› 连接本地gpu

[ ] ↳ *已隐藏 5 个单元格*

⌄ 数据预处理

```
import  numpy  as  np
import  tensorflow  as  tf
import  matplotlib
import  matplotlib.pyplot  as  plt
import  seaborn  as  sns
import  IPython
from  sklearn  import  metrics
from  sklearn  import  model_selection
from  sklearn.model_selection  import  train_test_split
from  sklearn.preprocessing  import  StandardScaler,  MinMaxScaler
from  sklearn.model_selection  import  train_test_split
from  tensorflow  import  keras
from  tensorflow.keras  import  layers
from  tensorflow.keras.models  import  Sequential
from  tensorflow.keras.layers  import  LSTM,  Dense



import  pandas  as  pd

!pip  install  h3
import  h3
import  folium
import  branca.colormap  as  cm

import  torch.utils.data
from  torch  import  optim,  nn


import  pytz

import  scipy.optimize
#  %matplotlib  widget
```

```
Requirement already satisfied: h3 in c:\python312\lib\site-packages (4.1.1)

[notice] A new release of pip is available: 24.0 -> 24.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

```
import  numpy  as  np
import  pandas  as  pd
import  h3
from  sklearn.preprocessing  import  LabelEncoder
from  tensorflow.keras  import  layers,  models,  Input
import  matplotlib.pyplot  as  plt


#  df  =  pd.read_csv('/content/all_waybill_info_meituan_0322.csv')
#  df.dropna(inplace=True)

```

```
#  ===  数据加载与初步处理  ===
#  模拟加载数据（替换为您的数据文件路径）
df  =  pd.read_csv('/mingxuan/Courses/24Fall/CS  Capstone/Meituan-INFORMS-TSL-Research-Challenge-main/all_waybill_info_meituan_0322.csv')
#  df  =  pd.read_csv('/content/drive/MyDrive/NYU/24Fall/CS  Cap/all_waybill_info_meituan_0322.csv')
#  df  =  pd.read_csv('/content/all_waybill_info_meituan_0322.csv')
df.dropna(inplace=True)
#  df.drop(['Zodiac'],  axis=1,  inplace=True)

df.reset_index(drop=True,  inplace=True)
```

```
#用来可视化的df2，不是df
#  df2  =  df[['is_courier_grabbed','is_prebook','platform_order_time','order_push_time','recipient_lng','recipient_lat']]
df2  =  df[['is_courier_grabbed','is_prebook','platform_order_time','order_push_time','estimate_meal_prepare_time','recipient_lng','recipient_l

#去重,只看接受了的订单,只看非预约订单
df2  =  df2[df2['is_prebook']  ==  0]
df2  =  df2[df2['is_courier_grabbed']  ==  1]
```

```
df2=  df2.sort_values(by='platform_order_time')
df2=  df2.sort_values(by='order_push_time')

df2.reset_index(drop=True,  inplace=True)
df2  =  df2.drop(columns=['is_courier_grabbed'])
df2  =  df2.drop(columns=['is_prebook'])

#  经纬度转换为浮点数
df2['recipient_lng']  =  pd.to_numeric(df2['recipient_lng'],  errors='coerce')
df2['recipient_lat']  =  pd.to_numeric(df2['recipient_lat'],  errors='coerce')

#转换time系列下单时间为date  time
#转换完后数据格式是pandas._libs.tslibs.timestamps.Timestamp
df2['platform_order_time_date']  =  pd.to_datetime(df2['platform_order_time'],  unit='s')
df2['order_push_time_date']  =  pd.to_datetime(df2['order_push_time'],  unit='s')
df2['estimate_meal_prepare_time_date']  =  pd.to_datetime(df2['estimate_meal_prepare_time'],  unit='s')

#时区换成UTC+8hour,不要多次按！每次按都会在原基础上+8！
df2['platform_order_time_date']  =  df2['platform_order_time_date'].dt.tz_localize('UTC').dt.tz_convert('Asia/Singapore')
df2['platform_order_time_date']  =  df2['platform_order_time_date'].dt.tz_localize(None)
df2['order_push_time_date']  =  df2['order_push_time_date'].dt.tz_localize('UTC').dt.tz_convert('Asia/Singapore')
df2['order_push_time_date']  =  df2['order_push_time_date'].dt.tz_localize(None)
df2['estimate_meal_prepare_time_date']  =  df2['estimate_meal_prepare_time_date'].dt.tz_localize('UTC').dt.tz_convert('Asia/Singapore')
df2['estimate_meal_prepare_time_date']  =  df2['estimate_meal_prepare_time_date'].dt.tz_localize(None)

#帮助df2的时间戳数据添加一天之内的特征辅助
day  =  24*60*60

df2['Day  sin']  =  np.sin(df2['platform_order_time']  *  (2  *  np.pi  /  day))
df2['Day  cos']  =  np.cos(df2['platform_order_time']  *  (2  *  np.pi  /  day))


```

H3处理

```

```

h3_index转化

```
#  转换为  H3  指数
def  compute_h3_and_boundaries(row,  resolution=9):
      lng  =  row['recipient_lng']  /  1e6
      lat  =  row['recipient_lat']  /  1e6
      h3_index  =  h3.latlng_to_cell(lat,  lng,  resolution)
      return  pd.Series([h3_index])

#  添加  H3  索引列
df2[['H3_Index']]  =  df2.apply(compute_h3_and_boundaries,  axis=1)

```

```
df2[['H3_Index']]
```

	H3_Index
0	89329b5888fffff
1	89329b58d13ffff
2	89329b58c7bffff
3	89329b5aabbffff
4	89329b585a3ffff
...	...
546360	8916cb6f587ffff
546361	89329b58cc7ffff
546362	89329b58823ffff
546363	89329b58943ffff
546364	89329b58893ffff

546365 rows × 1 columns

通过拆解h3_index，从index的角度找到两个数据区域的分割线

[ ] ↳ *已隐藏 4 个单元格*

分离大小区域

```
df_larger=df_top_group.copy()
df_smaller=df_other_groups.copy()

df_larger.reset_index(drop=True, inplace=True)
df_smaller.reset_index(drop=True, inplace=True)


df_larger = df_larger.drop(columns=['order_push_time','estimate_meal_prepare_time','estimate_meal_prepare_time'])
df_larger = df_larger.drop(columns=['recipient_lng','recipient_lat','sender_lng','sender_lat'])
# df_larger = df_larger.drop(columns=['is_weekend','recipient_lng','recipient_lat','sender_lng','sender_lat'])
df_larger = df_larger.drop(columns=['order_push_time_date','estimate_meal_prepare_time_date'])
df_larger = df_larger.drop(columns=['H3_Binary','H3_Binary_19','H3_Decimal','H3_Binary_Last45'])
df_larger = df_larger.drop(columns=['H3_Binary_Groups','H3_Hex_Groups'])

df_smaller = df_smaller.drop(columns=['order_push_time','estimate_meal_prepare_time','estimate_meal_prepare_time'])
df_smaller = df_smaller.drop(columns=['recipient_lng','recipient_lat','sender_lng','sender_lat'])
# df_smaller = df_smaller.drop(columns=['is_weekend','recipient_lng','recipient_lat','sender_lng','sender_lat'])
df_smaller = df_smaller.drop(columns=['order_push_time_date','estimate_meal_prepare_time_date'])
df_smaller = df_smaller.drop(columns=['H3_Binary','H3_Binary_19','H3_Decimal','H3_Binary_Last45'])
df_smaller = df_smaller.drop(columns=['H3_Binary_Groups','H3_Hex_Groups'])
```

```
df_larger = df_larger.drop(columns=['Day sin','Day cos'])

df_smaller = df_smaller.drop(columns=['Day sin','Day cos'])
```

```
df_larger
# df_top_group

```

	platform_order_time	platform_order_time_date	H3_Index
0	1665935379	2022-10-16 23:49:39	89329b5888fffff
1	1665935707	2022-10-16 23:55:07	89329b58d13ffff
2	1665935814	2022-10-16 23:56:54	89329b58c7bffff
3	1665935881	2022-10-16 23:58:01	89329b5aabbffff
4	1665935996	2022-10-16 23:59:56	89329b585a3ffff
...	...	...	...
506137	1666627167	2022-10-24 23:59:27	89329b5aaabffff
506138	1666627173	2022-10-24 23:59:33	89329b58cc7ffff
506139	1666627181	2022-10-24 23:59:41	89329b58823ffff
506140	1666627168	2022-10-24 23:59:28	89329b58943ffff
506141	1666627188	2022-10-24 23:59:48	89329b58893ffff

506142 rows × 3 columns

```
# df_smaller
```

```
# 提取 H3 索引（假设原始数据已计算出 H3_Index 列）
all_h3_indices = df_larger['H3_Index'].unique()  # 提取大区LARGER 所有出现的 H3 cells

```

```
all_h3_indices.shape
```

```
(1560,)
```

将h3坐标处理为二维整数对+偏移回原点坐标和第一象限

[ ] ↳ *已隐藏 12 个单元格*

查看内存

[ ] ↳ *已隐藏 4 个单元格*

时间切片

```
def  preprocess_order_time_h3_with_time_features(df,  time_col,  interval='10min'):
        import  numpy  as  np
        import  pandas  as  pd
        from  itertools  import  product

        #  #  确保时间列为  datetime  格式
        #  df[time_col]  =  pd.to_datetime(df[time_col],  unit='s',  errors='coerce')

        #  #  检查转换后的时间列是否有效
        #  if  df[time_col].isna().any():
        #          raise  ValueError(f"{time_col}  中存在无效值，无法转换为  datetime！请检查数据。")

        #  添加时间桶
        df['time_bucket']  =  df[time_col].dt.floor(interval)

        #  聚合订单量
        grouped  =  df.groupby(['time_bucket',  'i',  'j']).size().reset_index(name='order_volume')

        #  获取完整的时间段（从最早到最晚时间桶）
        full_time_range  =  pd.date_range(
                start=grouped['time_bucket'].min(),
                end=grouped['time_bucket'].max(),
                freq=interval
        )

        #  获取所有  i,  j  的最大值，构建完整的  (i,  j)  组合
        i_max  =  grouped['i'].max()
        j_max  =  grouped['j'].max()

        #  生成所有可能的  (i,  j)  组合
        size_max  =  np.max([i_max,  j_max])
        ij_pairs  =  list(product(range(size_max  +  1),  range(size_max  +  1)))

        #  构造完整索引
        full_index  =  pd.MultiIndex.from_product([full_time_range,  ij_pairs],  names=['time_bucket',  'ij'])

        #  将  (i,  j)  映射为元组，便于索引
        grouped['ij']  =  list(zip(grouped['i'],  grouped['j']))
        grouped  =  grouped.set_index(['time_bucket',  'ij'])['order_volume']

        #  补全所有时间桶的完整数据，填充缺失的订单量为  0
        filled  =  grouped.reindex(full_index,  fill_value=0).reset_index()
        filled.columns  =  ['time_bucket',  'ij',  'order_volume']    #  重命名列方便后续操作

        #  分离  i  和  j
        filled['i']  =  filled['ij'].apply(lambda  x:  x[0])
        filled['j']  =  filled['ij'].apply(lambda  x:  x[1])

        #  将补全数据转为二维矩阵  (时间步数,  网格数)
        grid_data  =  (
                filled.pivot(index='time_bucket',  columns='ij',  values='order_volume')
                .reindex(index=full_time_range,  fill_value=0)
                .reset_index(drop=True)
        )

        #  转换网格数据为数组
        grid_data_array  =  grid_data.to_numpy()

        #  构造仅包含订单量的  LSTM  输入数组
        lstm_array  =  np.expand_dims(grid_data_array,  axis=-1)

```

```
        return  lstm_array

```

```
#  应用函数
lstm_array  =  preprocess_order_time_h3_with_time_features(
        df_larger_with_ij,
        time_col='platform_order_time_date'
)

#  查看结果形状
print("LSTM  Array  Shape:",  lstm_array.shape)

```

```
LSTM Array Shape: (1155, 9409, 1)
```

有slide_window的数据集拆分 且部分分步处理数据

格式转换

```
#  转换为  ConvLSTM  格式，调整数据类型

def  reshape_to_convlstm_input(lstm_array,  rows,  cols):
        """
        将  LSTM  array  转换为  ConvLSTM  的输入格式
        :param  lstm_array:  原始  LSTM  array，形状为  (time_steps,  grid,  features)
        :param  rows:  网格行数
        :param  cols:  网格列数
        :return:  重塑后的数据，形状为  (time_steps,  rows,  cols,  features)
        """
        import  numpy  as  np

        grid_size  =  rows  *  cols
        if  lstm_array.shape[1]  !=  grid_size:
                raise  ValueError("LSTM  array  的网格数量与指定的  rows  x  cols  不匹配")

        #  重塑为  ConvLSTM  输入格式
        return  lstm_array.reshape(-1,  rows,  cols,  lstm_array.shape[2])

rows,  cols  =  97,  97
convlstm_input  =  reshape_to_convlstm_input(lstm_array,  rows,  cols).astype(np.float32)

```

分批次处理滑动窗口，然后合并

```
def  apply_sliding_window_with_targets_in_batches(data,  window_size,  step=1,  batch_size=100):
        """
        分批对时间序列数据应用滑动窗口，同时生成目标值
        :param  data:  输入数据，形状为  (time_steps,  rows,  cols,  channels)
        :param  window_size:  滑动窗口的时间步数
        :param  step:  滑动的步长
        :param  batch_size:  每批次生成的窗口数
        :return:  分批生成的滑动窗口数据和目标值
        """
        import  numpy  as  np

        time_steps,  rows,  cols,  channels  =  data.shape
        num_windows  =  (time_steps  -  window_size)  //  step

        for  start  in  range(0,  num_windows,  batch_size):
                end  =  min(start  +  batch_size,  num_windows)
                windows  =  np.array([
                        data[i:  i  +  window_size]
                        for  i  in  range(start  *  step,  end  *  step,  step)
                ],  dtype=np.float32)
                targets  =  np.array([
                        data[i  +  window_size,  :,  :,  0]
                        for  i  in  range(start  *  step,  end  *  step,  step)
                ],  dtype=np.float32)
                yield  windows,  targets

```

```
def  normalize_in_batches(data,  scaler=None,  batch_size=100):
        """
```

```
    分批归一化数据
    :param data: 输入数据，形状为 (num_windows, window_size, rows, cols, channels)
    :param scaler: 可选的 sklearn MinMaxScaler 实例
    :param batch_size: 批次大小
    :return: 归一化后的数据
    """
    import numpy as np
    from sklearn.preprocessing import MinMaxScaler

    if scaler is None:
        scaler = MinMaxScaler()

    data_reshaped = data.reshape(-1, data.shape[-1])
    num_samples = data_reshaped.shape[0]
    normalized_data = np.zeros_like(data_reshaped, dtype=np.float32)

    for start in range(0, num_samples, batch_size):
        end = min(start + batch_size, num_samples)
        normalized_data[start:end] = scaler.fit_transform(data_reshaped[start:end])

    return normalized_data.reshape(data.shape), scaler

# 归一化原始数据
convlstm_input_normalized, scaler = normalize_in_batches(convlstm_input)

```

```
# 一次性拟合 scaler
data_reshaped = convlstm_input.reshape(-1, convlstm_input.shape[-1])
scaler = MinMaxScaler()
scaler.fit(data_reshaped)

# 分批归一化
normalized_data = np.zeros_like(data_reshaped, dtype=np.float32)
num_samples = data_reshaped.shape[0]

for start in range(0, num_samples, batch_size):
    end = min(start + batch_size, num_samples)
    normalized_data[start:end] = scaler.transform(data_reshaped[start:end])

convlstm_input_normalized = normalized_data.reshape(convlstm_input.shape)

```

```
# 设置窗口参数
window_size = 10
step = 1
batch_size = 128

# # 初始化列表存储结果
# sliding_windows_list = []
# targets_list = []

# # 分批生成滑动窗口和目标值
# for batch_windows, batch_targets in apply_sliding_window_with_targets_in_batches(convlstm_input, window_size, step, batch_size):
#     sliding_windows_list.append(batch_windows)
#     targets_list.append(batch_targets)

# # 合并结果
# sliding_windows = np.concatenate(sliding_windows_list)
# targets = np.concatenate(targets_list)

# # 释放中间变量
# del sliding_windows_list, targets_list, convlstm_input
# import gc
# gc.collect()

```

```


# 滑动窗口生成
sliding_windows_list = []
targets_list = []

for batch_windows, batch_targets in apply_sliding_window_with_targets_in_batches(
    convlstm_input_normalized, window_size, step, batch_size
):
    sliding_windows_list.append(batch_windows)
    targets_list.append(batch_targets)

# 合并滑动窗口和目标
```

```
sliding_windows = np.concatenate(sliding_windows_list)
targets = np.concatenate(targets_list)

# 清理内存
del sliding_windows_list, targets_list
gc.collect()

# 数据集拆分
# train_windows, val_windows, train_targets, val_targets = split_dataset(sliding_windows, targets)

```

```
193
```

数据集拆分

```
def split_dataset(windows, targets, train_ratio=0.8):
    """
    划分数据集为训练集和验证集
    :param windows: 滑动窗口数据，形状为 (num_windows, window_size, rows, cols, channels)
    :param targets: 目标值，形状为 (num_windows, rows, cols)
    :param train_ratio: 训练集的比例
    :return: 训练集滑动窗口, 验证集滑动窗口, 训练集目标值, 验证集目标值
    """
    import numpy as np

    # 打乱索引
    indexes = np.arange(windows.shape[0])
    np.random.shuffle(indexes)

    # 划分数据集
    split_index = int(train_ratio * len(indexes))
    train_indexes = indexes[:split_index]
    val_indexes = indexes[split_index:]

    train_windows = windows[train_indexes]
    val_windows = windows[val_indexes]
    train_targets = targets[train_indexes]
    val_targets = targets[val_indexes]

    return train_windows, val_windows, train_targets, val_targets

```

```
# 数据集拆分
train_windows, val_windows, train_targets, val_targets = split_dataset(sliding_windows, targets)

# 释放中间变量
del sliding_windows, targets
gc.collect()

```

```
0
```

分批次归一化，然后合并

```
# def normalize_in_batches(data, scaler=None, batch_size=100):
#     """
#     分批归一化数据
#     :param data: 输入数据，形状为 (num_windows, window_size, rows, cols, channels)
#     :param scaler: 可选的 sklearn MinMaxScaler 实例
#     :param batch_size: 批次大小
#     :return: 归一化后的数据
#     """
#     import numpy as np
#     from sklearn.preprocessing import MinMaxScaler

#     if scaler is None:
#         scaler = MinMaxScaler()

#     data_reshaped = data.reshape(-1, data.shape[-1])
#     num_samples = data_reshaped.shape[0]
#     normalized_data = np.zeros_like(data_reshaped, dtype=np.float32)

#     for start in range(0, num_samples, batch_size):
#         end = min(start + batch_size, num_samples)
#         normalized_data[start:end] = scaler.fit_transform(data_reshaped[start:end])

```

```
#         return  normalized_data.reshape(data.shape),  scaler

```

```
#  #  对训练集归一化
#  train_normalized,  scaler  =  normalize_in_batches(train_windows)

#  #  使用相同  scaler  归一化验证集
#  val_normalized,  _  =  normalize_in_batches(val_windows,  scaler=scaler)

#  #  释放中间变量
#  del  train_windows,  val_windows
#  gc.collect()

```

```
#  def  normalize_in_batches_optimized(data,  scaler=None,  batch_size=100):
#          import  numpy  as  np
#          from  sklearn.preprocessing  import  MinMaxScaler

#          if  scaler  is  None:
#                  scaler  =  MinMaxScaler()

#          data_reshaped  =  data.reshape(-1,  data.shape[-1])
#          scaler.fit(data_reshaped)    #  全局拟合  scaler

#          num_samples  =  data_reshaped.shape[0]
#          normalized_data  =  np.zeros_like(data_reshaped,  dtype=np.float32)

#          for  start  in  range(0,  num_samples,  batch_size):
#                  end  =  min(start  +  batch_size,  num_samples)
#                  normalized_data[start:end]  =  scaler.transform(data_reshaped[start:end])

#          return  normalized_data.reshape(data.shape),  scaler

```

构建Tensorflow数据集

```
import  tensorflow  as  tf

batch_size  =  16

#  创建训练集和验证集
train_dataset  =  tf.data.Dataset.from_tensor_slices((train_normalized,  train_targets))
train_dataset  =  train_dataset.batch(batch_size).shuffle(buffer_size=100)

val_dataset  =  tf.data.Dataset.from_tensor_slices((val_normalized,  val_targets))
val_dataset  =  val_dataset.batch(batch_size)

```

验证

[ ] ↳ *已隐藏 2 个单元格*

模型构建

```
#  import  tensorflow  as  tf
#  import  numpy  as  np
#  from  tensorflow.keras  import  Sequential
#  from  tensorflow.keras.layers  import  ConvLSTM2D,  BatchNormalization,  Conv3D

#  #  模拟归一化后的训练和验证数据
#  #  假设  train_normalized  和  val_normalized  形状为  (num_samples,  time_steps,  height,  width,  channels)
#  num_samples  =  100
#  time_steps  =  10
#  height,  width,  channels  =  97,  97,  3


#  #  定义前  20  帧预测后  20  帧的逻辑
#  def  create_shifted_frames(data,  input_length,  predict_length):
#          x,  y  =  [],  []
#          for  i  in  range(data.shape[1]  -  input_length  -  predict_length  +  1):
#                  x.append(data[:,  i:i  +  input_length,  :,  :,  :])
#                  y.append(data[:,  i  +  input_length:i  +  input_length  +  predict_length,  :,  :,  :])
#          return  np.concatenate(x,  axis=0),  np.concatenate(y,  axis=0)
```

```
# input_length = 20
# predict_length = 20

# x_train, y_train = create_shifted_frames(train_normalized, input_length, predict_length)
# x_val, y_val = create_shifted_frames(val_normalized, input_length, predict_length)

# print("x_train shape:", x_train.shape)
# print("y_train shape:", y_train.shape)

# # 构建 tf.data.Dataset
# batch_size = 16
# train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
# train_dataset = train_dataset.shuffle(buffer_size=100).batch(batch_size).prefetch(buffer_size=tf.data.AUTOTUNE)

# val_dataset = tf.data.Dataset.from_tensor_slices((x_val, y_val))
# val_dataset = val_dataset.batch(batch_size).prefetch(buffer_size=tf.data.AUTOTUNE)

# # 构建模型
# model = Sequential([
#     ConvLSTM2D(filters=64, kernel_size=(5, 5), input_shape=(None, 97, 97, 3),
#                padding='same', return_sequences=True),
#     BatchNormalization(),
#     ConvLSTM2D(filters=64, kernel_size=(3, 3), padding='same', return_sequences=True),
#     BatchNormalization(),
#     ConvLSTM2D(filters=64, kernel_size=(1, 1), padding='same', return_sequences=True),
#     Conv3D(filters=1, kernel_size=(3, 3, 3), activation='sigmoid', padding='same')
# ])

# model.compile(loss='binary_crossentropy', optimizer='adadelta')

# # 打印模型结构
# model.summary()

# # 训练模型
# model.fit(train_dataset, validation_data=val_dataset, epochs=20)

```

加载数据到TensorFlow

[ ] ↳ *已隐藏 9 个单元格*

convLSTM模型构建

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import ConvLSTM2D, BatchNormalization, Conv2D, Flatten, Dense,Reshape

# def create_convlstm_model(input_shape, output_size):
#     model = Sequential([
#         ConvLSTM2D(filters=32, kernel_size=(3, 3), padding="same", return_sequences=False, input_shape=input_shape),
#         BatchNormalization(),
#         Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding="same"),
#         Flatten(),
#         Dense(128, activation='relu'),
#         Dense(output_size, activation='linear')
#     ])
#     return model

def create_convlstm_model(input_shape,output_size):

    model = Sequential([
        ConvLSTM2D(filters=32, kernel_size=(5, 5), padding="same", return_sequences=False, input_shape=input_shape),
        BatchNormalization(),
        Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding="same"),
        Flatten(),
        Dense(128, activation='relu'),
        Dense(output_size, activation='tanh'),  # 全连接层，输出展平的网格
        # Dense(output_size, activation='linear'),  # 全连接层，输出展平的网格
        Reshape((97, 97))  # 调整形状为 (97, 97)
    ])
    return model

```

```
# # 展平目标值
# train_targets = train_targets.reshape(train_targets.shape[0], -1)  # (batch_size, 9409)
# val_targets = val_targets.reshape(val_targets.shape[0], -1)        # (batch_size, 9409)
```

```
# # 模型构建核心代码，这里我们修改超参数与keras官方超参数一致
# model = Sequential([
#         keras.layers.ConvLSTM2D(filters=64, kernel_size=(5, 5),
#                                 input_shape=(None, 97, 97, 3),
#                                 padding='same', return_sequences=True),
#         keras.layers.BatchNormalization(),
#         keras.layers.ConvLSTM2D(filters=64, kernel_size=(3, 3),
#                                 padding='same', return_sequences=True),
#         keras.layers.BatchNormalization(),
#         keras.layers.ConvLSTM2D(filters=64, kernel_size=(1, 1),
#                                 padding='same', return_sequences=True),
#         keras.layers.Conv3D(filters=1, kernel_size=(3, 3, 3),
#                             activation='sigmoid',
#                             padding='same', data_format='channels_last')
# ])
# model.compile(loss='binary_crossentropy', optimizer='adadelta')
# model.summary()

```

```
# # 保存模型
# model.save('conv_lstm_model.h5')

# # 加载模型
# from tensorflow.keras.models import load_model
# model = load_model('conv_lstm_model.h5')

```

```
input_shape = (10, 97, 97, 1)   # 与你的训练数据形状匹配
# output_size = (97 , 97)   # 每个网格点的目标数量
output_size = 97 * 97   # 每个网格点的目标数量
# model = create_convlstm_model(input_shape)
model = create_convlstm_model(input_shape, output_size)


import tensorflow as tf

def weighted_mse(y_true, y_pred):
    weights = tf.where(y_true > 0, 5.0, 1.0)   # 非零值权重设为 10，零值权重为 1
    return tf.reduce_mean(weights * tf.square(y_true - y_pred))

model.compile(optimizer='adam', loss=weighted_mse, metrics=['mae'])



# model.compile(optimizer='adam', loss='mse', metrics=['mae'])
model.summary()

```

Model: "sequential_9"

Layer (type)	Output Shape	Param #
conv_lstm2d_9 (ConvLSTM2D)	(None, 97, 97, 32)	105,728
batch_normalization_9 (BatchNormalization)	(None, 97, 97, 32)	128
conv2d_9 (Conv2D)	(None, 97, 97, 64)	18,496
flatten_9 (Flatten)	(None, 602176)	0
dense_18 (Dense)	(None, 128)	77,078,656
dense_19 (Dense)	(None, 9409)	1,213,761
reshape_9 (Reshape)	(None, 97, 97)	0

Total params: 78,416,769 (299.14 MB)
Trainable params: 78,416,705 (299.14 MB)
Non-trainable params: 64 (256.00 B)

训练

```
# # 填充时间序列到统一长度（假设最大时间步为 20）
# max_time_steps = 20
# x_train = tf.keras.preprocessing.sequence.pad_sequences(x_train, maxlen=max_time_steps, dtype='float32', padding='post')
```

```
# y_train = tf.keras.preprocessing.sequence.pad_sequences(y_train, maxlen=max_time_steps, dtype='float32', padding='post')

```

双击（或按回车键）即可修改

```
epochs = 10
history = model.fit(
    train_dataset,
    # validation_data=val_dataset,
    epochs=epochs,
    verbose=1
)

```

```
Epoch 1/10
58/58 ━━━━━━━━━━━━━━━━━━━━ 95s 2s/step - loss: 1.3539e-04 - mae: 0.0014
Epoch 2/10
58/58 ━━━━━━━━━━━━━━━━━━━━ 94s 2s/step - loss: 1.1214e-04 - mae: 0.0013
Epoch 3/10
58/58 ━━━━━━━━━━━━━━━━━━━━ 96s 2s/step - loss: 1.2119e-04 - mae: 0.0014
Epoch 4/10
58/58 ━━━━━━━━━━━━━━━━━━━━ 97s 2s/step - loss: 1.2880e-04 - mae: 0.0014
Epoch 5/10
58/58 ━━━━━━━━━━━━━━━━━━━━ 97s 2s/step - loss: 1.3543e-04 - mae: 0.0014
Epoch 6/10
58/58 ━━━━━━━━━━━━━━━━━━━━ 95s 2s/step - loss: 1.3232e-04 - mae: 0.0014
Epoch 7/10
58/58 ━━━━━━━━━━━━━━━━━━━━ 95s 2s/step - loss: 1.2787e-04 - mae: 0.0014
Epoch 8/10
58/58 ━━━━━━━━━━━━━━━━━━━━ 95s 2s/step - loss: 1.3805e-04 - mae: 0.0014
Epoch 9/10
58/58 ━━━━━━━━━━━━━━━━━━━━ 95s 2s/step - loss: 1.1223e-04 - mae: 0.0014
Epoch 10/10
58/58 ━━━━━━━━━━━━━━━━━━━━ 96s 2s/step - loss: 1.3654e-04 - mae: 0.0014
```

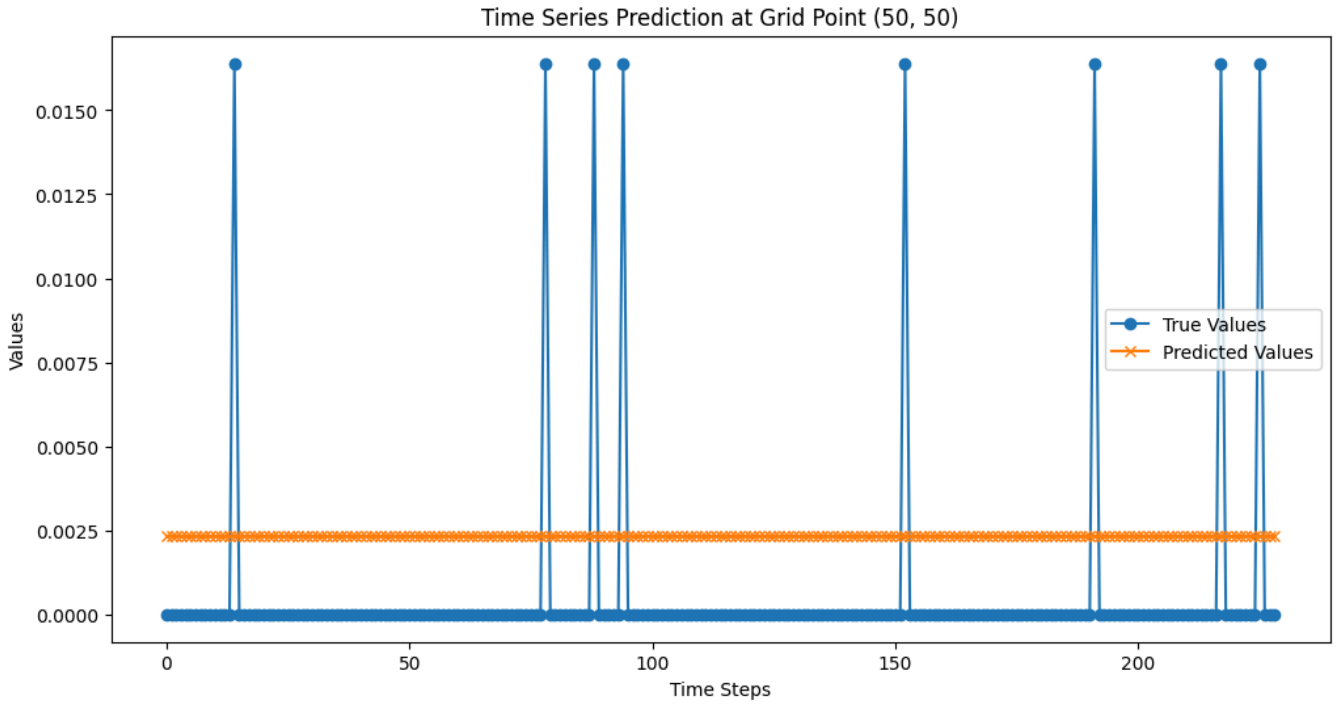
```

```

验证/评估

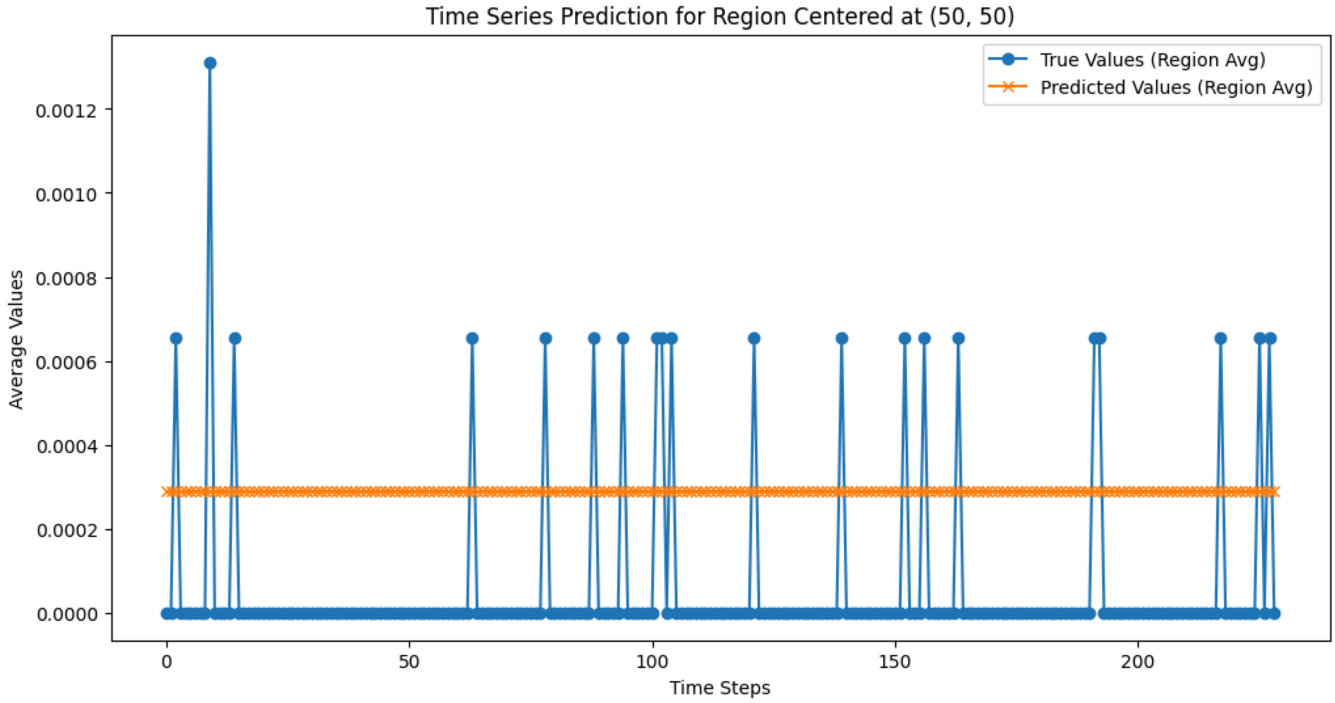
```
# val_loss, val_mae = model.evaluate(val_dataset)
# print(f"Validation Loss: {val_loss}, Validation MAE: {val_mae}")

```

```
# 评估模型整体性能
val_loss, val_mae = model.evaluate(val_dataset)
print(f"Validation Loss: {val_loss}, Validation MAE: {val_mae}")

# 获取验证集逐样本预测值和真实值
val_data = []
val_true = []

for batch in val_dataset:
    inputs, targets = batch
    val_data.append(inputs.numpy())
    val_true.append(targets.numpy())

val_data = np.concatenate(val_data, axis=0)
val_true = np.concatenate(val_true, axis=0)
val_predictions = model.predict(val_data).reshape(val_true.shape)

# 可视化分析：时间序列对比
grid_point = (50, 50)  # 选择一个网格点
window_idx = 0  # 选择一个时间窗口
true_series = val_true[:, grid_point[0], grid_point[1]]
pred_series = val_predictions[:, grid_point[0], grid_point[1]]

plt.figure(figsize=(12, 6))
plt.plot(true_series, label="True Values", marker="o")
plt.plot(pred_series, label="Predicted Values", marker="x")
plt.xlabel("Time Steps")
plt.ylabel("Values")
plt.title(f"Time Series Prediction at Grid Point {grid_point}")
plt.legend()
plt.show()

```

```
15/15 ━━━━━━━━━━━━━━━━━━━━ 7s 488ms/step - loss: 1.3704e-04 - mae: 0.0014
Validation Loss: 0.00012709625298157334, Validation MAE: 0.0014083877904340625
8/8 ━━━━━━━━━━━━━━━━━━━━ 8s 925ms/step
```




```
# 设置局部区域的范围
center_point = (50, 50)  # 选定中心点
radius = 2  # 半径范围（例：2表示以中心点为中心的5x5区域）
x_start, x_end = center_point[0] - radius, center_point[0] + radius + 1
y_start, y_end = center_point[1] - radius, center_point[1] + radius + 1

# 提取局部区域真实值和预测值
true_region = val_true[:, x_start:x_end, y_start:y_end].mean(axis=(1, 2))  # 区域平均
pred_region = val_predictions[:, x_start:x_end, y_start:y_end].mean(axis=(1, 2))  # 区域平均

# 绘制时间序列对比
plt.figure(figsize=(12, 6))
plt.plot(true_region, label="True Values (Region Avg)", marker="o")
plt.plot(pred_region, label="Predicted Values (Region Avg)", marker="x")
plt.xlabel("Time Steps")
plt.ylabel("Average Values")
plt.title(f"Time Series Prediction for Region Centered at {center_point}")
plt.legend()
plt.show()

```



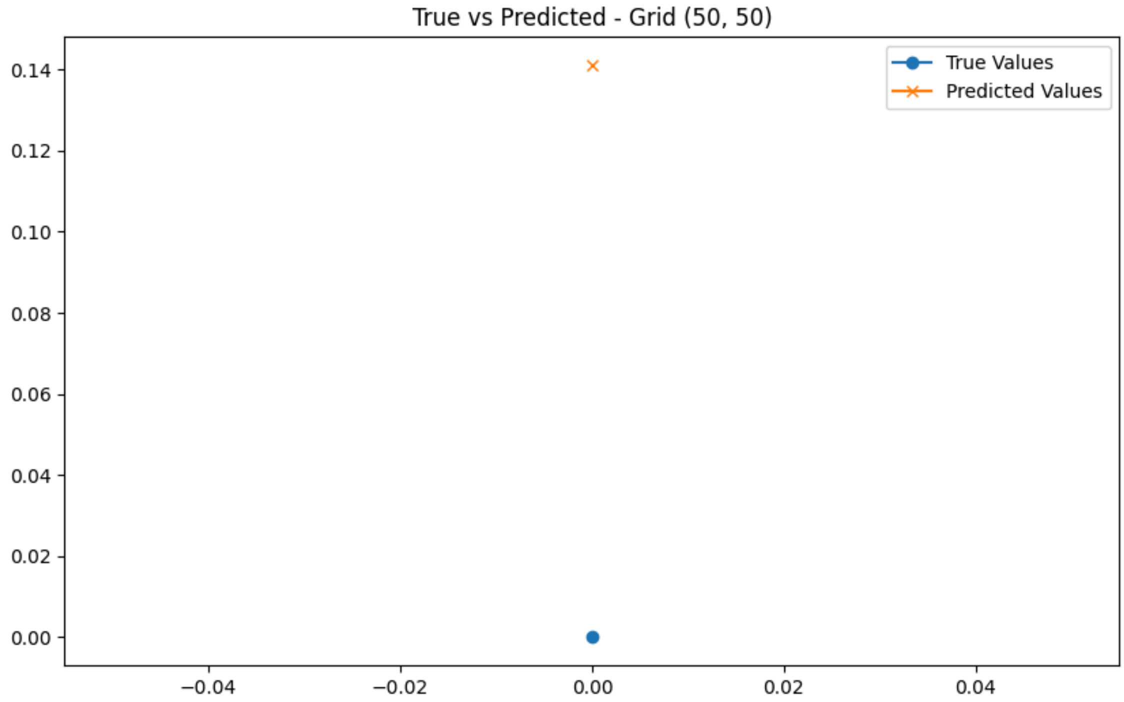
还原：返归一

```
# 反归一化
val_predictions_broadcasted = val_predictions.reshape(-1, 1)
val_true_broadcasted = val_true.reshape(-1, 1)

val_predictions_restored = scaler.inverse_transform(val_predictions_broadcasted).flatten()
val_true_restored = scaler.inverse_transform(val_true_broadcasted).flatten()

val_predictions_restored = val_predictions_restored.reshape(val_true.shape)
val_true_restored = val_true_restored.reshape(val_true.shape)
#
```

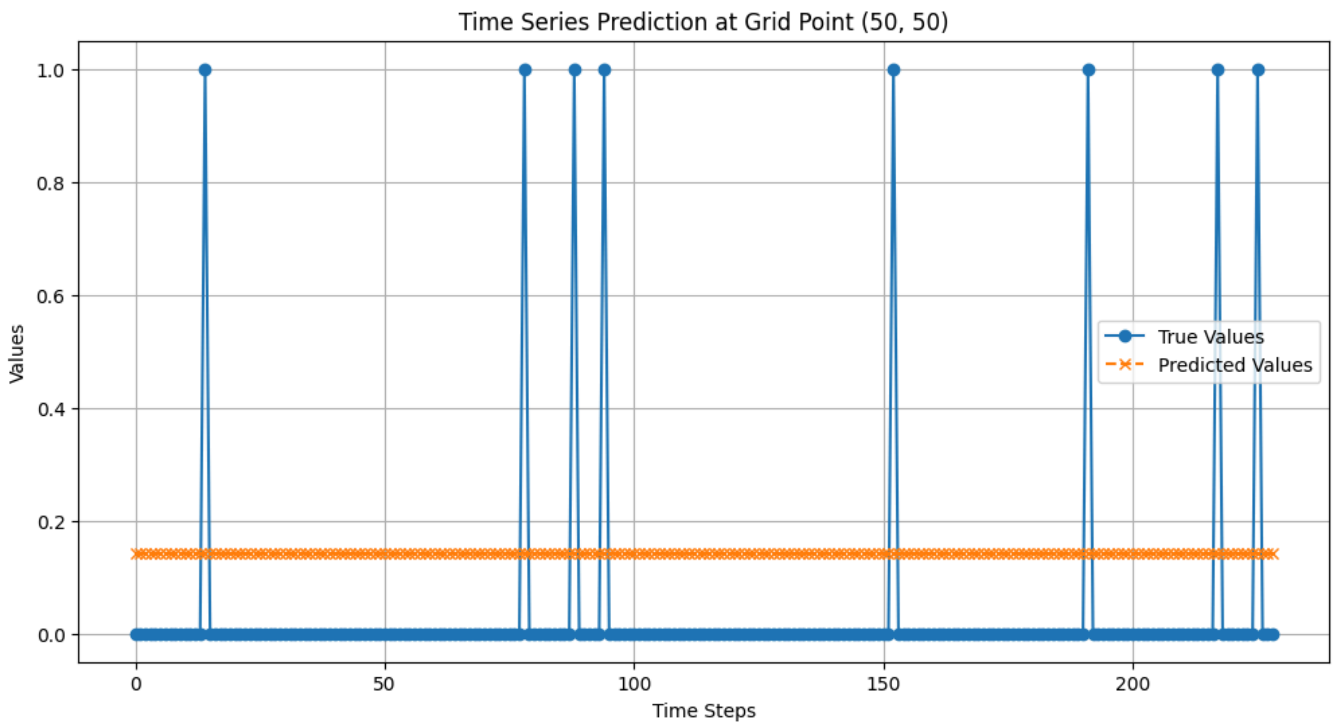
```
print("Restored predictions shape:", val_predictions_restored.shape)
print("Restored true values shape:", val_true_restored.shape)

```

```
Restored predictions shape: (229, 97, 97)
Restored true values shape: (229, 97, 97)
```

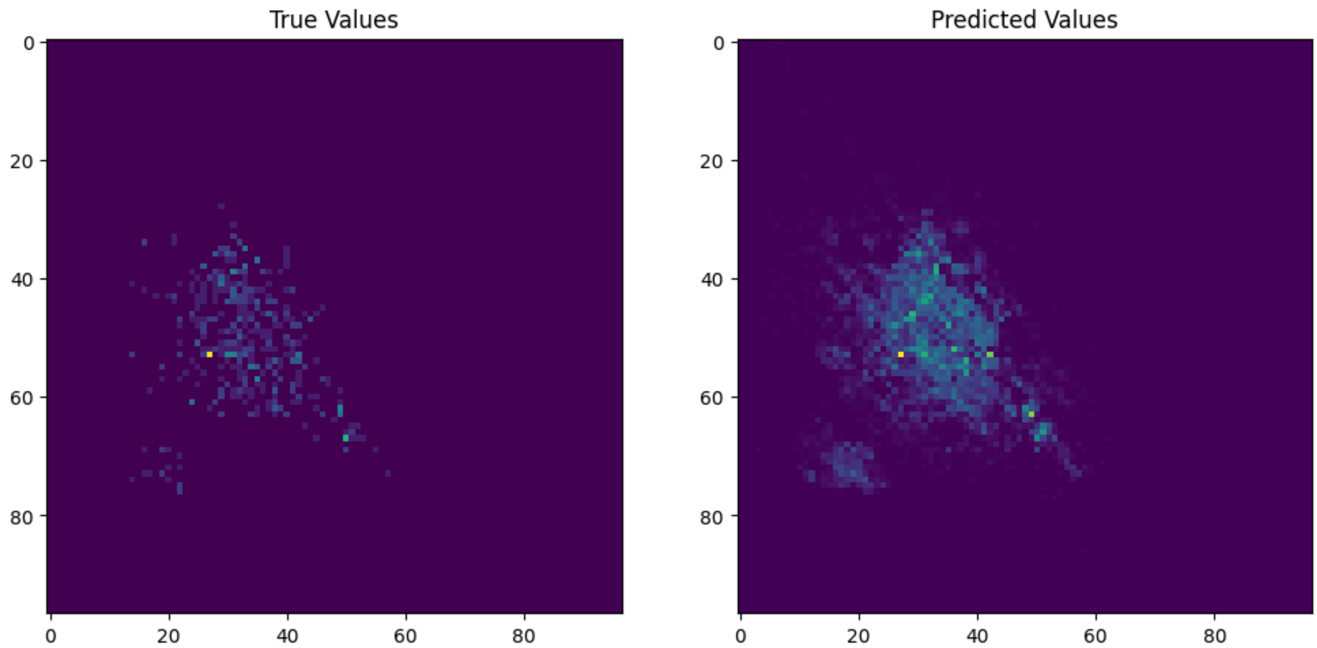
```
# 绘制第一个网格点的预测和真实值对比
grid_point = (50, 50)  # 选择一个网格点
window_idx = 0  # 选择第一个窗口

plt.figure(figsize=(10, 6))
plt.plot(val_true_restored[window_idx, grid_point[0], grid_point[1]], label='True Values', marker='o')
plt.plot(val_predictions_restored[window_idx, grid_point[0], grid_point[1]], label='Predicted Values', marker='x')
plt.legend()
plt.title(f"True vs Predicted - Grid {grid_point}")
plt.show()

```



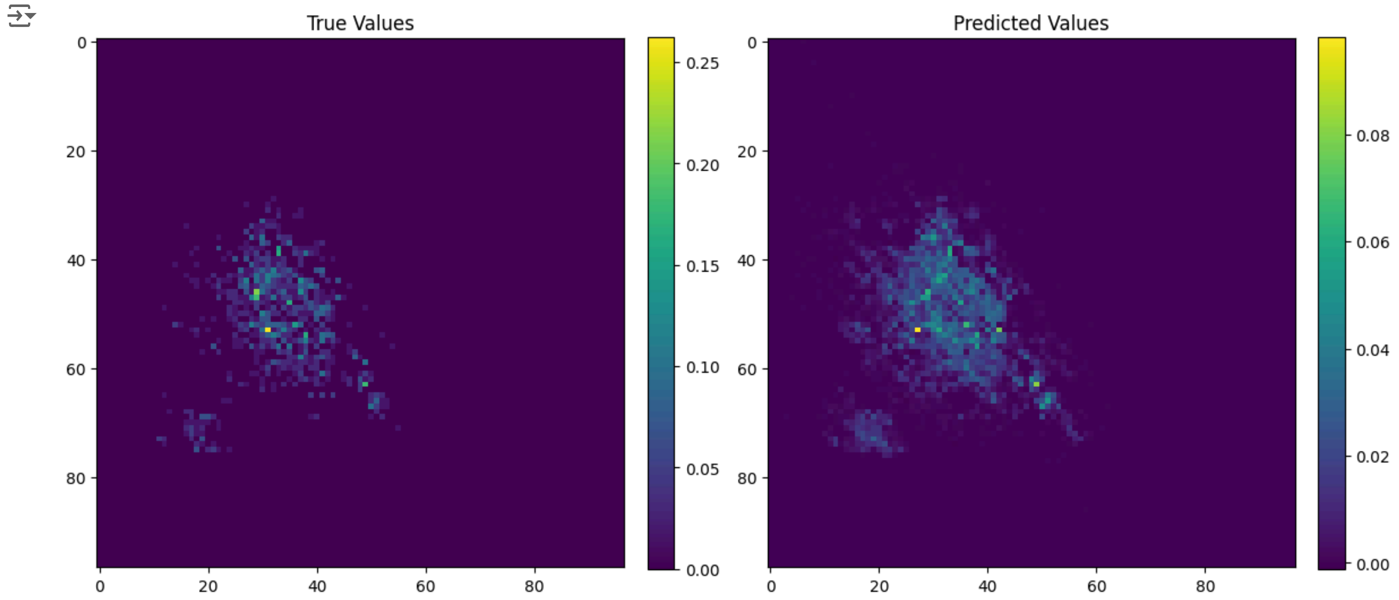

```
# 选择一个验证窗口和网格点
window_idx = 200  # 第一个滑动窗口
grid_point = (50, 50)  # 网格点 (50, 50)

# 获取真实值和预测值的时间序列
true_series = val_true_restored[:, grid_point[0], grid_point[1]]
pred_series = val_predictions_restored[:, grid_point[0], grid_point[1]]

# 创建时间轴
time_steps = np.arange(len(true_series))

# 绘图
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
plt.plot(time_steps, true_series, label='True Values', marker='o', linestyle='-')
plt.plot(time_steps, pred_series, label='Predicted Values', marker='x', linestyle='--')
plt.xlabel('Time Steps')
plt.ylabel('Values')
plt.title(f'Time Series Prediction at Grid Point {grid_point}')
plt.legend()
plt.grid()
plt.show()

```




```
# 选择一个验证窗口 index
window_idx =50   # 第一个滑动窗口

# 获取真实值和预测值的空间分布
true_grid = val_true[window_idx]
pred_grid = val_predictions[window_idx]

# 绘制空间分布对比
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

# 真实值
axes[0].imshow(true_grid, cmap='viridis')
axes[0].set_title('True Values')

# 预测值
axes[1].imshow(pred_grid, cmap='viridis')
axes[1].set_title('Predicted Values')

plt.show()




```



```python
#  选择一个验证窗口  index
window_idx  =  2    #  第一个滑动窗口

#  获取真实值和预测值的空间分布
true_grid  =  val_true[window_idx]
pred_grid  =  val_predictions[window_idx]

#  绘制空间分布对比
fig,  axes  =  plt.subplots(1,  2,  figsize=(12,  6))

#  真实值
im1  =  axes[0].imshow(true_grid,  cmap='viridis')
axes[0].set_title('True  Values')
plt.colorbar(im1,  ax=axes[0],  fraction=0.046,  pad=0.04)    #  添加颜色条

#  预测值
im2  =  axes[1].imshow(pred_grid,  cmap='viridis')
axes[1].set_title('Predicted  Values')
plt.colorbar(im2,  ax=axes[1],  fraction=0.046,  pad=0.04)    #  添加颜色条

plt.tight_layout()    #  调整布局以防止重叠
plt.show()

```



有slide_window的数据集拆分

转换成ConvLSTM需要的数据形式（batch size暂时不考虑，在处理完滑动窗口期分数据集之后，训练时引入动态batchsize）

[ ] ↳ *已隐藏 1 个单元格*

应用滑动窗口

[ ] ↳ *已隐藏 1 个单元格*

拆分数据集

[ ] ↳ *已隐藏 1 个单元格*

去除周末时间

[ ] ↳ *已隐藏 2 个单元格*

归一化

[ ] ↳ *已隐藏 1 个单元格*

完整流程

[ ] ↳ *已隐藏 6 个单元格*

无slide_window的数据集拆分

[ ] ↳ *已隐藏 3 个单元格*

Tensorflow

```
# from tensorflow.keras.models import Sequential
# from tensorflow.keras.layers import ConvLSTM2D, BatchNormalization, Conv2D, Flatten, Dense

# # 定义模型
# def create_convlstm_model(input_shape, output_size):
#     """
#     创建 ConvLSTM 模型
#     :param input_shape: 输入形状 (time_steps, rows, cols, channels)
#     :param output_size: 输出维度
#     :return: TensorFlow Keras 模型
#     """
#     model = Sequential([
#         # ConvLSTM2D 层
#         ConvLSTM2D(filters=32, kernel_size=(3, 3), padding="same", return_sequences=False, input_shape=input_shape),
#         BatchNormalization(),

#         # 卷积层
#         Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding="same"),
#         Flatten(),

#         # 全连接层
#         Dense(128, activation='relu'),
#         Dense(output_size, activation='linear')  # 线性激活用于回归
#     ])
#     return model

```

```
# input_shape = train_normalized_data.shape[1:]  # (time_steps, rows, cols, channels)
# output_size = 1  # 假设是单值回归任务

# model = create_convlstm_model(input_shape, output_size)
# model.summary()

```

```
# model.compile(
#     optimizer='adam',
#     loss='mse',  # 回归任务
#     metrics=['mae']  # 平均绝对误差
# )

```

```
# import tensorflow as tf

# # 创建 TensorFlow Dataset
# batch_size = 16

# train_dataset = tf.data.Dataset.from_tensor_slices((train_normalized_data, train_targets))
# train_dataset = train_dataset.batch(batch_size).shuffle(buffer_size=100)

# val_dataset = tf.data.Dataset.from_tensor_slices((val_normalized_data, val_targets))
# val_dataset = val_dataset.batch(batch_size)

```

```
# # 训练模型
# epochs = 50
# history = model.fit(
```

```
#         train_dataset,
#         validation_data=val_dataset,
#         epochs=epochs,
#         verbose=1
# )

```

手动归一化

```
# def min_max_normalize(data, min_val=None, max_val=None):
#         if min_val is None:
#               min_val = data.min()
#         if max_val is None:
#               max_val = data.max()
#         return (data - min_val) / (max_val - min_val), min_val, max_val

```

```
# def standardize(data, mean=None, std=None):
#         if mean is None:
#               mean = data.mean()
#         if std is None:
#               std = data.std()
#         return (data - mean) / std, mean, std

```

```
# train_dataset[..., 0], min_val, max_val = min_max_normalize(train_dataset[..., 0])
# val_dataset[..., 0] = (val_dataset[..., 0] - min_val) / (max_val - min_val)

```

```
# # 模拟雷达时间序列，240 帧，大小 100x100
# time_steps = 1154
# height, width = 97, 97
# # radar_data = np.random.rand(time_steps, height, width)  # 随机模拟序列数据


# # 滑窗参数
# window_size = 5   # 输入序列长度
# step_size = 3      # 滑动步长

# # 滑窗生成序列 block
# X_blocks = []
# for i in range(0, time_steps - window_size, step_size):
#         X_blocks.append(radar_data[i:i + window_size])  # 每个 block 包含 5 帧

# X_blocks = np.array(X_blocks)  # 转换为 NumPy 数组
# print("生成的 block 数量:", X_blocks.shape[0])  # 输出 block 数量
# print("单个 block 的形状:", X_blocks.shape[1:])  # 输出单个 block 的形状

```

```
# import numpy as np

# # 假设 8 天的订单数据，时间步为 192，网格为 32x32
# time_steps = 1154
# height, width = 97, 97
```

```

# # 随机生成订单需求数据（每个时间步每个网格的订单数量）
# # data = np.random.randint(0, 100, size=(time_steps, height, width))  # 0-100 随机订单

# # 滑动窗口生成样本 (n=6, m=1)
# n, m = 6, 1  # 输入过去 1 小时，预测未来 10 小时
# X, y = [], []
# for i in range(len(data) - n - m + 1):
#     X.append(data[i:i+n])          # 输入的 1 小时
#     y.append(data[i+n:i+n+m])      # 预测的 10分钟

# X = np.array(X)  # 转换为数组，形状为 (样本数, n, height, width)
# y = np.array(y)  # 转换为数组，形状为 (样本数, m, height, width)

# # 检查数据形状
# print("输入数据形状:", X.shape)  # (样本数, 6, 97, 97)
# print("输出数据形状:", y.shape)  # (样本数, 1, 97, 97)
```

```
# from tensorflow.keras.models import Sequential
# from tensorflow.keras.layers import ConvLSTM2D, BatchNormalization, Conv3D

# model = Sequential([
#     ConvLSTM2D(filters=64, kernel_size=(3, 3), activation='relu', input_shape=(time_steps, height, width, features)),
#     BatchNormalization(),
#     ConvLSTM2D(filters=32, kernel_size=(3, 3), activation='relu', return_sequences=False),
#     BatchNormalization(),
#     Conv3D(filters=1, kernel_size=(3, 3, 3), activation='sigmoid')
# ])
# model.compile(optimizer='adam', loss='mse')
# model.summary()

```

```
# def preprocess_order_data(df, time_col, interval='10min'):
#     # Ensure time_col is in datetime format
#     df = df.assign(y=df2["y"].reset_index(drop=True))

#     df[time_col] = pd.to_datetime(df[time_col])
# s
#     # Add cyclic time features
#     day = 24 * 60  # Total minutes in a day
```