Zhao_2024_分层电力预测

1. 数据预处理

1.1 导入必要的库

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from sklearn.preprocessing import MinMaxScaler
from collections import Counter
import os
import matplotlib.pyplot as plt
```

- **NumPy** 和 **Pandas**：用于数据处理和操作。
- **PyTorch**：用于构建和训练深度学习模型。
- **scikit-learn** 的 **MinMaxScaler**：用于数据归一化。
- **collections.Counter**：用于统计数据分布。
- **os** 和 **matplotlib.pyplot**：用于文件操作和可视化。

1.2 设置随机种子

```
torch.manual_seed(42)
np.random.seed(42)
```

- **目的**：确保实验的可复现性，即每次运行代码时得到相同的结果。

1.3 加载并预处理数据

```
def load_and_preprocess_data(file_path, skiprows=1):
    try:
        data = pd.read_csv(file_path, skiprows=skiprows, head
er=0)
        data['Date'] = pd.to_datetime(data.iloc[:, 0], format
='%Y/%m/%d')
        data = data.set_index('Date')
        data = data.iloc[:, 1:]
        data = data.select_dtypes(include=[np.number])
        data = data.astype(np.float32)
        if data.isnull().values.any():
            print(f"检测到来自{file_path}的数据中存在NaN值。删除包
含NaN的行。")
            data = data.dropna()
        return data
    except Exception as e:
        print(f"加载{file_path}时出错: {e}")
        return pd.DataFrame()
```

- **功能**：
  1. **读取CSV文件**：跳过前 `skiprows` 行（默认跳过第一行）。
  2. **解析日期**：假设第一列为日期，格式为 `%Y/%m/%d`，并将其设置为索引。
  3. **选择数值列**：仅保留数值类型的列。
  4. **类型转换**：将数据类型转换为 `float32`。
  5. **处理缺失值**：如果存在 NaN 值，删除包含 NaN 的行。
- **返回**：预处理后的 `DataFrame`。

1.4 加载所有数据

```
all_data_path = 'data/OEL_all.csv'
all_data = load_and_preprocess_data(all_data_path, skiprows=
1)
if all_data.empty:
```

```
    raise ValueError(f"无法从 '{all_data_path}' 加载数据。请检查
文件内容。")

all_columns = all_data.columns.tolist()
print(f"所有数据已加载。样本数量: {len(all_data)}, 列名: {all_colu
mns}")
```

- **操作**：
  1. 加载指定路径的CSV数据。
  2. 检查数据是否为空，若为空则抛出错误。
  3. 打印数据的样本数量和列名。

1.5 构建聚合数据集

```
def create_aggregated_datasets(all_data):
    aggregated_data = {}
    aggregated_data['daily_max'] = all_data.resample('D').max
()
    aggregated_data['weekly_max'] = all_data.resample('W').ma
x()
    aggregated_data['monthly_max'] = all_data.resample('M').m
ax()
    aggregated_data['yearly_max'] = all_data.resample('A').ma
x()
    return aggregated_data

aggregated_data = create_aggregated_datasets(all_data)
print("聚合数据集已创建。")
for level, df in aggregated_data.items():
    print(f"{level} 样本数量: {len(df)}")
```

- **功能**：
  1. **重采样**：基于原始数据按不同时间频率（每日、每周、每月、每年）计算最大值。

- **返回**：包含四个不同时间频率聚合后的数据集的字典。
- **打印**：每个层级的样本数量。

1.6 数据归一化

```
scaler = MinMaxScaler(feature_range=(0, 1))
scaler.fit(all_data)

def scale_dataset(df, scaler):
    scaled_array = scaler.transform(df)
    scaled_df = pd.DataFrame(scaled_array, columns=df.column
s, index=df.index)
    return scaled_df

scaled_all_data = scale_dataset(all_data, scaler)
scaled_daily_max = scale_dataset(aggregated_data['daily_ma
x'], scaler)
scaled_weekly_max = scale_dataset(aggregated_data['weekly_ma
x'], scaler)
scaled_monthly_max = scale_dataset(aggregated_data['monthly_m
ax'], scaler)
scaled_yearly_max = scale_dataset(aggregated_data['yearly_ma
x'], scaler)

print("数据归一化完成。")
```

- **步骤**：
  1. **初始化**：使用 `MinMaxScaler` 将数据缩放到 [0, 1] 范围。
  2. **拟合缩放器**：仅使用原始数据 `all_data` 拟合缩放器。
  3. **应用缩放**：对所有聚合后的数据集进行缩放。
- **输出**：归一化后的数据集。

1.7 定义层级编码（独热向量）

```
level_encodings = {
    'A': [1, 0, 0, 0],
    'B': [0, 1, 0, 0],
    'C': [0, 0, 1, 0],
    'D': [0, 0, 0, 1],
}

num_levels = len(level_encodings)  # 应为4
```

- **目的**：为每个层级（A, B, C, D）定义独热编码，用于后续模型输入。

1.8 计算每个层级的输入大小

```
input_sizes = {
    'A': 48 * len(all_columns) + num_levels,  # 48个半小时
    'B': 7 * len(all_columns) + num_levels,   # 7天
    'C': 4 * len(all_columns) + num_levels,   # 4周
    'D': 12 * len(all_columns) + num_levels   # 12个月
}

input_size = max(input_sizes.values())  # 15076
print(f"确定的最大输入大小: {input_size}")
```

- **解释**：
  - **Level A**：使用48个半小时的数据（即24小时），每个半小时的数据展平后再加上层级编码。
  - **Level B**：使用7天的数据。
  - **Level C**：使用4周的数据。
  - **Level D**：使用12个月的数据。
- **确定输入大小**：取所有层级中最大的输入大小（即15076），确保所有层级的输入序列可以被容纳。48 (单个channel最大输入长度) * 314 （channel数量） + 4（mark长度）= 15076

1.9 计算每个层级输入数据的均值用于填充

```
mean_values = {}
mean_values['A'] = scaled_all_data.mean().values  # Level A:
all_data
mean_values['B'] = scaled_daily_max.mean().values  # Level B:
daily_max
mean_values['C'] = scaled_weekly_max.mean().values  # Level
C: weekly_max
mean_values['D'] = scaled_monthly_max.mean().values  # Level
D: monthly_max
```

- **目的**：在输入序列长度不足时，使用每个层级输入数据的均值进行填充，确保输入序列长度一致。

1.10 定义数据集类 `UnifiedDataset`

```
class UnifiedDataset(Dataset):
    def __init__(self, data_list, input_size, mean_values):
        self.samples = []
        self.input_size = input_size
        self.mean_values = mean_values  # 用于填充
        for data_dict in data_list:
            input_sequences, targets, levels = self.prepare_l
evel_samples(
                data_dict['input_data'],
                data_dict['target_data'],
                data_dict['input_length'],
                data_dict['level_label']
            )
            self.samples.extend(zip(input_sequences, targets,
levels))
        # 如果有样本，则解包
        if self.samples:
            self.inputs, self.targets, self.levels = zip(*sel
f.samples)
```

```
            self.inputs = np.array(self.inputs)
            self.targets = np.array(self.targets)
            self.levels = np.array(self.levels)
        else:
            self.inputs = np.array([])
            self.targets = np.array([])
            self.levels = np.array([])

    def prepare_level_samples(self, input_data, target_data,
input_length, level_label):
        input_data_sorted = input_data.sort_index()
        target_data_sorted = target_data.sort_index()

        input_sequences = []
        targets = []
        levels = []

        for target_date in target_data_sorted.index:
            # 获取 input_length 个输入样本，结束于 target_date 前
一时间步
            if level_label == 'A':
                input_end = target_date - pd.Timedelta(minute
s=30)
                input_start = input_end - pd.Timedelta(minute
s=30 * input_length) + pd.Timedelta(minutes=30)
            elif level_label == 'B':
                input_end = target_date - pd.Timedelta(days=
1)
                input_start = input_end - pd.Timedelta(days=1
* input_length) + pd.Timedelta(days=1)
            elif level_label == 'C':
                input_end = target_date - pd.Timedelta(weeks=
1)
                input_start = input_end - pd.Timedelta(weeks=
1 * input_length) + pd.Timedelta(weeks=1)
            elif level_label == 'D':
```

```
                input_end = target_date - pd.offsets.MonthBeg
in(1)
                input_start = input_end - pd.offsets.MonthBeg
in(input_length) + pd.offsets.MonthBegin(1)
            else:
                print(f"未知的层级标签: {level_label}")
                continue

            try:
                input_seq = input_data_sorted.loc[input_star
t:input_end].values
                actual_length = len(input_seq)
                if actual_length < input_length:
                    pad_length = input_length - actual_length
                    pad_values = self.mean_values[level_labe
l]
                    pad_array = np.tile(pad_values, pad_lengt
h).reshape(pad_length, -1)
                    input_seq = np.vstack([input_seq, pad_arr
ay])
                elif actual_length > input_length:
                    input_seq = input_seq[-input_length:]
            except KeyError:
                print(f"层级 {level_label} 日期 {target_date}
的输入数据不足。跳过。")
                continue

            # 展平并添加层级编码
            input_with_level = np.concatenate([input_seq.flat
ten(), level_encodings[level_label]])

            # 填充或截断以达到最大输入大小
            if len(input_with_level) < self.input_size:
                padding_size = self.input_size - len(input_wi
th_level)
                padding = np.zeros(padding_size, dtype=np.flo
```

```
at32)
                input_with_level = np.concatenate([input_with
_level, padding])
            elif len(input_with_level) > self.input_size:
                input_with_level = input_with_level[:self.inp
ut_size]

            # 获取目标标签
            target = target_data_sorted.loc[target_date].valu
es

            # 检查目标标签是否包含NaN
            if np.isnan(target).any():
                print(f"层级 {level_label} 日期 {target_date}
的目标标签包含NaN。跳过。")
                continue

            input_sequences.append(input_with_level)
            targets.append(target)
            levels.append(level_label)

        return input_sequences, targets, levels
```

- **功能**：
  1. **初始化**：
     - 接收 `data_list`（包含各层级的输入数据、目标数据、输入序列长度、层级标签）。
     - 遍历 `data_list`，调用 `prepare_level_samples` 方法生成样本。
     - 将生成的样本存储在 `self.samples` 中，并解包为 `self.inputs`、`self.targets` 和 `self.levels`。
  2. `prepare_level_samples` **方法**：
     - **输入序列获取**：根据层级标签（A, B, C, D），确定输入序列的开始和结束时间。

- **填充**：如果实际获取的输入序列长度不足，使用层级对应的均值进行填充。
- **展平**：将输入序列展平，并添加层级的独热编码。
- **填充或截断**：确保输入特征的总长度与 `input_size` 一致。
- **目标标签获取**：获取目标日期的目标数据，检查是否包含 NaN 值，若包含则跳过。
- **返回**：输入序列、目标标签和层级标签的列表。

1.11 为每个层级准备数据

```
data_list = []

# Level A
data_list.append({
    'input_data': scaled_all_data,
    'target_data': scaled_daily_max,
    'input_length': 48,  # 48个半小时
    'level_label': 'A'
})

# Level B
data_list.append({
    'input_data': scaled_daily_max,
    'target_data': scaled_weekly_max,
    'input_length': 7,  # 7天
    'level_label': 'B'
})

# Level C
data_list.append({
    'input_data': scaled_weekly_max,
    'target_data': scaled_monthly_max,
    'input_length': 4,  # 4周
    'level_label': 'C'
})
```

```
# Level D
data_list.append({
    'input_data': scaled_monthly_max,
    'target_data': scaled_yearly_max,
    'input_length': 12,  # 12个月
    'level_label': 'D'
})
```

- **说明**：
  - **Level A**：使用半小时级别的所有数据 `scaled_all_data` 作为输入，目标为每日最大值 `scaled_daily_max`，输入序列长度为48（即24小时）。
  - **Level B**：使用每日最大值 `scaled_daily_max` 作为输入，目标为每周最大值 `scaled_weekly_max`，输入序列长度为7天。
  - **Level C**：使用每周最大值 `scaled_weekly_max` 作为输入，目标为每月最大值 `scaled_monthly_max`，输入序列长度为4周。
  - **Level D**：使用每月最大值 `scaled_monthly_max` 作为输入，目标为每年最大值 `scaled_yearly_max`，输入序列长度为12个月。

1.12 创建统一数据集

```
dataset_full = UnifiedDataset(data_list, input_size, mean_val
ues)
print(f"统一数据集创建完成。总样本数量: {len(dataset_full)}")
```

- **作用**：实例化 `UnifiedDataset` 类，将所有层级的数据整合成一个统一的数据集，并打印总样本数量。

1.13 检查每个层级的样本数量

```
if len(dataset_full) > 0:
    level_counts = Counter(dataset_full.levels)
    print(f"每个层级的样本数量: {level_counts}")
```

```
else:
    print("没有生成任何样本，请检查数据和参数设置。")
```

- **功能**：统计并打印每个层级生成的样本数量，确保数据准备过程正确。

1.14 按层级划分训练集和测试集

```
def split_dataset(dataset, train_ratio=0.8):
    train_samples = []
    test_samples = []
    levels = dataset.levels
    unique_levels = set(levels)
    for level in unique_levels:
        indices = np.where(levels == level)[0]
        if len(indices) == 0:
            continue
        np.random.shuffle(indices)  # 打乱每个层级的索引
        n_train = int(len(indices) * train_ratio)
        train_indices = indices[:n_train]
        test_indices = indices[n_train:]
        train_samples.extend([dataset.samples[i] for i in tra
in_indices])
        test_samples.extend([dataset.samples[i] for i in test
_indices])
    # 解包训练集和测试集
    if train_samples:
        train_inputs, train_targets, train_levels = zip(*trai
n_samples)
        train_inputs = np.array(train_inputs)
        train_targets = np.array(train_targets)
        train_levels = np.array(train_levels)
    else:
        train_inputs, train_targets, train_levels = np.array
([]), np.array([]), np.array([])
    if test_samples:
        test_inputs, test_targets, test_levels = zip(*test_sa
```

```
mples)
        test_inputs = np.array(test_inputs)
        test_targets = np.array(test_targets)
        test_levels = np.array(test_levels)
    else:
        test_inputs, test_targets, test_levels = np.array
([]), np.array([]), np.array([])
    return (train_inputs, train_targets, train_levels), (test
_inputs, test_targets, test_levels)

# 进行数据拆分
(train_inputs, train_targets, train_levels), (test_inputs, te
st_targets, test_levels) = split_dataset(dataset_full, train_
ratio=0.8)
print(f"训练集样本数量: {len(train_inputs)}, 测试集样本数量: {len
(test_inputs)}")
```

- **功能**：
  1. **按层级划分**：确保每个层级的数据按比例（80%训练，20%测试）分配到训练集和测试集中。
  2. **打乱顺序**：对每个层级的数据索引进行随机打乱，以避免时间序列中的潜在顺序偏差。
  3. **统计**：打印训练集和测试集的样本数量。

1.15 定义自定义数据集类 `CustomDataset`

```
class CustomDataset(Dataset):
    def __init__(self, inputs, targets, levels):
        self.inputs = inputs
        self.targets = targets
        self.levels = levels

    def __len__(self):
        return len(self.inputs)
```

```
    def __getitem__(self, idx):
        x = torch.tensor(self.inputs[idx], dtype=torch.float3
2)
        y = torch.tensor(self.targets[idx], dtype=torch.float
32)
        level = self.levels[idx]
        return x, y, level
```

- **功能**：将 NumPy 数组数据封装为 PyTorch 的 `Dataset`，便于后续使用 `DataLoader` 进行批处理。

1.16 创建 DataLoader

```
train_dataset = CustomDataset(train_inputs, train_targets, tr
ain_levels)
test_dataset = CustomDataset(test_inputs, test_targets, test_
levels)

# 创建DataLoader
train_loader = DataLoader(train_dataset, batch_size=32, shuff
le=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle
=False)

print("DataLoaders创建完成。")
```

- **作用**：
  - **训练集**：`train_loader`，批量大小为32，数据随机打乱。
  - **测试集**：`test_loader`，批量大小为32，不打乱数据顺序。
- **目的**：便于模型在训练和测试过程中高效地读取数据。

---

2. 模型架构

2.1 定义编码器和解码器模型（使用Transformer）

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super(PositionalEncoding, self).__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.floa
t).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).floa
t() * (-np.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(1)  # (max_len, 1, d_model)
        self.register_buffer('pe', pe)

    def forward(self, x):
        # x: (seq_len, batch_size, d_model)
        x = x + self.pe[:x.size(0), :]
        return x

class TransformerEncoderModel(nn.Module):
    def __init__(self, input_size, latent_size, num_layers=2,
nhead=4, dim_feedforward=256, dropout=0.1):
        super(TransformerEncoderModel, self).__init__()
        self.input_proj = nn.Linear(input_size, dim_feedforwa
rd)
        self.positional_encoding = PositionalEncoding(dim_fee
dforward)
        encoder_layer = nn.TransformerEncoderLayer(d_model=di
m_feedforward, nhead=nhead, dim_feedforward=dim_feedforward,
dropout=dropout)
        self.transformer_encoder = nn.TransformerEncoder(enco
der_layer, num_layers=num_layers)
        self.output_proj = nn.Linear(dim_feedforward, latent_
size)
```

```
    def forward(self, x):
        # x: (batch_size, input_size)
        x = self.input_proj(x)  # (batch_size, dim_feedforwar
d)
        x = x.unsqueeze(1)  # (batch_size, seq_len=1, dim_fee
dforward)
        x = x.permute(1, 0, 2)  # (seq_len=1, batch_size, dim
_feedforward)
        x = self.positional_encoding(x)
        x = self.transformer_encoder(x)  # (seq_len=1, batch_
size, dim_feedforward)
        x = x.squeeze(0)  # (batch_size, dim_feedforward)
        z = self.output_proj(x)  # (batch_size, latent_size)
        return z

class TransformerDecoderRecon(nn.Module):
    def __init__(self, latent_size, output_size, num_layers=
2, nhead=4, dim_feedforward=256, dropout=0.1):
        super(TransformerDecoderRecon, self).__init__()
        self.input_proj = nn.Linear(latent_size, dim_feedforw
ard)
        self.positional_encoding = PositionalEncoding(dim_fee
dforward)
        decoder_layer = nn.TransformerDecoderLayer(d_model=di
m_feedforward, nhead=nhead, dim_feedforward=dim_feedforward,
dropout=dropout)
        self.transformer_decoder = nn.TransformerDecoder(deco
der_layer, num_layers=num_layers)
        self.output_proj = nn.Linear(dim_feedforward, output_
size)

    def forward(self, z):
        # z: (batch_size, latent_size)
        z = self.input_proj(z)  # (batch_size, dim_feedforwar
d)
        z = z.unsqueeze(1)  # (batch_size, seq_len=1, dim_fee
```

```
dforward)
        z = z.permute(1, 0, 2)  # (seq_len=1, batch_size, dim
_feedforward)
        z = self.positional_encoding(z)
        # 使用z作为memory和target，以简化模型
        y = self.transformer_decoder(z, z)  # (seq_len=1, bat
ch_size, dim_feedforward)
        y = y.squeeze(0)  # (batch_size, dim_feedforward)
        x_recon = self.output_proj(y)  # (batch_size, output_
size)
        return x_recon

class TransformerDecoderPred(nn.Module):
    def __init__(self, latent_size, output_size, num_layers=
2, nhead=4, dim_feedforward=256, dropout=0.1):
        super(TransformerDecoderPred, self).__init__()
        self.input_proj = nn.Linear(latent_size, dim_feedforw
ard)
        self.positional_encoding = PositionalEncoding(dim_fee
dforward)
        decoder_layer = nn.TransformerDecoderLayer(d_model=di
m_feedforward, nhead=nhead, dim_feedforward=dim_feedforward,
dropout=dropout)
        self.transformer_decoder = nn.TransformerDecoder(deco
der_layer, num_layers=num_layers)
        self.output_proj = nn.Linear(dim_feedforward, output_
size)

    def forward(self, z):
        # z: (batch_size, latent_size)
        z = self.input_proj(z)  # (batch_size, dim_feedforwar
d)
        z = z.unsqueeze(1)  # (batch_size, seq_len=1, dim_fee
dforward)
        z = z.permute(1, 0, 2)  # (seq_len=1, batch_size, dim
_feedforward)
```

```
        z = self.positional_encoding(z)
        # 使用z作为memory和target，以简化模型
        y = self.transformer_decoder(z, z)  # (seq_len=1, bat
ch_size, dim_feedforward)
        y = y.squeeze(0)  # (batch_size, dim_feedforward)
        y_pred = self.output_proj(y)  # (batch_size, output_s
ize)
        return y_pred
```

- **组件**：
  1. **PositionalEncoding**：
     - **功能**：为Transformer模型添加位置编码，以保留输入序列中元素的位置信息。
  2. **TransformerEncoderModel**：
     - **结构**：包含输入映射层、位置编码、多个Transformer编码器层以及输出映射层。
     - **功能**：将高维输入数据编码为潜在表示 `z`，捕捉数据的主要特征和模式。
  3. **TransformerDecoderRecon**（重建解码器）：
     - **结构**：包含输入映射层、位置编码、多个Transformer解码器层以及输出映射层。
     - **功能**：基于潜在表示 `z` 重建原始输入数据（去除层级编码部分）。
  4. **TransformerDecoderPred**（预测解码器）：
     - **结构**：与重建解码器类似，包含输入映射层、位置编码、多个Transformer解码器层以及输出映射层。
     - **功能**：基于潜在表示 `z` 进行目标特征的预测。

2.2 实例化模型

```
hidden_size = 128  # Transformer的隐藏维度
latent_size = 64
output_size = len(all_columns)  # 用于预测的特征数量
```

```
# 实例化基于Transformer的模型
encoder = TransformerEncoderModel(input_size=input_size, latent_size=latent_size, num_layers=2, nhead=4, dim_feedforward=256, dropout=0.1)
decoder_recon = TransformerDecoderRecon(latent_size=latent_size, output_size=input_size - num_levels, num_layers=2, nhead=4, dim_feedforward=256, dropout=0.1)
decoder_pred = TransformerDecoderPred(latent_size=latent_size, output_size=output_size, num_layers=2, nhead=4, dim_feedforward=256, dropout=0.1)

print("基于Transformer的模型实例化完成。")
```

- **设置参数**：
  - **hidden_size**：隐藏层大小，设置为128。
  - **latent_size**：潜在向量大小，设置为64。
  - **output_size**：预测特征的数量，等于所有列的数量。
- **实例化**：
  - **encoder**：将输入数据编码为潜在表示 `z`。
  - **decoder_recon**：重建输入数据。
  - **decoder_pred**：预测目标特征。

2.3 定义损失函数和优化器

```
criterion_recon = nn.MSELoss()
criterion_pred = nn.MSELoss()
optimizer_encoder = optim.Adam(list(encoder.parameters()) + list(decoder_recon.parameters()), lr=0.001)
optimizer_decoder = optim.Adam(decoder_pred.parameters(), lr=0.001)
```

```
print("损失函数和优化器定义完成。")
```

- **损失函数**：
  1. **criterion_recon**：均方误差损失，用于衡量重建解码器的重建效果。
  2. **criterion_pred**：均方误差损失，用于衡量预测解码器的预测效果。
- **优化器**：
  1. **optimizer_encoder**：Adam优化器，优化编码器和重建解码器的参数。
  2. **optimizer_decoder**：Adam优化器，仅优化预测解码器的参数。

2.4 训练编码器

```
def train_encoder(encoder, decoder_recon, dataloader, criteri
on, optimizer, num_epochs=50):
    encoder.train()
    decoder_recon.train()
    for epoch in range(num_epochs):
        total_loss = 0
        for x_batch, _, _ in dataloader:
            optimizer.zero_grad()
            # 前向传播
            z = encoder(x_batch)
            x_recon = decoder_recon(z)
            # 排除层级编码部分
            x_original = x_batch[:, :-num_levels]
            loss = criterion(x_recon, x_original)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        avg_loss = total_loss / len(dataloader)
        if (epoch + 1) % 10 == 0 or epoch == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Encoder L
oss: {avg_loss:.6f}')
```

```
# 训练编码器
print("开始训练编码器...")
train_encoder(encoder, decoder_recon, train_loader, criterion
_recon, optimizer_encoder, num_epochs=100)
print("编码器训练完成。")
```

- **过程**：
  1. **设置模式**：将编码器和重建解码器设置为训练模式。
  2. **训练循环**：遍历指定的训练轮数（默认100轮）。
     - **前向传播**：
       - 输入批次数据 `x_batch` 通过编码器生成潜在表示 `z`。
       - 潜在表示 `z` 通过重建解码器重建输入数据 `x_recon`。
       - 提取原始输入数据 `x_original`（去除层级编码部分）。
     - **计算损失**：使用均方误差损失 `criterion_recon` 计算重建损失。
     - **反向传播**：反向传播误差并更新编码器和重建解码器的参数。
     - **累计损失**：累加每个批次的损失。
  3. **打印损失**：每隔10轮或第一轮打印一次平均损失。

2.5 冻结编码器参数

```
for param in encoder.parameters():
    param.requires_grad = False
print("编码器参数已冻结。")
```

- **目的**：在训练预测解码器时，保持编码器的参数不变，仅训练预测解码器的参数。

2.6 训练预测解码器

```
def train_decoder(decoder_pred, encoder, dataloader, criterio
n, optimizer, num_epochs=10):
    decoder_pred.train()
    for epoch in range(num_epochs):
```

```
        total_loss = 0
        for x_batch, y_batch, _ in dataloader:
            optimizer.zero_grad()
            # 前向传播
            z = encoder(x_batch)
            y_pred = decoder_pred(z)
            loss = criterion(y_pred, y_batch)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        avg_loss = total_loss / len(dataloader)
        if (epoch + 1) % 5 == 0 or epoch == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Decoder L
oss: {avg_loss:.6f}')

# 训练预测解码器
print("开始训练解码器...")
train_decoder(decoder_pred, encoder, train_loader, criterion_
pred, optimizer_decoder, num_epochs=50)
print("解码器训练完成。")
```

- **过程**：
  1. **设置模式**：将预测解码器设置为训练模式。
  2. **训练循环**：遍历指定的训练轮数（默认50轮）。
     - **前向传播**：
       - 输入批次数据 `x_batch` 通过编码器生成潜在表示 `z`。
       - 潜在表示 `z` 通过预测解码器生成预测结果 `y_pred`。
     - **计算损失**：使用均方误差损失 `criterion_pred` 计算预测损失。
     - **反向传播**：反向传播误差并更新预测解码器的参数。
     - **累计损失**：累加每个批次的损失。
  3. **打印损失**：每隔5轮或第一轮打印一次平均损失。

---

3. 结果的保存和可视化

3.1 测试模型

```
def test_model(encoder, decoder_pred, dataloader, criterion):
    encoder.eval()
    decoder_pred.eval()
    total_loss = 0
    all_preds = []
    all_targets = []
    all_levels = []
    with torch.no_grad():
        for x_batch, y_batch, levels in dataloader:
            z = encoder(x_batch)
            y_pred = decoder_pred(z)
            loss = criterion(y_pred, y_batch)
            total_loss += loss.item()
            all_preds.append(y_pred.cpu().numpy())
            all_targets.append(y_batch.cpu().numpy())
            all_levels.extend(levels)
    avg_loss = total_loss / len(dataloader)
    print(f'Test Loss: {avg_loss:.6f}')
    # 拼接所有预测和实际标签
    all_preds = np.vstack(all_preds)
    all_targets = np.vstack(all_targets)
    return all_preds, all_targets, all_levels

# 测试模型
print("开始测试模型...")
all_preds_test, all_targets_test, all_levels_test = test_mode
l(encoder, decoder_pred, test_loader, criterion_pred)
print("模型测试完成。")
```

- **功能**：
    1. **设置模式**：将编码器和预测解码器设置为评估模式。

2. **禁用梯度计算**：使用 `torch.no_grad()` 以节省内存和计算资源。
3. **前向传播**：
    - 输入数据通过编码器生成潜在表示 `z`。
    - 潜在表示 `z` 通过预测解码器生成预测结果 `y_pred`。
4. **计算损失**：累计所有批次的预测损失。
5. **收集结果**：收集所有预测结果、实际标签和层级标签。
6. **返回**：所有预测结果、实际标签和层级标签的 NumPy 数组。

3.2 逆归一化预测结果和实际标签

```
try:
    all_preds_original = scaler.inverse_transform(all_preds_t
est)
    all_targets_original = scaler.inverse_transform(all_targe
ts_test)
    print("逆归一化完成。")
except Exception as e:
    print(f"逆归一化时出错: {e}")
    all_preds_original = all_preds_test
    all_targets_original = all_targets_test
```

- **功能**：
    - **逆归一化**：将预测结果和实际标签从缩放后的值还原为原始值。
    - **异常处理**：若逆归一化过程中出现错误，则保留缩放后的值。

3.3 检查标签是否全为零

```
print("检查实际标签是否全为零...")
actual_zero = np.sum(all_targets_original == 0)
print(f"实际标签中值为零的数量: {actual_zero} (总标签数量: {all_tar
gets_original.size})")
```

- **目的**：确保实际标签中不全为零，验证数据的有效性。

3.4 创建结果 DataFrame 并保存

```
results_df = pd.DataFrame({
    'Level': all_levels_test,
})

# 为每个通道添加预测值和实际值
for i, col in enumerate(all_columns):
    results_df[f'Predicted_{col}'] = all_preds_original[:, i]
    results_df[f'Actual_{col}'] = all_targets_original[:, i]

# 创建输出目录（如果不存在）
output_dir = 'prediction_results'
os.makedirs(output_dir, exist_ok=True)

# 保存所有测试结果到一个CSV文件
results_df.to_csv(os.path.join(output_dir, 'prediction_result
s_test.csv'), index=False)
print("所有测试结果已保存到 'prediction_results_test.csv'。")
```

- **步骤**：
  1. **创建 DataFrame**：包含层级标签、每个通道的预测值和实际值。
  2. **保存**：将完整的测试结果保存为 `prediction_results_test.csv`。

3.5 按层级保存结果

```
unique_levels = sorted(set(all_levels_test))
for level in unique_levels:
    level_df = results_df[results_df['Level'] == level]
    filename = f'prediction_results_{level}_test.csv'
    level_df.to_csv(os.path.join(output_dir, filename), index
=False)
    print(f"层级 {level} 的测试结果已保存到 '{filename}'。")
```

- **功能**：将每个层级的测试结果分别保存为不同的 CSV 文件，便于分层分析。

3.6 打印部分结果以供验证

```
print("示例结果：")
print(results_df.head())
```

- **目的**：快速查看保存结果的前几行，确保数据正确。

3.7 保存标签和预测值的样本以供进一步检查

```
# 保存实际标签示例
labels_output_path = os.path.join(output_dir, 'actual_labels_
sample.csv')
actual_labels_sample = results_df[['Level'] + [f'Actual_{co
l}' for col in all_columns]].head(100)
actual_labels_sample.to_csv(labels_output_path, index=False)
print(f"实际标签示例已保存到 '{labels_output_path}'。")

# 保存预测结果示例
predictions_output_path = os.path.join(output_dir, 'predicted
_values_sample.csv')
predicted_values_sample = results_df[['Level'] + [f'Predicted
_{col}' for col in all_columns]].head(100)
predicted_values_sample.to_csv(predictions_output_path, index
=False)
print(f"预测结果示例已保存到 '{predictions_output_path}'。")
```

- **目的**：
  - **实际标签示例**：保存前100个实际标签，便于进一步检查。
  - **预测结果示例**：保存前100个预测值，便于进一步检查。

3.8 可视化部分预测结果与实际值

```
def plot_predictions(actual, predicted, channel, output_dir,
num_samples=100):
    plt.figure(figsize=(15, 5))
```

```
    plt.plot(actual[:num_samples], label='Actual')
    plt.plot(predicted[:num_samples], label='Predicted')
    plt.title(f'Channel: {channel} - Actual vs Predicted')
    plt.xlabel('Sample')
    plt.ylabel('Value')
    plt.legend()
    plt.savefig(os.path.join(output_dir, f'plot_{channel}.pn
g'))
    plt.close()

# 绘制前5个通道的预测结果
for i, col in enumerate(all_columns[:5]):
    plot_predictions(all_targets_original[:, i], all_preds_or
iginal[:, i], col, output_dir)
    print(f"预测与实际值对比图已保存为 'plot_{col}.png'。")
```

- **功能**：
  - **绘图函数**：`plot_predictions` 绘制实际值与预测值的对比图，并保存为 PNG 文件。
    - **横轴**：样本编号（从0到 `num_samples-1`）。
    - **纵轴**：数值（实际值和预测值）。
    - **标题**：指定通道名称。
    - **图例**：区分实际值和预测值。
  - **绘制示例**：绘制前5个通道的预测结果与实际值对比图。
- **图示解释**：
  - **横轴（Sample）**：表示样本的序号，展示从第一个样本到第 `num_samples` 个样本的变化。
  - **纵轴（Value）**：表示数据的数值大小，展示实际值和预测值的具体数值。
  - **图例（Legend）**：区分实际值（Actual）和预测值（Predicted），帮助识别模型预测的准确性。

- **图像意义**：通过对比实际值和预测值的曲线，直观展示模型在不同通道上的预测性能，判断模型是否能够有效捕捉数据的趋势和变化。

总结

您的代码主要分为三个部分：

1. 预处理

- **数据加载**：从 CSV 文件中读取数据，处理日期格式，并选择数值类型的列。
- **数据聚合**：按不同时间频率（每日、每周、每月、每年）聚合数据，计算最大值。
- **数据归一化**：使用 `MinMaxScaler` 将数据缩放到 [0, 1] 范围，便于模型训练。
- **样本生成**：通过 `UnifiedDataset` 类，将不同层级的数据生成统一的输入序列，处理长度不足时进行均值填充，确保所有输入序列长度一致。
- **数据分割**：按层级比例划分训练集和测试集，确保各层级的数据均衡分布。
- **DataLoader**：将数据集封装为 `DataLoader`，便于模型在训练和测试时高效读取数据。

2. 模型架构

- **编码器（Encoder）**：将高维输入数据编码为低维潜在表示 `z`，捕捉数据的主要特征和模式。
- **重建解码器（DecoderRecon）**：基于潜在表示 `z` 重建原始输入数据，用于训练编码器捕捉有效特征。
- **预测解码器（DecoderPred）**：基于潜在表示 `z` 预测目标特征，完成实际的预测任务。
- **训练过程**：
  - **阶段一**：训练编码器和重建解码器，使编码器能够有效地捕捉数据的主要特征。
  - **阶段二**：冻结编码器参数，仅训练预测解码器，使其能够基于潜在表示进行准确预测。

3. 结果的保存和可视化

- **结果保存**：

    - **完整结果**：将所有测试样本的预测值和实际值保存到 `prediction_results_test.csv`。
    - **按层级保存**：将每个层级的测试结果分别保存为独立的 CSV 文件，便于分层分析。
    - **示例保存**：保存前100个实际标签和预测值。
- **结果可视化**：
    - **对比图**：为前5个通道绘制实际值与预测值的对比图，横轴为样本编号，纵轴为数值大小。通过这些图表，直观地评估模型在不同通道上的预测性能。

设计理念

1. 统一处理不同输入的子任务

本项目将不同时间频率（每日、每周、每月、每年）的预测任务统一到一个框架中处理。通过统一的Transformer编码器和解码器，模型能够在不同层级间共享信息，提高整体预测的准确性和泛化能力。

2. 通过统一子任务学习综合表征

统一的模型架构使得模型可以在不同的子任务中学习到综合的特征表征。这种共享表征有助于模型捕捉到跨时间频率的潜在模式和依赖关系，从而提升预测性能。

3. 利用共同的特征空间和重建损失

尽管各个子任务的输入序列长度不同，但它们都是基于同一数据集进行采样，来源于相同的特征空间。通过使用重建损失，模型能够学习到一个统一的潜在表征，这不仅帮助编码器捕捉重要特征，还使得预测解码器能够更准确地进行目标预测。

4. 通用解码器用于下游任务

一个通用的预测解码器（`DecoderPred`）能够基于统一的潜在表征执行不同的下游任务，如电力负荷预测。这种设计使得模型架构更加简洁，同时保持了灵活性和扩展性。

使用说明

1. **克隆仓库**：

```
git clone <https://github.com/yourusername/OEL.git>
cd OEL
```

2. **创建数据目录**：

   在项目根目录下创建 `data` 文件夹，并将 `OEL_all.csv` 数据文件放入其中。

```
mkdir data
# 将 OEL_all.csv 放入 data/ 目录
```

3. **安装依赖**：

   确保您已经安装了Python（推荐3.7或更高版本）。建议使用虚拟环境：

```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

   requirements.txt 示例内容：

```
numpy
pandas
torch
scikit-learn
matplotlib
```

4. **运行主代码**：

   打开并运行 `main.ipynb`，执行整个数据处理、模型训练、评估和结果可视化的流程。

```
jupyter notebook main.ipynb
```

5. **查看结果**：

   - **预测结果**：在 `prediction_results/` 目录下查看 `prediction_results_test.csv` 以及按层级划分的预测结果文件。

- **可视化图表**：在 `prediction_results/plots/` 目录下查看各通道的实际值与预测值对比图（如 `plot_0.png`）。
- **样本数据**：查看 `actual_labels_sample.csv` 和 `predicted_values_sample.csv` 以了解具体的预测表现。
