1. 数据预处理

In [1]:

```
import numpy as np
import scipy.io
from datetime import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

1.1 数据读取函数定义

注： mat 文件数据结构见 BatteryDataset 下的 txt 文件

In [2]:

```
def TimeConvert(hmm):
    """
        转换时间格式，将字符串转换成 datatime 格式
    Args:
        hmm: 待输入的原始时间数据 (ndarray)
    Returns:
        标准化后的时间数据
    """
    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)

def LoadMat(mat_file):
    """
        加载 mat 文件数据
    Args:
        mat_file: 待加载的文件路径 (string)
    Returns:
        读取的数据 (list)，其中每个元素为一个嵌套的 dict 类型
    """
    # 函数返回一个字典，其中键是 mat 文件中变量的名称，值是对应的数据数组
    data = scipy.io.loadmat(mat_file)
    # 从文件路径中提取文件名(不包含扩展名),用于访问字典的值
    fileName = mat_file.split('/')[-1].split('.')[0]
    col = data[fileName] # 获取整个数据(一个(1 x N)的四层结构化数组)
    col = col[0][0][0][0] # 去除冗余维度，访问包含所有循环数据的(616,)结构化数组
    size = col.shape[0] # 获取数组的大小(cycle 的数量)
    # print("data['B0005'].dtype:",data['B0005'].dtype,"value:",data['B0005'])
    # print("data['B0005'][0][0][0][0].dtype:",data['B0005'][0][0][0][0].dtype,
    #       "value:",data['B0005'][0][0][0][0])
    # print("data['B0005'][0][0][0][0][0][3][0].dtype:",data['B0005'][0][0][0][0][0][3][0].dty
    #       "value:",data['B0005'][0][0][0][0][0][3][0])

    data = []
    for i in range(size): # 遍历每个 cycle 的数据
        """ dtype.fields 方法用于访问 NumPy 结构化数组的字段信息，它返回一个字典，其中:
        键: 是结构化数组中每个字段的名称（字符串）;
        值: 是描述每个字段的元组，包含字段的数据类型、字节偏移量以及可选的标题。 """
        k = list(col[i][3][0].dtype.fields.keys()) # 获取结构化数组(data 字段)中所有子字段名称
        d1, d2 = {}, {}
        if str(col[i][0][0]) != 'impedance': # 去除 impedance 类型的数据
            for j in range(len(k)): # 遍历(data 字段)数组中的每个子字段
                t = col[i][3][0][0][j][0] # 获取该字段的数组数据
                l = [t[m] for m in range(len(t))] # 遍历提取数组中每个数据转为列表
                d2[k[j]] = l # 保存该数据及其对应的字段名称(以键值对的形式存在)
        # 将每个样本(cycle)的类型、温度、时间和数据存储到字典 d1 中
        d1['type'], d1['temp'], d1['time'], d1['data'] = \
```

```python
            str(col[i][0][0]), int(col[i][1][0]), str(TimeConvert(col[i][2][0])), d2
        data.append(d1)

    return data

def GetBatteryCapacity(Battery):
    """
        获取单个锂电池的容量数据
    Args:
        Battery: 单个电池的数据 (dict)
    Returns:
        获取的电池容量数据 (list), 包含两个元素, 第一个为放电周期, 第二个为容量数据
    """
    cycle, capacity = [], []
    i = 1
    for Bat in Battery:
        if Bat['type'] == 'discharge': # 放电状态下获取容量数据
            capacity.append(Bat['data']['Capacity'][0])
            cycle.append(i)
            i += 1
    return [cycle, capacity]

def GetBatteryValues(Battery, Type='charge'):
    """
        获取单个锂电池充电或放电时的测试数据(默认为充电状态的数据)
    Args:
        Battery: 单个电池的数据 (dict)
        Type: 指定要读取的数据类型 (string)
    Returns:
        获取的电池数据,  list 类型
    """
    data = []
    for Bat in Battery:
        if Bat['type'] == Type:
            data.append(Bat['data'])
    return data
```

In [3]:
```python
# 用于测试数据
# Battery_list = 'B0005'
# dir_path = r'BatteryDataset/'

# path = dir_path + Battery_list + '.mat'
# data = LoadMat(path)
```

1.2 读取数据

In [4]:
```python
Battery_list = ['B0005', 'B0006', 'B0007', 'B0018']
dir_path = r'BatteryDataset/'

capacity, charge, discharge = {}, {}, {}
for name in Battery_list:
    print('Loading Dataset ' + name + '.mat ...')
    path = dir_path + name + '.mat'
    data = LoadMat(path)
    capacity[name] = GetBatteryCapacity(data) # 放电时的容量数据
    charge[name] = GetBatteryValues(data, 'charge') # 充电数据
    discharge[name] = GetBatteryValues(data, 'discharge') # 放电数据
```

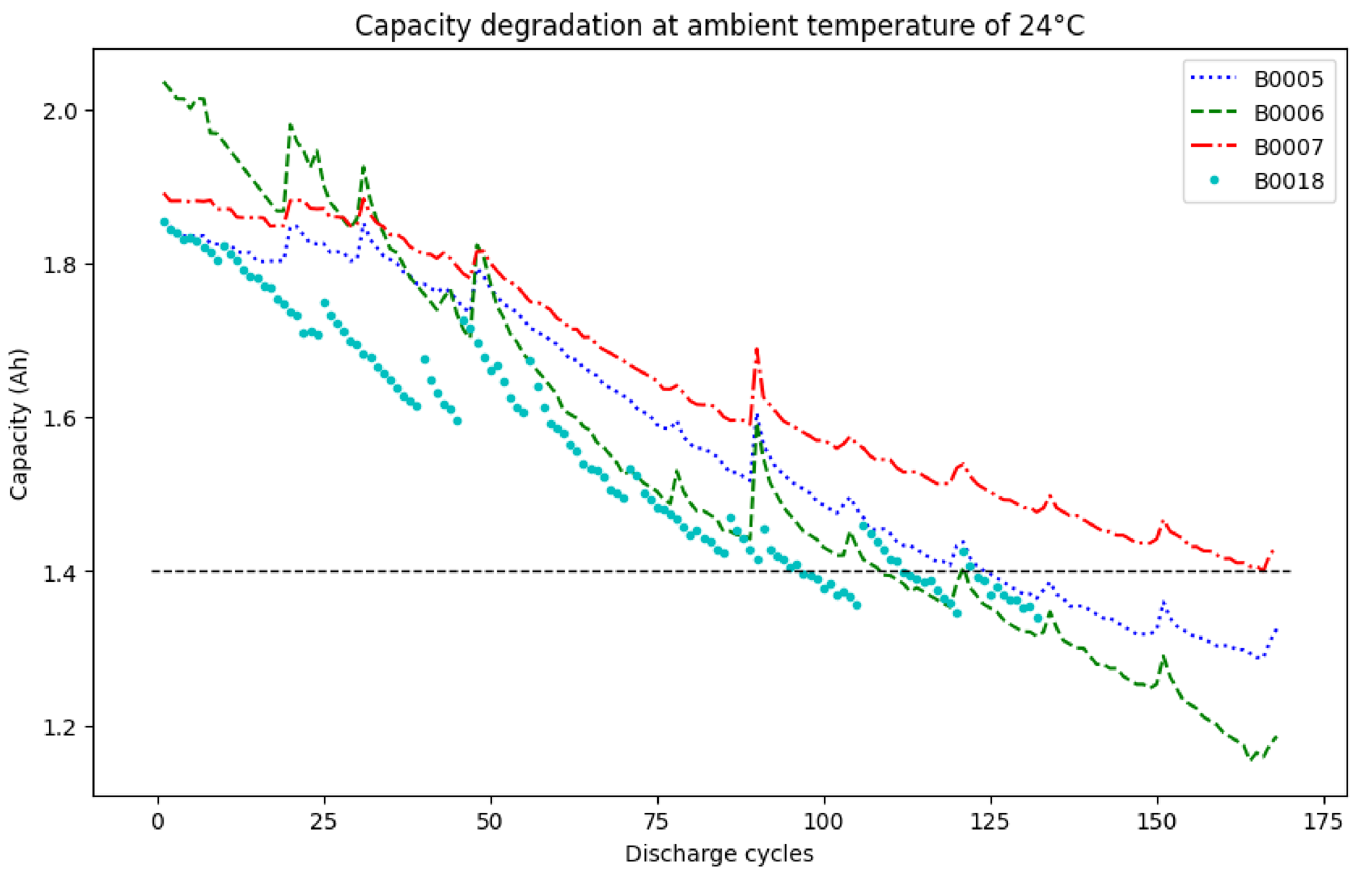
```
Loading Dataset B0005.mat ...
Loading Dataset B0006.mat ...
Loading Dataset B0007.mat ...
Loading Dataset B0018.mat ...
```

1.3 数据展示

1.3.1 不同电池容量 vs. 充放电周期曲线

In [5]:
```
fig, ax = plt.subplots(1, figsize=(10, 6))
color_list = ['b:', 'g--', 'r-.', 'c.']
c = 0
for name,color in zip(Battery_list, color_list):
    df_result = capacity[name]
    ax.plot(df_result[0], df_result[1], color, label=name)
# 临界点直线(电池容量下降30%则认为报废)
plt.plot([-1,170],[2.0*0.7,2.0*0.7],c='black',lw=1,ls='--')
ax.set(xlabel='Discharge cycles', ylabel='Capacity (Ah)',
       title='Capacity degradation at ambient temperature of 24°C')
plt.legend()
```

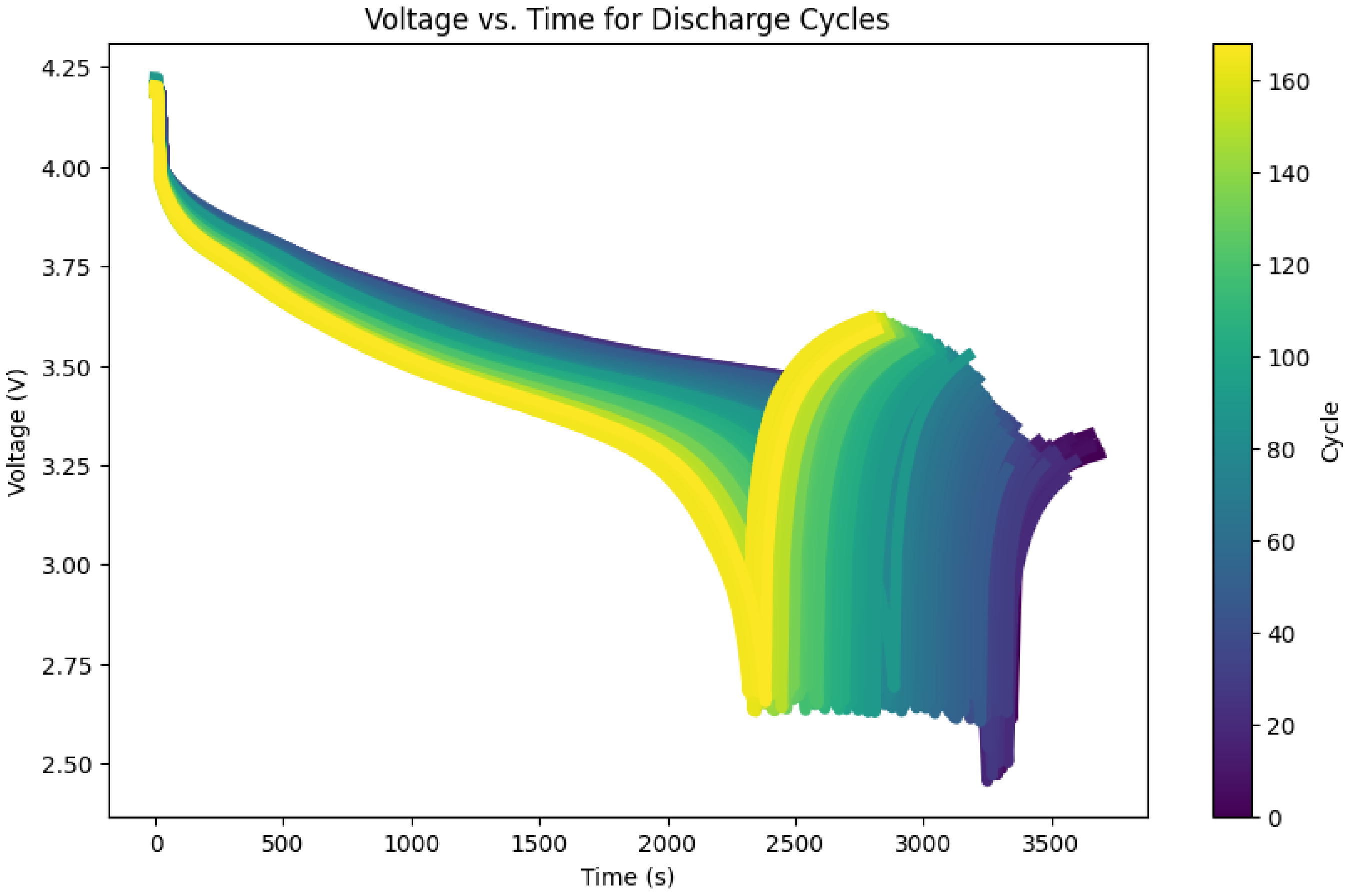
Out[5]: <matplotlib.legend.Legend at 0x7fbae69d47c0>



1.3.2 不同放电周期下，B0005号电池电压 vs. 放电周期曲线

In [6]:
```
fig, ax = plt.subplots(1, figsize=(10, 6))
cmap = plt.get_cmap('viridis') # 选择合适的颜色映射, 'hsv', 'jet', 'viridis' 等

name = 'B0005' # 仅绘制第 B0005 号电池
for i, cycle_data in enumerate(discharge[name]):
    # 使用 plot 绘制，并根据循环次数着色
    ax.plot(cycle_data['Time'], cycle_data['Voltage_measured'],
            c=cmap(i / len(discharge[name])), linewidth=5.0)
# 添加颜色条
sm = plt.cm.ScalarMappable(cmap=cmap,
                           norm=plt.Normalize(vmin=0, vmax=len(discharge[name])))
sm.set_array([])  # 这是为了让colorbar工作，即使没有明确的映射数组
cbar = plt.colorbar(sm, ax=ax)
cbar.set_label('Cycle') # 添加颜色条标签

ax.set(xlabel='Time (s)', ylabel='Voltage (V)',
       title='Voltage vs. Time for Discharge Cycles')
```

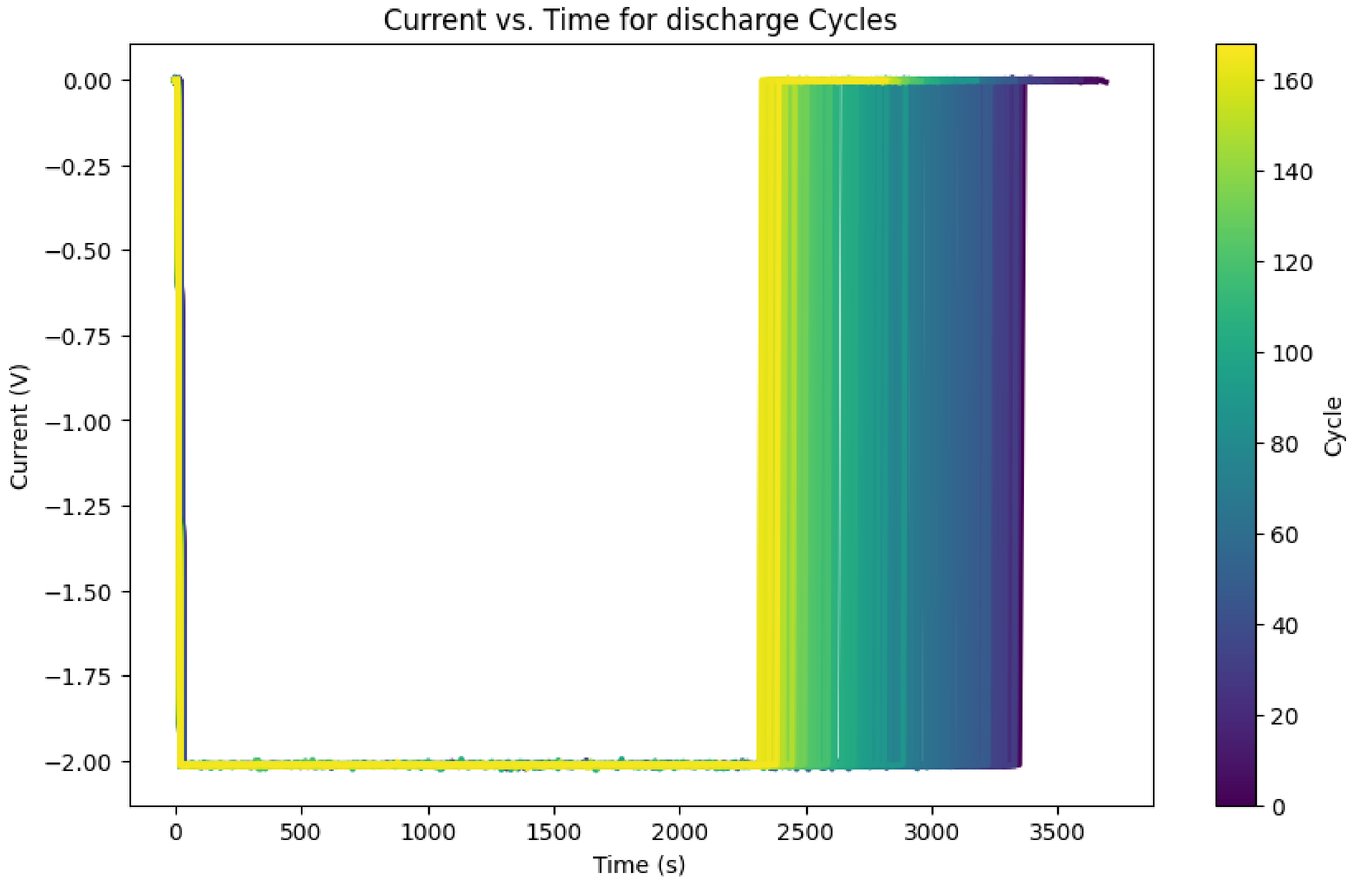
```
Out[6]: [Text(0.5, 0, 'Time (s)'),
 Text(0, 0.5, 'Voltage (V)'),
 Text(0.5, 1.0, 'Voltage vs. Time for Discharge Cycles')]
```



1.3.3 不同放电周期下，B0005号电池电流 vs. 充电周期曲线

```
In [7]: fig, ax = plt.subplots(1, figsize=(10, 6))
cmap = plt.get_cmap('viridis') # 选择合适的颜色映射, 'hsv', 'jet', 'viridis' 等

name = 'B0005' # 仅绘制第 B0005 号电池
for i, cycle_data in enumerate(discharge[name]):
    # 使用 plot 绘制, 并根据循环次数着色
    ax.plot(cycle_data['Time'], cycle_data['Current_measured'],
            c=cmap(i / len(charge[name])), linewidth=2.5)
# 添加颜色条
sm = plt.cm.ScalarMappable(cmap=cmap,
                           norm=plt.Normalize(vmin=0, vmax=len(discharge[name])))
sm.set_array([])  # 这是为了让colorbar工作, 即使没有明确的映射数组
cbar = plt.colorbar(sm, ax=ax)
cbar.set_label('Cycle') # 添加颜色条标签

ax.set(xlabel='Time (s)', ylabel='Current (V)',
       title='Current vs. Time for discharge Cycles')
```

```
Out[7]: [Text(0.5, 0, 'Time (s)'),
 Text(0, 0.5, 'Current (V)'),
 Text(0.5, 1.0, 'Current vs. Time for discharge Cycles')]
```



1.4 创建数据集

1.4.1 数据集类定义

In [8]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, Subset, ConcatDataset
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
from sklearn.metrics import mean_absolute_error
from math import sqrt
import random
import os
# 注册设备
device = "cuda" if torch.cuda.is_available() else "cpu"
```

In [9]:
```python
# 定义数据集类
class TimeSeriesDataset(Dataset):
    def __init__(self, data, window_size):
        self.window_size = window_size
        self.data = torch.tensor(data, dtype=torch.float32).to(device)
        # 计算最大索引
        self.max_index = self.data.shape[0] - self.window_size - 1

    def __len__(self):
        return self.max_index + 1 #  返回有效数据长度

    def __getitem__(self, index):
        if index > self.max_index:
            raise IndexError(f"Index {index} is out of bounds."
                             f"Max index is {self.max_index}")
        x = self.data[index:index + self.window_size]
        y = self.data[index + self.window_size]
        return x.unsqueeze(1), y.unsqueeze(0) # 添加一个特征维度
```

1.4.2 数据集获取函数定义

In [10]:
```python
def get_split_dataset(data_dict, name, window_size=8, shuffle=True,
                      capacity_threshold=0.0, train_ratio=0.0, batch_size=32):
    """
        获取分割后的训练集和测试集 DataLoader
    Args:
        data_dict: 字典类型，键为电池名称，值为包含电池信息的元组，
                   其中第二个元素是容量数据列表 (list)
        name: 指定为测试集的电池数据名称 (str)
        window_size: 用于创建时间序列的窗口大小 (int)
        shuffle: 是否打乱训练集 (bool)
        train_ratio: 用于训练数据划分的比例 (float32)
        capacity_threshold: 用于训练数据划分的阈值 (float32)
        batch_size: 训练的批大小 (int)
    Returns:
        包含训练数据和测试数据 DataLoader 的元组。
    """
    data = data_dict[name][1]
    # 创建数据集
    dataset = TimeSeriesDataset(data, window_size)
    # 划分数据集
    if capacity_threshold > 0: # 优先使用阈值前的数据训练，阈值后的数据测试
        max_capacity = max(data)
        capacity = max_capacity * capacity_threshold
        point = next((i for i, val in enumerate(data) if val < capacity), None)
    else: # 否则按照指定的比例进行划分
        if (0 < train_ratio <= 1):
            point = int(len(data) * train_ratio)
        else:
            raise ValueError("Train ratio must be between 0 and 1.")
    # 使用 Subset 创建训练集和验证集，保持时间顺序
    train_dataset = Subset(dataset, range(point))
    test_dataset = Subset(dataset, range(point, len(dataset)))

    # 创建 DataLoader
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=shuffle)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    return train_loader, test_loader

def get_data(data_dict, name, window_size=8, shuffle=True, batch_size=32):
    """
        留一法获取训练集和测试集 DataLoader,每次留一个电池的数据作为测试集
    Args:
```

```
        data_dict: 字典类型，键为电池名称，值为包含电池信息的元组，
                其中第二个元素是容量数据列表 (list)
        name: 指定为测试集的电池数据名称 (str)
        window_size: 用于创建时间序列的窗口大小 (int)
        shuffle: 是否打乱训练集 (bool)
        batch_size: 训练的批大小 (int)
    Returns:
        包含训练数据和测试数据 DataLoader 的元组。
    """
    test_data = data_dict[name][1]
    test_dataset = TimeSeriesDataset(test_data, window_size)

    train_datasets = []
    for k, v in data_dict.items():
        if k != name:
            dataset = TimeSeriesDataset(v[1], window_size)
            train_datasets.append(dataset)
    train_dataset = ConcatDataset(train_datasets) # 使用 ConcatDataset 拼接多个数据集

    # 创建 DataLoader
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=shuffle)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    # 打印 DataLoader
    # print("train_loader_num:", len(train_loader.dataset))
    # print("test_loader_num:", len(test_loader.dataset))
    # for x, y in train_loader:
    #     print("x:", x.shape) # 输出: x: (batch_size, window_size, num_features)
    #     print("y:", y.shape) # 输出: y: (batch_size, 1)
    #     break
    return train_loader, test_loader
```

In [11]:

```
# window_size = 8
# for i in range(len(Battery_list)):
#     name = Battery_list[i]
#     get_data(capacity, name, window_size)
```

2. 模型建立

In [12]:

```
# 定义CNN层
class CNNLayer(nn.Module):
    def __init__(self, num_channels, out_dim, kernel_size=1):
        super(CNNLayer, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=num_channels,
                               out_channels=out_dim,
                               kernel_size=kernel_size)

    def forward(self, x):
        x = F.relu(self.conv1(x)) # x.shape([batch_size, out_dim, 1])
        return x

# 定义LSTM层
class LSTMLayer(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, bidirectional):
        super(LSTMLayer, self).__init__()
        self.hidden_size = hidden_dim
        self.num_layers = num_layers
        self.bidirectional = bidirectional
        self.lstm = nn.LSTM(input_size=input_dim, hidden_size=hidden_dim,
                            num_layers=num_layers, batch_first=True,
                            bidirectional=bidirectional)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers*2 if self.bidirectional else
```

```python
                         self.num_layers, x.size(0),
                         self.hidden_size).to(device) # 初始化隐藏状态h0
        c0 = torch.zeros(self.num_layers*2 if self.bidirectional else
                         self.num_layers, x.size(0),
                         self.hidden_size).to(device)  # 初始化记忆状态c0
        output, (hidden, cell) = self.lstm(x, (h0, c0))
        # output.shape([batch_size, 1, hidden_dim*2 if bidirectional else hidden_dim])
        return output

# 定义Attention层
class AttentionLayer(nn.Module):
    def __init__(self, feature_dim, step_dim):
        super(AttentionLayer, self).__init__()
        self.attention = nn.Linear(feature_dim, step_dim)
        self.context_vector = nn.Linear(step_dim, 1, bias=False)

    def forward(self, x):
        # 将输出值限制在 -1 到 1 之间, shape: [batch_size, feature_dim, step_dim]
        attention_weights = torch.tanh(self.attention(x))
        # 为每个时间步计算一个未归一化的注意力权重, shape: [batch_size, 1]
        attention_weights = self.context_vector(attention_weights).squeeze(2)
        attention_weights = F.softmax(attention_weights, dim=1) # 对权重归一化
        # 将注意力权重与输入 x 相乘, shape: [batch_size, feature_dim]
        context_vector = torch.bmm(attention_weights.unsqueeze(1), x).squeeze(1)
        return context_vector, attention_weights

# 建立组合模型 CNN-LSTM-Attention
class CLAM(nn.Module):
    def __init__(self, num_channels, out_dim, kernel_size, hidden_dim,
                 num_layers, bidirectional, step_dim, output_dim):
        super(CLAM, self).__init__()
        self.cnn = CNNLayer(num_channels, out_dim, kernel_size)
        self.lstm = LSTMLayer(out_dim, hidden_dim,
                              num_layers, bidirectional)
        self.attention = AttentionLayer(hidden_dim * 2 if bidirectional else
                                        hidden_dim, step_dim)
        self.fc = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim, output_dim)

    def forward(self, x):
        x = x.transpose(1, 2) # 交换维度
        x = self.cnn(x)
        x = x.transpose(1, 2)
        x = self.lstm(x)
        x, _ = self.attention(x)
        x = self.fc(x)
        # x = self.fc(x[:, -1, :]) # 取序列最后一个时间步的输出作为预测
        return x

    # 建立组合模型 CNN-LSTM
class CLM(nn.Module):
    def __init__(self, num_channels, out_dim, kernel_size, hidden_dim,
                num_layers, bidirectional, output_dim):
        super(CLM, self).__init__()
        self.cnn = CNNLayer(num_channels, out_dim, kernel_size)
        self.lstm = LSTMLayer(out_dim, hidden_dim,
                              num_layers, bidirectional)
        self.fc = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim, output_dim)

    def forward(self, x):
        x = x.transpose(1, 2) # 交换维度
        x = self.cnn(x)
        x = x.transpose(1, 2)
        x = self.lstm(x)
        # x = self.fc(x)
        x = self.fc(x[:, -1, :]) # 取序列最后一个时间步的输出作为预测
        return x
```

```python
# 建立组合模型 LSTM-Attention
class LAM(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers,
                 bidirectional, step_dim, output_dim):
        super(LAM, self).__init__()
        self.lstm = LSTMLayer(input_dim, hidden_dim,
                              num_layers, bidirectional)
        self.attention = AttentionLayer(hidden_dim * 2 if bidirectional else
                                        hidden_dim, step_dim)
        self.fc = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim, output_dim)

    def forward(self, x):
        x = self.lstm(x)
        x, _ = self.attention(x)
        x = self.fc(x)
        # x = self.fc(x[:, -1, :]) # 取序列最后一个时间步的输出作为预测
        return x

    # 建立模型 LSTM
class LM(nn.Module):
    def __init__(self, input_dim, hidden_dim,
                 num_layers, bidirectional, output_dim):
        super(LM, self).__init__()
        self.lstm = LSTMLayer(input_dim, hidden_dim,
                              num_layers, bidirectional)
        self.fc = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim, output_dim)

    def forward(self, x):
        x = self.lstm(x)
        # x = self.fc(x)
        x = self.fc(x[:, -1, :]) # 取序列最后一个时间步的输出作为预测
        return x
```

3. 模型训练

3.1 训练函数定义

3.1.1 模型获取函数和批训练函数定义

In [13]:

```python
def get_model(num_channels, out_dim, kernel_size, feature_dim, hidden_dim, num_layers,
              bidirectional, step_dim, output_dim, learn_rate, model_name='BNO-CBiLAM'):
    """
        获取模型，并指定优化器和损失计算方法
    Args:
        num_channels: 输入模型的通道数，即窗口大小
        out_dim: 卷积的输出特征维度
        feature_dim: 输入数据的特征维度
        kernel_size: 卷积核大小
        hidden_dim: LSTM隐藏状态维度
        num_layers: LSTM层的数目
        bidirectional: 是否使用双向LSTM
        step_dim: 时间步维度
        output_dim: 输出维度(预测目标维度)
        model_name: 指定的模型名称
        learn_rate: 学习率
    Returns:
        指定的模型、损失函数和优化器的元组
    """
    if model_name == 'BNO-CBiLA':
        model = CLAM(num_channels, out_dim, kernel_size, hidden_dim,
                     num_layers, bidirectional, step_dim, output_dim)
```

```python
    elif model_name == 'CBiLA':
        model = CLAM(num_channels, out_dim, kernel_size, hidden_dim,
                num_layers, bidirectional, step_dim, output_dim)
    elif model_name == 'CBiL':
        model = CLM(num_channels, out_dim, kernel_size, hidden_dim,
                    num_layers, bidirectional, output_dim)
    elif model_name == 'BiLSTM':
        model = LM(feature_dim, hidden_dim, num_layers,
                    bidirectional, output_dim)
    else:
        model = LM(feature_dim, hidden_dim, num_layers,
                  bidirectional=False, output_dim=output_dim)
    loss_fn = nn.MSELoss() # 使用均方误差
    optimizer = optim.Adam(model.parameters(),
                          lr=learn_rate, betas=(0.5,0.999)) # 使用Adam优化器
    return model, loss_fn, optimizer


def train_batch(x, y, model, optimizer, loss_fn):
    """
        批训练函数
    Args:
        x: 输入的训练数据
        y: 输入的真实目标数据
        model: 指定的模型
        optimizer: 指定的优化器
        loss_fn: 指定的损失函数
    Returns:
        计算的损失标量
    """
    model.train() # 设置为训练
    prediction = model(x) # 输入数据
    # print("Prediction shape:", prediction.shape)
    batch_loss = loss_fn(prediction, y) # 计算损失
    batch_loss.backward() # 进行反向传播
    optimizer.step() # 梯度下降
    optimizer.zero_grad() # 清空梯度
    return batch_loss.item()
```

3.1.2 评估函数定义

In [14]:

```python
def relative_error(y_test, y_predict, threshold):
    """
        计算预测值与真实值之间在达到特定阈值时的相对误差
    Args:
        y_test: 真实的电池容量衰减数据
        y_predict: 模型预测的电池容量衰减数据
        threshold: 定义电池寿命结束的容量阈值
    Returns:
        计算的相对误差分数
    """
    true_re, pred_re = len(y_test), 0

    for i in range(len(y_test) - 1):
        if y_test[i] <= threshold >= y_test[i+1]:
            true_re = i - 1 # 第一个下降到阈值前的数据的放电次数
            break
    for i in range(len(y_predict) - 1):
        if y_predict[i] <= threshold:
            pred_re = i - 1 # 预测的次数
            break
    # 计算相对误差，公式为 |真实剩余寿命 - 预测剩余寿命| / 真实剩余寿命
    score = abs(true_re - pred_re) / true_re
    if score > 1: score = 1
```

```python
    return score


def evaluation(y_test, y_predict):
    """
        计算模型的评价指标
    Args:
        y_test: 真实的电池容量衰减数据
        y_predict: 模型预测的电池容量衰减数据
    Returns:
        一个字典，包含所有计算的指标
    """
    rmse = sqrt(mean_squared_error(y_test, y_predict))
    crmsd = np.sqrt(np.mean(((y_test - np.mean(y_test)) - \
                            (y_predict - np.mean(y_predict)))**2))
    mad = np.median(np.abs(y_test - y_predict))
    mae = mean_absolute_error(y_test, y_predict)
    mbe = np.abs(np.mean(y_predict - y_test))
    rsquare = r2_score(y_test, y_predict)
    metrics_dict = {
        "RMSE": rmse,
        "CRMSD": crmsd,
        "MAD": mad,
        "MAE": mae,
        "MBE": mbe,
        "R2": rsquare
    }
    return metrics_dict

def setup_seed(seed):
    """
        设置环境的随机种子，保证训练结果的一致性
    Args:
        seed: 指定的随机种子
    Returns:
        None.
    """
    np.random.seed(seed) # 设置 NumPy 模块的随机种子
    random.seed(seed) # 设置 Python 内置 random 模块的随机种子
    os.environ['PYTHONHASHSEED'] = str(seed) # 设置 Python 的哈希种子
    torch.manual_seed(seed) # 设置 PyTorch 的随机种子
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed) # 为当前 GPU 设置随机种子
        torch.cuda.manual_seed_all(seed) # 为所有可用的 GPU 设置随机种子
        torch.backends.cudnn.benchmark = False # 禁用 cuDNN 的 benchmark 模式
        torch.backends.cudnn.deterministic = True # 启用 cuDNN 的确定性模式
```

3.2 开始训练

3.2.1 贝叶斯优化原理及流程

贝叶斯优化是一种求解函数最优值的算法，它最普遍的使用场景是在机器学习过程中对超参数进行调优。贝叶斯优化算法的核心框架是SMBO (Sequential Model-Based Optimization)，而贝叶斯优化（Bayesian Optimization）狭义上特指代理模型为高斯过程回归模型的SMBO。

- SMBO (Sequential Model-Based Optimization)：
  SMBO是一套优化框架，也是贝叶斯优化所使用的核心框架。它有两个重要组成部分：
  - 一个代理模型（surrogate model），用于对目标函数进行建模。代理模型通常有确定的公式或者能计算梯度，又或者有已知的凹凸性、线性等特性，总之就是更容易用于优化。更泛化地讲，其实它就是一个学习模型，输入是所有观测到的函数值点，训练后可以在给定任意x的情况下给出对 $f(x)$ 的估计。

- 一个优化策略（optimization strategy），决定下一个采样点的位置，即下一步应在哪个输入 x 处观测函数值 $f(x)$ 。通常它是通过采集函数（acquisition function） 来实现的。

- 代理模型（Surrogate Model）: 高斯过程(Caucsian Process) 是一类随机过程 $\{F(x), x \in A\}$ ，它的任意n维分布 $\{F(x_1), \ldots, F(x_n)\}$ （n也是任意的）都服从多元正态分布,即：对任意有限个 $x_1, \ldots, x_n \in A$,$F(x_1), \ldots, F(x_n)$ 的任意线性组合 $a_1 F(x_1) + \ldots + a_n F(x_n)$ 都是一个正态分布。
正如一个正态分布可以通过指定均值和方差来确定，一个高斯过程可以通过指定均值函数 $m(x)$ 和协方差函数 $K(x, x')$ 唯一确定:

$$m(x) = E[F(x)]$$

$$K(x, x') = E[(F(x) - m(x))(F(x') - m(x'))]$$

则高斯过程可以表示为：

$$F(x) \sim GP(m(x), K(x, x'))$$

均值函数定义了每个索引 x 对应的随机变量（同时也是正态分布变量） $F(x)$ 的均值；而协方差函数不仅定义了每个索引的方差 $K(x, x')$ ，还定义了任意两个索引 x_1, x_2 对应的随机变量 $F(x_1)F(x_2)$ 之间的相关性 $K(x_1, x_2)$ 。在高斯过程模型里，协方差函数也被称作核函数（Kernel function）。

- 采集函数（Acquisition Function）
由于代理模型输出了函数 f 的后验分布 $F(x)|F(x_{1:t}) = f(x_{1:t})$ ，我们可以利用这个后验分布去评估下一个采样点应该在哪个位置。由于在采集函数阶段我们讨论的都是后验分布，因此后文中将省略条件部分，提到 $F(x)$ 时指的都是 $F(x)|F(x_{1:t}) = f(x_{1:t})$ 。通常做法是设计一个采集函数 $A(x, F(x)|F(x_{1:t}) = f(x_{1:t}))$ ，它的输入相当于对每个采样点 x 进行打分，分数越高的点越值得被采样。
一般来说，采集函数需要满足下面的要求：

1. 在已有的采样点处采集函数的值更小，因为这些点已经被探索过，再在这些点处计算函数值对解决问题没有什么用
2. 在置信区间更宽（方差更大）的点处采集函数的值更大，因为这些点具有更大的不确定性，更值得探索
3. 对最大（小）化问题，在函数均值更大（小）的点处采集函数的值更大，因为均值是对该点处函数值的估计值，这些点更可能在极值点附近。 有非常多种采集函数可供选择，如：
   - Expected Improvement (EI)
   当我们已经采样过 t 个点之后，总会有一个最优点 x_m ，使得：

   $$f_t^* = max_{i<t} f(x_i) = f(x_m)$$

   假设我们还可以再观测一轮，得到 $F(x) = f(x)$ ，最优点将在 $f(x)$ 和 f_t^* 之间产生。不妨令

   $$[F(x) - f_t^*]^+ = \max(0, F(x) - f_t^*)$$

   由于现在

   $$[F(x) - f_t^*]^+$$

   是一个随机变量，因此我们可以计算它的期望：

   $$\begin{aligned} EI_t(x) &= E[[F(x) - f_t^*]^+] \\ &= \sigma(x)\phi(\frac{\mu(x) - f_t^*}{\sigma(x)}) + (\mu(x) - f_t^*)\Phi(\frac{\mu(x) - f_t^*}{\sigma(x)}) \end{aligned}$$

其中，$\mu(x)$ 和 $\sigma(x)$ 是正态分布 $F(x)$ 的均值和标准差，即后验均值和标准差 $\sqrt{\sigma^2(x)}$ 。
$\varphi(x)$ 为标准正态分布的概率密度函数：

$$\varphi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

而 $\Phi(x)$ 为标准正态分布的分布函数：

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt$$

$EI_t(x)$ 也是一个仅以x为自变量的函数，它的最大值点就是下一个采样点。

$$\hat{x} = \text{argmax}_x EI_t(x)$$

由于 $EI_t(x)$ 有公式，计算不费劲，也可以求梯度，找到它的最大值/极大值有很多种现成的方案可以做到，相比于求原目标函数 $f(x)$ 的最值要简单得多。

- 贝叶斯优化的一般步骤：
  Step1: 定义需要拟合的目标函数f(x)及其x定义域（自变量不一定是一个，也可能是很多个）；
  Step2: 确定有限个(n)个观测点，并求解出这些观测点的观测值，目标函数值；
  Step3: 根据有限个观测值对目标函数进行估计，计算该次估计的最大值或最小值（本库使用的是最大值）；
  Step4: 根据某种规则（本库使用的是高斯过程），以确定下一个需要计算的观测点；
  Step5: 重复以上2~4步骤，直至达到观测阈值或资源耗尽（指定观测次数）。

3.2.2 贝叶斯优化目标函数定义

In [15]:

```python
from bayes_opt import BayesianOptimization
```

In [16]:

```python
def objective(window_size, out_dim, kernel_size, hidden_dim, num_layers, step_dim,
                          seed, metric, learn_rate, model_list, Rated_Capacity=2.0,
                          bidirectional=True, feature_dim=1, output_dim=1, epochs=100):
    # 将超参数转换为整数
    window_size = int(window_size)
    out_dim = int(out_dim)
    kernel_size = int(kernel_size)
    hidden_dim = int(hidden_dim)
    num_layers = int(num_layers)
    step_dim = int(step_dim)
    setup_seed(seed) # 设置种子

    model_metrics = {}  # 用于存储每个模型的最终指标值
    model_best_score = []
    for model_name in model_list:
        battery_metrics = {} # 存储每个电池的 metrics

        model, loss_fn, optimizer = get_model(feature_dim, out_dim, kernel_size, feature_dim,
                                              hidden_dim, num_layers, bidirectional,
                                              step_dim, output_dim, learn_rate, model_name)
        for i in range(len(Battery_list)): # 四折交叉验证
            name = Battery_list[i]
            train_loader, test_loader = get_data(capacity, name, window_size)
            model = model.to(device) # 注册模型到设备

            train_loss = [0]
            score, best_score = float(1), float(1)
            epoch_metrics = {}
            for epoch in range(epochs):
                train_epoch_loss = []
```

```python
                model.train()  # 设置为训练模式
                for index, batch in enumerate(iter(train_loader)):
                    x, y = batch
                    # 归一化
                    x /= torch.tensor(Rated_Capacity).to(device)
                    y /= torch.tensor(Rated_Capacity).to(device)
                    batch_loss = train_batch(x, y, model, optimizer, loss_fn)
                    train_epoch_loss.append(batch_loss)

                train_epoch_loss = np.array(train_epoch_loss).mean()
                train_loss.append(train_epoch_loss)

                if (epoch + 1) % 10 == 0:
                    model.eval() # 设置为验证模式
                    tesy_pred, test_y = [], []
                    with torch.no_grad():
                        for index, batch in enumerate(iter(test_loader)):
                            x, y = batch
                            # 归一化
                            x /= torch.tensor(Rated_Capacity).to(device)
                            y /= torch.tensor(Rated_Capacity).to(device)
                            pred = model(x)

                            # 将预测值和真实值转换为 NumPy 数组并展平
                            pred_np = (pred * Rated_Capacity).cpu().numpy().flatten()
                            y_np = (y * Rated_Capacity).cpu().numpy().flatten()
                            tesy_pred.extend(pred_np)
                            test_y.extend(y_np)
                    metrics = evaluation(np.array(test_y), np.array(tesy_pred))
                    metrics['RE'] = relative_error(np.array(test_y),
                                        np.array(tesy_pred), Rated_Capacity * 0.7)
                    score = metrics[metric]
                    if epoch + 1 == 10:
                        best_score = score
                        epoch_metrics = metrics
                    else:
                        # 使用指标分数进行模型选择
                        if (batch_loss < 1e-4) and (score < best_score):
                            best_score = score
                            epoch_metrics = metrics
                            # break
            model_best_score.append(best_score)
            battery_metrics[name] = epoch_metrics
        model_metrics[model_name] = battery_metrics
    model_score = np.mean(model_best_score) # 四个电池的平均分数

    # 打印最终指标值
    # for model_name, model_metric in model_metrics.items():
    #     print(f"Model: {model_name}")
    #     for bat_name, bat_metric in model_metric.items():
    #         print(f"  Battery: {bat_name}")
    #         for metric_name, metric_value in bat_metric.items():
    #             print(f"    Metric: {metric_name}: {metric_value:.4f}")
    #     print('------------------------------------------------------------------')

    return model_score
```

In [17]:

```python
def bayes_optim(model_name, pbounds, seed, metric, epochs, hidden_dim=32):
    """
    使用贝叶斯优化搜索最佳超参数。
    """
    optimizer = BayesianOptimization(
        f=lambda window_size, out_dim, kernel_size, num_layers, step_dim, learn_rate: \
            objective(window_size, out_dim, kernel_size, hidden_dim, num_layers, step_dim,
                      seed, metric, learn_rate, model_name, epochs=epochs),
```

```
        pbounds=pbounds,
        random_state=seed,
        verbose=2
    )
    optimizer.maximize(init_points=5, n_iter=50) #  调整 init_points 和 n_iter
    return optimizer.max
```

3.2.3 开始优化

In [ ]:

```
epochs = 100 # 训练次数
seed = 42 # 随机种子
metric = 'R2'
model_list = ['BNO-CBiLA']

# 定义超参数搜索空间
pbounds = {
    'window_size': (8, 32),     # 窗口大小
    'out_dim': (1, 128),        # 卷积输出维度
    'kernel_size': (2, 5),      # 卷积核大小
    'num_layers': (1, 3),       # LSTM层数
    'step_dim': (1, 128),        # 预测步长维度
    'learn_rate': (0.0001, 0.01) # 学习率
}

# 执行贝叶斯优化
best_hps = bayes_optim(model_list, pbounds, seed, metric, epochs, hidden_dim=64)

print("Best hyperparameters:", best_hps)
```

iter	target	kernel...	learn_...	num_la...	out_dim	step_dim	windo w...
1	0.8518	3.124	0.009512	2.464	77.03	20.81	11.74
2	0.7063	2.174	0.008675	2.202	90.93	3.614	31.28
3	0.9254	4.497	0.002202	1.364	24.29	39.64	20.59
4	0.9399	3.296	0.002983	2.224	18.72	38.1	16.79
5	0.8378	3.368	0.007873	1.399	66.31	76.24	9.115
6	0.9541	2.0	0.01	3.0	17.46	8.377	8.0
7	0.8373	2.883	0.008879	1.36	20.3	37.09	16.7
8	0.9273	2.631	0.009926	1.428	90.03	63.95	14.9
9	0.8324	4.369	0.005402	2.789	59.08	54.65	23.69
10	0.7928	3.313	0.001698	1.281	1.193	127.7	20.79
11	0.9233	4.185	0.004424	1.844	4.864	10.6	23.99
12	0.8297	3.256	0.007281	2.042	95.19	10.93	8.785
13	0.9162	4.212	0.007936	2.153	50.59	73.68	9.787
14	0.9627	4.166	0.000384	2.547	85.65	88.63	20.87
15	0.9538	4.627	0.003412	1.672	120.5	70.55	13.38
16	0.8986	3.316	0.009813	1.53	67.66	36.8	21.71
17	0.7141	2.356	0.003583	2.49	87.54	72.67	16.58
18	0.9359	3.042	0.006808	1.975	12.64	2.93	13.3
19	0.9384	3.949	0.008714	2.118	43.0	121.7	21.37
20	0.9431	3.571	0.001483	1.954	68.75	15.09	15.15
21	0.9417	2.677	0.001145	1.316	69.02	101.7	26.07
22	0.8345	2.666	0.006296	2.183	2.622	105.8	14.47
23	0.9528	2.313	0.007445	2.167	118.3	24.96	12.76
24	0.9635	4.743	0.001414	1.999	99.43	64.94	9.972
25	0.9467	3.891	0.002116	1.625	26.2	21.26	31.33
26	0.9698	4.486	0.007054	1.233	54.36	9.684	8.897
27	0.9249	2.856	0.005747	2.484	117.0	11.07	24.97
28	0.6717	3.609	0.009897	2.327	1.458	64.51	15.08
29	0.8887	2.196	0.00325	1.866	127.8	9.167	24.71
30	0.902	3.602	0.004782	1.042	74.47	80.96	23.17
31	0.9353	3.694	0.005804	2.411	11.24	19.29	11.43

```
| 32        | 0.9246    | 4.475     | 0.0007705 | 2.914     | 44.16     | 7.093     | 24.54     |
| 33        | 0.9156    | 3.786     | 0.000296  | 1.067     | 28.12     | 33.88     | 12.4      |
| 34        | 0.7976    | 2.098     | 0.001755  | 2.802     | 113.5     | 98.41     | 13.45     |
| 35        | 0.8877    | 2.674     | 0.00763   | 1.543     | 5.408     | 20.6      | 31.37     |
| 36        | 0.9453    | 2.946     | 0.003238  | 1.832     | 82.81     | 25.12     | 27.26     |
| 37        | 0.4442    | 2.057     | 0.008385  | 2.941     | 89.08     | 118.3     | 31.17     |
| 38        | 0.9609    | 4.358     | 0.006051  | 2.401     | 100.5     | 65.05     | 9.343     |
| 39        | 0.845     | 4.52      | 0.0001    | 3.0       | 15.79     | 40.85     | 17.66     |
| 40        | 0.9347    | 3.807     | 0.004628  | 1.579     | 55.05     | 6.904     | 10.93     |
| 41        | 0.9544    | 2.02      | 0.004035  | 1.328     | 16.16     | 7.038     | 11.87     |
| 42        | 0.9047    | 2.189     | 0.002412  | 2.515     | 14.52     | 12.73     | 10.07     |
| 43        | 0.9532    | 3.967     | 0.003975  | 2.001     | 17.54     | 3.113     | 9.026     |
| 44        | 0.8082    | 2.04      | 0.003462  | 2.295     | 11.7      | 6.747     | 9.185     |
| 45        | 0.962     | 3.146     | 0.001828  | 2.582     | 19.77     | 8.069     | 11.85     |
| 46        | 0.87      | 4.961     | 0.005698  | 2.68      | 17.4      | 4.272     | 13.88     |
| 47        | 0.9722    | 3.272     | 0.001588  | 1.771     | 21.4      | 8.361     | 8.197     |
| 48        | 0.9075    | 4.688     | 0.004592  | 2.458     | 23.94     | 3.923     | 8.543     |
| 49        | 0.9702    | 4.118     | 0.003996  | 1.916     | 56.39     | 14.32     | 10.87     |
| 50        | 0.9676    | 4.951     | 0.008913  | 1.0       | 59.55     | 11.04     | 8.617     |
| 51        | 0.947     | 2.519     | 0.005286  | 2.251     | 62.21     | 11.91     | 12.72     |
| 52        | 0.9329    | 5.0       | 0.002067  | 1.0       | 61.97     | 16.88     | 9.904     |
| 53        | 0.9524    | 3.451     | 0.007608  | 1.858     | 51.11     | 13.82     | 12.81     |
| 54        | 0.838     | 5.0       | 0.01      | 1.0       | 56.6      | 11.65     | 15.85     |
| 55        | 0.8993    | 2.0       | 0.0001    | 3.0       | 56.78     | 12.27     | 8.0       |
=================================================================================================
Best hyperparameters: {'target': 0.9722461489707898, 'params': {'kernel_size': 3.2722288340418677, 'learn_rate': 0.0015877021947692592, 'num_layers': 1.7706637586190885, 'out_dim': 21.403853494539913, 'step_dim': 8.36115568751329, 'window_size': 8.196710925882549}}
```

3.2.4 使用优化后的参数训练

In [ ]:

```python
# 访问优化后的超参数值并转为 int :
best_params = best_hps['params']

best_window_size = int(best_params['window_size'])
best_out_dim = int(best_params['out_dim'])
best_kernel_size = int(best_params['kernel_size'])
best_num_layers = int(best_params['num_layers'])
best_step_dim = int(best_params['step_dim'])
best_learn_rate = best_params['learn_rate']
```

```python
best_hidden_dim = 64


window_size = 16
out_dim = 32
feature_dim = 1
kernel_size = 2
hidden_dim = 32
num_layers = 1
bidirectional = True
step_dim = 64
output_dim = 1
learn_rate = 0.001
Rated_Capacity = 2.0 # 额定容量
epochs = 300 # 训练次数

seed = 42 # 随机种子
setup_seed(seed) # 设置种子
metric = 'RE'
model_list = ['BNO-CBiLA', 'CBiLA', 'CBiL', 'BiLSTM', 'LSTM']
metrics_list = ['RMSE', 'CRMSD', 'MAD', 'MAE', 'MBE', 'R2', 'RE']

model_metrics = {}  # 用于存储每个模型的最终指标值
for model_name in model_list:
    battery_metrics = {} # 存储每个电池的 metrics
    if model_name != 'BNO-CBiLA':
        model, loss_fn, optimizer = get_model(feature_dim, out_dim, kernel_size,
                                              feature_dim, hidden_dim, num_layers,
                                              bidirectional, step_dim, output_dim,
                                              learn_rate, model_name)
        for i in range(len(Battery_list)): # 四折交叉验证
            name = Battery_list[i]
            train_loader, test_loader = get_data(capacity, name, window_size)
            model = model.to(device) # 注册模型到设备

            train_loss = [0]
            score, best_score = float(1), float(1)
            epoch_metrics = {}
            for epoch in range(int(epochs/2)):
                train_epoch_loss = []
                model.train()  # 设置为训练模式
                for index, batch in enumerate(iter(train_loader)):
                    x, y = batch
                    # 归一化
                    x /= torch.tensor(Rated_Capacity).to(device)
                    y /= torch.tensor(Rated_Capacity).to(device)
                    batch_loss = train_batch(x, y, model, optimizer, loss_fn)
                    train_epoch_loss.append(batch_loss)

                train_epoch_loss = np.array(train_epoch_loss).mean()
                train_loss.append(train_epoch_loss)

                if (epoch + 1) % 10 == 0:
                    model.eval() # 设置为验证模式
                    tesy_pred, test_y = [], []
                    with torch.no_grad():
                        for index, batch in enumerate(iter(test_loader)):
                            x, y = batch
                            # 归一化
                            x /= torch.tensor(Rated_Capacity).to(device)
                            y /= torch.tensor(Rated_Capacity).to(device)
                            pred = model(x)

                            # 将预测值和真实值转换为 NumPy 数组并展平
                            pred_np = (pred * Rated_Capacity).cpu().numpy().flatten()
                            y_np = (y * Rated_Capacity).cpu().numpy().flatten()
```

```python
                    tesy_pred.extend(pred_np)
                    test_y.extend(y_np)
                metrics = evaluation(np.array(test_y), np.array(tesy_pred))
                metrics['RE'] = relative_error(np.array(test_y),
                                        np.array(tesy_pred), Rated_Capacity * 0.7)
                score = metrics[metric]
                if epoch + 1 == 10:
                    best_score = score
                    epoch_metrics = metrics
                else:
                    # 使用指标分数进行模型选择
                    if (batch_loss < 1e-4) and (score < best_score):
                        best_score = score
                        epoch_metrics = metrics
                        # break
        battery_metrics[name] = epoch_metrics
    model_metrics[model_name] = battery_metrics
else:
    model, loss_fn, optimizer = get_model(feature_dim, best_out_dim, best_kernel_size,
                                          feature_dim, best_hidden_dim, best_num_layers,
                                          bidirectional, best_step_dim, output_dim,
                                          best_learn_rate, model_name)
    for i in range(len(Battery_list)): # 四折交叉验证
        name = Battery_list[i]
        train_loader, test_loader = get_data(capacity, name, best_window_size)
        model = model.to(device) # 注册模型到设备

        train_loss = [0]
        score, best_score = float(1), float(1)
        epoch_metrics = {}
        for epoch in range(int(epochs)):
            train_epoch_loss = []
            model.train()  # 设置为训练模式
            for index, batch in enumerate(iter(train_loader)):
                x, y = batch
                # 归一化
                x /= torch.tensor(Rated_Capacity).to(device)
                y /= torch.tensor(Rated_Capacity).to(device)
                batch_loss = train_batch(x, y, model, optimizer, loss_fn)
                train_epoch_loss.append(batch_loss)

            train_epoch_loss = np.array(train_epoch_loss).mean()
            train_loss.append(train_epoch_loss)

            if (epoch + 1) % 10 == 0:
                model.eval() # 设置为验证模式
                tesy_pred, test_y = [], []
                with torch.no_grad():
                    for index, batch in enumerate(iter(test_loader)):
                        x, y = batch
                        # 归一化
                        x /= torch.tensor(Rated_Capacity).to(device)
                        y /= torch.tensor(Rated_Capacity).to(device)
                        pred = model(x)

                        # 将预测值和真实值转换为 NumPy 数组并展平
                        pred_np = (pred * Rated_Capacity).cpu().numpy().flatten()
                        y_np = (y * Rated_Capacity).cpu().numpy().flatten()
                        tesy_pred.extend(pred_np)
                        test_y.extend(y_np)
                metrics = evaluation(np.array(test_y), np.array(tesy_pred))
                metrics['RE'] = relative_error(np.array(test_y),
                                        np.array(tesy_pred), Rated_Capacity * 0.7)
                score = metrics[metric]
                if epoch + 1 == 10:
                    best_score = score
```

```
                        epoch_metrics = metrics
                    else:
                        # 使用指标分数进行模型选择
                        if (batch_loss < 1e-4) and (score < best_score):
                            best_score = score
                            epoch_metrics = metrics
                            # break
            battery_metrics[name] = epoch_metrics
        model_metrics[model_name] = battery_metrics

# 打印最终指标值
for model_name, model_metric in model_metrics.items():
    print(f"Model: {model_name}")
    for bat_name, bat_metric in model_metric.items():
        print(f"  Battery: {bat_name}")
        for metric_name, metric_value in bat_metric.items():
            print(f"    Metric: {metric_name}: {metric_value:.4f}")
    print('------------------------------------------------------------------')
```

```
Model: BNO-CBiLA
  Battery: B0005
    Metric: RMSE: 0.0171
    Metric: CRMSD: 0.0158
    Metric: MAD: 0.0045
    Metric: MAE: 0.0096
    Metric: MBE: 0.0066
    Metric: R2: 0.9914
    Metric: RE: 0.0087
  Battery: B0006
    Metric: RMSE: 0.0252
    Metric: CRMSD: 0.0248
    Metric: MAD: 0.0114
    Metric: MAE: 0.0158
    Metric: MBE: 0.0043
    Metric: R2: 0.9884
    Metric: RE: 0.0000
  Battery: B0007
    Metric: RMSE: 0.0126
    Metric: CRMSD: 0.0119
    Metric: MAD: 0.0034
    Metric: MAE: 0.0064
    Metric: MBE: 0.0041
    Metric: R2: 0.9934
    Metric: RE: 0.0187
  Battery: B0018
    Metric: RMSE: 0.0239
    Metric: CRMSD: 0.0221
    Metric: MAD: 0.0053
    Metric: MAE: 0.0114
    Metric: MBE: 0.0091
    Metric: R2: 0.9716
    Metric: RE: 0.0115
------------------------------------------------------------------
Model: CBiLA
  Battery: B0005
    Metric: RMSE: 0.0285
    Metric: CRMSD: 0.0269
    Metric: MAD: 0.0118
    Metric: MAE: 0.0195
    Metric: MBE: 0.0095
    Metric: R2: 0.9748
    Metric: RE: 0.0093
  Battery: B0006
    Metric: RMSE: 0.0277
    Metric: CRMSD: 0.0268
    Metric: MAD: 0.0055
    Metric: MAE: 0.0138
    Metric: MBE: 0.0073
    Metric: R2: 0.9842
    Metric: RE: 0.0000
  Battery: B0007
    Metric: RMSE: 0.0138
    Metric: CRMSD: 0.0121
    Metric: MAD: 0.0045
    Metric: MAE: 0.0076
    Metric: MBE: 0.0066
    Metric: R2: 0.9915
    Metric: RE: 0.0263
  Battery: B0018
    Metric: RMSE: 0.0235
    Metric: CRMSD: 0.0224
    Metric: MAD: 0.0051
    Metric: MAE: 0.0112
    Metric: MBE: 0.0072
    Metric: R2: 0.9669
```

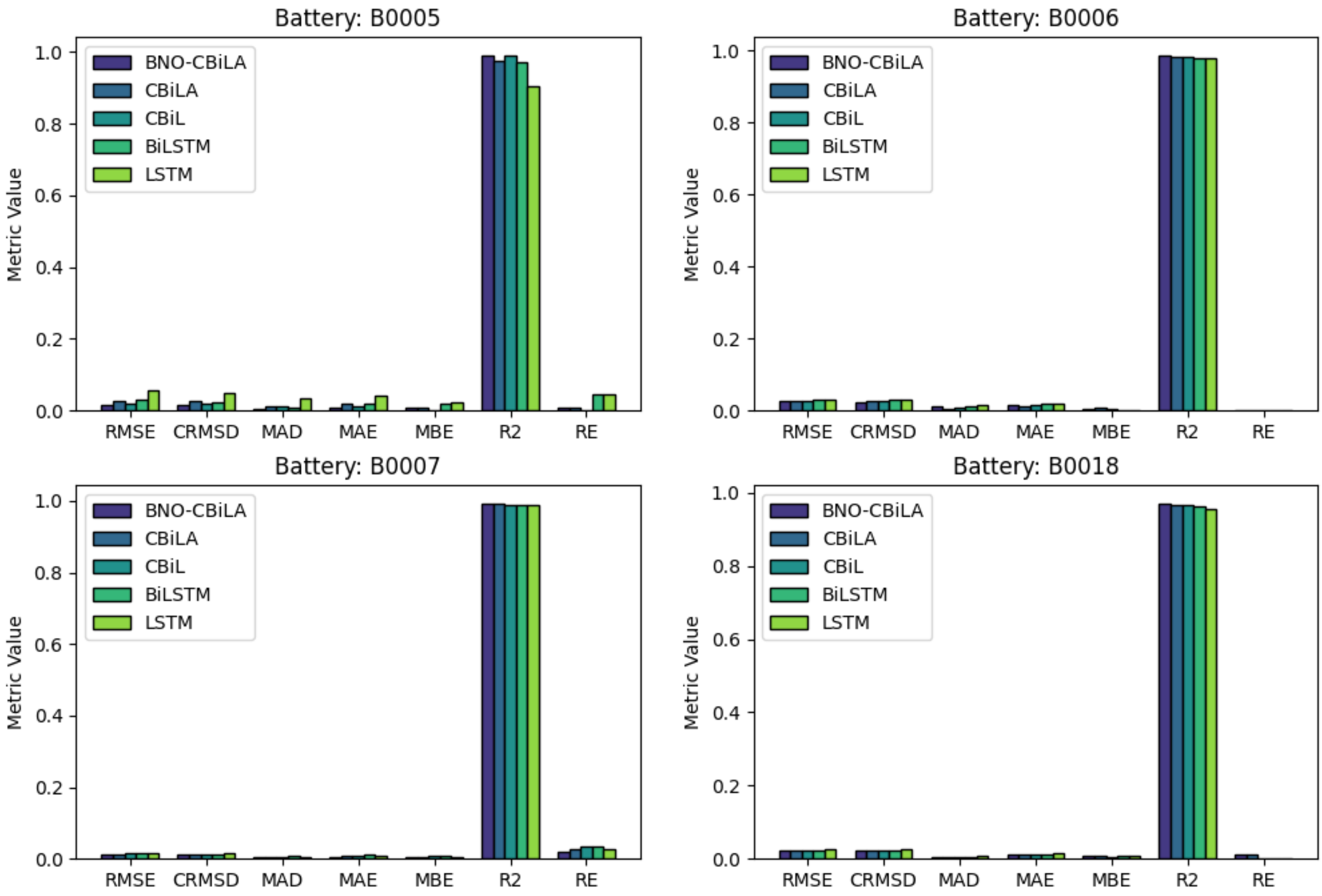
```
    Metric: RE: 0.0127
------------------------------------------------------------------
Model: CBiL
  Battery: B0005
    Metric: RMSE: 0.0183
    Metric: CRMSD: 0.0181
    Metric: MAD: 0.0107
    Metric: MAE: 0.0137
    Metric: MBE: 0.0028
    Metric: R2: 0.9897
    Metric: RE: 0.0000
  Battery: B0006
    Metric: RMSE: 0.0280
    Metric: CRMSD: 0.0277
    Metric: MAD: 0.0089
    Metric: MAE: 0.0156
    Metric: MBE: 0.0039
    Metric: R2: 0.9839
    Metric: RE: 0.0000
  Battery: B0007
    Metric: RMSE: 0.0157
    Metric: CRMSD: 0.0125
    Metric: MAD: 0.0062
    Metric: MAE: 0.0098
    Metric: MBE: 0.0096
    Metric: R2: 0.9889
    Metric: RE: 0.0329
  Battery: B0018
    Metric: RMSE: 0.0241
    Metric: CRMSD: 0.0236
    Metric: MAD: 0.0059
    Metric: MAE: 0.0122
    Metric: MBE: 0.0048
    Metric: R2: 0.9651
    Metric: RE: 0.0000
------------------------------------------------------------------
Model: BiLSTM
  Battery: B0005
    Metric: RMSE: 0.0303
    Metric: CRMSD: 0.0236
    Metric: MAD: 0.0083
    Metric: MAE: 0.0195
    Metric: MBE: 0.0191
    Metric: R2: 0.9715
    Metric: RE: 0.0467
  Battery: B0006
    Metric: RMSE: 0.0305
    Metric: CRMSD: 0.0305
    Metric: MAD: 0.0125
    Metric: MAE: 0.0201
    Metric: MBE: 0.0003
    Metric: R2: 0.9810
    Metric: RE: 0.0000
  Battery: B0007
    Metric: RMSE: 0.0170
    Metric: CRMSD: 0.0138
    Metric: MAD: 0.0074
    Metric: MAE: 0.0111
    Metric: MBE: 0.0099
    Metric: R2: 0.9871
    Metric: RE: 0.0329
  Battery: B0018
    Metric: RMSE: 0.0246
    Metric: CRMSD: 0.0232
    Metric: MAD: 0.0051
    Metric: MAE: 0.0127
```

```
    Metric: MBE: 0.0079
    Metric: R2: 0.9638
    Metric: RE: 0.0000
------------------------------------------------------------------
Model: LSTM
  Battery: B0005
    Metric: RMSE: 0.0554
    Metric: CRMSD: 0.0501
    Metric: MAD: 0.0332
    Metric: MAE: 0.0433
    Metric: MBE: 0.0235
    Metric: R2: 0.9051
    Metric: RE: 0.0467
  Battery: B0006
    Metric: RMSE: 0.0314
    Metric: CRMSD: 0.0313
    Metric: MAD: 0.0154
    Metric: MAE: 0.0213
    Metric: MBE: 0.0019
    Metric: R2: 0.9798
    Metric: RE: 0.0000
  Battery: B0007
    Metric: RMSE: 0.0154
    Metric: CRMSD: 0.0146
    Metric: MAD: 0.0056
    Metric: MAE: 0.0095
    Metric: MBE: 0.0051
    Metric: R2: 0.9894
    Metric: RE: 0.0263
  Battery: B0018
    Metric: RMSE: 0.0268
    Metric: CRMSD: 0.0260
    Metric: MAD: 0.0101
    Metric: MAE: 0.0164
    Metric: MBE: 0.0066
    Metric: R2: 0.9567
    Metric: RE: 0.0000
------------------------------------------------------------------
```

3.3 各模型指标对比直方图

In [ ]:
```python
import seaborn as sns

# 绘制直方图
fig, axes = plt.subplots(2, 2, figsize=(12, 8))
axes = axes.flatten()

colors = sns.color_palette("viridis", n_colors=len(model_list))

for i, battery_name in enumerate(Battery_list):
    ax = axes[i]
    x = np.arange(len(metrics_list))
    width = 0.15  # 每个柱子的宽度

    for j, model_name in enumerate(model_list):
        try:
            metrics = model_metrics[model_name][battery_name]
            values = [metrics[metric_name] for metric_name in metrics_list]
            ax.bar(x + j * width, values, width,
                   label=model_name, color=colors[j], edgecolor='black')
        except KeyError:
            print(f"Warning: Metrics not found for model
                  {model_name} on battery {battery_name}")

    ax.set_xticks(x + width * (len(model_list) - 1) / 2)
```

```
    ax.set_xticklabels(metrics_list, ha="center")
    ax.set_ylabel('Metric Value')
    ax.set_title(f"Battery: {battery_name}")
    ax.legend()

plt.show()
```



In [ ]: