README

本模型使用时需输入小分子的SMILES结构式，获取其所预测的RNA适配体输出
由于核糖开关适配体数据量有限，故将DNA原件全部转为RNA进行探索预测
模型采用类Transformer架构，将小分子与aptamer序列视为翻译任务
详细代码见 **RNA_smiles (1).ipynb** 文件

调用模型的代码如下，可进一步封装为api

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from tokenizers import Tokenizer

# --------------------------
# 1. 重新定义模型（确保嵌入层维度=256）
# --------------------------
class Mol2Aptamer(nn.Module):
    def __init__(self, smiles_vocab_size, rna_vocab_size, d_model=256, nhead=8, num_encoder_layers=2, num_decoder_layers=3):
        super().__init__()
        self.d_model = d_model  # 显式保存模型维度，便于验证

        # Encoder: 2层 TransformerEncoder（d_model=256）
        self.encoder = nn.TransformerEncoder(
            encoder_layer=nn.TransformerEncoderLayer(
                d_model=d_model,        # 注意力层期望的维度=256
                nhead=nhead,
                dim_feedforward=2048,
                batch_first=True
            ),
            num_layers=num_encoder_layers
        )

        # Decoder: 3层 TransformerDecoder（d_model=256）
        self.decoder = nn.TransformerDecoder(
            decoder_layer=nn.TransformerDecoderLayer(
                d_model=d_model,
                nhead=nhead,
                dim_feedforward=2048,
                batch_first=True
            ),
            num_layers=num_decoder_layers
```

```
        )

        # 嵌入层：明确设置输出维度=d_model=256
        self.smiles_embedding = nn.Embedding(smiles_vocab_size, d_model) # 输出 (seq_len, 256)
        self.rna_embedding = nn.Embedding(rna_vocab_size, d_model)
        self.pos_embedding = nn.Embedding(512, d_model)  # 位置编码维度也=256

        # 输出层
        self.fc_out = nn.Linear(d_model, rna_vocab_size)

    def forward(self, smiles_ids, rna_inp):
        batch_size, seq_len_smi = smiles_ids.shape
        batch_size, seq_len_rna = rna_inp.shape
        device = smiles_ids.device

        # --------------------------
        # 关键：验证嵌入层输出维度（确保=256）
        # --------------------------
        smiles_emb = self.smiles_embedding(smiles_ids)
        assert smiles_emb.shape[-1] == self.d_model, \
            f"嵌入层输出维度错误：期望 {self.d_model}，实际 {smiles_emb.shape[-1]}"

        # 叠加位置编码（维度需与嵌入层一致）
        smiles_pos = torch.arange(seq_len_smi, device=device).unsqueeze(0).repeat(batch_size, 1)
        smiles_emb += self.pos_embedding(smiles_pos)
        assert smiles_emb.shape[-1] == self.d_model, \
            f"位置编码后维度错误：期望 {self.d_model}，实际 {smiles_emb.shape[-1]}"

        # Encoder前向（输入维度=256，匹配注意力层期望）
        memory = self.encoder(smiles_emb)

        # RNA嵌入与位置编码（同样验证维度）
        rna_emb = self.rna_embedding(rna_inp)
        assert rna_emb.shape[-1] == self.d_model, \
            f"RNA嵌入层输出维度错误：期望 {self.d_model}，实际 {rna_emb.shape[-1]}"
        rna_pos = torch.arange(seq_len_rna, device=device).unsqueeze(0).repeat(batch_size, 1)
        rna_emb += self.pos_embedding(rna_pos)
```

```
71
72          # Decoder前向
73          tgt_mask = nn.Transformer.generate_square_subsequent_mask(seq_l
    en_rna, device=device)
74          decoder_out = self.decoder(tgt=rna_emb, memory=memory, tgt_mask
    =tgt_mask)
75
76          # 输出层
77          logits = self.fc_out(decoder_out)
78          return logits
79
80  # --------------------------
81  # 2. 加载分词器 + 初始化模型（显式确认嵌入层维度）
82  # --------------------------
83  smiles_tokenizer = Tokenizer.from_file("/root/autodl-tmp/rna/smiles_tok
    enizer.json")
84  rna_tokenizer = Tokenizer.from_file("/root/autodl-tmp/rna/rna_tokenize
    r.json")
85
86  # 词汇表大小
87  smiles_vocab_size = len(smiles_tokenizer.get_vocab())
88  rna_vocab_size = len(rna_tokenizer.get_vocab())
89
90  # 设备与模型初始化（d_model=256，嵌入层输出维度=256）
91  device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
92  model = Mol2Aptamer(
93      smiles_vocab_size=smiles_vocab_size,
94      rna_vocab_size=rna_vocab_size,
95      d_model=256,           # 强制嵌入层输出维度=256
96      nhead=8,
97      num_encoder_layers=2,
98      num_decoder_layers=3
99  ).to(device)
100
101 # --------------------------
102 # 3. 修复权重加载：确保嵌入层权重被正确加载
103 # --------------------------
104 # 加载原始权重
105 state_dict = torch.load("/root/autodl-tmp/model_epoch_59.pth", map_loca
    tion=device)
106
107 # 查看所有权重键，确认嵌入层键是否存在（如 "smiles_embedding.weight"）
108 print("原始权重中的嵌入层相关键：")
109 for key in state_dict.keys():
```

```
        if "embedding" in key:
            print(f"- {key}: 形状 {state_dict[key].shape}")
    print("-" * 50)

    # 过滤权重：保留所有与当前模型匹配的键（包括嵌入层、位置编码、输出层）
    filtered_state_dict = {}
    model_keys = set(model.state_dict().keys())  # 当前模型需要的键
    for key, value in state_dict.items():
        if key in model_keys:
            # 额外验证嵌入层权重维度是否正确（如 smiles_embedding.weight 形状应为 (vocab_size, 256)）
            if "embedding.weight" in key:
                assert value.shape[-1] == model.d_model, \
                    f"{key} 维度错误：期望 {model.d_model}，实际 {value.shape[-1]}"
            filtered_state_dict[key] = value
        else:
            print(f"跳过冗余键：{key}")

    # 加载过滤后的权重
    model.load_state_dict(filtered_state_dict, strict=False)  # strict=False：允许模型有未加载的键（如无）
    model.eval()
    print("权重加载成功！")

    # --------------------------
    # 4. 修复生成函数：确保SMILES编码后维度正确
    # --------------------------
    def generate_aptamers(
        model, smiles, smiles_tokenizer, rna_tokenizer,
        max_len=80, num_return=5,
        strategy="topk", top_k=10, top_p=0.9, temperature=0.8,
        device=None
    ):
        if device is None:
            device = next(model.parameters()).device

        model.eval()
        model.to(device)

        # 验证特殊token
        required_tokens = ["<pad>", "<bos>", "<eos>", "<unk>"]
        for token in required_tokens:
            token_id = rna_tokenizer.token_to_id(token)
```

```
        if token_id is None:
            raise ValueError(f"RNA tokenizer missing required token: {token}")

    bos_id = rna_tokenizer.token_to_id("<bos>")
    eos_id = rna_tokenizer.token_to_id("<eos>")
    pad_id = rna_tokenizer.token_to_id("<pad>")
    unk_id = rna_tokenizer.token_to_id("<unk>")
    smiles_pad_id = smiles_tokenizer.token_to_id("<pad>")
    if smiles_pad_id is None:
        raise ValueError("SMILES tokenizer missing <pad> token")

    # --------------------------
    # 修复SMILES编码：确保输入到嵌入层的张量格式正确
    # --------------------------
    try:
        smi_encoded = smiles_tokenizer.encode(smiles)
        max_smi_len = 128  # 与训练时一致
        # 补全/截断SMILES到max_smi_len
        smi_ids = smi_encoded.ids[:max_smi_len]
        smi_ids += [smiles_pad_id] * (max_smi_len - len(smi_ids))
        # 转换为张量：(batch_size=1, seq_len=128)
        smi_ids = torch.tensor(smi_ids, dtype=torch.long).unsqueeze(0).to(device)
        assert smi_ids.shape == (1, max_smi_len), \
            f"SMILES张量形状错误：期望 (1, {max_smi_len})，实际 {smi_ids.shape}"
    except Exception as e:
        raise ValueError(f"Failed to encode SMILES: {str(e)}")

    results = []
    with torch.no_grad():
        # --------------------------
        # 验证Encoder输入维度（嵌入层输出应为256）
        # --------------------------
        smiles_emb = model.smiles_embedding(smi_ids)
        assert smiles_emb.shape == (1, max_smi_len, model.d_model), \
            f"SMILES嵌入后形状错误：期望 (1, {max_smi_len}, {model.d_model})，实际 {smiles_emb.shape}"

        # 预计算Encoder输出
        memory = model.encoder(smiles_emb)

        for _ in range(num_return):
```

```
            generated = [bos_id]
            has_unk = False

            for _ in range(max_len - 1):
                # RNA输入张量: (1, current_len)
                rna_inp = torch.tensor([generated], dtype=torch.long).to(device)

                # 模型前向传播
                logits = model(smi_ids, rna_inp)[:, -1, :]  # (1, rna_vocab_size)

                # 温度调节
                logits = logits / temperature

                # 采样策略
                if strategy == "greedy":
                    next_id = torch.argmax(logits, dim=-1).item()
                elif strategy == "topk":
                    topk_probs, topk_ids = torch.topk(logits, k=top_k)
                    topk_probs = F.softmax(topk_probs, dim=-1)
                    idx = torch.multinomial(topk_probs, 1).item()
                    next_id = topk_ids[0, idx].item()
                elif strategy == "topp":
                    sorted_logits, sorted_ids = torch.sort(logits, descending=True)
                    sorted_probs = F.softmax(sorted_logits, dim=-1)
                    cum_probs = torch.cumsum(sorted_probs, dim=-1)
                    cutoff = max(1, torch.sum(cum_probs <= top_p).item())
                    filtered_probs = sorted_probs[:, :cutoff]
                    filtered_ids = sorted_ids[:, :cutoff]
                    idx = torch.multinomial(filtered_probs, 1).item()
                    next_id = filtered_ids[0, idx].item()
                else:
                    raise ValueError(f"Unknown strategy: {strategy}")

                # 终止条件
                if next_id == unk_id:
                    has_unk = True
                if next_id == eos_id:
                    break
                generated.append(next_id)

```

```
            # 后处理
            filtered_ids = [id for id in generated if id not in [bos_id, eos_id, pad_id, unk_id]]
            try:
                seq = rna_tokenizer.decode(filtered_ids, skip_special_tokens=True)
                if seq and not has_unk and seq not in results:
                    results.append(seq)
            except Exception as e:
                print(f"Warning: Failed to decode sequence: {str(e)}")

    # 补全候选数量
    while len(results) < num_return:
        results.append(results[-1] if results else "")
    return results[:num_return]

# --------------------------
# 5. 测试生成（成功运行）
# --------------------------
if __name__ == "__main__":
    # 输入SMILES（苯酚）
    smiles = "C1=CC=C(C=C1)O"

    # 生成前先验证模型嵌入层维度
    print(f"模型嵌入层输出维度：{model.d_model}")
    print(f"SMILES嵌入层权重形状：{model.smiles_embedding.weight.shape}")
    print(f"RNA嵌入层权重形状：{model.rna_embedding.weight.shape}")

    # 生成Aptamer
    candidates = generate_aptamers(
        model=model,
        smiles=smiles,
        smiles_tokenizer=smiles_tokenizer,
        rna_tokenizer=rna_tokenizer,
        max_len=80,
        num_return=5,
        strategy="topk",
        top_k=10,
        temperature=0.8,
        device=device
    )

    # 打印结果
```

```
    print("\n候选Aptamer序列：")
    for i, seq in enumerate(candidates, 1):
        print(f"{i}. {seq}")
```

输出：

```
原始权重中的嵌入层相关键：
- smiles_embedding.weight: 形状 torch.Size([188, 256])
- rna_embedding.weight: 形状 torch.Size([100, 256])
- pos_embedding.weight: 形状 torch.Size([512, 256])
--------------------------------------------------
跳过冗余键：decoder_layer.self_attn.in_proj_weight
跳过冗余键：decoder_layer.self_attn.in_proj_bias
跳过冗余键：decoder_layer.self_attn.out_proj.weight
跳过冗余键：decoder_layer.self_attn.out_proj.bias
跳过冗余键：decoder_layer.multihead_attn.in_proj_weight
跳过冗余键：decoder_layer.multihead_attn.in_proj_bias
跳过冗余键：decoder_layer.multihead_attn.out_proj.weight
跳过冗余键：decoder_layer.multihead_attn.out_proj.bias
跳过冗余键：decoder_layer.linear1.weight
跳过冗余键：decoder_layer.linear1.bias
跳过冗余键：decoder_layer.linear2.weight
跳过冗余键：decoder_layer.linear2.bias
跳过冗余键：decoder_layer.norm1.weight
跳过冗余键：decoder_layer.norm1.bias
跳过冗余键：decoder_layer.norm2.weight
跳过冗余键：decoder_layer.norm2.bias
跳过冗余键：decoder_layer.norm3.weight
跳过冗余键：decoder_layer.norm3.bias
权重加载成功!
模型嵌入层输出维度：256
SMILES嵌入层权重形状：torch.Size([188, 256])
RNA嵌入层权重形状：torch.Size([100, 256])

候选Aptamer序列：
1. Ġ GGU AAU AC GCA GA CGU GAGG GAU GCA CU CGG AU GCGU AGG GG GUU GAU C
A
2. Ġ A GCA GCA CA GA GGU CAU CUU GAU CU CGG CUU GAU AGG GU CGU CC GUAA
 CU CC C
3. Ġ CC GG AA CU A CUU CA CGU AC GACU GU CACA CCU GAGG GGU CGU AA CAA G
U GGU AU GCGU
4. Ġ CC GG GGU GG AA AC GAU CU CAU CC GG GUU GU
5. Ġ A GCA GCA CA GA GGU CA GAU GCA CU CGG ACC CC AUU CU CC UU CC AUCC
```

```
 CU CAU CC GUCC ACC CU AU
```

可进一步更改生成参数调控输出:

```python
# 自定义生成参数（适合调整多样性、长度等）
aptamers = generate_aptamers(
    model=model,
    smiles="CC(=O)OC1=CC=CC=C1C(=O)O",
    smiles_tokenizer=smiles_tokenizer,
    rna_tokenizer=rna_tokenizer,
    max_len=100,                # 最长生成100个token
    num_return=3,               # 返回3个候选序列
    strategy="topp",            # 使用top-p采样（核采样）
    top_p=0.95,                 # 累积概率阈值0.95
    temperature=0.8,            # 降低多样性（值越小越确定）
)

print("生成的Aptamer序列：")
for i, seq in enumerate(aptamers):
    print(f"候选{i+1}: {seq}")
```

之后调用RNAfold包计算适配体序列稳定性并排序获得最终输出，后续可接入3D可视化工具，并进行分子动力学验证

```python
#过滤函数（RNAfold计算ΔG）
import RNA

def filter_by_rnafold(sequences, min_length=20, max_length=80, max_homopolymer=6, max_candidates=5):
    """
    sequences: list of str
    返回过滤+排序后的序列
    """
    results = []
    for seq in sequences:
        # 长度限制
        if len(seq) < min_length or len(seq) > max_length:
            continue
        # 去掉长同聚核苷酸 (AAAAAAA)
        if any(base*max_homopolymer in seq for base in "ACGU"):
            continue
        # 用 RNAfold 预测 ΔG
        structure, mfe = RNA.fold(seq)
```

```
            results.append((seq, mfe))

    # 按 ΔG 从低到高排序（越低越稳定）
    results = sorted(results, key=lambda x: x[1])
    return results[:max_candidates]

def generate_and_filter(
    model, smiles, smiles_tokenizer, rna_tokenizer,
    num_generate=50, return_top=5,
    strategy="topk", top_k=10, top_p=0.9, temperature=0.8
):
    # Step1: 生成候选
    candidates = generate_aptamers(
        model, smiles, smiles_tokenizer, rna_tokenizer,
        max_len=80, num_return=num_generate,
        strategy=strategy, top_k=top_k, top_p=top_p, temperature=temperature,
        device=device
    )

    # Step2: 过滤 & 打分
    filtered = filter_by_rnafold(candidates, max_candidates=return_top)

    return filtered

smiles = "C1=CC=C(C=C1)O"  # phenol

top_candidates = generate_and_filter(
    model, smiles, smiles_tokenizer, rna_tokenizer,
    num_generate=100, return_top=5,
    strategy="topp", top_p=0.9, temperature=0.7
)

print("Top 适配体候选（按ΔG排序）: ")
for seq, mfe in top_candidates:
    print(f"{seq}   ΔG={mfe:.2f}")

```

输出

```
Top 适配体候选（按ΔG排序）:
Ġ CGA GAGG AGU GGU GG GGU CA GAU GCA CU CGG ACC CC AUU CU CC C   ΔG=-4.10
Ġ CA AUGG CC ACC CC GG GGU GG GCGC GAA AGU GGU   ΔG=-3.10
```

```
4 Ġ CU CU CGG GA CGA CC CA CGU CC GGGU GG CUU GAU AGG GG GGU GGU CC AUCC
   CU CC   ΔG=-2.20
5 Ġ CC GGU ACA CA GG AGG CU GGU GCGC GGU GAA GU GCC GA GU CGU AA C   ΔG=-
  2.10
6 Ġ CGU AC GACU CA GG GCC A GAGG GAU CGG GU GGU CGU GGU CCU GAU GCA AUCU
   CU CC C   ΔG=-1.80
```