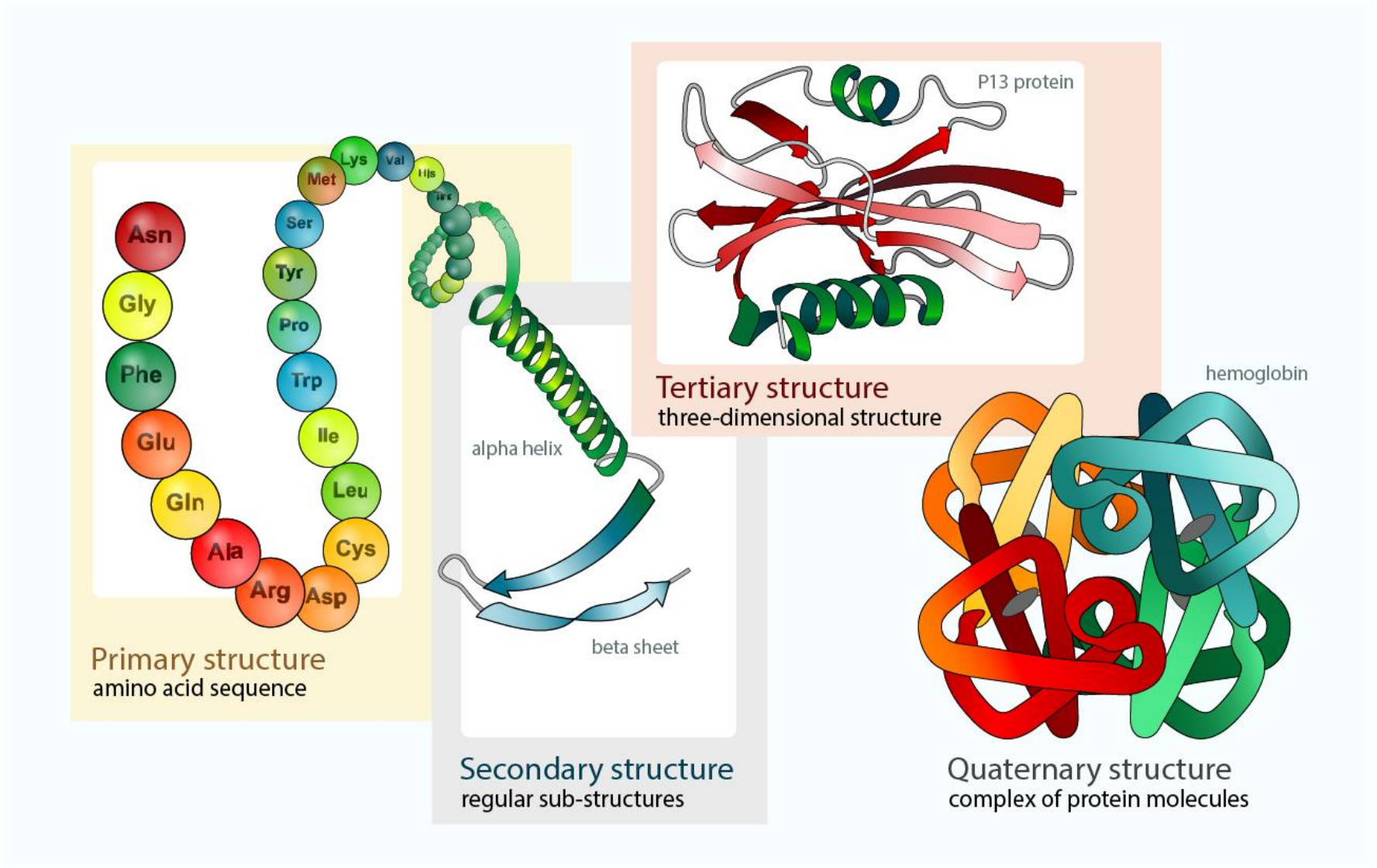
第五章 蛋白质结构预测与蛋白质语言模型

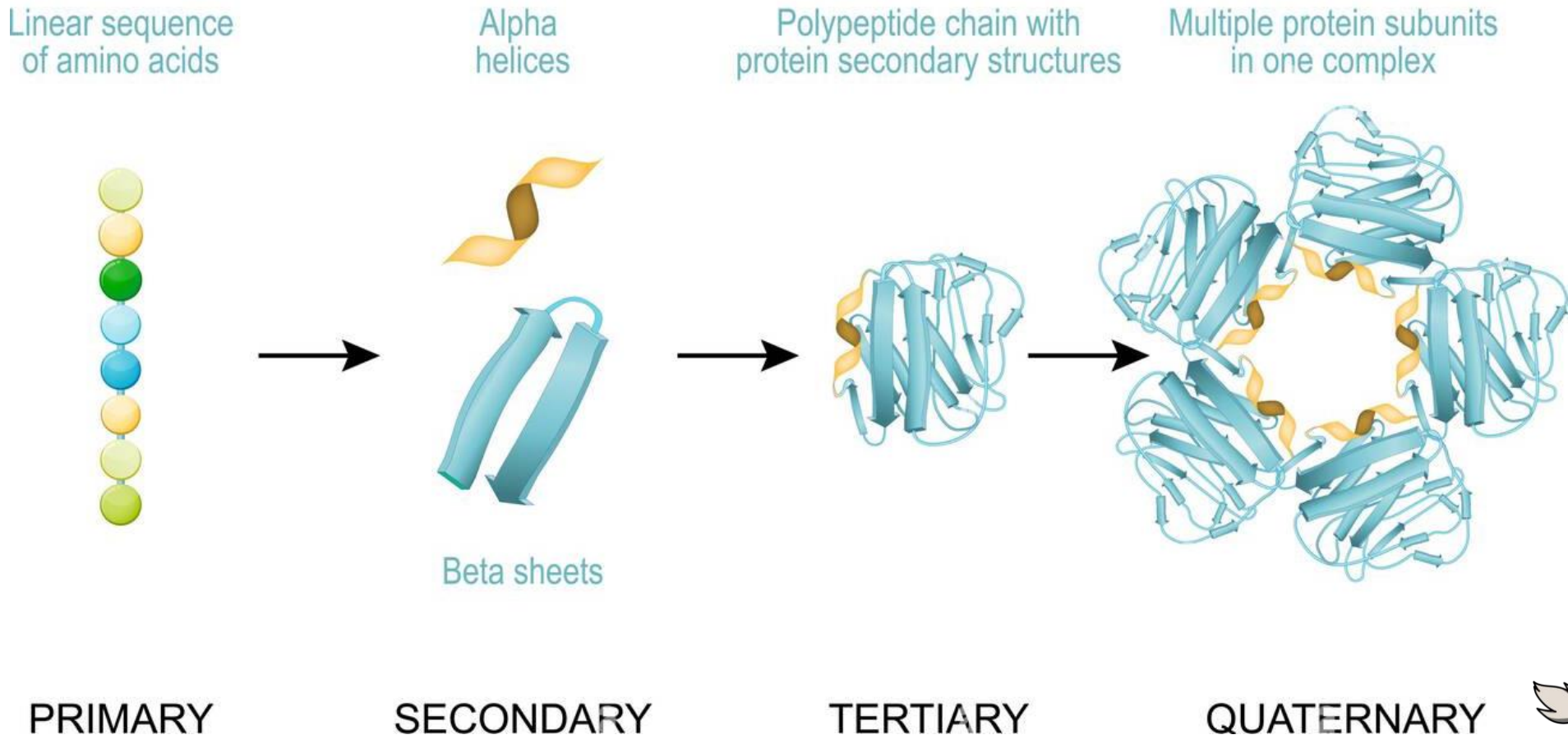
蛋白质结构预测

基于AlphaFold预测蛋白质结构及蛋白相互作用实战

蛋白质结构与功能紧密相连



从序列到结构能否实现 ？


Linear sequence of amino acids
Alpha helices
Polypeptide chain with protein secondary structures
Multiple protein subunits in one complex
Beta sheets
PRIMARY
SECONDARY
TERTIARY
QUATERNARY

蛋白质结构预测发展的里程碑事件

第一阶段： 理论奠基与早期探索

第二阶段： 同源建模兴起

PDB数据库大量扩张

第三阶段： 计算模拟与早期计算革命

进化耦合分析 (ECA)

AlphaFold

第四阶段： 深度学习革命

加速推理算法层

Diffusion model

1950s – 1970s	1980s – 1990s	1990s –2010s	2016	2018	2020	present

Christian Anfinsen

”蛋白质的天然构象由其氨基酸序列唯一决定，且处于热力学最稳定状态（自由能最低）。

论证了理论可行性（1972年诺贝尔化学奖）

二级结构预测准确率达到60%

Adrej Sali

”进化信息是关键。蛋白质结构比序列更保守，亲缘关系较近的蛋白质结构高度相似。

进化信息隐含在蛋白质结构特征中

同源建模方法取得成功，并被广泛应用

David Baker

”将长序列拆成短片段,从已知结构中提取相似片段作为“积木”，再用蒙特卡洛算法在构象空间搜索最优组合。

不依赖于整体结构模板（2024年诺贝尔化学奖）

Rosetta**开启从头预测时代**

Jinbo Xv

”利用卷积神经网络从高质量多序列比对中提取氨基酸接触信息，得到共进化特征。

提高了氨基酸接触预测的精度

开启了深度学习蛋白质结构预测时代

AlphaFold2

”深度挖掘进化信息和空间约束，直接生成原子坐标。

提出了蛋白质结构预测的革命性架构（2024年诺贝尔化学奖）

结构预测精度已接近实验水平

AlphaFold3

AlphaFold-multimer

RoseTTAFold

FastFold

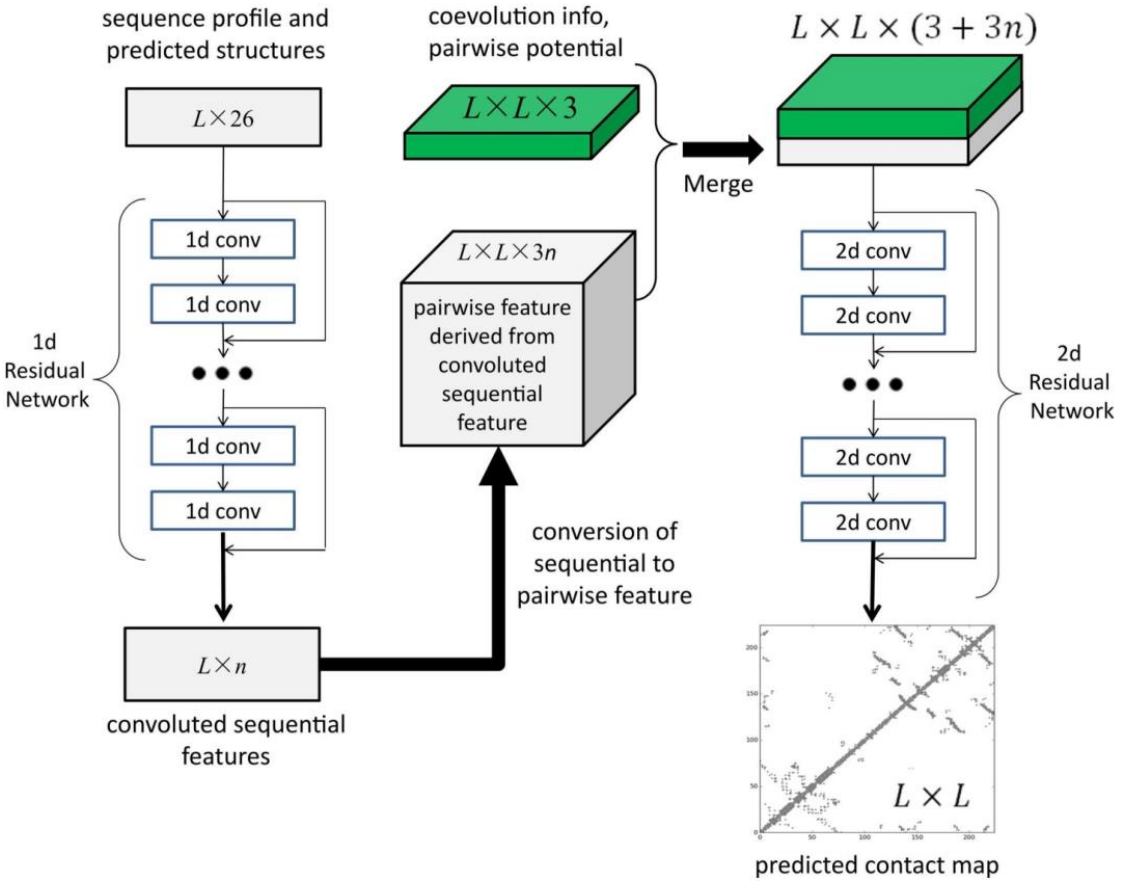
ESMFold

关键词：“动态”、“高效”、“复合体预测”、“小分子”…

基于大模型及深度学习的蛋白质结构及相互作用预测处于飞速发展水平

AlphaFold前的尝试：深度CNN提取氨基酸接触信息

- **RaptorX-Contact**

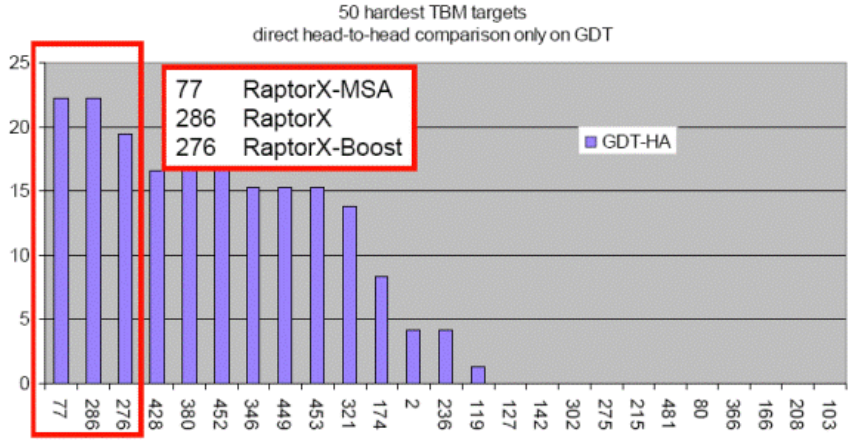


通过**两个深度残差神经网络**形成的**超深神经网络**，整合进化耦合和序列守恒信息来预测接触。

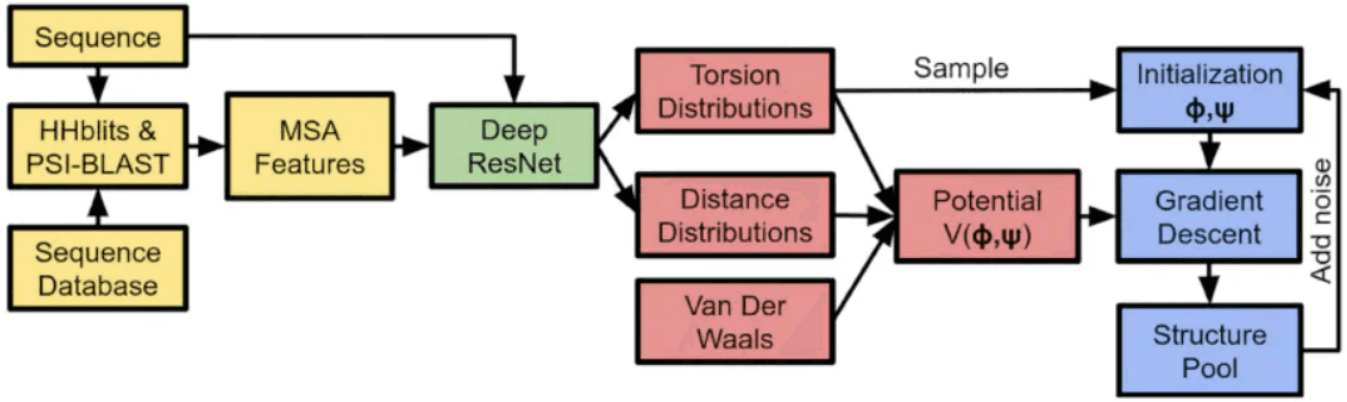
- **第一个残差网络**对序列特征进行一系列的一维卷积变换；
- **第二个残差网络**对第一个残差网络的输出、 进化耦合（evolutionary coupling，EC）信息和成对电位进行一系列二维卷积变换。

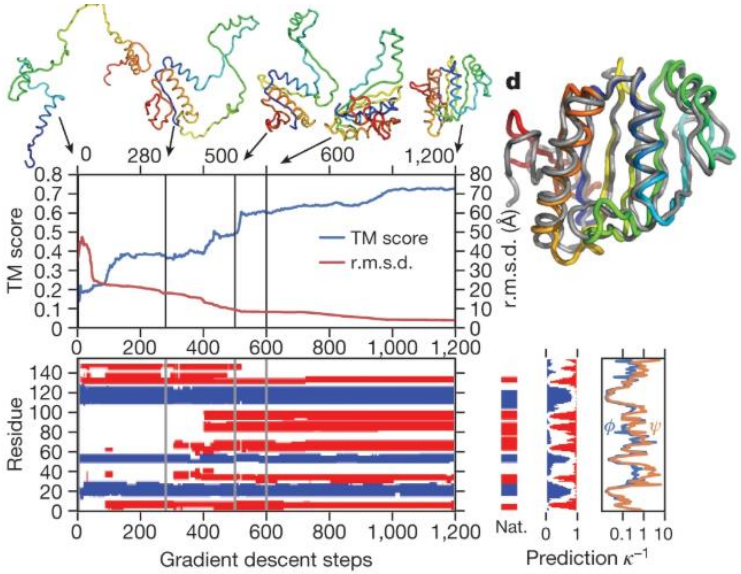
精确地模拟接触发生模式和复杂的序列结构关系，从而获得更高质量的接触预测，忽略蛋白质有多少序列同源物。

在CASP10上取得很好的效果



技术突破：AlphaFold





- **特征工程：**

序列和MSA特征抽取，把氨基酸链的输入转换到特征空间

- **Deep ResNet：**

依据特征工程中的特征预测氨基酸链的一些性质，如距离分布，夹角分布

- **评估函数构建：**

构造评估函数，评估神经网络输出解的合理程度

- **结构生成：**

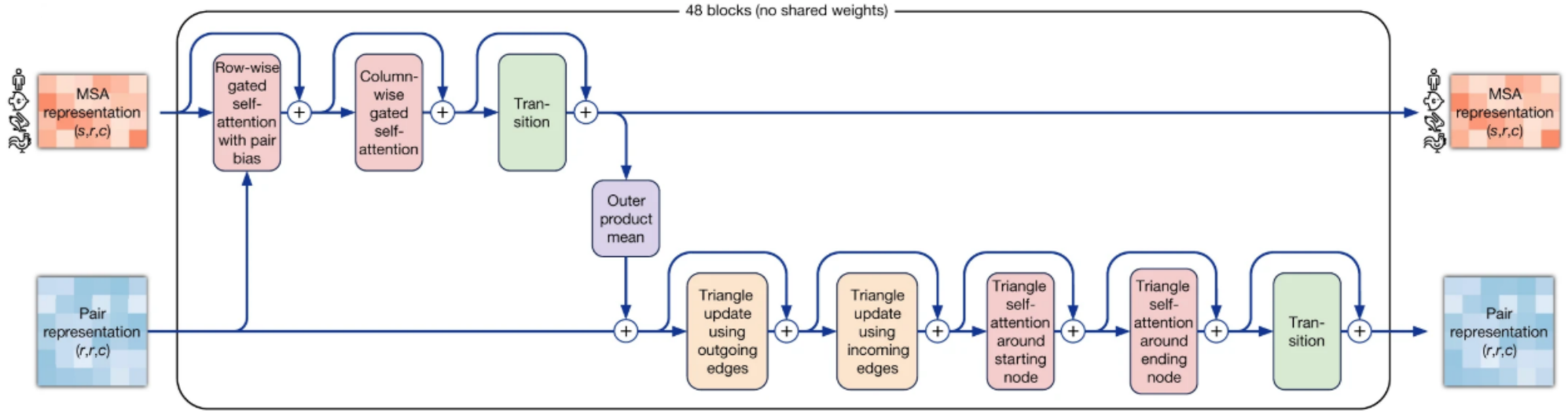
评估Loss，梯度下降法取得收敛解，得出最终预测结构

精度提升：AlphaFold2

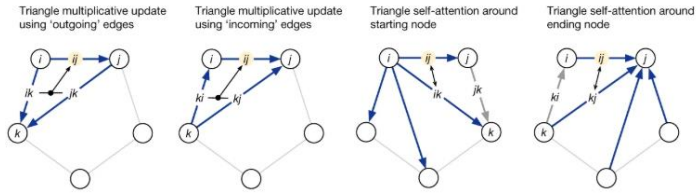
- **Evoformer模块的引入 – 将问题转化为3D图形推理问题**

高精度MSA（进化信息）

模板信息提取（初始接触概率）



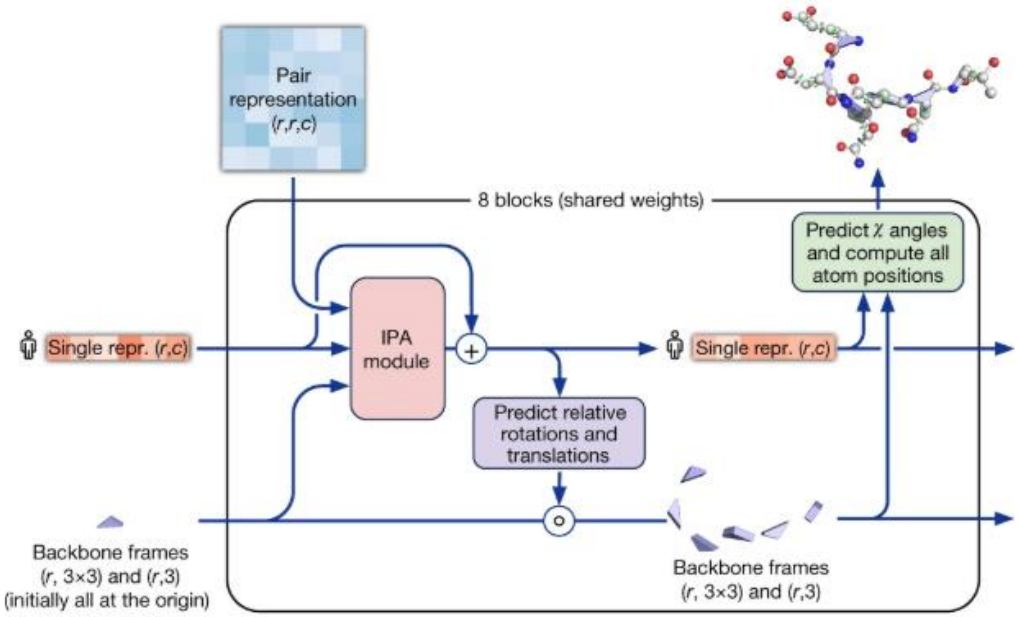
进化信息与氨基酸接触概率交互更新（反复迭代）



引入自注意机制（更新氨基酸接触概率）

精度提升：AlphaFold2

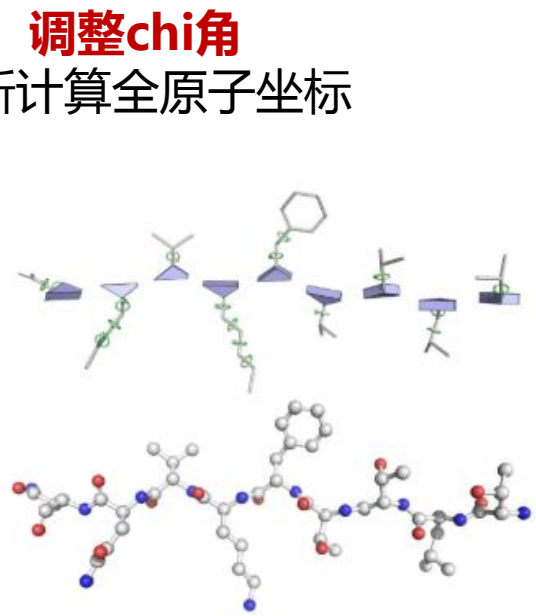
- **端到端（End-to-End）结构生成**



IPA（invariant point attention）模块

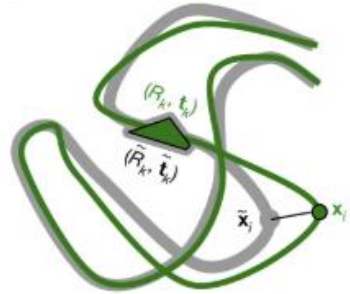
空间感知注意力

如同捏陶土一般，基于Evoformer迭代的氨基酸接触知识，8层网络连续调整主链N-C-O走向与侧链接触，但保证整体符合物理化学规律，通过Loss计算重复迭代至收敛



调整chi角

重新计算全原子坐标



模型弛豫

消除违反物理化学规律的键长与键角

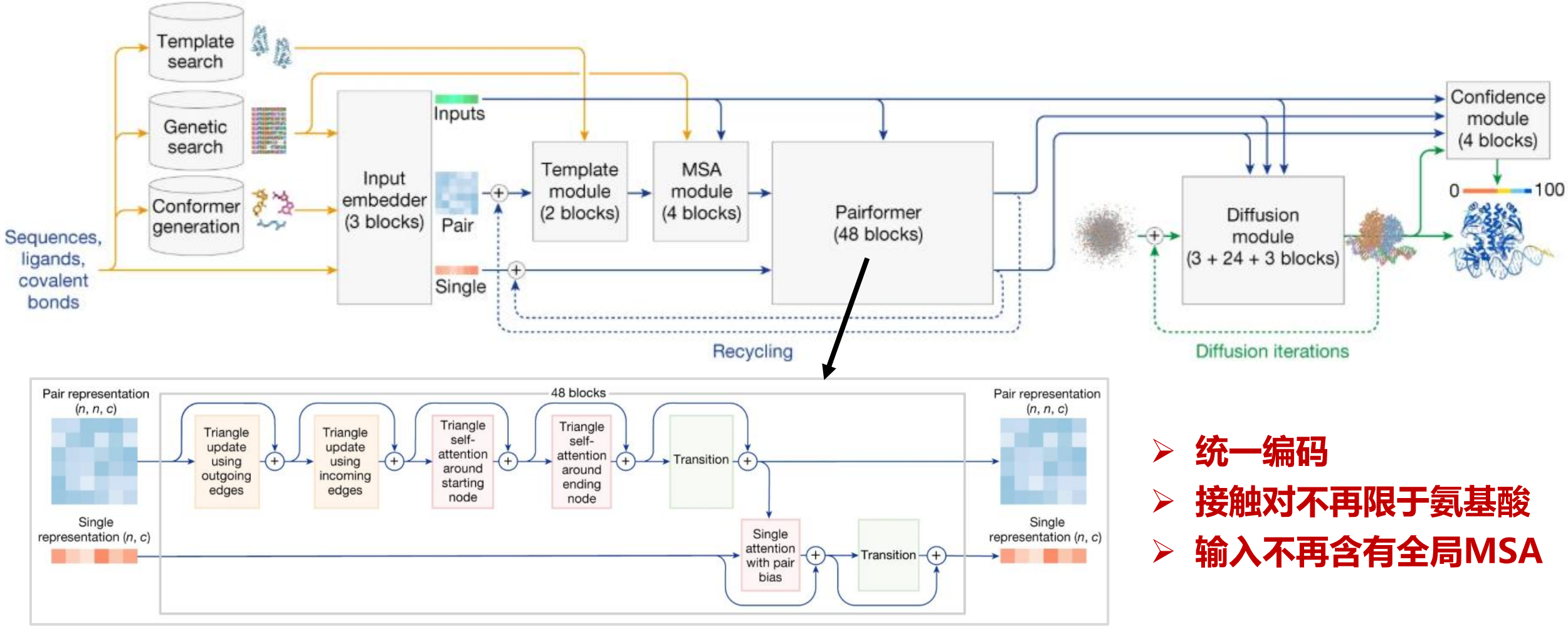
空间坐标变换

得到最终结构模型

将结构预测从“计算优化问题”转变为“表征学习问题”

泛化革命：AlphaFold3

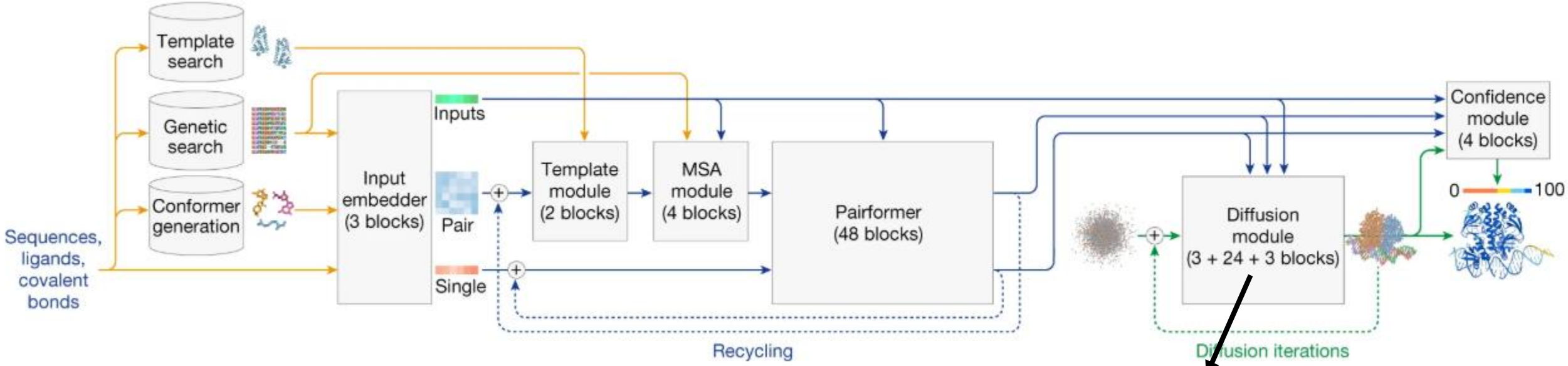
- **Pairformer 模块 – 比 Evoformer 更轻量更泛化**

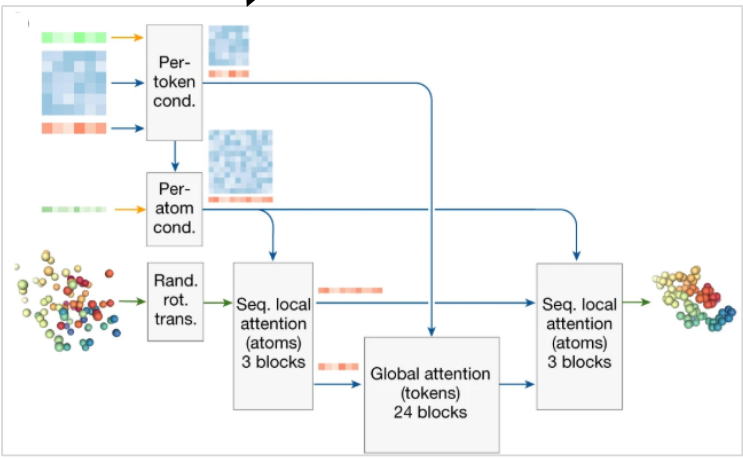


- 统一编码
- 接触对不再限于氨基酸
- 输入不再含有全局MSA

泛化革命：AlphaFold3

- **Diffusion model（扩散模型）的引入**





- **正向过程：向真实结构添加高斯噪声，逐步破坏结构**
- **逆向过程：神经网络学习从噪声中重建结构，每一步预测噪声并减去**

多模态输入引导扩散步骤 + 几何等变扩散 + 由骨架到侧链依次去噪

AlphaFold2/3能做什么？

- 蛋白质结构预测（单体）- **使用预训练模型**
- 复合体结构预测（蛋白质-蛋白质、蛋白质-核酸、蛋白质-小分子等） - **使用预训练模型**
- 自主模型训练（Open-source AF2 only）

实战：部署AlphaFold3

- **在HPC环境下通过Singularity部署AlphaFold3**

  - 加载 Singularity 模块

    ```
    $ module load singularity
    ```

  - 获取 alphafold3 docker容器映像（已打包，数据截止日期2022-05-22）

    ```
    $ wget https://path/to/alphafold3.0.1.sif
    ```

  - 拉取 alphafold3 源码

    ```
    $ git clone https://github.com/google-deepmind/alphafold3
    ```

- ✓ docker：一种将配置好的程序打包为容器的工具，本质上类似于虚拟机，打包容器理论上可以在任何平台运行。
- ✓ Singularity：专门为HPC环境打造的轻量级docker，功能与docker类似。

!NOTE: AF3的源码中，我们只取其内获取并下载数据库的脚本，运行时所用的主程序以及其他依赖被包含在了.sif映像中。

自行构建环境参考：https://github.com/google-deepmind/alphafold3/blob/main/docs/installation.md

实战：部署AlphaFold3

- **Database及model parameters获取**

  - 获取 alphafold3 model parameters

    ```
    $ wget https://path/to/af3.bin.zst
    ```

    （放至某一目录中，比如 $HOME/af3_model_param/）

  - 下载 databases

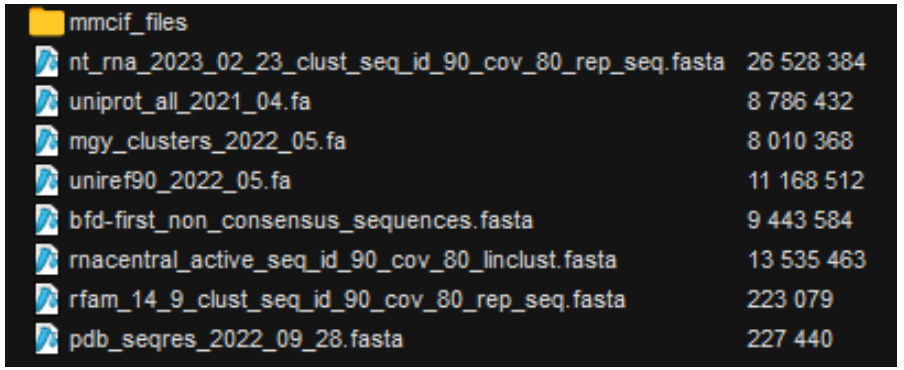
    （进入alphafold3源码路径）

    ```
    $ cd alphafold3
    ```

    （运行DB下载脚本）

    ```
    $ bash fetch_databases.sh
    ```

    【默认条件下，脚本会在 $HOME/ 路径下新建一个 public_databases/ 文件夹容纳DB】



!NOTE: 文件较大，需要耐心等待一段时间下载。

自行构建环境参考：https://github.com/google-deepmind/alphafold3/blob/main/docs/installation.md

实战：了解AlphaFold3的输入

- **JSON文件格式**

  - AF3的输入（input）是JSON格式的文件，包含了**序列（蛋白质、核酸、小分子等）信息**、**链指定**、**模型指定**、**翻译后修饰设置**、**额外共价键设置**、**模板指定**、**MSA指定等等需要告诉程序的信息**。
  - JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，常用于前后端之间的数据传输和存储。它以键值对的形式表示数据，具有简洁、易读、跨平台兼容等特点。

参考：https://github.com/google-deepmind/alphafold3/blob/main/docs/input.md

实战：了解AlphaFold3的输入

- **一个基础的AF3 input JSON文件**

```
{
  "name": "test",
  "modelSeeds": [1, 2],
  "sequences": [
  {
    "protein": {
    "id": "A",
    "sequence":
"MSSMNPEYDYLFKLLLIGDSGVGKSCLLLRFADDTYTESYIS
TIGVDFKIRTIELDGKTIKLQIWDTAGQERFRTITSSYYRGAHG
IIVVYDVTDQESFNNVKQWLQEIDRYASENVNKLLVGNKCD
LTTKKVVDYTTAKEFADSLGIPFLETSAKNATNVEQSFMTM
AAEIKKRMGPGATAGGAEKSNVKIQSTPVKQSGGGCC"
    }
  }
  ],
  "dialect": "alphafold3",
  "version": 1
}
```

- "sequences":[ ] 定义输入的序列，可以是蛋白质，核酸，小分子等。多条序列以花括号“{}”分隔，内容中需要指定序列类型，如protein、dna、rna、ligand等。
- "modelSeeds":[ ] 定义使用的random seeds，保持可重复性
- "dialect": 主程序选择，这里我们使用“alphafold3 “
- "version": 1：基础参数；
  2：适用于指定模板以及MSA；
  3：适用于指定大分子以及小分子构象参数

参考：https://github.com/google-deepmind/alphafold3/blob/main/docs/input.md

实战：了解AlphaFold3的I/O文件架构

- **在运行之前，文件架构应类似：**

```
$HOME
    |-----  alphafold3.0.1.sif      # 镜像容器
    |-----  public_databases        # 数据库文件
    |-----  af3_model_param         # 模型参数
    |-----  af3_jobs                # 预测任务的工作目录（自建）
              |-----  output        # 输出目录（自建）
              |-----  input         # 输入目录（自建）
                        |-----  test.json      # 输入.json文件
```

实战：运行AlphaFold3 ！

- **以 Human_Rab1A 蛋白为例，编写.json文件（如前所述）**
- **运行AlphaFold3：**

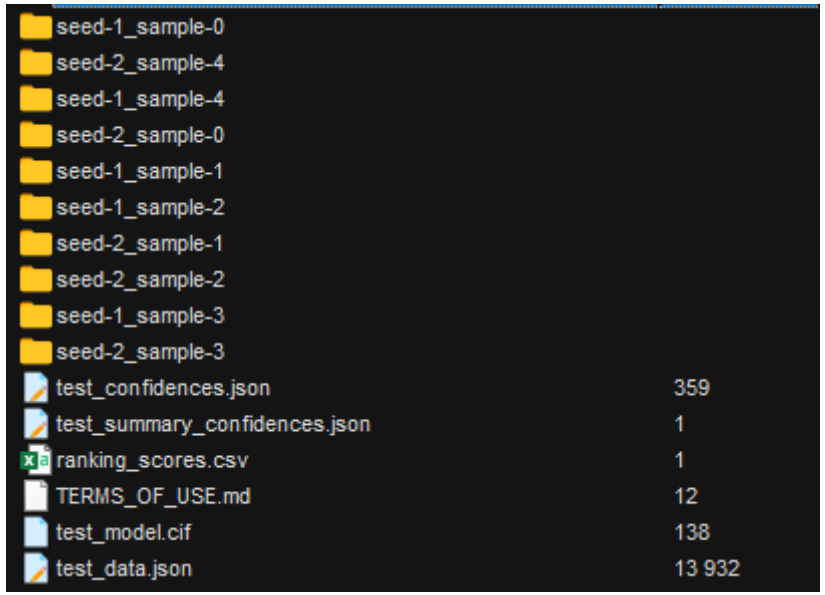
```
$ module load singularity      # 适用于HPC环境

$ singularity exec \      # 单次执行容器环境
    --nv \                # 启动gpu支持
    --bind $HOME/af3jobs/input:/root/af_input \                  # 挂载input文件夹至容器环境
    --bind $HOME/af3jobs/output:/root/af_output \                # 挂载output文件夹至容器环境
    --bind $HOME/af3_model_param:/root/models \                  # 挂载af3_model_param文件夹至容器环境
    --bind $HOME/public_databases:/root/public_databases \       # 挂载public_databases文件夹至容器环境
    $HOME/alphafold3.0.1.sif \                                   # singularity执行的容器环境，以下为在容器环境内运行
    python /app/alphafold/run_alphafold.py \                     # 运行AF3主程序，注意这里的路径是恒定的
    --json_path=/root/af_input/test.json \                       # 指定输入参数文件夹
    --model_dir=/root/models \                                   # 指定模型参数文件夹
    --db_dir=/root/public_databases \                            # 指定数据库文件夹
    --output_dir=/root/af_output                                 # 指定输出文件夹
```

✓ 对于 Nvidia A40 (40GB) 而言，使用两个seeds，205个氨基酸的单体蛋白质总共运行时间为 58s。

实战：结果分析

- **查看output文件夹中的内容：**



每个seeds会得到5个模型，这里我们使用了两个seeds，得到共10个模型

“***_confidences**”：保存了预测结果的.json文件，其中包含了pLDDT等信息，可以用于绘制氨基酸置信度图（省略）。

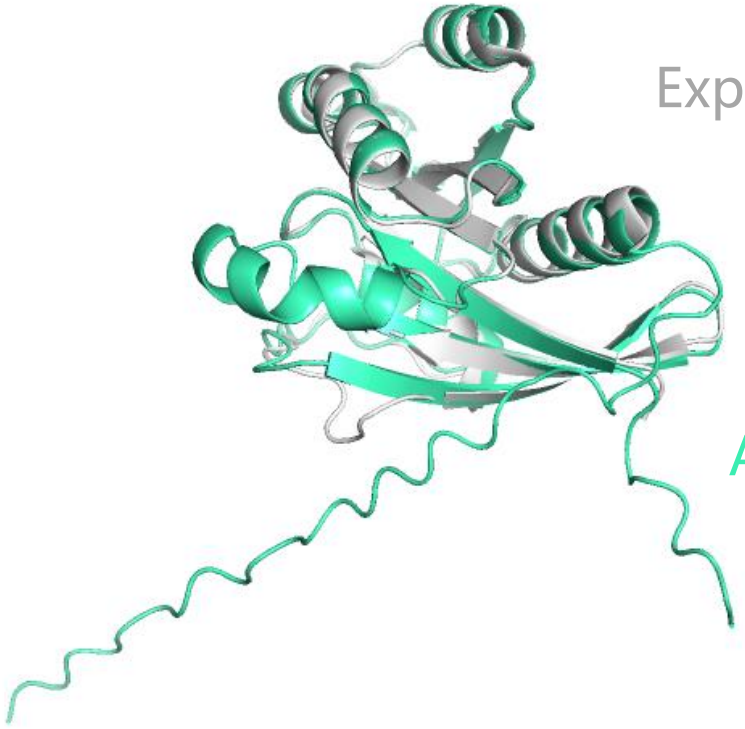
“**ranking_scores**”：对所有模型的置信度（根据pLDDT）进行排序，越高的打分代表置信度越高。

“***_model**”：置信度最高的模型

同时要注意，程序认为的置信度最高有时并不代表最接近实际的情况，可以尝试思考下为什么。

实战：结果分析

- **使用Pymol等软件可视化预测结果**



Experimental (PDB: 2FOL)

RMSD = 0.321 nm

AF3 predicted

实战：多条链的输入

- 一个分子，多个copy（同源二聚体）

```
{
  "name": "homodimer",
  "modelSeeds": [1],
  "sequences": [
  {
    "protein": {
    "id": ["A", "B"],
    "sequence":
"MSSMNPEYDYLFKLLLIGDSGVGKSCLLLRFADDTYTESYIS
TIGVDFKIRTIELDGKTIKLQIWDTAGQERFRTITSSYYRGAHG
IIVVYDVTDQESFNNVKQWLQEIDRYASENVNKLLVGNKCD
LTTKKVVDYTTAKEFADSLGIPFLETSAKNATNVEQSFMTM
AAEIKKRMGPGATAGGAEKSNVKIQSTPVKQSGGGCC"
    }
  }
  ],
  "dialect": "alphafold3",
  "version": 1
}
```

- 多个蛋白分子（预测相互作用）

```
{
  "name":  “multimer",
  "modelSeeds": [1],
  "sequences": [
  {
    "protein": {
    "id": "A",
    "sequence": "MSRATSWEGK"
    }
  },
  {
    "protein": {
    "id": "B",
    "sequence": "MASPTBTA"
    }
  }
  ],
  "dialect": "alphafold3",
  "version": 1
}
```

参考：https://github.com/google-deepmind/alphafold3/blob/main/docs/input.md

Hints, Tips, & FAQs

- **计算瓶颈在哪里？**

目前的计算瓶颈主要在于前期的MSA，过长的序列会使计算量骤增，虽然并行计算极大增加了MSA的速度，但目前半开源的AF3仍不支持多卡并行。

- **最大支持的序列长度？**

AF3的token是动态规划的，也就是说理论上可以没有限制。但过长的序列会大量占据显存，因此如果对于Nvidia A40 (40GB)，在单序列在6000-8000个token以下，多序列在5000个token以下，都是可以正常运行的，否则会报显存溢出的错误。

- **为什么与预期结果不一样？**

预测（inference）只是给出根据先验知识得到的最优化结果，模型永远只能预测它所接触过的知识，然后根据一定规则泛化。蛋白质结构复杂多样，三维折叠以及相互作用的规律是非线性的，因此预测能达到的效果会随着先验知识的增多而更收敛于真实情况。

随堂小测 & 课外学习

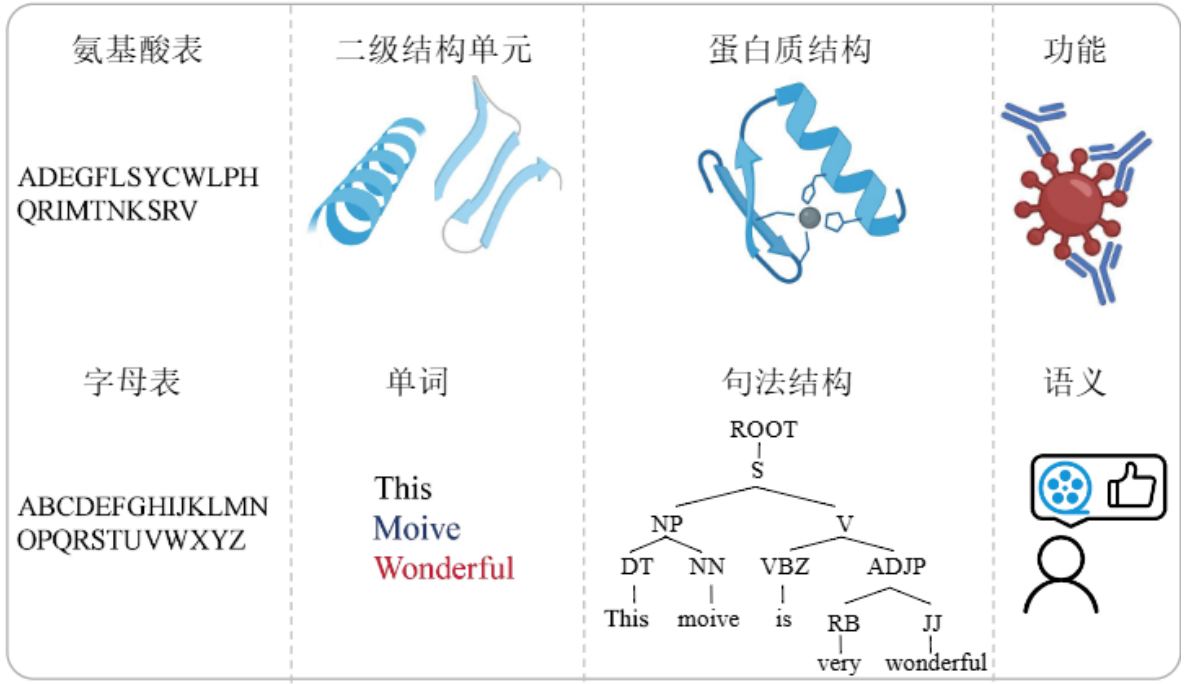
- **作业：**
  - 练习结构可视化软件的使用（Pymol、Chimera、VMD等）
  - 运行一个同源二聚体结构预测
  - 运行两个（或多个）蛋白质的相互作用复合体结构预测
  - 尝试“蛋白质-核酸”以及“蛋白质-小分子”预测体系

- **参考资料：**
  - **AF original：** Senior, A.W., Evans, R., Jumper, J. et al. Improved protein structure prediction using potentials from deep learning. Nature 577, 706–710 (2020). https://doi.org/10.1038/s41586-019-1923-7
  - **AF2：** Jumper, J., Evans, R., Pritzel, A. et al. Highly accurate protein structure prediction with AlphaFold. Nature 596, 583–589 (2021). https://doi.org/10.1038/s41586-021-03819-2
  - **AF3：** Abramson, J., Adler, J., Dunger, J. et al. Accurate structure prediction of biomolecular interactions with AlphaFold 3. Nature 630, 493–500 (2024). https://doi.org/10.1038/s41586-024-07487-w
  - **AF3官方文档：** https://github.com/google-deepmind/alphafold3/tree/main

蛋白质语言模型

Protein Language Models(PLMs)

从自然语言处理到蛋白质语言



以 ChatGPT 为代表的语言模型通过在海量文本上预训练，能够学习文本的统计规律，掌握基本的语法和上下文中单词的语义，能够进行高质量的文本理解和生成。

自然语言 (NLP)：字母 -> 单词-> 语法-> 语义

蛋白质语言：氨基酸 -> 序列-> 结构 -> 功能

PLM 能建模蛋白质序列的共进化信息，学习残基之间的相互依赖关系和进化约束， 就好比自然语言LM能够学习文本的语法一样。

模型原理

掩码 (Masked) 语言模型 (BERT)原理

1.从序列中随机选取部分氨基酸进行掩盖；

2.用特殊标记[mask]替换 ；

3.输入模型中，在上下文条件下预测被掩盖的氨基酸。

原始序列：

MKVSKILLERSEY

在构造训练数据时随机掩盖部分氨基酸后：

M[mask]VSKILLER[mask]EY

模型目标：预测[MASK]=‘ K ‘和‘ S ‘

模型原理

自回归(Autoregressive)语言模型(GPT)原理

1.给定一个序列的前缀，预测下一个氨基酸

2.目标是最小交叉熵损失：

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(x_t | x_{<t})$$

原始序列：

MKVSKILLERSEY

输入M->预测K，

输入MK->预测V，

……依此类推

按照顺序从左到右预测下一个氨基酸

模型Tokenizer

tokenizer的任务是把原始文本（自然语言字符序列）转换为模型可以处理的离散 token 序列，再进一步映射到向量空间。

输入：MKVSKILLERSEY

常见分词方法：

1. 基于字符（Character-level）
   每个字符是一个 token，例如："M" "K" "V" "S" …
   输出：[20,15,7,8,15,12,4,4,9,10,8,9,19]

2. 基于子词（Subword-level）
   利用统计方法把词拆成子词单元。
   如，BPE (Byte Pair Encoding)：不断合并高频字符对，得到子词单元。
   'MKVSKILLERSEY' 可能被拆分为"MKV" "SKILL" "ERSEY"
   输出：[4,5,6]

索引	字符	子词
0	<cls>	<cls>
1	<pad>	<pad>
2	<eos>	<eos>
3	<unk>	<unk>
4	L	MKV
5	A	SKILL
6	G	ERSEY
7	V	KVSI
8	S	ILLE
9	E	LERS
10	R	VSKI
…	…	…

模型Padding

token序列输入模型前，通常会 padding 到固定长度，比如 15。

input = M K V S K I L L E R S E Y

input_ids =[20,15,7,8,15,12,4,4,9,10,8,9,19,1,1]

attention_mask=[1 ,1 ,1,1,1 ,1 ,1,1,1,1 ,1,1,1 ,0,0]

attention_mask本质：告诉模型哪些位置需要计算，哪些是 padding。

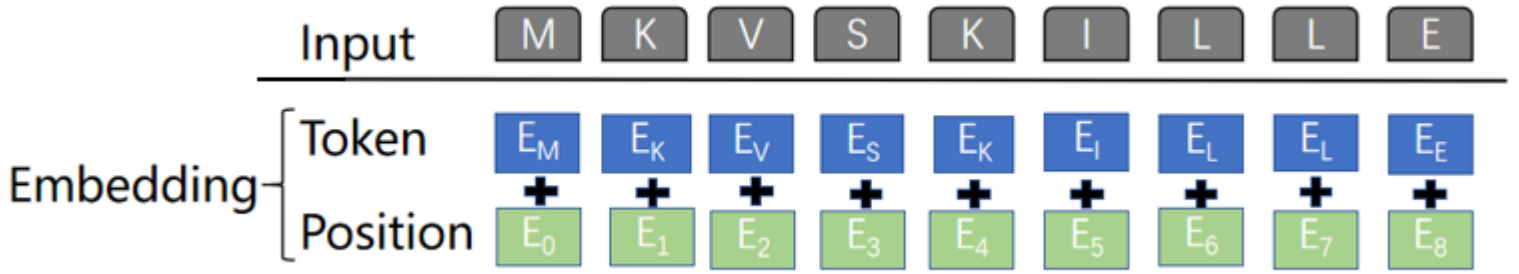
值通常是 1 或 0：

1 → 有效 token（需要注意力计算)

0 → padding（忽略）

模型embedding

将离散的token映射到连续的向量空间



[20,15,7,8,15,12,4,4,9,10,8,9,19,1,1]

	d1	d2	d3	d4	d5	…
20	0.31	0.44	0.35	0.41	0.38	…
15	0.28	0.52	0.37	0.37	0.35	…
7	0.26	0.24	0.34	0.53	0.24	…
…	…	…	…	…	…	…

token embedding矩阵

	d1	d2	d3	d4	d5	…
0	0.13	0.47	0.15	0.14	0.23	…
1	0.27	0.56	0.47	0.21	0.26	…
2	0.25	0.15	0.56	0.47	0.75	…
…	…	…	…	…	…	…

Transformer 不知道词序，需要额外加位置编码 Positional Embedding，训练时 embedding 会随着梯度更新，逐渐学到语义结构

模型输出

模型输出形状：[batch_size, seq_len, hidden_dim]

- batch_size：输入序列数量
- seq_len：序列长度（已padding）
- hidden_dim：token 的向量维度

```
[
  # 第1个样本 (batch 0)
  [
    [ 0.12,  0.98, -0.33],   # token 1
    [ 0.05, -0.20,  0.44],   # token 2
    [ 0.88,  0.10, -0.55],   # token 3
    [ 0.00,  0.00,  0.00],   # token 4
  ],

  # 第2个样本 (batch 1)
  [
    [ 0.22, -0.11,  0.77],   # token 1
    [-0.33,  0.66,  0.12],   # token 2
    [ 0.19,  0.05, -0.25],   # token 3
    [-0.04,  0.55,  0.09],   # token 4
  ]
]
```

模型损失

输出: [batch_size, seq_len, hidden_dim]

通过一个线性层把 hidden_dim → vocab_size：

logits: [batch_size, seq_len, vocab_size]

对每个位置的 logits 取 softmax，得到概率： $P(y_t|y_{<t}) = \text{Softmax}(\text{logits}_t)$

计算交叉熵损失： $\mathcal{L} = -\sum_{t=1}^{T} \log P(x_t|x_{<t})$

```
hidden_states[0, 0] = [0.12, -0.33, 0.87, ..., 0.05]   # 第一个样本第一个 token 的向量
logits[0, 0] = [ 2.3, -1.1, 0.7, ..., 4.5 ]
P[0,0] = [0.02, 0.0003, 0.005, ..., 0.12]   #每个氨基酸在该 token 的概率
```

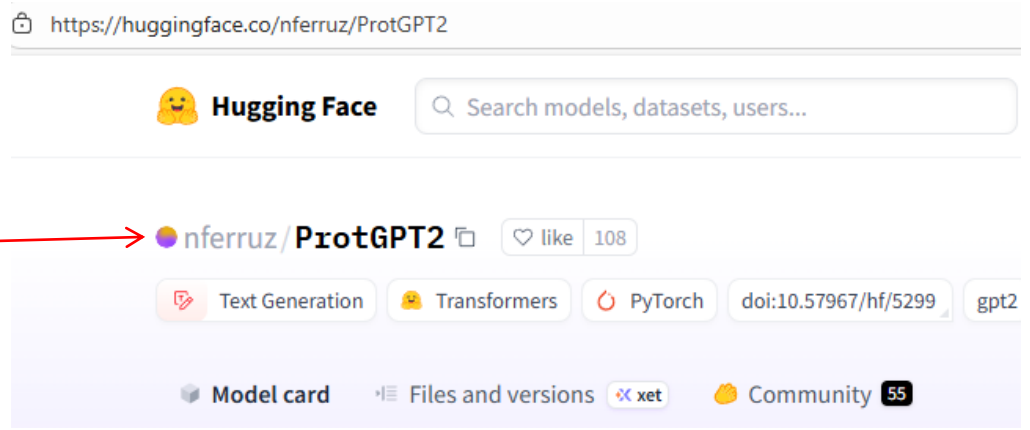
计算损失后，反向传播，更新参数

利用抗菌肽数据微调ProtGPT2模型

```python
import pandas as pd
import torch
from torch.utils.data import Dataset
from transformers import GPT2LMHeadModel, GPT2Tokenizer, Trainer, TrainingArguments
from sklearn.model_selection import train_test_split
import math

# 1. 配置
DATA_FILE = "D:/Downloads/amp_data.csv"
MODEL_NAME = "nferruz/ProtGPT2"
OUTPUT_DIR = "./AMPGPT"
MAX_LENGTH = 15
BATCH_SIZE = 16
EPOCHS = 3
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
N_FREEZE = 34
```

镜像网站https://hf-mirror.com



从huggingface中加载模型

MAX_LENGTH：输入序列的最大长度，超过会截断，不足会填充

BATCH_SIZE: 每次训练使用的样本数量，影响显存占用和梯度稳定性

EPOCHS: 数据集完整迭代次数

冻结模型层数

```python
# 2. 数据集
class AMPDataset(Dataset):
    def __init__(self, sequences, tokenizer, max_length=15):
        self.sequences = sequences
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.sequences)

    def __getitem__(self, idx):
        seq = self.sequences[idx]
        tokenized = self.tokenizer(
            seq,
            max_length=self.max_length,
            padding="max_length",
            truncation=True,
            return_tensors="pt"
        )
        input_ids = tokenized["input_ids"].squeeze()
        attention_mask = tokenized["attention_mask"].squeeze()
        labels = input_ids.clone()
        labels[labels == tokenizer.pad_token_id] = -100
        return {"input_ids": input_ids, "attention_mask": attention_mask, "labels": labels}
```

输入参数

返回样本数量

labels 是模型在训练时要预测的目标，padding token 在计算 loss 时不希望影响梯度，需要设置为 -100，CrossEntropyLoss 会忽略 -100 的位置

truncation=True是当序列长度超过 max_length 时自动截断
return_tensors="pt" 会返回 [1, max_length] 的张量，所以用 .squeeze() 去掉第一维，得到 [max_length]

```python
# 3. 读取数据
df = pd.read_csv(DATA_FILE)
sequences = df['Sequence'].tolist()

# 4. Tokenizer & Model
tokenizer = GPT2Tokenizer.from_pretrained(MODEL_NAME)
if tokenizer.pad_token is None:
    tokenizer.add_special_tokens({'pad_token': '<pad>'})
model = GPT2LMHeadModel.from_pretrained(MODEL_NAME)
model.resize_token_embeddings(len(tokenizer))
model.to(DEVICE)
```

GPT2 tokenizer 默认没有 pad_token。
添加 pad_token 后，要 resize 模型 embedding，否则训练时会报错

resize_token_embeddings 会把新加入的 pad_token 加入 embedding 层

```python
# 5. 冻结部分层
for i, block in enumerate(model.transformer.h):
    if i < N_FREEZE:
        for param in block.parameters():
            param.requires_grad = False

# LM head训练
for param in model.lm_head.parameters():
    param.requires_grad = True
```

model.transformer.h 是ProtGPT2 的所有transformer 层。
i < N_FREEZE：表示前 N_FREEZE 层不参与梯度更新。

lm_head 是输出线性层，用于把 hidden states 映射回 token vocabulary。

```python
# 6. 数据集拆分
train_seqs, eval_seqs = train_test_split( *arrays: sequences, test_size=0.25, random_state=42)
train_dataset = AMPDataset(train_seqs, tokenizer, MAX_LENGTH)
eval_dataset = AMPDataset(eval_seqs, tokenizer, MAX_LENGTH)

# 7. 指标函数（计算困惑度）
def compute_metrics(eval_pred):
    loss = eval_pred.metrics["eval_loss"]
    perplexity = math.exp(loss)
    return {"perplexity": perplexity}
```

```python
print(train_dataset[0])
```

```
{'input_ids': tensor([  706,   380,   610,   292,   863,   416,  3002,   351, 50257, 50257,
        50257, 50257, 50257, 50257, 50257, 50257]),
 'attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]),
 'labels': tensor([  706,   380,   610,   292,   863,   416,  3002,   351, 50257, 50257,50257, 50257, 50257, 50257, 50257, 50257])}
```

Trainer 在验证集上评估模型时，计算困惑度(Perplexity)
对自回归语言模型来说，Perplexity 越低，模型预测能力越好

$$\text{Perplexity} = e^{\text{loss}}$$

```python
# 8. 训练配置
training_args = TrainingArguments(
    output_dir=OUTPUT_DIR,
    overwrite_output_dir=True,
    num_train_epochs=EPOCHS,
    per_device_train_batch_size=BATCH_SIZE,
    per_device_eval_batch_size=BATCH_SIZE,
    save_steps=100,
    save_total_limit=2,
    logging_steps=100,
    eval_strategy="steps",
    eval_steps=100,
    learning_rate=1e-5,
    warmup_steps=100,
    weight_decay=0.01,
    load_best_model_at_end=True,
    metric_for_best_model="perplexity"
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics
)

# 9. 微调训练
trainer.train()
model.save_pretrained(OUTPUT_DIR)
tokenizer.save_pretrained(OUTPUT_DIR)
```

learning_rate=1e-5　　初始学习率
warmup_steps=100　　线性学习率预热步数
weight_decay=0.01　　权重衰减（L2 正则）
load_best_model_at_end=True　　保存最优模型

warmup_steps作用：训练初期做线性学习率预热。前100 步，学习率从 0 线性增加到 learning_rate，之后再按照原策略衰减。稳定训练，避免刚开始梯度过大导致 loss 发散。

weight_decay作用：对权重做 L2 正则化，避免过拟合。

使用微调后的模型进行推理

```python
import torch
from transformers import GPT2LMHeadModel, GPT2Tokenizer
# 配置
MODEL_PATH = "./AMPGPT"  # 微调后模型路径
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
GENERATE_MAX_LENGTH = 15 # 最大序列长度
# 加载模型与 tokenizer
tokenizer = GPT2Tokenizer.from_pretrained(MODEL_PATH)
model = GPT2LMHeadModel.from_pretrained(MODEL_PATH)
model.to(DEVICE)
model.eval()
#prompt
prompts = ["M"]
inputs = tokenizer(prompts, return_tensors="pt", padding=True).to(DEVICE)
# 生成序列
outputs = model.generate(
    **inputs,
    max_length=GENERATE_MAX_LENGTH,
    do_sample=True,    # 启用采样生成多样性
    top_k=50,          # top-k 采样
    top_p=0.95,        # nucleus 采样
    temperature=1.0,
    num_return_sequences=1,
    pad_token_id=tokenizer.pad_token_id,
    eos_token_id=None
)
# 解码输出
generated_seqs = tokenizer.batch_decode(outputs, skip_special_tokens=True)
# 打印生成结果
print("generated_seq:", generated_seqs)
```

加载微调后的 ProtGPT2 模型和词表，用 "M" 作为起始氨基酸序列

```
generated_seq: ['MALSILKGLEKGG\n']
```

利用抗菌肽与非抗菌肽数据微调ESM2模型

```python
import pandas as pd
import torch
import os
from torch.utils.data import Dataset
from transformers import EsmForSequenceClassification, EsmTokenizer, Trainer, TrainingArguments
import numpy as np
from sklearn.metrics import accuracy_score, f1_score
from sklearn.model_selection import train_test_split

# 1 数据准备：合并 AMP 与非 AMP

def load_amp_data(amp_file, non_amp_file):
    df_amp = pd.read_csv(amp_file)
    df_amp['label'] = 1  # AMP
    df_non = pd.read_csv(non_amp_file)
    df_non['label'] = 0  # 非 AMP
    df = pd.concat( objs: [df_amp, df_non], ignore_index=True)
    df = df.sample(frac=1).reset_index(drop=True)
    return df
```

	Sequence	label
0	KKKLFWKIPKFLHLAKKF	1
1	FLGKVFKGASKVFGAVFGKV	1
2	SENTGAIGKVFPRGNHWAVGHLM	0
3	PFVYLI	0
4	FFHHIFRGIVHVGRTIHRLVTGG	1

合并数据集，label=1说明是抗菌肽

```python
# 2 Dataset 定义
class AMPDataset(Dataset):

    def __init__(self, dataframe, tokenizer, max_length=40):
        self.data = dataframe
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        seq = self.data.iloc[idx]['Sequence']
        label = int(self.data.iloc[idx]['label'])

        encoding = self.tokenizer(seq,
                                  padding='max_length',
                                  truncation=True,
                                  max_length=self.max_length,
                                  return_tensors="pt")
        item = {key: val.squeeze(0) for key, val in encoding.items()}
        item['labels'] = torch.tensor(label, dtype=torch.long)
        return item
```

labels → 分类标签（注意和语言建模时的 labels = input_ids 不一样）

```python
# 3 加载模型与 tokenizer
OUTPUT_DIR = "./esm2_amp"
model_name = "facebook/esm2_t12_35M_UR50D"
tokenizer = EsmTokenizer.from_pretrained(model_name)
model = EsmForSequenceClassification.from_pretrained(model_name, num_labels=2)

# 训练 ESM2
for param in model.esm.parameters():
    param.requires_grad = True

#训练分类头
for param in model.classifier.parameters():
    param.requires_grad = True

# 4 加载数据并拆分训练/验证
df = load_amp_data( amp_file: "D:/Downloads/amp_data.csv",  non_amp_file: "D:/Downloads/nonamp_data.csv")
train_seqs, eval_seqs = train_test_split( *arrays: df, test_size=0.25, random_state=42)
train_dataset = AMPDataset(train_seqs, tokenizer)
eval_dataset = AMPDataset(eval_seqs, tokenizer)
```

Checkpoint name	Num layers	Num parameters
esm2_t48_15B_UR50D	48	15B
esm2_t36_3B_UR50D	36	3B
esm2_t33_650M_UR50D	33	650M
esm2_t30_150M_UR50D	30	150M
esm2_t12_35M_UR50D	12	35M
esm2_t6_8M_UR50D	6	8M

模型参数较少，未进行冻结

```python
print(train_dataset[0])
```

```
{'input_ids': tensor([ 0, 12, 15,  4,  8, 14, 15, 11, 15, 15, 17,  4, 15, 15,  7,  4, 15,  6,
         5, 12, 15,  6,  5, 12,  5,  7,  5, 15, 20,  7,  2,  1,  1,  1,  1,  1,
         1,  1,  1,  1]), 'attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]), 'labels': tensor(1)}
```

```python
# 5 训练参数
training_args = TrainingArguments(
    output_dir=OUTPUT_DIR,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    num_train_epochs=3,
    learning_rate=1e-5,
    eval_strategy="epoch",
    save_strategy="epoch",
    logging_dir="./logs",
    logging_steps=10,
    save_total_limit=2,
    load_best_model_at_end=True,
    metric_for_best_model="accuracy",
)

# 6 Metric
def compute_metrics(pred):
    labels = pred.label_ids
    preds = np.argmax(pred.predictions, axis=1)
    acc = accuracy_score(labels, preds)
    f1 = f1_score(labels, preds)
    return {"accuracy": acc, "f1": f1}
```

```python
# 7 Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    compute_metrics=compute_metrics,
)

# 8 训练
trainer.train()

if not os.path.exists(OUTPUT_DIR):
    os.makedirs(OUTPUT_DIR)

model.save_pretrained(OUTPUT_DIR)
tokenizer.save_pretrained(OUTPUT_DIR)
```

pred.predictions → 模型的输出 logits，形状 [batch_size, num_labels]
np.argmax(..., axis=1) → 取最大概率对应的类别

```
[0.8, -0.2],  # 样本1
[-0.5, 1.3],  # 样本2
[0.1, -0.1],  # 样本3
[-1.2, 2.0]  # 样本4
```

```python
import torch
from transformers import EsmTokenizer, EsmForSequenceClassification
#1 配置
MODEL_DIR = "./esm2_amp"
MAX_LEN = 40
#2 加载模型和分词器
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
tokenizer = EsmTokenizer.from_pretrained(MODEL_DIR)
model = EsmForSequenceClassification.from_pretrained(MODEL_DIR).to(device)
model.eval()
#3 推理函数
def predict_sequences(sequences):
    inputs = tokenizer(
        sequences,
        padding=True,
        truncation=True,
        max_length=MAX_LEN,
        return_tensors="pt"
    ).to(device)

    with torch.no_grad():
        outputs = model(**inputs)
        probs = torch.softmax(outputs.logits, dim=-1)
        preds = torch.argmax(probs, dim=-1)
    results = []
    for seq, pred, prob in zip(sequences, preds, probs):
        label = "AMP" if pred.item() == 1 else "Non-AMP"
        confidence = prob[pred].item()
        results.append({
            "sequence": seq,
            "pred_id": pred.item(),
            "pred_label": label,
            "confidence": round(confidence, 4)
        })
    return results
#4 示例调用
if __name__ == "__main__":
    test_sequences = [
        "ALSILKGLEKLAKMGIALTNCKATKKC",
        "KAELCMSADKGALI"
    ]
    preds = predict_sequences(test_sequences)
    for p in preds:
        print(p)
```

outputs.logits形状 [batch_size, num_labels]，是 Trainer 里 pred.predictions 的来源
torch.softmax(..., dim=-1)把 logits 转换成概率分布，所有类别的概率加起来 = 1
torch.argmax(..., dim=-1)取最大概率对应的索引 = 预测的类别 ID

```
{'sequence': 'ALSILKGLEKLAKMGIALTNCKATKKC', 'pred_id': 1, 'pred_label': 'AMP', 'confidence': 0.9056}
{'sequence': 'KAELCMSADKGALI', 'pred_id': 0, 'pred_label': 'Non-AMP', 'confidence': 0.9149}
```

随堂小测 & 课外学习

参考资料

完全图解GPT-2- 知乎

读懂BERT - 知乎

huggingface transformers文档