (\mathbf{r}_i - \mathbf{r}_{\text{ref}})^3 \rangle \tag{4}$$

This approach is similar to the pair-wise distance representation. However, by selecting only a few reference points, the vector length is reduced to $N_{\text{stat\_descriptor}} \times N_{\text{ref}}$ instead of $O(N^2)$ as in the full distance matrix. The USR vector is invariant to rotations and translations, very fast to compute, and useful for tasks like shape-based similarity search and virtual screening.

USR can also be combined with other features (e.g., atomic types or learned embeddings) for machine learning tasks. More advanced variants incorporate additional statistical descriptors or use multiple reference points for higher-resolution shape encoding.

Example: Ultrafast Shape Recognition for H_2O

We illustrate USR with the H_2O molecule.

Step 1: Define reference points for the molecule. Following the original USR method, we select four reference points:

1. **Centroid (C)**: the average position of all atoms

$$\mathbf{r}_{\text{centroid}} = \frac{1}{3}\sum_i \mathbf{r}_i = \frac{1}{3}\begin{bmatrix} 0 + 0.958 - 0.239 \\ 0 + 0 + 0.927 \\ 0 + 0 + 0 \end{bmatrix} = \begin{bmatrix} 0.2397 \\ 0.309 \\ 0 \end{bmatrix}$$

2. **Closest atom to centroid:** Oxygen (atom 0)
3. **Farthest atom from centroid:** Hydrogen (atom 1)
4. **Farthest atom from the previous farthest:** Hydrogen (atom 2)

Step 2: Compute distances from all atoms to each reference point. For example, for the centroid reference point:

$$d_i = \|\mathbf{r}_i - \mathbf{r}_{\text{centroid}}\|$$

Step 3: Compute statistical descriptors. For each reference point, compute the mean, variance, and skewness of the distances. For H_2O, this produces a vector of length $4 \times 3 = 12$.

The final USR vector for H_2O is a compact 12-dimensional descriptor encoding its 3D shape, invariant to rotation and translation:

$$\mathbf{v}_{\text{USR}} = [\text{mean}_1, \text{var}_1, \text{skew}_1, \text{mean}_2, \ldots, \text{skew}_4]$$

This vector can be used directly as input for similarity search or machine learning tasks.

USR is implemented in `rdkit`:

```python
from rdkit import Chem
from rdkit.Chem import rdMolDescriptors

mol = Chem.MolFromSmiles("CCO")
mol = Chem.AddHs(mol)

usr_vector = rdMolDescriptors.GetUSR(mol, confId=0)  # Use conformer with ID 0
```

The similarity between two USR vectors is evaluated by their Euclidean distance

$$d_{\text{USR}} = \|\mathbf{v}_1 - \mathbf{v}_2\|, \tag{5}$$

and the USR-based similarity between the two molecules is

$$\frac{1}{1 + d_{\text{USR}}}. \tag{6}$$

2 Latent Representations: Learned Embeddings

Human-selected chemical representations and features can be redundant. In previous lectures, we discussed how descriptor-based representations may contain redundant information and reviewed basic dimension reduction techniques such as principal component analysis (PCA). While one-hot encoding is highly generalizable, it is also sparse and high-dimensional, which can make learning inefficient. Moreover, one-hot vectors do not capture any notion of similarity between tokens.

In this section, we introduce an advanced dimension reduction technique: **learned embeddings**, which automatically learn a *dense, continuous vector* representation for each token during training.

This learned, low-dimensional representation space is called a **latent space**.

Latent Space In machine learning, a latent space is a lower-dimensional, continuous vector space that encodes high-dimensional input data in a compact, informative form. The term "latent" refers to features that are *not directly observed* in the raw data but are inferred or learned by the model.

The latent space is typically learned by the model and is *task-specific*. The primary human input is the choice of latent dimension, unlike feature selection or PCA, where a human-defined rule is applied. Latent spaces are usually optimized *nonlinearly* to capture patterns most relevant to the learning objective.

2.1 Tokens to embedded vectors

Instead of representing each token as a sparse one-hot vector, we assign a **trainable vector** of dimension d to each token. The embedding vectors are initialized *randomly* and updated through gradient descent, along with other parameters of the neural network.

Example: Learned embedding vs one-hot encoding for small molecules

Consider a set of small molecule,

$$\{H_2O,\ CH_4,\ NH_3,\ HF,\ PH_3,\ H_2S,\ HCl,\ HBr,\ HI\}$$

The corresponding vocabulary is

$$\mathcal{V} = \{\text{H, C, N, O, F, P, S, Cl, Br, I}\} \quad (|\mathcal{V}| = 10)$$

The one-hot encoding for H_2O is

$$X_{\text{one-hot}} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} \text{O} \\ \text{H} \\ \text{H} \end{matrix}$$

Even though each molecule is small, the large vocabulary makes one-hot inefficient and sparse.
A learned embedding in 3D can be

$$X_{\text{embed}} = \begin{bmatrix} 0.42 & -0.10 & 0.05 \\ 0.88 & 0.03 & -0.07 \\ 0.88 & 0.03 & -0.07 \end{bmatrix} \begin{matrix} \text{O} \\ \text{H} \\ \text{H} \end{matrix}$$

The learned embedding vector is much denser, and continuous, and can still capture the underlying pattern.

Remarks

- Embeddings are learned *during training*, so they are task-specific.
- Similar atoms or tokens can end up with similar embedding vectors, capturing chemical intuition automatically.
- The one-hot encoding and embedding are two independent encoding systems, and we do not need to have a one-hot encoding first. Simply map your tokens to embedded vectors.

2.2 Neural-Network Realization

Learned embeddings have become a standard practice for handling discrete inputs, such as in large language models (LLMs). In most machine learning workflows, the embedding layer is treated as the **first layer** of the neural network. The embedding vectors are trainable parameters and are updated during training, just like the weights in other layers of the network.

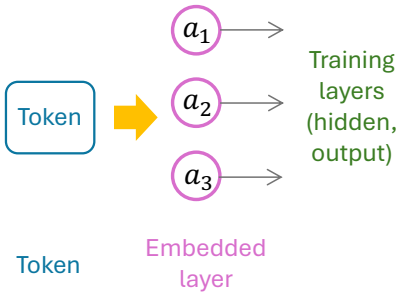

Figure 4: Embedding layer as an extra layer in a neural network.

The tokenized representation, e.g., one-hot encoding, and the embedding vectors have a one-to-one correspondence stored as a **lookup table** (the embedding matrix). Whenever a token's embedding vector is updated during training, the corresponding row in the lookup table is updated automatically.

For example, the embedding matrix for a vocabulary with {O, H} is

$$X_{\text{embed}} = \begin{bmatrix} 0.42 & -0.10 & 0.05 \\ 0.88 & 0.03 & -0.07 \end{bmatrix} \begin{matrix} \text{O} \\ \text{H} \end{matrix}$$

2.2.1 PyTorch Implementation

In `PyTorch`, an embedding layer is implemented as `nn.Embedding(vocab_size, embedding_dim)`. It is essentially a lookup table of size (vocab_size × embedding_dim). Each row corresponds to a trainable vector for one token in the vocabulary. `PyTorch` provides a module to add an embedding layer to your neural network.

```python
import torch
import torch.nn as nn

# Vocabulary of size 6, embedding dimension 3
vocab = {"H": 0, "C": 1, "O": 2, "N": 3, "S": 4, "=": 5}
vocab_size = len(vocab)
embedding_dim = 3

embedding = nn.Embedding(vocab_size, embedding_dim)

# Batch of token indices (not one-hot)
# 0 -> "H", 5 -> "="
```

```
inputs = torch.tensor([0, 5], dtype=torch.long) # torch.long is long integer data
↪  type
embedded = embedding(inputs)  # embedding matrix of shape: (2, 3)
```

2.3 Pooling: Aggregating Atom Embeddings

In many molecular learning tasks, we start with *per-atom embeddings*. To make a molecule-level prediction, we need a single vector that summarizes all the atom information. This is where **pooling** becomes essential. We cannot simply concatenate all embeddings, as the resulting vector would vary in size across molecules and could become excessively long.

Pooling is an operation that aggregates multiple vectors into one. It also ensures the representation is **invariant to atom order**, a crucial property for molecular learning. Common types of pooling include:

- **Sum pooling:** Add all atom embeddings element-wise.
- **Mean (average) pooling:** Take the element-wise average of all atom embeddings.
- **Max pooling:** Take the element-wise maximum across all atom embeddings.

Example: Mean pooling

Suppose a molecule has 3 atoms, each with a 3-dimensional embedding:

$$X_{\text{atoms}} = \begin{bmatrix} 0.1 & 0.5 & 0.2 \\ 0.3 & 0.7 & 0.4 \\ 0.2 & 0.2 & 0.9 \end{bmatrix}$$

Mean pooling:

$$\mathbf{h}_{\text{mol}} = \frac{1}{3}\sum_{i=1}^{3}\mathbf{x}_i = \frac{1}{3}\begin{bmatrix} 0.1+0.3+0.2 \\ 0.5+0.7+0.2 \\ 0.2+0.4+0.9 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.467 \\ 0.5 \end{bmatrix}$$

The resulting vector $\mathbf{h}_{\text{mol}}$ provides a **fixed-length, molecule-level representation** that can be fed into a fully connected neural network to predict molecular properties.

3 Lab

1. Ultrafast shape recognition.

https://colab.research.google.com/drive/1wB0CCWXuGFHlnZOvmAg1pUDxfP-3thBr?usp=sharing

2. Simple learned embedding for molecules. `https://colab.research.google.com/drive/1X7Ug8fkjCVIjvcQIwOjfOAyWewkcTh5e?usp=sharing`

Problems

1. Design a vocabulary for the following set of SMILES strings:

`CCO`, `ClC=O`, `CC(=O)O`, `C1CC1`, `BrC=C`

Then use your vocabulary to encode `CCO` and `CC(=O)O` as one-hot matrices.