仓库名称

基于改进 Transformer 的锂离子电池健康状态预测

Improved Transformer-based Lithium-ion Battery State of Health Prediction

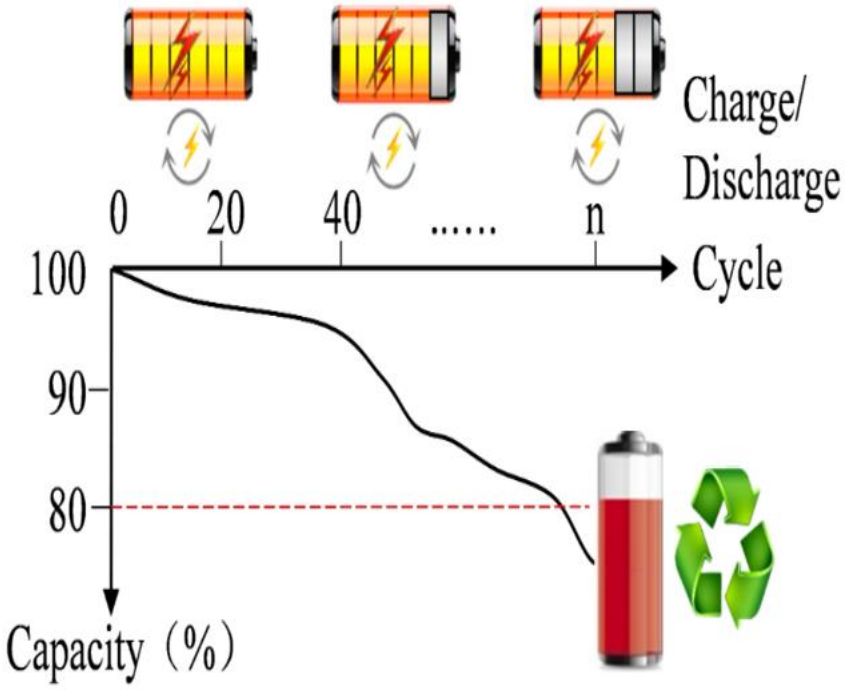
摘要：

本文在深度学习和数据分析方面的创新主要体现在特征提取与融合、多尺度特征处理和高维特征建模。提出的 **ADTC-Transformer** 架构结合了自适应扩张时序卷积（ADTC）与 Transformer 编码器，能够有效捕捉局部与全局特征，通过加权融合机制提升 SOH 预测的精度与鲁棒性。同时，将**特征金字塔网络（FPN）与 U 型网络（Unet）结合，形成 FUnet 模块**，增强了多尺度特征融合。最后，引入 Kolmogorov-Arnold 网络（KAN）作为预测模块，有效处理高维特征，提升了预测性能。实验表明，所提方法在 NASA、CALCE 和 WRBD 数据集上显著提高了预测准确性，特别是在长期 SOH 预测中表现突出，为电池健康管理与性能优化提供了强有力支持。

1. 目的

Transformer 基准模型在锂电池方向的 python 相关复现代码仍然较少。为此，我整理并上传了一篇基于改进 Transformer 的锂离子电池 SOH 预测工作。由于实验阶段的源代码较为杂乱，此处提供的是重新整理后的精简版本，部分实现可能与原实验存在不完全对应之处，还请谅解。

2. 背景

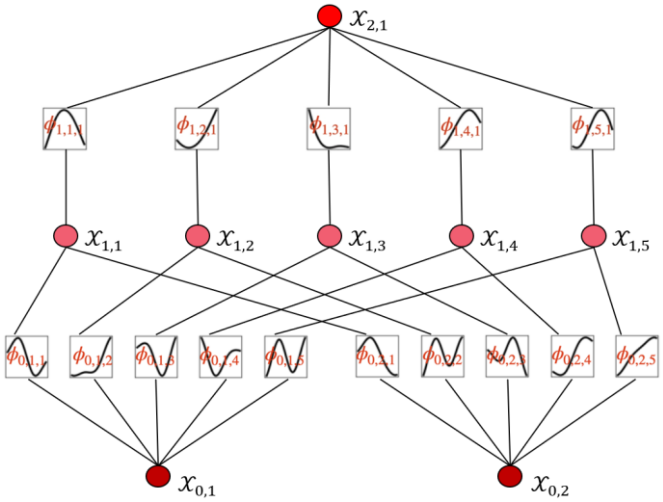


随着充放电次数的增加，锂电池的性能逐渐下降。电池的性能可以用容量来表示，相关指标可以定义如下：

$SOH(t) = Ct/C0 \times 100\%$,

其中，$C0$ 表示额定容量，Ct 表示 t 时刻的容量。等到 SOH 降到 70% 时，电池视为失效。我们要做的是用电池的历史数据，比如电流、电压和容量，对电池的下降趋势进行建模。然后，用训练好的模型来预测电池的 SOH。

3.1 个别模块（KAN 模块）



KAN（Kolmogorov-Arnold Network）基于 Kolmogorov-Arnold 表示定理设计，能够将任何连续的多维函数表示为若干单变量函数的嵌套组合，从而为复杂高维数据的处理提供理论支撑。在锂离子电池预测任务中，高维数据间常存在复杂非线性关系，KAN 通过其特有的非线性激活和线性组合特性，可有效降低高维数据的复杂性，实现特征的高效建模与利用。因此，KAN 模块在整个架构中被置于 FUnet 模块之后，作为最终预测模块。

KAN 的总体公式如下：

$$KAN(x)=f(x_1,x_2,\ldots,x_n)=\sum_{q=1}^{2n+1}\Phi_q\left(\sum_{p=1}^{n}\varphi_{q,p}(x_p)\right) \quad (1)$$

具体而言，首先，KAN 通过输入特征 x_p 的分布动态生成网格节点位置，基于 b 样条基函数对每个输入特征通过网格节点实现插值，所有插值结果被线性组合以生成聚合结果 $\varphi_{q,p}(x_p)$，公式如下：

$$B_{i,k}(x_p)=\frac{x_p-t_i}{t_{i+k}-t_i}B_{i,k-1}(x_p)+\frac{t_{i+k+1}-x_p}{t_{i+k+1}-t_{i+1}}B_{i+1,k-1}(x_p) \quad (2)$$

$$\varphi_{q,p}(x_p)=\sum_{i=1}^{N}w_iB_{i,k}(x_p) \quad (3)$$

其中，t_i 为插值节点的位置，k 为 b 样条的度数，w_i 为可学习权值，N 为插值节点数。

之后，对样条结果进行加权求和，得到所有输入特征 x_p 的局部插值结果 $\sum_{p=1}^{n}\varphi_{q,p}(x_p)$，结合非线性激活函数和偏置项，利用 Φ_q 对进行全局非线性处理。公式如下：

$$\Phi_q(y_q)=\sigma\left(\sum_{j=1}^{m}v_{q,j}\cdot y_{q,j}+b_q\right) \quad (4)$$

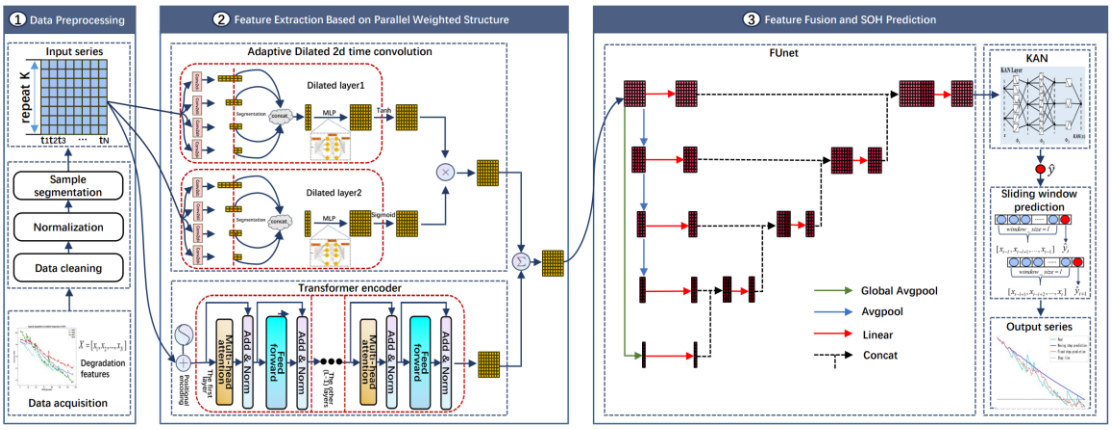
其中，Φ_q 为 KAN 模型中的全局非线性变换因子，y_q 表示插值后所有输入特征的局部插值结果之和，$\sigma(\cdot)$ 为非线性激活函数，$v_{q,j}$ 为全局非线性变换层的可学习权值，b_q 为偏置项，

表示偏移量，m 为全局函数层的节点数。

KAN 通过可学习的样条权值和激活函数参数，使其能够动态适配不同输入数据分布，实现了对并行加权结构与 FUnet 处理后的复杂高维特征的有效建模，确保最后的 SOH 预测结果具有更高的准确性与鲁棒性。

3.2 总框架介绍

整个框架分为数据预处理、特征提取、模型训练与评估四个阶段。在预处理阶段完成数据归一化与训练集/测试集划分；在特征提取阶段并行运行 ADTC 与 Transformer 模块；在模型训练阶段，通过 FUnet 模块对加权的高维特征进行多尺度融合，通过 KAN 模块的非线性建模优化最终输出；最后在测试阶段完成单步预测与多步预测任务。总体框架的各模块之间高效协作，形成从原始数据到 SOH 预测的完整闭环。



在预处理阶段，对原始健康状态数据 X 预处理，得到 ADTC 模块的初始输入 X_{re}，通过逐渐增大卷积核 $[k_1,k_2,k_3,k_4]$ 的膨胀卷积层,经过提取得到时间依赖的多尺度特征 X_1,X_2,X_3,X_4，卷积过程中不进行 padding 填充操作:

$$X_{re}\in\mathbb{R}^{C\times N\times T}=reshape(X) \quad (5)$$

$$X_i\in\mathbb{R}^{(C/4)\times N\times(T-S_i)}=\mathrm{Conv2d}_i(X_{re}) \ , \mathrm{i}=(1,2,3,4) \quad (6)$$

$$\begin{cases} S_i=d_i^{'}\cdot(k_i-1) \ , \mathrm{i}=(1,2,3,4) \\ k=[2,4,8,\dfrac{3T}{4}] \\ \mathrm{Conv2d}_i(\cdot)=(C,\dfrac{C}{4},(1,k_i),d_i^{'}) \\ d_i^{'}=(1,\max(1,\dfrac{T+1}{2k_i})) \end{cases} \quad (7)$$

其中，X 是原始健康状态数据，X_{re} 表示重塑后的 X，$\in\mathbb{R}^{C\times N\times T}$ 表示所属的实数张量形状，C 表示特征数量，N 表示序列个数，T 表示时间步长，$,X_i,i=(1,2,3,4)$ 是 X_{re} 通过

不同卷积核 $k_i, i=(1,2,3,4)$ 进行卷积后的四个多尺度特征，$\mathrm{Conv2d}_i(\cdot)$ 表示二维卷积的映射函数，S_i ,i=(1,2,3,4) 指卷积操作的步长，$k=[2,4,8,\frac{3T}{4}]$ 分别对应 k_1,k_2,k_3,k_4 四个卷积核的大小，$(C,\frac{C}{4},(1,k_i),d_i^{'})$ 指二维卷积的输入通道数为 C，输出通道数为 $\frac{C}{4}$，$(1,k_i)$ 指二维卷积核的大小，$d_i^{'}$ 指膨胀因子；

对不同卷积核进行卷积后，生成四个多尺度特征 X_1,X_2,X_3,X_4。为了统一多尺度特征的时间维度，以时间维度最短的特征为标准，进行一致化处理，截取长度为 $(T-S_{\max})$，得到四个截取后的多尺度特征 $X_1^{'},X_2^{'},X_3^{'},X_4^{'}$：

$$X_i^{'}\in\mathbb{R}^{(C/4)\times N\times(T-S_{\max})}=X_i[\ldots,-(T-S_{\max}):] \text{ , i=(1,2,3,4)} \quad (8)$$

其中，$[\ldots,-(T-S_{\max}):]$ 指在时间维度上统一截取最后 $(T-S_{\max})$ 个时间步长的值，$S_{\max}$ 为 S_i ,i=(1,2,3,4) 中的最大值；

随后，这些特征沿通道维度拼接，恢复与 ADTC 初始输入 X_{re} 相同的特征数量，得到拼接后的特征张量 X_T：

$$X_T\in\mathbb{R}^{C\times N\times(T-S_{\max})}=\mathrm{Concat}(X_1^{'},X_2^{'},X_3^{'},X_4^{'}) \quad (9)$$

接着，拼接后的特征张量 X_T 经多层感知机（MLP）处理生成第一个膨胀初始层的输出 X_T'：

$$X_T'\in\mathbb{R}^{C\times N\times T}=\mathrm{MLP}(X_T) \quad (10)$$

接着，通过类似处理，得到第二个膨胀初始层 DIL 的输出 X_S'。

随后，X_T'、X_S'分别通过 Tanh 激活函数和 Sigmoid 激活函数处理，再通过点积运算生成 ADTC 模块的最终输出 Y_{ADTC}：

$$Y_{ADTC}\in\mathbb{R}^{C\times N\times T}=\mathrm{Tanh}(X_T')\odot\sigma(X_S') \quad (11)$$

其中，$\mathrm{Tanh}(\cdot)$ 表示 tanh 激活函数，$\sigma(\cdot)$ 表示 Sigmoid 激活函数，$\odot$ 表示点积乘法。

并行结构在将 X_{re} 输入 ADTC 模块的同时，将 X_{re} 输入到 Transformer 编码器模块，通过多头注意力和全连接层的处理生成全局特征表示 Y_{TRM}：

$$Y_{TRM} = TransformerEncoder(X_{re}) \quad （12）$$

其中，$TransformerEncoder(\cdot)$ 表示 Transformer 模块的流程函数，包含公式（1）~（8）。

接着，对 ADTC 模块与 Transformer 编码器模块的输出进行加权融合，生成 FUnet 模块的输入特征张量：

$$Z = \alpha_{ADTC} \cdot Y_{ADTC} + \alpha_{TRM} \cdot Y_{TRM} \quad （13）$$

在 FUnet 模块中，构建特征金字塔结构，从第二层开始通过池化层提取特征，得到提取后的特征 Z_2, Z_3, Z_4, Z_5：

$$\begin{cases} Z_1 \in \mathbb{R}^{C\times N\times T_1} = Z \quad ,\mathrm{T}_1 = T \\ Z_i \in \mathbb{R}^{C\times N\times T_i} = AvgPool(Z_{i-1}) \quad ,\mathrm{i}=(2,3,4) \\ Z_5 \in \mathbb{R}^{C\times N\times T_5} = GlobalAvgPool(Z_1) \quad ,\mathrm{T}_5 = 1 \end{cases} \quad （14）$$

其中，$T_i \ ,i=(2,3,4)$ 由计算公式 $\mathrm{len}(\cdot)$ 得出，公式表达为：

$$\mathrm{len}(T_i) = \frac{T_{i-1} + 2\times padding - \mathrm{kernel\_size}}{\mathrm{stride}} + 1 \ \ ,\mathrm{i}=(2,3,4) \quad （15）$$

随后，各层特征 Z_1, Z_2, Z_3, Z_4, Z_5 经过线性层映射为中间特征 E_1, E_2, E_3, E_4, E_5：

$$E_i = \mathrm{Linear}(Z_i), \mathrm{i} = (1,2,3,4,5) \quad （16）$$

随后，基于金字塔结构，从上到下依次进行中间特征 E_1, E_2, E_3, E_4, E_5 在时间维度的拼接，并通过线性层生成下一层中间特征 D_{i+1}，重复此操作直至输出第 1 层特征 D_1：

$$\begin{cases} D_i = \mathrm{Linear}(\mathrm{Concat}(D_{i+1}, E_i)) \ \ ,\mathrm{i}=(1,2,3) \\ D_4 = \mathrm{Linear}(\mathrm{Concat}(E_5, E_4)) \end{cases} \quad （17）$$

随后，把第 1 层的新特征 D_1 当作多尺度特征的融合特征 O，经过线性层和 reshape 操作得到最终融合特征 O'：

$$O' \in \mathbb{R}^{N\times T\times C} = \mathrm{reshape}(W_O \in \mathbb{R}^{C\times C} \cdot O) \quad （18）$$

其中，$W_O \in \mathbb{R}^{C\times C}$ 表示线性层的变换矩阵，且其是一个 $C\times C$ 的变换矩阵；

最后，基于从 FUnet 模块获得的高维特征，采用 Kolmogorov-Arnold 网络（KAN）将融合特征 O' 转化为点预测结果 O''，公式表达如下：

$$O'' \in \mathbb{R}^{\in 1\times 1} = \mathrm{KAN}(O') \quad (19)$$

其中，$KAN(\cdot)$ 指 KAN 模块的流程函数。

在测试阶段，经过预处理后的测试数据 $X_{\text{test}} \in \mathbb{R}^{C\times N\times T}$ 输入到训练好的模型 $\mathrm{Model}(\cdot)$ 中，通过滑动窗口累计预测点，生成完整的健康状态预测结果 O''_{pred}：

$$O''_{pred} \in \mathbb{R}^{1\times T_{pred}} = \mathrm{Model}(X_{\text{test}}) \quad (20)$$

其中，$O''_{pred} \in \mathbb{R}^{1\times T_{pred}}$ 指测试结果 O''_{pred} 是包含 T_{pred} 个点数据标量，T_{pred} 指滑动窗口从预测开始点到预测截止点（电池的记录最大放电循环数）的滑动步数。

4. 实验

4.1 数据集

为了验证所提模型的有效性，本文选用了 NASA、CALCE 和 WRBD 三大数据集，这些数据集提供了在不同实验条件下的电池退化信息。

NASA 数据集

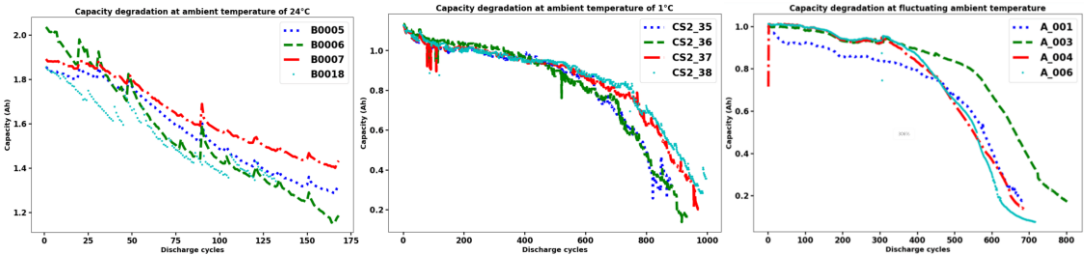
NASA 数据集包括 18650 锂离子电池（B0005、B0006、B0007、B0018）的测试数据。这些测试在 24°C 的室温条件下进行，采用恒流（CC）模式以 1.5 A 电流充电至 4.2 V，随后切换至恒压（CV）充电，直到电流降至 20 mA。放电过程在恒流 2 A 下进行，截止电压分别为 2.7 V、2.5 V、2.2 V 和 2.5 V。电池寿命终止（EOL）标准为电池容量衰减至初始额定容量的 30%，即从 2 Ah 降至约 1.4 Ah。该数据集详细记录了不同工况下的退化趋势，是电池 SOH 预测领域中常用的基准数据集。

CALCE 数据集

CALCE 数据集由先进生命周期工程中心（CALCE）提供，包含四个电池（CS2_35、CS2_36、CS2_37、CS2_38）的数据。测试在 1°C 的控制温度下进行，充电采用恒流模式至 4.2 V 后切换至恒压充电，直至电流降至 20 mA。放电以恒流模式进行，截止电压为 2.7 V。EOL 标准同样定义为额定容量下降 30%，即从 1.1 Ah 降至约 0.77 Ah。该数据集捕捉了在严格实验室条件下的退化特性，为 SOH 预测建模提供了宝贵的数据支持。

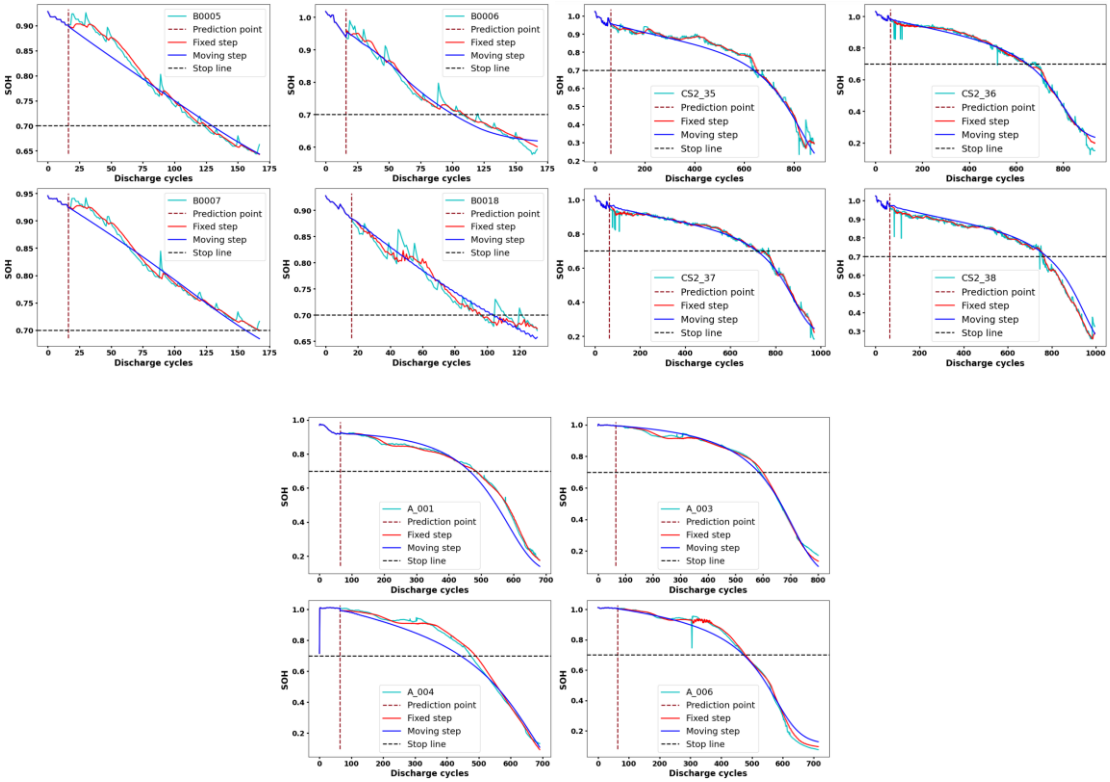
WRBD 数据集

温州随机电池数据集（WRBD）(Lyu et al., 2024，详见原文)由本实验室采集与处理，用以补充 NASA 和 CALCE 的公共数据集。WRBD 数据集包含四个锂离子电池（A_001、A_003、A_004、A_006）的数据，这些电池均在标准的恒流/恒压协议下测试。充电过程以 1C 电流充至 4.2 V 后切换至恒压模式，直到电流降至 0.05C。放电过程以 1C 电流至截止电压 2.75 V 完成。测试均在室温条件下进行，并定期记录容量衰减情况。EOL 标准为容量从 1.0 Ah 降至约 0.7 Ah。WRBD 数据集提供了独特的、高质量的退化数据，增加了训练和评估过程的多样性与鲁棒性。

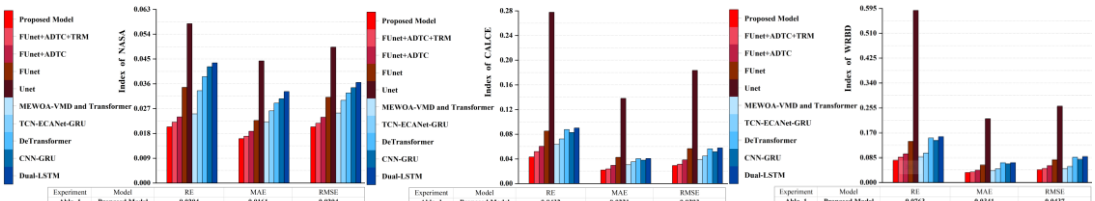


4.2 结果与讨论

4.2.1 单步预测与多步预测



4.2.2 消融实验+对比实验

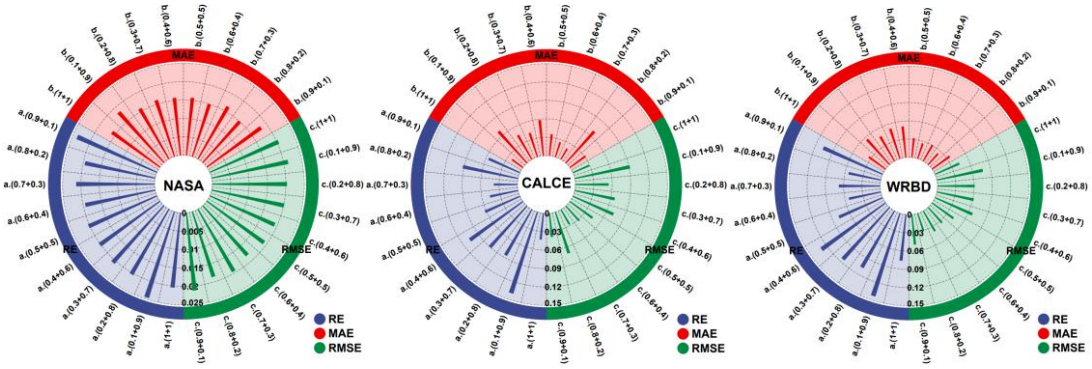


Experiment	Model	RE	MAE	RMSE
Abla. 1	**Proposed Model**	**0.0204**	**0.0161**	**0.0204**
Abla. 2	FUnet+ADTC+TRM	0.0221	0.0170	0.0217
Abla. 3	FUnet+ADTC	0.0240	0.0188	0.0239
Abla. 4	FUnet	0.0347	0.0227	0.0311
Abla. 5	Unet	0.0578	0.0443	0.0493
Comp. 1	MEWOA-VMD and Transformer	0.0251	0.0221	0.0254
Comp. 2	TCN-ECANet-GRU	0.0335	0.0262	0.0300
Comp. 3	DeTransformer	0.0386	0.0290	0.0327
Comp. 4	CNN-GRU	0.0422	0.0306	0.0346
Comp. 5	Dual-LSTM	0.0436	0.0332	0.0365

Experiment	Model	RE	MAE	RMSE
Abla. 1	**Proposed Model**	**0.0432**	**0.0221**	**0.0293**
Abla. 2	FUnet+ADTC+TRM	0.0520	0.0240	0.0315
Abla. 3	FUnet+ADTC	0.0605	0.0296	0.0386
Abla. 4	FUnet	0.0853	0.0426	0.0568
Abla. 5	Unet	0.2778	0.1381	0.1835
Comp. 1	MEWOA-VMD and Transformer	0.0636	0.0306	0.0387
Comp. 2	TCN-ECANet-GRU	0.0724	0.0353	0.0447
Comp. 3	DeTransformer	0.0876	0.0405	0.0564
Comp. 4	CNN-GRU	0.0829	0.0380	0.0516
Comp. 5	Dual-LSTM	0.0902	0.0407	0.0578

Experiment	Model	RE	MAE	RMSE
Abla. 1	**Proposed Model**	**0.0763**	**0.0341**	**0.0437**
Abla. 2	FUnet+ADTC+TRM	0.0871	0.0362	0.0482
Abla. 3	FUnet+ADTC	0.0980	0.0423	0.0579
Abla. 4	FUnet	0.1408	0.0601	0.0781
Abla. 5	Unet	0.5881	0.2180	0.2608
Comp. 1	MEWOA-VMD and Transformer	0.0884	0.0410	0.0476
Comp. 2	TCN-ECANet-GRU	0.1006	0.0472	0.0550
Comp. 3	DeTransformer	0.1522	0.0678	0.0866
Comp. 4	CNN-GRU	0.1440	0.0637	0.0793
Comp. 5	Dual-LSTM	0.1566	0.0682	0.0888

4.2.3 权值调试实验



讨论详见原文。

5 代码讲解

5.1 数据预处理

这段代码实现了锂电池健康状态预测任务的数据预处理流程。首先，通过 loadMat 函数加载 .mat 文件并提取电池的各类数据，包括电池类型、温度、时间戳和容量等信息。接着，使用 convert_to_time 函数将时间戳转换为标准的日期时间格式。然后，提取电池的放电数据（discharge）作为容量数据，通过 getBatteryCapacity 函数获取每个周期的容量信息。针对不同的电池类型（如充电或放电），还可以使用 getBatteryValues 函数提取相关数据。为了准备训练数据，代码通过滑动窗口方式，使用 build_instances 函数将容量数据切分为特征和目标值，方便后续模型训练。同时，split_dataset 函数根据给定的训练集比例或容量阈值，将数据分割为训练集和测试集。最后，通过 get_train_test 函数实现了基于“留一法”评估的训练集和测试集构建，确保一个电池的数据用于测试，其余数据用于训练。整个过程的目标是将原始电池数据转化为适合深度学习模型训练的格式，为后续的健康状态预测模型提供准备工作。

```
def convert_to_time(hmm):
    year, month, day, hour, minute, second = int(hmm[0]), int(hmm[1]), int(hmm[2]),
int(hmm[3]), int(hmm[4]), int(hmm[5])
    return datetime(year=year, month=month, day=day, hour=hour, minute=minute,
second=second)



# load .mat data
def loadMat(matfile):
    data = scipy.io.loadmat(matfile)
    filename = matfile.split("/")[-1].split(".")[0]
    col = data[filename]
    col = col[0][0][0][0]
    print("col_shape:",col.shape)
    size = col.shape[0]
    print("size_shape:",size)

    data = []
    for i in range(size):
```

```
        k = list(col[i][3][0].dtype.fields.keys())
        d1, d2 = {}, {}
        if str(col[i][0][0]) != 'impedance':
            for j in range(len(k)):
                t = col[i][3][0][0][j][0];
                l = [t[m] for m in range(len(t))]
                d2[k[j]] = l
        d1['type'], d1['temp'], d1['time'], d1['data'] = str(col[i][0][0]), int(col[i][1][0]),
str(convert_to_time(col[i][2][0])), d2
        data.append(d1)

    return data


# get capacity data
def getBatteryCapacity(Battery):
    cycle, capacity = [], []
    i = 1
    for Bat in Battery:
        if Bat['type'] == 'discharge':
            capacity.append(Bat['data']['Capacity'][0])
            cycle.append(i)
            i += 1
    return [cycle, capacity]


# get the charge data of a battery
def getBatteryValues(Battery, Type='charge'):
    data=[]
    for Bat in Battery:
        if Bat['type'] == Type:
            data.append(Bat['data'])
    return data

# P1.1
Battery_list = ['B0005', 'B0006', 'B0007', 'B0018']
dir_path = 'nasa/'

Battery = {}
for name in Battery_list:
    print('Load Dataset ' + name + '.mat ...')
    path = dir_path + name + '.mat'
    data = loadMat(path)
    Battery[name] = getBatteryCapacity(data)
```

```
# P2.1
def build_instances(sequence, window_size):
    # sequence: list of capacity
    x, y = [], []
    for i in range(len(sequence) - window_size):
        features = sequence[i:i + window_size]
        target = sequence[i + window_size]

        x.append(features)
        y.append(target)

    return np.array(x).astype(np.float32), np.array(y).astype(np.float32)


def split_dataset(data_sequence, train_ratio=0.0, capacity_threshold=0.0):
    if capacity_threshold > 0:
        max_capacity = max(data_sequence)
        capacity = max_capacity * capacity_threshold
        point = [i for i in range(len(data_sequence)) if data_sequence[i] < capacity]
    else:
        point = int(train_ratio + 1)
        if 0 < train_ratio <= 1:
            point = int(len(data_sequence) * train_ratio)
    train_data, test_data = data_sequence[:point], data_sequence[point:]

    return train_data, test_data


# leave-one-out evaluation: one battery is sampled randomly; the remainder are used for
training.
def get_train_test(data_dict, name, window_size=8):
    data_sequence = data_dict[name][1]
    train_data, test_data = data_sequence[:window_size + 1], data_sequence[window_size +
1:]
    train_x, train_y = build_instances(train_data, window_size)
    for k, v in data_dict.items():
        if k != name:
            data_x, data_y = build_instances(v[1], window_size)
            train_x, train_y = np.r_[train_x, data_x], np.r_[train_y, data_y]

    return train_x, train_y, list(train_data), list(test_data)
```

5.2 模块定义

在具体的模块设计方面，本文实现了多个创新性的神经网络模块。例如，block_model 模块通过线性变换和归一化操作提取特征；Model2 模块则结合下采样和特征融合策略，在不同尺度上捕捉时序数据特征；dilated_inception2 模块采用扩张卷积来增强感受野并有效处理时序信息；temporal_conv2 结合多种扩张卷积并应用非线性变换，捕捉复杂的时序模式；SEAttention 和 LocalSEAttention 则通过通道和局部信息加权来增强特征表示能力。此外，TransformerEncoderLayer 和 TransformerEncoderLayers 通过堆叠多个自注意力层，进一步增强了对时序依赖关系的建模能力；Trend_aware_attention 则通过引入趋势感知机制，使模型能更好地理解和捕捉时序数据的动态变化。最后，PositionalEncoding 模块为输入数据添加了位置编码，帮助模型识别不同时间步的顺序信息。通过这些模块的组合与优化，本文提出的 ADTC-Transformer 架构在处理锂电池健康状态预测任务时展示了卓越的性能，尤其在长期预测中取得了显著的效果。

```
class block_model(nn.Module):
    """
    Decomposition-Linear
    """

    def __init__(self, input_channels, input_len, out_len):
        super(block_model, self).__init__()
        self.channels = input_channels
        self.input_len = input_len
        self.out_len = out_len

        self.Linear_channel = nn.Linear(self.input_len, self.out_len)
        self.ln = nn.LayerNorm(out_len)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        # (B,C,N,T) --> (B,C,N,T)
        output = self.Linear_channel(x)
        return output


class Model2(nn.Module):
    def __init__(self, input_channels=64, out_channels=64, seq_len=720, pred_len=720):
        super(Model2, self).__init__()

        self.input_channels = input_channels
        self.out_channels = out_channels
        self.input_len = seq_len
        self.out_len = pred_len

        # 下采样设定
        n1 = 1
```

```
        filters = [n1, n1 * 2, n1 * 4, n1 * 8, n1 * int(self.input_len)]
        down_in = [int(self.input_len / filters[i]) for i in range(5)]
        down_out = [int(self.out_len / filters[i]) for i in range(5)]

        # 最大池化层
        self.Maxpool1 = nn.AvgPool2d(kernel_size=(1, 3), stride=(1, 2), padding=(0, 1))
        self.Maxpool2 = nn.AvgPool2d(kernel_size=(1, 3), stride=(1, 2), padding=(0, 1))
        self.Maxpool3 = nn.AvgPool2d(kernel_size=(1, 3), stride=(1, 2), padding=(0, 1))
        self.Maxpool4 = nn.AvgPool2d(kernel_size=(1, 3), stride=(1, 2), padding=(0, 1))
        self.GlobalAvgPool = nn.AdaptiveAvgPool2d((1, 1))  # 全局平均池化

        # 左边特征提取层
        self.down_block1 = block_model(self.input_channels, down_in[0], down_out[0])
        self.down_block2 = block_model(self.input_channels, down_in[1], down_out[1])
        self.down_block3 = block_model(self.input_channels, down_in[2], down_out[2])
        self.down_block4 = block_model(self.input_channels, down_in[3], down_out[3])
        self.down_block5 = block_model(self.input_channels, down_in[4], down_out[4])  # 由于
全局平均池化的输出长度为 1

        # 右边特征融合层
        self.up_block4 = block_model(self.input_channels, down_out[3] + down_out[4],
down_out[3])
        self.up_block3 = block_model(self.input_channels, down_out[2] + down_out[3],
down_out[2])
        self.up_block2 = block_model(self.input_channels, down_out[1] + down_out[2],
down_out[1])
        self.up_block1 = block_model(self.input_channels, down_out[0] + down_out[1],
down_out[0])

        # 输出映射
        self.linear_out = nn.Linear(self.input_channels, self.out_channels)

    def forward(self, x):
        x1 = x.permute(0, 3, 1, 2)  # (B,N,T,C) -> (B,C,N,T)
        e1 = self.down_block1(x1)  # (B,C,N,T) -> (B,C,N,T)

        x2 = self.Maxpool1(x1)  # (B,C,N,T) -> (B,C,N,T/2)
        e2 = self.down_block2(x2)  # (B,C,N,T/2) -> (B,C,N,T/2)

        x3 = self.Maxpool2(x2)  # (B,C,N,T/2) -> (B,C,N,T/4)
        e3 = self.down_block3(x3)  # (B,C,N,T/4) -> (B,C,N,T/4)

        x4 = self.Maxpool3(x3)  # (B,C,N,T/4) -> (B,C,N,T/8)
        e4 = self.down_block4(x4)  # (B,C,N,T/8) -> (B,C,N,T/8)
```

```
        # 全局平均池化
        x5 = self.GlobalAvgPool(x1)  # (B,C,N,T) -> (B,C,1,1)
        e5 = self.down_block5(x5)  # (B,C,1,1) -> (B,C,1,1)

        # 第五层向第四层融合
        d4 = torch.cat((e4, e5), dim=-1)  # (B,C,N,T/8) + (B,C,1,1) -> (B,C,N,T/8+1)
        d4 = self.up_block4(d4)  # (B,C,N,T/8+1) -> (B,C,N,T/8)

        # 第四层向第三层融合
        d3 = torch.cat((e3, d4), dim=-1)  # (B,C,N,T/4) + (B,C,N,T/8) -> (B,C,N,3T/8)
        d3 = self.up_block3(d3)  # (B,C,N,3T/8) -> (B,C,N,T/4)

        # 第三层向第二层融合
        d2 = torch.cat((e2, d3), dim=-1)  # (B,C,N,T/2) + (B,C,N,T/4) -> (B,C,N,3T/4)
        d2 = self.up_block2(d2)  # (B,C,N,3T/4) -> (B,C,N,T/2)

        # 第二层向第一层融合
        d1 = torch.cat((e1, d2), dim=-1)  # (B,C,N,T) + (B,C,N,T/2) -> (B,C,N,3T/2)
        out = self.up_block1(d1)  # (B,C,N,3T/2) -> (B,C,N,T)

        out = self.linear_out(out.permute(0, 2, 3, 1))  # (B,C,N,T) -> (B,N,T,C)
        return out




class dilated_inception2(nn.Module):
    def __init__(self, cin, cout, seq_len, kernel_set=None, base_dilation_factor=1):
        super(dilated_inception2, self).__init__()
        self.tconv = nn.ModuleList()
        self.padding = 0  # No padding
        self.seq_len = seq_len
        self.base_dilation_factor = base_dilation_factor
        if kernel_set is None:
            self.kernel_set = [2, 4, 8, 3*int(cin)//4]  # Default kernel sizes
        else:
            self.kernel_set = kernel_set
        cout = int(cout / len(self.kernel_set))  # Divide output channels by number of kernels

        # Calculate appropriate dilation factors for each kernel
        self.dilation_factors = self.calculate_dilation_factors(self.seq_len, self.kernel_set,
```

```
self.base_dilation_factor)

        for kern, dilation_factor in zip(self.kernel_set, self.dilation_factors):
            self.tconv.append(nn.Conv2d(cin, cout, (1, kern), dilation=(1, dilation_factor)))

        # Calculate input size for the fully connected layer
        min_time_dim = min([self.seq_len - dilation_factor * (kern - 1) for kern, dilation_factor
in
                            zip(self.kernel_set, self.dilation_factors)])
        lin_input_size = min_time_dim

        self.out = nn.Sequential(
            nn.Linear(lin_input_size, cin),
            nn.ReLU(),
            nn.Linear(cin, self.seq_len)
        )

    def calculate_dilation_factors(self, seq_len, kernel_set, base_dilation_factor):
        # A simple strategy to calculate dilation factors
        # Here we use a heuristic to spread dilation factors across the kernel sizes
        dilation_factors = [max(1, base_dilation_factor * (seq_len // (2 * k))) for k in kernel_set]
        return dilation_factors

    def forward(self, input):
        # input: (B, C, N, T)
        x = []
        for i in range(len(self.kernel_set)):
            x.append(self.tconv[i](input))  # Perform dilated convolutions with different kernel
sizes

        # Align the time dimension by truncating to the minimum length
        min_time_dim = min([xi.size(3) for xi in x])
        for i in range(len(self.kernel_set)):
            x[i] = x[i][..., -min_time_dim:]

        x = torch.cat(x, dim=1)  # Concatenate along the channel dimension
        x = self.out(x)  # Apply fully connected layers
        return x

class temporal_conv2(nn.Module):
    def __init__(self, cin, cout, seq_len, base_dilation_factor=1):
        super(temporal_conv2, self).__init__()

        self.filter_convs = dilated_inception2(cin=cin, cout=cout, seq_len=seq_len,
```

```
base_dilation_factor=base_dilation_factor)
        self.gated_convs = dilated_inception2(cin=cin, cout=cout, seq_len=seq_len,
base_dilation_factor=base_dilation_factor)
        self.silu_convs = dilated_inception2(cin=cin, cout=cout, seq_len=seq_len,
base_dilation_factor=base_dilation_factor)
        self.silu_activation = nn.SiLU()  # Instantiate the SiLU activation function

    def forward(self, X):
        # X:(B,C,N,T)
        filter = self.filter_convs(X)  # 执行左边的 DIL 层: (B,C,N,T)-->(B,C,N,T)
        filter = torch.tanh(filter)  # 左边的 DIL 层后接一个 tanh 激活函数,生成输出:(B,C,N,T)--
>(B,C,N,T)
        silu = self.silu_convs(X)
        silu = self.silu_activation(silu)  # Apply SiLU activation function to the tensor
        gate = self.gated_convs(X)  # 执行右边的 DIL 层: (B,C,N,T)-->(B,C,N,T)
        gate = torch.sigmoid(gate)  # 右边的 DIL 层后接一个 sigmoid 门控函数,生成权重表
示:(B,C,N,T)-->(B,C,N,T)
        # out = filter * gate * silu # 执行逐元素乘法: (B,C,N,T) * (B,C,N,T) = (B,C,N,T)
        out = filter * gate
        return out




from src.efficient_kan import KAN1,KANLinear1

from torch.nn import init

"Squeeze-and-Excitation Networks"

class SEAttention(nn.Module):

    def __init__(self, channel=512,reduction=16):
        super().__init__()
        # 在空间维度上,将 H×W 压缩为 1×1
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        # 包含两层全连接,先降维,后升维。最后接一个 sigmoid 函数
        self.fc = nn.Sequential(
            nn.Linear(channel, channel // reduction, bias=False),
            torch.nn.SiLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            torch.nn.SiLU()
        )
```

```
    def init_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                init.kaiming_normal_(m.weight, mode='fan_out')
                if m.bias is not None:
                    init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                init.constant_(m.weight, 1)
                init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                init.normal_(m.weight, std=0.001)
                if m.bias is not None:
                    init.constant_(m.bias, 0)

    def forward(self, x):
        # (B,C,H,W)
        B, C, H, W = x.size()
        # Squeeze: (B,C,H,W)-->avg_pool-->(B,C,1,1)-->view-->(B,C)
        y = self.avg_pool(x).view(B, C)
        # Excitation: (B,C)-->fc-->(B,C)-->(B, C, 1, 1)
        y = self.fc(y).view(B, C, 1, 1)
        # scale: (B,C,H,W) * (B, C, 1, 1) == (B,C,H,W)
        out = x * y
        return out

class LocalSEAttention(nn.Module):
    def __init__(self, channel=512, reduction=16, kernel_size=3):
        super(LocalSEAttention, self).__init__()
        # 局部卷积层，提取局部信息
        self.conv = nn.Conv2d(channel, channel, kernel_size=kernel_size,
padding=kernel_size//2, groups=channel)
        # 全局平均池化
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        # 全连接层，先降维，后升维，最后接一个 sigmoid 函数
        self.fc = nn.Sequential(
            nn.Linear(channel, channel // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        # (B, C, H, W)
```

```
        B, C, H, W = x.size()
        # 局部卷积，提取局部信息
        local_feature = self.conv(x)
        # Squeeze: (B,C,H,W) --> avg_pool --> (B,C,1,1) --> view --> (B,C)
        y = self.avg_pool(local_feature).view(B, C)
        # Excitation: (B,C) --> fc --> (B,C) --> (B, C, 1, 1)
        y = self.fc(y).view(B, C, 1, 1)
        # scale: (B,C,H,W) * (B, C, 1, 1) == (B,C,H,W)
        out = x * y
        return out



from torch import Tensor
from typing import Optional


class TransformerEncoderLayer(nn.Module):
    def __init__(self, embed_dim, dense_dim, num_heads, dropout_rate, norm_first=False,
batch_first=False):
        super(TransformerEncoderLayer, self).__init__()

        self.self_attn = nn.MultiheadAttention(embed_dim, num_heads, batch_first=batch_first)
        self.layernorm1 = nn.LayerNorm(embed_dim)
        self.dropout1 = nn.Dropout(dropout_rate)

        self.dense1 = nn.Linear(embed_dim, dense_dim)
        self.dense2 = nn.Linear(dense_dim, embed_dim)
        self.layernorm2 = nn.LayerNorm(embed_dim)
        self.dropout2 = nn.Dropout(dropout_rate)

        self.norm_first = norm_first  # 允许 layernorm 在注意力和前馈网络之前或之后执行

    def forward(self, src: Tensor, src_mask: Optional[Tensor] = None,
                src_key_padding_mask: Optional[Tensor] = None) -> Tensor:
        # fastpath 选项：启用高效路径以减少推理时的内存占用
        is_fastpath_enabled = torch.backends.mha.get_fastpath_enabled() and not self.training

        # 使用注意力机制
        if is_fastpath_enabled:
            # 如果启用了 fastpath，使用更高效的路径
            attn_output, _ = self.self_attn(src, src, src, attn_mask=src_mask,
key_padding_mask=src_key_padding_mask,
                                            need_weights=False)
```

```
        else:
            attn_output, _ = self.self_attn(src, src, src, attn_mask=src_mask,
key_padding_mask=src_key_padding_mask)

        attn_output = self.dropout1(attn_output)

        if self.norm_first:
            # 如果启用了 norm_first，先执行 LayerNorm
            src = src + attn_output
            out1 = self.layernorm1(src)
        else:
            out1 = self.layernorm1(src + attn_output)

        # 前馈网络
        dense_output = self.dense1(out1)
        dense_output = self.dense2(dense_output)
        dense_output = self.dropout2(dense_output)

        if self.norm_first:
            # 如果启用了 norm_first，先执行 LayerNorm
            src = out1 + dense_output
            out2 = self.layernorm2(src)
        else:
            out2 = self.layernorm2(out1 + dense_output)

        return out2


class TransformerEncoderLayers(nn.Module):
    def __init__(self, embed_dim, dense_dim, num_heads, dropout_rate, num_layers,
norm_first=False, batch_first=False):
        super(TransformerEncoderLayers, self).__init__()
        self.layers = nn.ModuleList([
            TransformerEncoderLayer(embed_dim, dense_dim, num_heads, dropout_rate,
norm_first, batch_first)
            for _ in range(num_layers)
        ])

    def forward(self, src: Tensor, src_mask: Optional[Tensor] = None,
                src_key_padding_mask: Optional[Tensor] = None) -> Tensor:
        x = src
        for layer in self.layers:
            # 每层都支持 mask 和 padding mask
            x = layer(x, src_mask=src_mask, src_key_padding_mask=src_key_padding_mask)
```

```
        return x


import torch
import torch.nn.functional as F
from torch import nn, Tensor
from typing import Optional


class Trend_aware_attention(nn.Module):
    '''
    Trend_aware_attention 机制
    X:      [batch_size, num_step, num_vertex, D]
    K:      注意力头数
    d:      每个注意力头的输出维度
    return: [batch_size, num_step, num_vertex, D]
    '''

    def __init__(self, K, d, kernel_size):
        super(Trend_aware_attention, self).__init__()
        D = K * d
        self.d = d
        self.K = K
        self.FC_v = nn.Linear(D, D)
        self.FC = nn.Linear(D, D)
        self.kernel_size = kernel_size
        self.padding = self.kernel_size - 1
        self.cnn_q = nn.Conv2d(D, D, (1, self.kernel_size), padding=(0, self.padding))
        self.cnn_k = nn.Conv2d(D, D, (1, self.kernel_size), padding=(0, self.padding))
        self.norm_q = nn.BatchNorm2d(D)
        self.norm_k = nn.BatchNorm2d(D)

        # 调试：打印 kernel_size 和 padding
        print(f"Initialized Trend_aware_attention with kernel_size={self.kernel_size} and
padding={self.padding}")

    def forward(self, X):
        batch_size = X.shape[0]
        print("Input X shape:", X.shape)  # 调试：检查输入 X 的形状

        X_ = X.permute(0, 3, 2, 1)  # (B, T, N, D) --> (B, D, N, T)
        print("X_ shape after permute:", X_.shape)  # 调试

        query = self.norm_q(self.cnn_q(X_))[:, :, :, :-self.padding].permute(0, 3, 2, 1)  # 生成
```

```
query
        key = self.norm_k(self.cnn_k(X_))[:, :, :, :-self.padding].permute(0, 3, 2, 1)  # 生成 key
        value = self.FC_v(X)  # 生成 value

        print("Query shape:", query.shape)  # 调试
        print("Key shape:", key.shape)  # 调试
        print("Value shape:", value.shape)  # 调试

        query = torch.cat(torch.split(query, self.d, dim=-1), dim=0)
        key = torch.cat(torch.split(key, self.d, dim=-1), dim=0)
        value = torch.cat(torch.split(value, self.d, dim=-1), dim=0)

        print("Query shape after split:", query.shape)  # 调试
        print("Key shape after split:", key.shape)  # 调试
        print("Value shape after split:", value.shape)  # 调试

        query = query.permute(0, 2, 1, 3)  # (B*k, N, T, d)
        key = key.permute(0, 2, 3, 1)  # (B*k, N, d, T)
        value = value.permute(0, 2, 1, 3)  # (B*k, N, T, d)

        print("Query shape after permute:", query.shape)  # 调试
        print("Key shape after permute:", key.shape)  # 调试
        print("Value shape after permute:", value.shape)  # 调试

        attention = (query @ key) * (self.d ** -0.5)  # 点积注意力
        print("Attention shape:", attention.shape)  # 调试

        attention = F.softmax(attention, dim=-1)

        X = (attention @ value)  # 加权 value
        print("X shape after attention:", X.shape)  # 调试

        X = torch.cat(torch.split(X, batch_size, dim=0), dim=-1)
        X = self.FC(X)
        return X.permute(0, 2, 1, 3)  # (B, N, T, D) --> (B, T, N, D)


class TransformerEncoderLayer2(nn.Module):
    def __init__(self, embed_dim, dense_dim, num_heads, dropout_rate, kernel_size,
norm_first=False, batch_first=True):
        super(TransformerEncoderLayer2, self).__init__()

        # 使用 TAA 替换原有的 MultiheadAttention
        self.self_attn = Trend_aware_attention(num_heads, embed_dim // num_heads,
```

```
kernel_size)
        self.layernorm1 = nn.LayerNorm(embed_dim)
        self.dropout1 = nn.Dropout(dropout_rate)

        self.dense1 = nn.Linear(embed_dim, dense_dim)
        self.dense2 = nn.Linear(dense_dim, embed_dim)
        self.layernorm2 = nn.LayerNorm(embed_dim)
        self.dropout2 = nn.Dropout(dropout_rate)

        self.norm_first = norm_first

    def forward(self, src: Tensor, src_mask: Optional[Tensor] = None,
                src_key_padding_mask: Optional[Tensor] = None) -> Tensor:

        batch_size, num_step, embed_dim = src.size()
        src = src.view(batch_size, num_step, 1, embed_dim)  # 调整输入格式

        attn_output = self.self_attn(src)
        attn_output = attn_output.view(batch_size, num_step, embed_dim)
        attn_output = self.dropout1(attn_output)

        if self.norm_first:
            src = src.view(batch_size, num_step, embed_dim) + attn_output
            out1 = self.layernorm1(src)
        else:
            out1 = self.layernorm1(src.view(batch_size, num_step, embed_dim) + attn_output)

        dense_output = self.dense1(out1)
        dense_output = self.dense2(dense_output)
        dense_output = self.dropout2(dense_output)

        if self.norm_first:
            src = out1 + dense_output
            out2 = self.layernorm2(src)
        else:
            out2 = self.layernorm2(out1 + dense_output)

        return out2


class TransformerEncoderLayers2(nn.Module):
    def __init__(self, embed_dim, dense_dim, num_heads, dropout_rate, num_layers,
kernel_size, norm_first=False,
                 batch_first=True):
```

```
        super(TransformerEncoderLayers2, self).__init__()
        self.layers = nn.ModuleList([
            TransformerEncoderLayer2(embed_dim, dense_dim, num_heads, dropout_rate,
kernel_size, norm_first, batch_first)
            for _ in range(num_layers)
        ])

    def forward(self, src: Tensor, src_mask: Optional[Tensor] = None,
                src_key_padding_mask: Optional[Tensor] = None) -> Tensor:
        x = src
        for layer in self.layers:
            x = layer(x, src_mask=src_mask, src_key_padding_mask=src_key_padding_mask)
        return x




# P2.23 Transformer 位置编码
class PositionalEncoding(nn.Module):
    def __init__(self, feature_len, feature_size, dropout=0.0):
        '''
        Args:
            feature_len: the feature length of input data (required).
            feature_size: the feature size of input data (required).
            dropout: the dropout rate (optional).
        '''
        super(PositionalEncoding, self).__init__()

        pe = torch.zeros(feature_len, feature_size)
        position = torch.arange(0, feature_len, dtype=torch.float).unsqueeze(1)

        div_term = torch.exp(torch.arange(0, feature_size, 2).float() * (-math.log(10000.0) /
feature_size))

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        pe = pe.unsqueeze(0)

        self.register_buffer('pe', pe)

    def forward(self, x):
```

```
        print("x_size:", x.shape)
        print("pe_size:",self.pe.shape)
        x = x + self.pe

        return x
```

5.3 主网络

Net 模型通过结合多个深度学习模块（如 Transformer 编码器、卷积神经网络和自注意力机制），有效地提升了锂电池健康状态（SOH）预测的能力。通过对多特征时序数据的处理、特征融合和加权机制，模型能够在捕捉电池数据中的短期和长期依赖关系方面取得显著的效果。整体架构使得模型能够处理复杂的时序特征，适应不同的数据输入，并提高了预测的准确性。

```
class Net(nn.Module):
    def __init__(self, feature_size=16, hidden_dim=32, feature_num=1, num_layers=1, nhead=1, dropout=0.0,
                 noise_level=0.01,alpha_3_1=1, alpha_3_2=1):
        '''
        Args:
            feature_size: the feature size of input data (required).
            hidden_dim: the hidden size of Transformer block (required).
            feature_num: the number of features, such as capacity, voltage, and current; set 1 for only sigle feature (optional).
            num_layers: the number of layers of Transformer block (optional).
            nhead: the number of heads of multi-attention in Transformer block (optional).
            dropout: the dropout rate of Transformer block (optional).
            noise_level: the noise level added in Autoencoder (optional).
        '''
        super(Net, self).__init__()
        self.auto_hidden = int(feature_size / 2)
        input_size = self.auto_hidden

        if feature_num == 1:
            # Transformer treated as an Encoder when modeling for a sigle feature like only capacity data
            self.pos = PositionalEncoding(feature_len=feature_num, feature_size=input_size)
            encoder_layers = nn.TransformerEncoderLayer(d_model=input_size, nhead=nhead, dim_feedforward=hidden_dim,
                                                        dropout=dropout, batch_first=True)
        elif feature_num > 1:
            # Transformer treated as a sequence model when modeling for multi-features like capacity, voltage, and current data
            self.pos = PositionalEncoding(feature_len=16, feature_size=16)
```

```
        encoder_layers = nn.TransformerEncoderLayer(d_model=feature_num,
nhead=nhead, dim_feedforward=hidden_dim,
                                                    dropout=dropout, batch_first=True)
        self.cell = nn.TransformerEncoder(encoder_layers, num_layers=num_layers)
        self.cell2 = TransformerEncoderLayers(embed_dim, dense_dim, num_heads,
dropout_rate, num_layers, norm_first, batch_first)
        self.cell3 = TransformerEncoderLayers2(embed_dim, dense_dim, num_heads,
dropout_rate, num_layers, 4, norm_first, batch_first)



        self.linear = nn.Linear(feature_num * feature_size, 1)
        # self.autoencoder = Autoencoder(input_size=feature_size,
hidden_dim=self.auto_hidden, noise_level=noise_level)  #不知道，如果 feature_num =2 的
话，input_size 会不会取 2*feature_size 呢？主要是 input_size 是否是 x 的所有尺寸乘积还
是只要 feature_size 这一个维度？
        # 实例化 Model2 并存储为一个属性
        self.model2 = Model2(input_channels=feature_num, out_channels=feature_num,
seq_len=feature_size,pred_len=feature_size)
        # self.model4 = OptimizedBayesianCNN()
        # self.model3 = ECAAttention(kernel_size=3)
        # self.model_3_4 = Multi_GTU(num_of_timesteps=16, in_channels=16, time_strides=1,
kernel_size=[3,5,7], pool=True)
        self.model_3_5 = temporal_conv2(cin=feature_num, cout=feature_num,
base_dilation_factor=1,seq_len=feature_size)
        self.kan1 = KANLinear1(feature_num * feature_size, 1)
        self.model_1_1 = SEAttention(channel=16,reduction=4)
        self.model_1_1gaidong = LocalSEAttention(channel=16, reduction=4, kernel_size=3)



    def forward(self, x):

        batch_size, feature_num, feature_size = x.shape
        print("shape_x:",x.shape)
        # out, decode = self.autoencoder(x)
        # print("shape_x_autoencoder 后的：",out.shape)
        out1 = x
        if feature_num > 1:
            out1 = out1.reshape(batch_size, -1, feature_num)
        print("Encoded output shape:", out1.shape)
        # # out = self.pos(out)
        #
        # out1 = self.pos(out1)
        # (B,N,T,C)
        out1 = out1.reshape(batch_size,-1,feature_size,feature_num)
```

```
print("out reshape 适应 Unet：",out1.shape)

out1 = out1.permute(0, 3, 1, 2)
# out1 = self.model_3_5(out1)
out1 = self.model_3_5(out1)
# out1 = self.model_3_5(out1)
out1 = out1.permute(0, 2, 3, 1)
# out1 = self.model2(out1)



out2 = x
if feature_num > 1:
    out2 = out2.reshape(batch_size, -1,
                        feature_num)
print("并列 trm out2 初次 reshape shape:", out2.shape)
out2 = self.pos(out2)
out2 = self.cell3(
    out2)

# (B,N,T,C)
out2 = out2.reshape(batch_size, -1, feature_size,
                    feature_num)
print("并列 trm out2 reshape 适应 Unet：", out2.shape)
# out2 = self.model2(out2)

out0 = alpha_3_1 * out1 + alpha_3_2 * out2
out0 = out0.permute(0, 3, 1, 2)   #B,C,N,T
out0 = self.model_larry_4(out0)

# out0 = out1




# out0 = self.model2(out0)  # (B,N,T,C)-->(B,N,T,C)
# print("out0_model2 后的:",out0.shape)
# out = out.permute(0, 3, 1, 2)
# out = self.model_3_5(out)
# out = out.permute(0, 2, 3, 1)
# out = self.model3(out)
# out = out.reshape(batch_size,-1,feature_num)
```

```
        # print("out3_reshape 回 3 维：",out.shape)

        # out = out.reshape(batch_size,feature_num,feature_size)
        # out, decode = self.autoencoder(x)
        # print("shape_x_autoencoder 后的：",out.shape)
        out0 = out0.reshape(batch_size,-1,feature_num)



        # out = self.pos(out)
        # out = self.cell(
        #     out)  # sigle feature: (batch_size, feature_num, auto_hidden) or multi-features:
(batch_size, auto_hidden, feature_num)
        out0 = out0.reshape(batch_size, -1)  # (batch_size, feature_num*auto_hidden)
        print("out3_cell+reshape 后:", out0.shape)
        # out0 = self.kan1(out0)  # out shape: (batch_size, 1);
        out0 = self.linear(out0)

        return out0
```

5.4 训练函数

该训练函数实现了一个完整的锂电池健康状态（SOH）预测模型的训练流程，涵盖了数据预处理、模型初始化、前向传播、损失计算、优化更新、评估指标计算以及早停机制等多个方面。通过循环训练多个电池的数据，评估每个电池模型的性能，并记录每轮训练中的评估指标，最终提供每个电池的预测结果和最优性能。

```
def train(lr=0.01, feature_size=8, feature_num=1, hidden_dim=32, num_layers=1, nhead=1,
dropout=0.0, epochs=1000,
          weight_decay=0.0, seed=0, alpha=0.0, noise_level=0.0, metric='re', device=('cuda:0' if
torch.cuda.is_available() else 'cpu'),
          alpha_3_1=1, alpha_3_2=1):
    '''
        Args:
            lr: learning rate for training (required).
            feature_size: the feature size of input data (required).
            feature_num: the number of features, such as capacity, voltage, and current; set 1
for only sigle feature (optional).
            hidden_dim: the hidden size of Transformer block (required).
            num_layers: the number of layers of Transformer block (optional).
            nhead: the number of heads of multi-attention in Transformer block (optional).
            dropout: the dropout rate of Transformer block (optional).
            epochs:
            weight_decay:
            seed: (optional).
            alpha: (optional).
            noise_level: the noise level added in Autoencoder (optional).
```

```
        metric: (optional).
        device: the device for training (optional).
    '''
    score_list, fixed_result_list, moving_result_list = [], [], []

    min_rmse_record = {}

    setup_seed(seed)
    for i in range(4):
        name = Battery_list[i]
        train_x, train_y, train_data, test_data = get_train_test(Battery, name, feature_size)

        print(f"--- Battery: {name} ---")   #这个打印一下真的可以.

        print("train_x:")
        print(type(train_x))
        if isinstance(train_x, np.ndarray):
            print(f"ShapeShape: {train_x.shape}")
        print(train_x)

        print("train_y:")
        print(type(train_y))
        if isinstance(train_y, np.ndarray):
            print(f"Shape: {train_y.shape}")
        print(train_y)

        print("train_data:")
        print(type(train_data))
        print(f"Length: {len(train_data)}")
        print(train_data)

        print("test_data:")
        print(type(test_data))
        print(f"Length: {len(test_data)}")
        print(test_data)

        print("----------------------")

        test_sequence = train_data + test_data
        print(f"Shape of test_sequence: {np.array(test_sequence).shape}")
        # print('sample size: {}'.format(len(train_x)))

        model = Net(feature_size=feature_size, hidden_dim=hidden_dim, feature_num=K,
num_layers=num_layers,
```

```
                nhead=nhead, dropout=dropout, noise_level=noise_level,
alpha_3_1=alpha_3_1, alpha_3_2=alpha_3_2)
        model = model.to(device)


        optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=weight_decay)
        criterion = nn.MSELoss()

        test_x = train_data.copy()
        loss_list, y_fixed_slice, y_moving_slice = [0], [], []
        rmse, re = 1, 1
        score_, score = [1], [1]

        metrics_record = {}  # Dictionary to store metrics

        for epoch in range(epochs):
            print(f'第{i}个电池，第{epoch}次训练，已完成{epoch / epochs}')
            print(f'第{seed}个种子，第{epoch}次训练，已完成{epoch / epochs}')
            x, y = np.reshape(train_x / Rated_Capacity, (-1, feature_num, feature_size)),
np.reshape(
                train_y / Rated_Capacity, (-1, 1))
            print("shape_x1:",x.shape,"shape_y1:",y.shape)
            x, y = torch.from_numpy(x).to(device), torch.from_numpy(y).to(device)
            print("shape_x2:", x.shape, "shape_y2:", y.shape)
            x = x.repeat(1, K, 1)
            print("shape_x3:", x.shape, "shape_y3:", y.shape)
            output = model(x)
            print("shape_x4:", output.shape, "shape_decode:")
            output = output.reshape(-1, 1)
            print("shape_x5:",output.shape,"shape_y4:",y.shape)
            loss = criterion(output, y) # + alpha * criterion(x)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            if (epoch + 1) % 10 == 0:
                test_x = train_data.copy()
                fixed_point_list, moving_point_list = [], []
                t = 0
                while (len(test_x) - len(train_data)) < len(test_data):
                    x = np.reshape(np.array(test_x[-feature_size-1:-1]) / Rated_Capacity,
                                   (-1, feature_num, feature_size)).astype(np.float32)
                    x = torch.from_numpy(x).to(device)
                    print("while 里的 test_x 后 feature_size 个:",x.shape,"while 里的 test_x:
```

```
",np.array(test_x).shape)
                x = x.repeat(1, K, 1)
                pred = model(x)
                next_point = pred.data.cpu().numpy()[0, 0] * Rated_Capacity
                test_x.append(
                    next_point)  # The test values are added to the original sequence to
continue to predict the next point
                fixed_point_list.append(
                    next_point)  # Saves the predicted value of the last point in the output
sequence
                print(f"Length of fixed_point_list: {len(fixed_point_list)}")

                x = np.reshape(np.array(test_sequence[t:t + feature_size]) / Rated_Capacity,
                               (-1, 1, feature_size)).astype(np.float32)
                x = torch.from_numpy(x).to(device)
                x = x.repeat(1, K, 1)
                pred = model(x)
                next_point = pred.data.cpu().numpy()[0, 0] * Rated_Capacity
                moving_point_list.append(
                    next_point)  # Saves the predicted value of the last point in the output
sequence
                print(f"Length of moving_point_list: {len(moving_point_list)}")
                t += 1
                print("t 多少：  ",t)

            y_fixed_slice.append(fixed_point_list)  # Save all the predicted values
            y_moving_slice.append(moving_point_list)

            loss_list.append(loss)
            # rmse = evaluation(y_test=test_data, y_predict=y_fixed_slice[-1])
            # re = relative_error(y_test=test_data, y_predict=y_fixed_slice[-1])
            re, mae, rmse = calculate_metrics(test_data, y_fixed_slice[-1])
            print(f'Epoch: {epoch + 1}, RE: {re:.4f}, MAE: {mae:.4f}, RMSE: {rmse:.4f}')
            # print('epoch:{:<2d} | loss:{:<6.4f} | RMSE:{:<6.4f} | RE:{:<6.4f}'.format(epoch, loss,
rmse, re))
            # Store metrics every 10 epochs
            metrics_record[epoch] = {'RE': re, 'MAE': mae, 'RMSE': rmse}

            # 打印模型结构和参数量
            summary(model, input_size=(1, 16, feature_size))
            # 在每个 epoch 结束时打印模型参数量
            total_params = sum(p.numel() for p in model.parameters())
            print(f"Total number of parameters: {total_params}")
```

```
            if metric == 're':
                score = [re]
            elif metric == 'rmse':
                score = [rmse]
            else:
                score = [re, rmse]
            # if (loss < 1e-3) and (score_[0] < score[0]):
            # 设置不同的早停条件
            if i == 0:
                if rmse < 2e-2:
                    break
            elif i == 1:
                if rmse < 2e-2:
                    break
            elif i == 2:
                if rmse < 2e-2:
                    break
            elif i == 3:
                if rmse < 2e-2:
                    break
            score_ = score.copy()

        score_list.append(score_)
        fixed_result_list.append(train_data.copy() + y_fixed_slice[-1])
        moving_result_list.append(train_data.copy() + y_moving_slice[-1])

        # Find and record minimum RMSE for the current battery
        min_rmse_epoch = min(metrics_record, key=lambda k: metrics_record[k]['RMSE'])
        min_rmse = metrics_record[min_rmse_epoch]['RMSE']
        min_rmse_record[name] = {'min_rmse': min_rmse, 'epoch': min_rmse_epoch}

        print(f"Battery {name} - Minimum RMSE: {min_rmse:.4f} at Epoch {min_rmse_epoch}")

        min_rmse_record[name] = {'min_rmse': min_rmse, 'epoch': min_rmse_epoch,
'alpha_3_1': alpha_3_1,
                                 'alpha_3_2': alpha_3_2}

    return score_list, fixed_result_list, moving_result_list, min_rmse_record
```

5.5 主函数

这段代码实现了一个基于多个种子和超参数组合的模型评估过程，通过遍历不同的超参数设置，计算并比较了不同种子和参数下的模型性能。最终，它输出了最佳种子和最优超参数配置的预测结果，并将相关评估指标保存到 CSV 文件中，便于后续分析和比较。此外，还进行了性能汇总和分组分析，帮助研究人员了解哪些超参数配置在锂电池健康状态预测

任务中最为有效。

```
def evaluate_and_print3_2(seeds, lr=lr, feature_size=feature_size, feature_num=feature_num,
hidden_dim=hidden_dim,
                          num_layers=num_layers, weight_decay=weight_decay,
                          epochs=epochs, device=('cuda:0' if torch.cuda.is_available() else 'cpu'),
                          alpha_3_1=1, alpha_3_2=1, K=16):
    model_fixed_preds = []
    model_moving_preds = []
    seed_metrics = []
    all_min_rmse_records = []

    seed_average_min_rmse = {}
    seed_battery_metrics = {}

    for seed in seeds:
        setup_seed(seed)
        score_list, fixed_result_list, moving_result_list, min_rmse_record = train_2(
            lr=lr, feature_size=feature_size, feature_num=feature_num,
            hidden_dim=hidden_dim, num_layers=num_layers, nhead=nhead,
            dropout=dropout, noise_level=noise_level, weight_decay=weight_decay,
            epochs=epochs, seed=seed, device=device,
            alpha_3_1=alpha_3_1, alpha_3_2=alpha_3_2, K=K
        )
        model_fixed_preds.append(fixed_result_list)
        model_moving_preds.append(moving_result_list)
        all_min_rmse_records.append(min_rmse_record)

        # 收集每个电池的指标
        battery_metrics = min_rmse_record  # 包含每个电池的 min_rmse, min_re, min_mae,
epoch

        # 计算 4 个电池的平均最小 RMSE
        total_min_rmse = sum([battery_metrics[name]['min_rmse'] for name in Battery_list])
        avg_min_rmse = total_min_rmse / len(Battery_list)
        seed_average_min_rmse[seed] = avg_min_rmse
        seed_battery_metrics[seed] = battery_metrics

    # 找到平均最小 RMSE 最低的种子
    best_seed = min(seed_average_min_rmse, key=seed_average_min_rmse.get)
    best_seed_metrics = seed_battery_metrics[best_seed]
    avg_min_re = sum([best_seed_metrics[name]['min_re'] for name in Battery_list]) /
len(Battery_list)
    avg_min_mae = sum([best_seed_metrics[name]['min_mae'] for name in Battery_list]) /
len(Battery_list)
```

```
    avg_min_rmse = seed_average_min_rmse[best_seed]

    # 返回最佳种子的预测结果
    seed_index = seeds.index(best_seed)
    best_fixed_preds = model_fixed_preds[seed_index]
    best_moving_preds = model_moving_preds[seed_index]

    # 将最佳种子的电池指标保存到 'nasa 系列_多个种子各自分数.csv'
    with open('nasa 系列_多个种子各自分数.csv', 'w', newline='') as csvfile:
        fieldnames = ['Battery', 'Seed', 'Min_RE', 'Min_MAE', 'Min_RMSE', 'Epoch', 'alpha_3_1',
'lr']
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

        writer.writeheader()
        for battery_name in Battery_list:
            metrics = best_seed_metrics[battery_name]
            writer.writerow({
                'Battery': battery_name,
                'Seed': best_seed,
                'Min_RE': metrics['min_re'],
                'Min_MAE': metrics['min_mae'],
                'Min_RMSE': metrics['min_rmse'],
                'Epoch': metrics['epoch'],
                'alpha_3_1': alpha_3_1,
                'lr': lr
            })

        # 写入平均最小 RE、MAE、RMSE
        writer.writerow({
            'Battery': 'Average',
            'Seed': best_seed,
            'Min_RE': avg_min_re,
            'Min_MAE': avg_min_mae,
            'Min_RMSE': avg_min_rmse,
            'Epoch': '',
            'alpha_3_1': alpha_3_1,
            'lr': lr
        })

    # 保存最佳种子的电池预测结果
    for i, battery_name in enumerate(Battery_list):
        test_data = Battery[battery_name][1]
        fixed_predict_data = best_fixed_preds[i][-len(test_data):]
        moving_predict_data = best_moving_preds[i][-len(test_data):]
```

```
        x = list(range(len(test_data)))
        threshold = [Rated_Capacity * 0.7] * len(test_data)

        with open(f'nasa 系列，单个电池多个种子平均分数_battery_{battery_name}.csv', 'w',
newline='') as csvfile:
            fieldnames = ['x', 'test_data', 'fixed_predict_data', 'moving_predict_data', 'threshold']
            writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

            writer.writeheader()
            for k in range(len(test_data)):
                row = {
                    'x': x[k],
                    'test_data': test_data[k],
                    'fixed_predict_data': fixed_predict_data[k],
                    'moving_predict_data': moving_predict_data[k],
                    'threshold': threshold[k]
                }
                writer.writerow(row)

    # 打印摘要
    print(f"\n 最佳种子: {best_seed}，平均最小 RMSE: {avg_min_rmse:.4f}")
    for battery_name in Battery_list:
        metrics = best_seed_metrics[battery_name]
        print(f"电池 {battery_name} - 最小 RMSE: {metrics['min_rmse']:.4f}, 最小 RE:
{metrics['min_re']:.4f}, 最小 MAE: {metrics['min_mae']:.4f}，发生在 Epoch {metrics['epoch']}")

    return all_min_rmse_records, model_fixed_preds, model_moving_preds



if __name__ == "__main__":
    # 定义需要遍历的参数值
    alpha_3_1_values = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
    lr_values = [0.005]
    seeds = [0,1,3]
    results = []
    start_time = time.time()

    for alpha_3_1 in alpha_3_1_values:
        alpha_3_2 = 1 - alpha_3_1  # 计算 alpha_3_2
        # alpha_3_2 = 0  # 计算 alpha_3_2
        # alpha_3_2 = 0  # 计算 alpha_3_2
        print(f"alpha_3_1: {alpha_3_1}, calculated alpha_3_2: {alpha_3_2}")  # 打印计算得到的
alpha_3_2 值
```

```
        for lr in lr_values:
            print(f"\nEvaluating for alpha_3_1: {alpha_3_1}, lr: {lr}")
            # 调用修改后的 evaluate_and_print3_2 函数
            all_min_rmse_records, model_fixed_preds, model_moving_preds =
evaluate_and_print3_2(
                seeds, lr=lr, alpha_3_1=alpha_3_1, alpha_3_2=alpha_3_2, K=K
            )

            # 记录每个种子的平均最小 RMSE
            seed_average_min_rmse = {}
            for seed_index, min_rmse_record in enumerate(all_min_rmse_records):
                seed = seeds[seed_index]
                total_min_rmse = sum([metrics['min_rmse'] for metrics in
min_rmse_record.values()])
                avg_min_rmse = total_min_rmse / len(Battery_list)
                seed_average_min_rmse[seed] = avg_min_rmse

                # 记录每个电池的最小 RMSE、RE、MAE
                for battery_name, metrics in min_rmse_record.items():
                    result = {
                        'alpha_3_1': alpha_3_1,
                        'alpha_3_2': alpha_3_2,
                        'lr': lr,
                        'seed': seed,
                        'battery_name': battery_name,
                        'min_rmse': metrics['min_rmse'],
                        'min_re': metrics['min_re'],
                        'min_mae': metrics['min_mae'],
                        'epoch': metrics['epoch']
                    }
                    results.append(result)
                    print(f"\nalpha_3_1: {alpha_3_1}, alpha_3_2: {alpha_3_2}, lr: {lr}, Seed {seed},
Battery {battery_name} - "
                          f"Min RMSE: {metrics['min_rmse']:.4f}, Min RE: {metrics['min_re']:.4f}, "
                          f"Min MAE: {metrics['min_mae']:.4f} at Epoch {metrics['epoch']}")

            # 找到平均最小 RMSE 最低的种子（小组）
            best_seed = min(seed_average_min_rmse, key=seed_average_min_rmse.get)
            best_avg_min_rmse = seed_average_min_rmse[best_seed]
            print(f"\nBest Seed: {best_seed} with Average Min RMSE: {best_avg_min_rmse:.4f},
'alpha_3_1': {alpha_3_1}, 'alpha_3_2': {alpha_3_2}, 'lr': {lr}")

            # 保存最佳种子的结果到文件
```

```
        best_seed_metrics = all_min_rmse_records[seeds.index(best_seed)]
        avg_min_re = sum([metrics['min_re'] for metrics in best_seed_metrics.values()]) /
len(Battery_list)
        avg_min_mae = sum([metrics['min_mae'] for metrics in best_seed_metrics.values()]) /
len(Battery_list)
        avg_min_rmse = best_avg_min_rmse

        # 将最佳种子的电池指标保存到 '论文 1，nasa 系列_多个种子各自分数.csv'
        with open('论文 1，nasa 系列_多个种子各自分数.csv', 'w', newline='') as csvfile:
            fieldnames = ['Battery', 'Seed', 'Min_RE', 'Min_MAE', 'Min_RMSE', 'Epoch',
'alpha_3_1', 'alpha_3_2', 'lr']
            writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

            writer.writeheader()
            for battery_name in Battery_list:
                metrics = best_seed_metrics[battery_name]
                writer.writerow({
                    'Battery': battery_name,
                    'Seed': best_seed,
                    'Min_RE': metrics['min_re'],
                    'Min_MAE': metrics['min_mae'],
                    'Min_RMSE': metrics['min_rmse'],
                    'Epoch': metrics['epoch'],
                    'alpha_3_1': alpha_3_1,
                    'alpha_3_2': alpha_3_2,
                    'lr': lr
                })

            # 写入平均最小 RE、MAE、RMSE
            writer.writerow({
                'Battery': 'Average',
                'Seed': best_seed,
                'Min_RE': avg_min_re,
                'Min_MAE': avg_min_mae,
                'Min_RMSE': avg_min_rmse,
                'Epoch': '',
                'alpha_3_1': alpha_3_1,
                'alpha_3_2': alpha_3_2,
                'lr': lr
            })

        # 保存最佳种子的电池预测结果
        fixed_preds = model_fixed_preds[seeds.index(best_seed)]
        moving_preds = model_moving_preds[seeds.index(best_seed)]
```

```
            for i, battery_name in enumerate(Battery_list):
                test_data = Battery[battery_name][1]
                fixed_predict_data = fixed_preds[i][-len(test_data):]
                moving_predict_data = moving_preds[i][-len(test_data):]
                x = list(range(len(test_data)))
                threshold = [Rated_Capacity * 0.7] * len(test_data)

                with open(f'论文 1，nasa 系列，单个电池多个种子平均分数
_battery_{battery_name}.csv', 'w', newline='') as csvfile:
                    fieldnames = ['x', 'test_data', 'fixed_predict_data', 'moving_predict_data',
'threshold']
                    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

                    writer.writeheader()
                    for k in range(len(test_data)):
                        row = {
                            'x': x[k],
                            'test_data': test_data[k],
                            'fixed_predict_data': fixed_predict_data[k],
                            'moving_predict_data': moving_predict_data[k],
                            'threshold': threshold[k]
                        }
                        writer.writerow(row)

    # 汇总打印结果
    print("\nSummary of all results:")
    for result in results:
        print(f"alpha_3_1: {result['alpha_3_1']}, alpha_3_2: {result['alpha_3_2']}, lr: {result['lr']},
Seed: {result['seed']}, "
              f"Battery: {result['battery_name']}, Min RMSE: {result['min_rmse']:.4f}, "
              f"Min RE: {result['min_re']:.4f}, Min MAE: {result['min_mae']:.4f}, Epoch:
{result['epoch']}")

    # **添加以下代码来计算并打印平均分数**
    # 根据 alpha_3_1、alpha_3_2、lr、seed 进行分组
    from collections import defaultdict

    grouped_results = defaultdict(lambda: {'rmse': [], 're': [], 'mae': []})

    for result in results:
        key = (result['alpha_3_1'], result['alpha_3_2'], result['lr'], result['seed'])
        grouped_results[key]['rmse'].append(result['min_rmse'])
        grouped_results[key]['re'].append(result['min_re'])
```

```
        grouped_results[key]['mae'].append(result['min_mae'])

    # 计算并打印每个组合的平均分数
    print("\nAverage scores per combination of alpha_3_1, alpha_3_2, lr, seed:")
    for key, metrics in grouped_results.items():
        avg_rmse = sum(metrics['rmse']) / len(metrics['rmse'])
        avg_re = sum(metrics['re']) / len(metrics['re'])
        avg_mae = sum(metrics['mae']) / len(metrics['mae'])
        alpha_3_1, alpha_3_2, lr, seed = key
        print(f"alpha_3_1: {alpha_3_1}, alpha_3_2: {alpha_3_2}, lr: {lr}, Seed: {seed}, "
              f"Average Min RMSE: {avg_rmse:.4f}, Average Min RE: {avg_re:.4f}, Average Min
MAE: {avg_mae:.4f}")

    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Total runtime: {elapsed_time:.2f} seconds")
```