Lecture 17

Crystal Structure Design with Autoregressive Transformer

Contents

Handling sequences, such as strings and time series, is an important task in chemistry. In the previous lectures, we learned about recurrent neural networks (RNN) and its variances, such as the long short-term memory (LSTM) and gated recurrent units (GRU). These models are *autoregressive*, where the previous output is used as the input for the next generation step. However, these methods have several limitations dealing with complicated data such as natural language, including long-term memory loss and limited parallelization on GPUs.

The key advancement happens in 2017, with the seminal work by Vaswani et al. [1], named *Attention is All You Need*. The model proposed in this paper is called a **Transformer**. Popular models based on Transformer include Bidirectional Encoder Representations from Transformers (BERT) and Generative Pre-trained Transformer (GPT).

What does *Attention is All You Need* mean? Remember that **recurrence** deals with the time-dependent sequences, while **attention** connects the input and output tokens. This title simply means that we only need attention, and can ignore recurrence in sequence learning. More specifically, Transformer relies on **self-attention**, which relates different positions in a single sequence.

In chemical science, the most famous example is AlphaFold2/3, which is based on Transformer-style attention mechanism. Of course, there are more complicated design, such as a geometric network that predicts 3D coordinates and iterative structure refinement model. Other applications include molecular and materials generation, reaction prediction and retrosynthesis, time-resolved spectroscopy, and quantum chemistry.

In this lecture, we will discuss materials design and how transformers enter the cycle to accelerate materials discovery.

1 Crystal Structure Modeling

A crystal structure can be considered as a giant molecule with repeated units, called **unit cells**. With the unit cell information, we know the information of the whole crystalline lattice. A unit cell

is composed of two parts:

- Cell dimension.
- Composition of the cell (atoms and positions).

An example of a unit cell for diamond crystal is shown in Fig. 1.

Cell dimension. Since most crystals are three-dimensional, a unit cell is often described by three vectors $(\mathbf{a}, \mathbf{b}, \mathbf{c})$:

$$\mathbf{a} = (x_a, y_a, z_a),$$
$$\mathbf{b} = (x_b, y_b, z_b),$$
$$\mathbf{c} = (x_c, y_c, z_c).$$

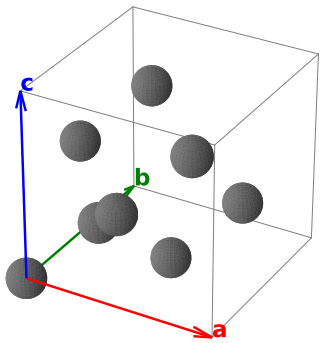

Figure 1: Unit cell for the diamond crystal.

Alternatively, one can replace the cell vectors $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ with the absolute lengths of the cell edges and angles between them:

$$(\mathbf{a}, \mathbf{b}, \mathbf{c}) \rightarrow (a, b, c, \alpha, \beta, \gamma),$$

where $a = |\mathbf{a}|, b = |\mathbf{b}|, c = |\mathbf{c}|$. $\alpha = \angle(\mathbf{b}, \mathbf{c}), \beta = \angle(\mathbf{a}, \mathbf{c}), \gamma = \angle(\mathbf{a}, \mathbf{b})$.

In many cases, the second representation of the cell is preferred, as it is *rotational invariant*.

Cell composition. The cell composition depends on the atomic types and positions in the unit cell.

$$\{\text{atom}_i, \text{position}_i\}_{i=1}^{n}$$

where atom_i is the atomic symbol and position_i are the 3D positions of the atom.

There are two choices of representing the atomic positions in a unit cell:

- The absolute coordinate. The positions are the vectors from the origin

$$\mathbf{r}_i = (x_i, y_i, z_i) \tag{1}$$

- The fractional coordinates. The positions are the fractions of the absolute coordinates with respect to the cell vectors

$$\mathbf{s}_i = (u_i, v_i, w_i), \tag{2}$$

where $\mathbf{r}_i = u_i\mathbf{a} + v_i\mathbf{b} + w_i\mathbf{c}$.

Evaluating the fractional values in $\mathbf{s}_i$ is non-trivial:

$$u_i = \frac{\mathbf{r}_i \cdot (\mathbf{b} \times \mathbf{c})}{\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})}, \quad v_i = \frac{\mathbf{r}_i \cdot (\mathbf{c} \times \mathbf{a})}{\mathbf{b} \cdot (\mathbf{c} \times \mathbf{a})}, \quad w_i = \frac{\mathbf{r}_i \cdot (\mathbf{a} \times \mathbf{b})}{\mathbf{c} \cdot (\mathbf{a} \times \mathbf{b})}, \tag{3}$$

where the cross product between two vectors are evaluated as

$$\mathbf{b} \times \mathbf{c} = \left(\begin{vmatrix} b_2 & b_3 \\ c_2 & c_3 \end{vmatrix}, \begin{vmatrix} b_1 & b_3 \\ c_1 & c_3 \end{vmatrix}, \begin{vmatrix} b_1 & b_2 \\ c_1 & c_2 \end{vmatrix} \right) \tag{4}$$

and $\begin{vmatrix} b_2 & b_3 \\ c_2 & c_3 \end{vmatrix} = b_2c_3 - b_3c_2$ is a determinant.

1.1 Data Formats for a Crystal Structure

Common data formats for storing a crystal structure includes CIF (`.cif`) and POSCAR (`.poscar`). In the following, we provide details about `.cif` files.

CIF file. A CIF file is schema-driven, and can contain one or multiple crystal structures, each belonging to a different *data block*, inside the data block, we see the following syntax:

- A data block starts with `data_XXX`, e.g., `data_Si`.
- The information is stored as *key-value* pairs, where the key starts with underscore _, e.g., `_cell_length_a`. A key-value pair is key and value together in a line, e.g.,

```
_cell_length_a  4.25504975
```

- For repeated data, e.g., atoms, we can use `loop_` to assign multiple values to one key, with the following syntax

```
loop_
_key1
_key2
 val1_1  val2_1
 val1_2  val2_2
 val1_3  val2_3
```

In most cases, you do not need to manually write or load a CIF file. Popular parsers[1] in Python Libraries, such as `pymatgen` and `ASE` reads `.cif` files automatically.

```
from pymatgen.core import Structure
```

[1] A parser is a component (software module or algorithm) that takes structured text or data and converts it into a formal internal representation that a program can operate on.

```
structure = Structure.from_file("file.cif", primitive=False)
```

By looking at the length of `structures`, you can see if this `.cif` file has one or multiple crystal structures.

```
from ase.io import read

atoms = read("file.cif") # one structure
atoms_list = read("file.cif", index=":") # multiple structures
```

Similarly, you can save the unit cell information to a `.cif` file. More details will be provided in the Lab.

1.2 Databases

You can download `.cif` files or other standard file formats from databases such as

- Materials Project: `https://next-gen.materialsproject.org/`
- AFLOW: `https://www.aflow.org/`
- Inorganic Crystal Structure Database (ICSD): `https://icsd.fiz-karlsruhe.de/`
- Cambridge Structural Database (CSD) for organic and metal–organic crystals: `https://www.ccdc.cam.ac.uk/`
- Crystallography Open Database: `https://www.crystallography.net/cod/`

1.3 Representing a Crystal Structure

Common representations of a crystal includes the **graph** representation and **3D coordinates** for the unit cell.

The 3D representation is straightforward, we just need the cell dimensions and the cell composition. Here we will focus on the graph representation. The key difficulty lies in the periodic boundary conditions (PBC) of the unit cell, which stems from the fact that the crystalline is composed of repeatedly tiling the unit cells in the three dimensions.

We define a **periodic** graph

$$G = (V, E)$$

where:

- V are nodes represented by atoms in the unit cell.
- E are edges connects atoms that *interact* with each other.

Each edge is a triplet $(i, j, \mathbf{n}_{ij})$, where:

- $i, j \in V$ are atom indices,
- $\mathbf{n}_{ij} \in \mathbb{Z}^3$ is a cell translation vector. For example,
  - $\mathbf{n}_{ij} = (1, 0, 0)$ means a translation of one unit cell along the **a** direction going from node i to j.
  - $\mathbf{n}_{ij} = (-1, 0, 0)$ means a translation of one unit cell along the opposite of the **a** direction going from node i to j.

Edge construction

Edges are defined using a **distance-based** cutoff r_c. For each pair (i, j) and lattice vector $\mathbf{n}_{ij}$, define the displacement:

$$\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i + \mathbf{L} \cdot \mathbf{n}_{ij}, \tag{5}$$

where $\mathbf{L} = (\mathbf{a}, \mathbf{b}, \mathbf{c})$.

An edge exists if:

$$\|\mathbf{r}_{ij}\| \leq r_c \tag{6}$$

In practice, we loop over a set of $\mathbf{n}_{ij}$ values to include edges due to the periodic boundary condition (PBC). This construction ensures that interactions with periodic images are properly included.

1.3.1 Node and Edge Features

Each node i carries features such as: (1) atomic number Z_i, (2) position $\mathbf{r}_i$ (or $\mathbf{s}_i$), and (3) optional properties (charge, spin, etc.).

Each edge $(i, j, \mathbf{n}_{ij})$ may include: (1) distance $\|\mathbf{r}_{ij}\|$, (2) direction $\hat{\mathbf{r}}_{ij} = \mathbf{r}_{ij}/\|\mathbf{r}_{ij}\|$, and (3) radial basis expansions of distance.

2 Transformer

A transformer is a **parametrized mapping** from one sequence to another sequence

$$(x_1, x_2, \cdots, x_{T_x}) \rightarrow (y_1, y_2, \cdots, y_{T_y})$$

where, in general, $T_x \neq T_y$.

Therefore, a transformer is a flexible **sequence-to-sequence model** that can be adapted to different tasks:

1. **Sequence translation.** For example, reaction prediction and retrosynthesis, where an input sequence is mapped to a different output sequence.
2. **Time series generation.** In autoregressive settings (e.g., GPT), the output sequence $\{y_t\}$ is generated sequentially according to

$$p(y_1, \cdots, y_{T_y}) = \prod_{t=1}^{T_y} p(y_t \mid y_{<t}),$$

which is commonly used for generative tasks such as molecular or chemical sequence generation.

In this lecture, we focus on the autoregressive transformer architecture and its key technical components. The central mechanism of a transformer is **self-attention**, which implements content-dependent interactions between tokens in a sequence. For autoregressive transformers, we additionally impose **causal masking**, which enforces temporal ordering by restricting each token to attend only to previous tokens during generation.

2.1 Self-Attention

Attention measures interactions between tokens in two sequences, while self-attention measures interactions between tokens within the **same sequence**.

Suppose there are T tokens in a sequence. We construct a $T \times T$ matrix, denoted by A, to represent the pairwise interaction weights (or **attention scores**) between these tokens.

The matrix element A_{ij} denotes the weight by which the ith token attends to the jth token, i.e., how much information from token j contributes to the representation of token i. In this sense,

- row i corresponds to the query (receiver),
- column j corresponds to the key/value (source).

How do we construct the self-attention matrix A? In the transformer model, each token is associated with three vectors: the *query*, *key*, and *value*, with the following interpretations:

- Query vector $\mathbf{q}_i$: what token i is looking for,
- Key vector $\mathbf{k}_i$: what token i offers,
- Value vector $\mathbf{v}_i$: the content to be aggregated from token i.

These vectors are obtained via learned linear mappings from the token embedding. Let $\mathbf{x}_i$ denote

the embedding of token i. We introduce trainable matrices W_Q, W_K, and W_V, such that

$$\mathbf{q}_i = W_Q \mathbf{x}_i, \quad \mathbf{k}_i = W_K \mathbf{x}_i, \quad \mathbf{v}_i = W_V \mathbf{x}_i, \tag{7}$$

where the transformation matrices are learned during training.

The self-attention matrix is then computed as

$$A_{ij} = \text{softmax}\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d}}\right), \tag{8}$$

where d is the dimension of the query and key vectors.

Stacking the vectors as *rows* of matrices Q, K, and V, we obtain the compact form

$$A = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right). \tag{9}$$

2.1.1 Updating the token vector

Now we have the attention matrix, we can update the embedding representation of the token by applying the attention matrix to the value vectors:

$$\mathbf{h}_i = \sum_{j=1}^{T} A_{ij} \mathbf{v}_j, \tag{10}$$

where $\mathbf{h}_i$ will replace $\mathbf{x}_i$. In matrix form,

$$H = AV. \tag{11}$$

2.2 Multi-head attention

In practice, transformers use **multi-head attention**, where multiple sets of query, key, and value projections are learned. Suppose we have L heads, and for each head,

$$Q^{(l)} = XW_Q^{(l)}, \quad K^{(l)} = XW_K^{(l)}, \quad V^{(l)} = XW_V^{(l)}. \tag{12}$$

Each head produces an output

$$H^{(l)} = \text{softmax}\left(\frac{Q^{(l)} K^{(l)\top}}{\sqrt{d}}\right) V^{(l)}. \tag{13}$$

The outputs from all heads are then concatenated along the feature dimension:

$$H = \text{Concat}\big(H^{(1)}, H^{(2)}, \cdots, H^{(L)}\big), \tag{14}$$

where $H \in \mathbb{R}^{T\times(Ld_v)}$. This concatenated representation is subsequently mapped back to the model dimension via a linear transformation:

$$\text{MultiHead}(X) = HW_O, \tag{15}$$

where $W_O \in \mathbb{R}^{(Ld_v)\times d_{\text{model}}}$ is a learned projection matrix.

2.3 Causal Masking

For autoregressive transformers, a **causal mask** is applied to prevent tokens from attending to future positions:

$$A_{ij} = 0 \quad \text{for } j > i. \tag{16}$$

This enforces temporal ordering in sequence generation.

In practice, causal masking is applied to the attention logits before the softmax. Let

$$S_{ij} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d}}.$$

We define the masked logits as

$$S_{ij} = \begin{cases} \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d}}, & j \le i, \\ -\infty, & j > i, \end{cases}$$

and then compute

$$A_{ij} = \text{softmax}(S_{ij}).$$

This ensures that tokens cannot attend to future positions while preserving the normalization of the attention weights.

2.4 Autoregressive Sequence Generation

In autoregressive sequence generation, the joint probability of a sequence is factorized into a product of conditional probabilities:

$$p(y_1, y_2, \cdots, y_T) = \prod_{t=1}^{T} p(y_t \mid y_{<t}), \tag{17}$$

where $y_{<t} = (y_1, \cdots, y_{t-1})$ denotes the prefix of the sequence.

In a transformer model, this factorization is implemented by enforcing **causal masking** in the self-attention mechanism, which ensures that the representation at position t depends only on tokens at positions $\leq t$. As a result, the model predicts each token based solely on previously generated tokens.

During training, the model is given the full sequence $(y_1, \cdots, y_T)$ and is optimized to predict the next token at each position:

$$\mathcal{L} = -\sum_{t=1}^{T} \log p(y_t \mid y_{<t}), \tag{18}$$

which corresponds to the negative log-likelihood (cross-entropy loss).

During inference, sequence generation proceeds sequentially. Starting from an initial token (or prompt), the model repeatedly:

1. computes the conditional distribution $p(y_t \mid y_{<t})$,
2. samples (or selects) the next token y_t,
3. appends y_t to the sequence.

This iterative process continues until a stopping criterion is met (e.g., a special end-of-sequence token or a maximum length).

Therefore, autoregressive transformers define a probabilistic model over sequences and generate outputs by sequentially sampling from the learned conditional distributions.

2.5 Conditional Sequence Generation

Conditional sequence generation extends the autoregressive framework by generating an output sequence $\{y_t\}$ conditioned on an input sequence $\{x_s\}$. The joint probability is factorized as

$$p(y_1, y_2, \cdots, y_{T_y} \mid x_1, x_2, \cdots, x_{T_x}) = \prod_{t=1}^{T_y} p(y_t \mid y_{<t}, x_1, \cdots, x_{T_x}). \tag{19}$$

In a transformer, this is typically implemented using an **encoder-decoder architecture**:

- **Encoder:** processes the input sequence $\{x_s\}$ and produces contextualized representations.
- **Decoder:** generates the output sequence autoregressively, attending to both the previously generated tokens $y_{<t}$ and the encoder representations of the input.

The attention in the decoder consists of two components:

1. **Self-attention:** attends to previously generated tokens with causal masking, enforcing autoregression.
2. **Cross-attention:** attends to the encoder output, allowing the decoder to incorporate information from the input sequence.

During training, the model maximizes the conditional likelihood of the output sequence given the input:

$$\mathcal{L} = -\sum_{t=1}^{T_y} \log p(y_t \mid y_{<t}, x_1, \cdots, x_{T_x}), \tag{20}$$

typically using teacher forcing. At inference, generation proceeds autoregressively, sampling each token conditioned on both the previously generated tokens and the encoded input sequence.

This framework enables a wide range of conditional generation tasks, including:

- Machine translation: translating sentences from one language to another.
- Text summarization: generating a summary conditioned on a longer text.
- Reaction prediction: generating chemical products conditioned on reactants.

3 Lab

- Processing CIF files: `https://colab.research.google.com/drive/1C6_nz0J9Fi8wpP8c_eG-Rs6Ww6f6zN7K?usp=sharing`

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.