

# Blood Glucose Level Time Series Prediction

## 血糖水平时间序列预测实验报告

小组成员：

2151136 朱开来

2152967 沈昊景

2153051 刘杰

2153604 郭晓杰

### 一. 领域知识理解

#### 血糖调节机制

血糖调节是一个复杂的生理过程，主要由胰岛素和胰高血糖素这两种激素来调控。胰岛素由胰腺中的 $\beta$ 细胞分泌，其主要功能是降低血糖水平。胰岛素通过促进细胞对葡萄糖的摄取和利用，以及抑制肝脏葡萄糖的生成，来实现这一目标。相反，胰高血糖素由胰腺中的 $\alpha$ 细胞分泌，其主要功能是升高血糖水平，主要通过促进肝脏分解糖原和生成新的葡萄糖来实现。

#### 影响血糖水平的因素

血糖水平受多种因素影响，这些因素可以是内源性的（如体内激素的变化）或外源性的（如饮食、运动等）。

##### ➤ 饮食

饮食是影响血糖水平的主要外源性因素之一。碳水化合物的摄入直接影响血糖水平，因为碳水化合物在消化道内被分解成葡萄糖并被吸收入血。不同种类的碳水化合物对血糖的影响不同。例如，高血糖指数（GI）的食物会导致血糖快速升高，而低 GI 食物则会导致血糖缓慢上升。此外，饮食中的脂肪和蛋白质也会影响血糖，但其影响机制和程度不同于碳水化合物。

##### ➤ 运动

运动是另一个重要的外源性因素。运动可以通过多种机制降低血糖水平。首先，运动时肌肉对葡萄糖的需求增加，这会促使更多的葡萄糖从血液中被摄取。其次，长期规律的运动可以提高胰岛素敏感性，使得体细胞对胰岛素的反应更好，进而更有效地降低血糖水平。然而，过度运动或剧烈运动也可能导致低血糖，特别是对使用胰岛素治疗的患者。

##### ➤ 药物

药物是糖尿病管理的重要组成部分。根据作用机制的不同，降糖药物可以分为多种类型，包括胰岛素、口服降糖药和非胰岛素类注射药物。胰岛素治疗主要用于 1 型糖尿病患者以及某些 2 型糖尿病患者。口服降糖药则主要用于 2 型糖尿病患者，包括磺脲类、双胍类、 $\alpha$ -糖苷酶抑制剂等。每种药物都有其特定的作用机制和适用情况，需要根据患者的具体情况进行个性化选择和调整。

糖尿病的类型及其管理

糖尿病主要分为 1 型糖尿病和 2 型糖尿病，此外还有妊娠糖尿病和其他特定类型的糖尿病。在本次研究中，主要关注了 1 型糖尿病和 2 型糖尿病病人的数据，对其血糖含量进行预测。

◆ 1 型糖尿病

1 型糖尿病是由于胰岛β细胞被自身免疫系统破坏，导致胰岛素绝对缺乏。这类患者需要终生依赖胰岛素治疗，包括皮下注射胰岛素或使用胰岛素泵。除了胰岛素治疗，1 型糖尿病患者还需要密切监测血糖水平，并注意饮食和运动的调整，以保持血糖的稳定。

◆ 2 型糖尿病

2 型糖尿病是由于胰岛素抵抗和胰岛素分泌不足共同导致的。这类患者通常通过生活方式的改变（如饮食控制、增加运动）来管理血糖。此外，还可能需需要口服降糖药或胰岛素治疗。2 型糖尿病的管理强调个体化治疗，根据患者的具体情况制定综合的管理计划，包括药物治疗、饮食调整、运动指导和血糖监测。










◆ 妊娠糖尿病

妊娠糖尿病是指妊娠期间首次发现或发生的糖尿病。管理妊娠糖尿病的关键在于通过饮食和运动来控制血糖，同时可能需要胰岛素治疗，以确保母亲和胎儿的健康。

二. 数据收集和预处理

根据数据集介绍的论文：《Chinese diabetes datasets for data-driven machine learning》，我们可以得知，尽管 CGM 技术能够提供大量连续血糖数据，但公开的血糖数据集较少，尤其是 T2DM 的数据集。这些数据集在患者数量、地理范围和数据类型上存在限制。常用的 T1DM 数据集包括 UVA/Padova 模拟器、OhioT1DM 和 D1NAMO 等，而 T2DM 数据集较少，如 Maryland 数据集和 Maastricht 研究。为了填补这一空白，数据集的作者构建了包含中国上海 T1DM 和 T2DM 患者详细信息两个新数据集，提供了饮食、临床特征、实验室测量和药物信息，支持数据驱动的机器学习研究。

接下来是对数据预处理的过程。我们对两种糖尿病的数据进行了分批处理。数据清理代码文件的目录如下图所示：

 .idea	2024/6/8 20:37	文件夹	
 generated_data	2024/6/9 12:39	文件夹	
 original_data	2024/6/8 20:38	文件夹	
 output	2024/6/8 20:38	文件夹	
 venv	2024/6/8 20:41	文件夹	
 pre_T1DM.py	2024/6/3 14:18	Python File	3 KB
 pre_T2DM.py	2024/6/3 14:14	Python File	3 KB
 T1DM.py	2024/6/3 14:19	Python File	4 KB
 T2DM.py	2024/5/31 17:48	Python File	4 KB

下一部分将展示一下数据处理的代码，这里主要讲解一下对于 T1DM 数据的处理过程，对于

T2DM 的数据进行类似地处理即可：

## pre\_T1DM

```
import openpyxl
import os
import glob

for index in range(1000, 1013): # 这个循环只会执行一次
    try:
        file_pattern = os.path.join('original_data/Shanghai_T1DM/', f'{index}'.xlsx')
        all_files = glob.glob(file_pattern)

        # 提取文件名
        all_file_names = [os.path.basename(file) for file in all_files]

        # 打印文件名
        for filename in all_file_names:
            try:
                print(filename)
                workbook = openpyxl.load_workbook("original_data/Shanghai_T1DM/" + filename)

                # 获取第一个工作表（这里只有一个）
                sheet = workbook.active

                # 获取合并单元格
                merged_cells = sheet.merged_cells

                # 强制加入11列，防止出现空表（无合并单元格）情况
                sheet.cell(1, 12).value = "CSII - basal insulin (Novolin R, IU / H)"

                # 遍历每一个合并单元格
                for merged_cell in merged_cells:
                    start_row, start_column, end_row, end_column = merged_cell.min_row, merged_cell.min_col, merged_cell.max_row, merged_cell.max_col

                    # 获取合并单元格的值
                    merged_value = sheet.cell(start_row, start_column).value

                    # 拆分合并单元格，并填充值
                    for row in range(start_row, end_row + 1):
                        for col in range(start_column, end_column + 1):
                            # 建立一个新列，把数据放进去
                            sheet.cell(row, col + 2).value = merged_value

                # 处理合并单元格间的单个单元格
                for row in sheet.iter_rows():
                    cell = row[11] # 新列11
                    if cell.value is None:
                        cell.value = row[9].value
                    # 暂停时全部置为0
                    if cell.value == "temporarily suspend insulin delivery":
                        cell.value = 0

                workbook.save("generated_data/Shanghai_T1DM/" + filename)
            except Exception as e:
                print(f"Error processing file {filename}: {e}")
    except Exception as e:
        print(f"Error during processing index {index}: {e}")
```

pre\_T1DM.py 文件的主要功能是处理一组特定目录下的 Excel 文件。其作用可以总结如下：

- ◆ 导入必要的库：
  - openpyxl：用于处理 Excel 文件。
  - os 和 glob：用于文件操作和查找。
- ◆ 文件匹配与查找：
  - 使用 glob 模块查找文件名模式匹配 original\_data/Shanghai\_T1DM/ 目录下以 1000 到 1012 为前缀的 Excel 文件。
- ◆ 文件处理：
  - 打开每一个查找到的匹配的 Excel 文件。
  - 获取第一个工作表并处理合并单元格。
  - 将合并单元格的值分配到单个单元格中。
  - 在第 12 列添加一个新列标题。
  - 将一些特定单元格值替换为 0（如 "temporarily suspend insulin delivery"）。
  - 保存处理后的文件到 generated\_data/Shanghai\_T1DM/ 目录。

## T1DM

```
import warnings

import pandas as pd
import glob
import os
from pandas.core.common import SettingWithCopyWarning

warnings.simplefilter(action="ignore", category=SettingWithCopyWarning)

for index in range(1000, 1013):
    file_pattern = os.path.join('generated_data/Shanghai_T1DM/', f'{index}*')
    all_files = glob.glob(file_pattern)
    # 提取文件名
    all_file_names = [os.path.basename(file) for file in all_files]
    # 打印文件名
    # print(all_file_names)

    # # 匹配模式
    # file_pattern = os.path.join('data/Shanghai_T1DM/', '2008*')
    # all_files = glob.glob(file_pattern)
    #
    # # 提取文件名
    # all_file_names = [os.path.basename(file) for file in all_files]
    #
    # # 打印文件名
    # print(all_file_names)
```

```

for filename in all_file_names:
    # 匹配的名字没有文件后缀
    Match_name = filename.replace('.xlsx', '').replace('.xls', '')

    # 从总表中读取包含要匹配数据的表
    summary_data_table = pd.read_excel('generated_data/Shanghai_T1DM_Summary.xlsx')
    # 根据文件名从总表中找到匹配行
    filtered_dt = summary_data_table[summary_data_table['Patient Number'].str.contains(Match_name)]
    # 提取前五列 (ID, 性别, 年龄, 身高, 体重)
    summary_needed_columns = filtered_dt.iloc[:, :5]

    # 将提取数据和对应的病人的表合并
    patient_data_table = pd.read_excel('generated_data/Shanghai_T1DM/' + filename)

    patient_data_table['Date'] = pd.to_datetime(patient_data_table['Date'])

    # 提取时间部分并转换为字符串
    patient_data_table['Time'] = patient_data_table['Date'].dt.strftime('%H:%M')

    # 删除原始 'Date' 列
    patient_data_table.drop(columns=['Date'], inplace=True)
    Time_column = patient_data_table.pop('Time')
    patient_data_table.insert(0, 'Time', Time_column)

    # 提取病人表中时间、CGM (连续血糖监测)、CSII - basal insulin (Novolin R IU / H) (皮下注射胰岛素剂量)。
    index = 11
    patient_needed_columns = patient_data_table.iloc[:, [0, 1, index]]
    # 获取第三列的列索引
    column_index = 2
    patient_needed_columns.iloc[:, column_index] = patient_needed_columns.iloc[:, column_index].fillna(0)

    # print(patient_needed_columns)

    # 将 summary_needed_columns 重复拼接接到与 patient_data_table 行数相匹配
    summary_needed_columns_repeated = pd.concat([summary_needed_columns] * len(patient_needed_columns),
                                                ignore_index=True)

    # 将 patient_data_table 和 summary_needed_columns_repeated 拼接到一起
    result = pd.concat([summary_needed_columns_repeated, patient_needed_columns], axis=1)

    print('输出名字: ' + Match_name)
    # 将 result 存储为 Excel 表格
    output_file = 'output/T1DM/excel/' + Match_name + 'out_put.xlsx'
    result.to_excel(output_file, index=False) # 如果不想保存索引, 可以设置 index=False

    # 将 result 存储为csv表格
    output_file = 'output/T1DM/csv/' + Match_name + 'out_put.csv'
    result.to_csv(output_file, index=False) # 如果不想保存索引, 可以设置 index=False

```

T1DM.py 文件的主要功能是从处理过的（即经过 pre\_T1DM.py 处理完的）Excel 文件中提取特定数据，并将这些数据与汇总表中的信息进行匹配和合并。其作用可以总结如下：

- ◆ 导入必要的库：
  - warnings: 用于处理警告信息。
  - pandas: 用于数据处理和分析。
  - glob 和 os: 用于文件操作和查找。
- ◆ 文件匹配与查找：
  - 使用 glob 模块查找文件名模式匹配 generated\_data/Shanghai\_T1DM/ 目录下以 1000 到 1012 为前缀的文件。
- ◆ 数据处理：
  - 从 generated\_data/Shanghai\_T1DM\_Summary.xlsx 文件中读取汇总表，并根据文件名找到匹配的行。

提取匹配行的前五列（ID、性别、年龄、身高、体重）。

读取对应病人的数据文件，并将日期列拆分为时间列，删除原始日期列。

提取病人表中的时间、CGM（连续血糖监测）、CSII - basal insulin（皮下注射胰岛素剂量）列，并填充空值。

将汇总表的提取数据重复拼接接到与病人数据表行数相匹配的行数。

将病人数据表和汇总表的提取数据合并。

- ◆ 结果输出：  
将合并后的结果保存为 Excel 文件和 CSV 文件，输出到 output/T1DM/excel/ 和 output/T1DM/csv/ 目录。

T2DM 数据集的处理方式类似，这里不再赘述。下图为处理前的表格数据和处理后的表格数据的比较：

原先数据集中的数据：

Date	CGM (mg / CBG (mg / Blood Keto Dietary intake	饮食	Insulin dose/Non-insulin CSII - bolus R (U / H) Insulin dose - iv
2021/7/30 16:43	113.4		
2021/7/30 16:58	124.2		
2021/7/30 17:13	129.6		4
2021/7/30 17:28	142.2	data not available	未记录
2021/7/30 17:43	156.6		
2021/7/30 17:58	162		
2021/7/30 18:13	163.8		
2021/7/30 18:28	165.6		
2021/7/30 18:43	169.2		
2021/7/30 18:58	167.4	192.6	
2021/7/30 19:13	169.2		
2021/7/30 19:28	174.6		
2021/7/30 19:43	176.4		0.3
2021/7/30 19:58	174.6		
2021/7/30 20:13	172.8		
2021/7/30 20:28	171		
2021/7/30 20:43	171		
2021/7/30 20:58	172.8	194.4	
2021/7/30 21:13	169.2		
2021/7/30 21:28	158.4		
2021/7/30 21:43	149.4		
2021/7/30 21:58	153		0.2
2021/7/30 22:13	154.8		
2021/7/30 22:28	151.2		
2021/7/30 22:43	154.8		
2021/7/30 22:58	156.6		
2021/7/30 23:13	151.2		
2021/7/30 23:28	147.6		
2021/7/30 23:43	140.4		

处理后的表格：



Time	Gender (F)	Age (years)	Height (m)	Weight (kg)	CGM (mg / dl)	CSII - basal insulin (Novolin R, IU / H)			
16:43	1	66	1.5	60	113.4	0.3			
16:58	1	66	1.5	60	124.2	0.3			
17:13	1	66	1.5	60	129.6	0.3			
17:28	1	66	1.5	60	142.2	0.3			
17:43	1	66	1.5	60	156.6	0.3			
17:58	1	66	1.5	60	162	0.3			
18:13	1	66	1.5	60	163.8	0.3			
18:28	1	66	1.5	60	165.6	0.3			
18:43	1	66	1.5	60	169.2	0.3			
18:58	1	66	1.5	60	167.4	0.3			
19:13	1	66	1.5	60	169.2	0.3			
19:28	1	66	1.5	60	174.6	0.3			
19:43	1	66	1.5	60	176.4	0.3			
19:58	1	66	1.5	60	174.6	0.2			
20:13	1	66	1.5	60	172.8	0.2			
20:28	1	66	1.5	60	171	0.2			
20:43	1	66	1.5	60	171	0.2			
20:58	1	66	1.5	60	172.8	0.2			
21:13	1	66	1.5	60	169.2	0.2			
21:28	1	66	1.5	60	158.4	0.2			
21:43	1	66	1.5	60	149.4	0.2			
21:58	1	66	1.5	60	153	0.2			
22:13	1	66	1.5	60	154.8	0.2			
22:28	1	66	1.5	60	151.2	0.2			
22:43	1	66	1.5	60	154.8	0.2			
22:58	1	66	1.5	60	156.6	0.2			
23:13	1	66	1.5	60	151.2	0.2			
23:28	1	66	1.5	60	147.6	0.2			
23:43	1	66	1.5	60	140.4	0.2			

本次数据预处理的过程，重要的步骤在下方提炼得出，并阐述了为何需要此步骤  
**总结：**

- **数据匹配和提取：**从汇总表中提取病人的基本信息（如 ID、性别、年龄、身高、体重），并与每个病人的详细数据文件进行匹配。将汇总信息与详细数据结合，提供了全面的病人信息，方便后续分析。
- **时间列处理：**将日期列拆分为时间列，使得时间信息更加明确和独立。有助于时间序列分析，方便按时间段进行数据统计和分析。
- **数据清洗和填充：**处理空值，确保每个数据点都有明确的值。数据完整性得到保证，避免分析过程中因缺失值导致的问题。

通过这样的预处理和数据整理，可以确保数据的质量和一致性。这为后续的统计分析、机器学习模型训练以及其他数据分析任务提供了可靠的基础。处理后的数据更加结构化，减少了人为干预和错误的可能性，提高了数据处理和分析的效率。

### 三．数据特征工程

#### 数据集总览分析

在 T1DM 和 T2DM 的总览数据表中，我们可发现表中有如下的数据：

**患者特征数据:**

Patient Number (患者编号)

Gender (Female=1, Male=2) (性别 (女=1, 男=2))

Age (years) (年龄 (年))

Height (m) (身高 (米))

Weight (kg) (体重 (千克))

BMI (kg/m<sup>2</sup>) (体质指数 (千克/平方米))

Smoking History (pack year) (年数)

Alcohol Drinking History (drinker/non-drinker) (饮酒历史 (喝酒/不喝酒))

**糖尿病相关信息:**

Type of Diabetes (糖尿病类型)

Duration of diabetes (years) (糖尿病病程 (年))

Acute Diabetic Complications (急性糖尿病并发症)

Diabetic Macrovascular Complications (糖尿病大血管并发症)

Diabetic Microvascular Complications (糖尿病微血管并发症)

Comorbidities (合并症)

Hypoglycemic Agents (降血糖药物)

Other Agents (其他药物)

**实验室测量数值:**

Fasting Plasma Glucose (mg/dl) (空腹血浆葡萄糖 (毫克/分升))

2-hour Postprandial Plasma Glucose (mg/dl) (餐后 2 小时血浆葡萄糖 (毫克/分升))

Fasting C-peptide (nmol/L) (空腹 C 肽 (纳摩尔/升))

2-hour Postprandial C-peptide (nmol/L) (餐后 2 小时 C 肽 (纳摩尔/升))

Fasting Insulin (pmol/L) (空腹胰岛素 (皮摩尔/升))

2-hour Postprandial insulin (pmol/L) (餐后 2 小时胰岛素 (皮摩尔/升))

HbA1c (mmol/mol) (糖化血红蛋白 (毫摩尔/摩尔))

Glycated Albumin (%) (糖化白蛋白 (%))

Total Cholesterol (mmol/L) (总胆固醇 (毫摩尔/升))

Triglyceride (mmol/L) (甘油三酯 (毫摩尔/升))

High-Density Lipoprotein Cholesterol (mmol/L) (高密度脂蛋白胆固醇 (毫摩尔/升))

Low-Density Lipoprotein Cholesterol (mmol/L) (低密度脂蛋白胆固醇 (毫摩尔/升))

Creatinine (umol/L) (肌酐 (微摩尔/升))

Estimated Glomerular Filtration Rate (ml/min/1.73m<sup>2</sup>) (估算肾小球滤过率 (毫升/分钟/1.73 平方米))

Uric Acid (mmol/L) (尿酸 (毫摩尔/升))

Blood Urea Nitrogen (mmol/L) (血尿素氮 (毫摩尔/升))

Hypoglycemia (yes/no) (低血糖 (是/否))

**为了能够预测病人的血糖含量, 我们需要这些数据:**

Gender (Female=1, Male=2) (性别 (女=1, 男=2))

Age (years) (年龄 (年))

Height (m) (身高 (米))



Weight (kg) (体重 (千克))

BMI (kg/m<sup>2</sup>) (体质指数 (千克/平方米))

为了预测病人的血糖水平，选择这些数据作为输入数据有以下几个原因：性别：性别影响胰岛素敏感性和血糖代谢，男女在体脂分布和激素水平上的差异会对血糖水平产生不同的影响；年龄：随着年龄的增长，人体的代谢率和胰岛素敏感性都会发生变化，年龄是影响血糖水平的重要因素；体质指数（BMI）：BMI 是身高和体重的综合指标，常用于评估肥胖程度。高 BMI 通常与胰岛素抵抗有关，这会显著影响血糖水平。BMI 是预测糖尿病风险的重要指标之一。当最后预测的时候，为了使用方便，会先输入身高体重，由系统自动生成 BMI。这些数据可以帮助建立病人的基本体格特征，为血糖预测模型提供关键的输入信息。体格特征与血糖水平之间存在显著关联，通过这些数据可以更准确地预测病人的血糖变化趋势。此外，这些数据易于获取和测量，能够明显地区别各个病人，适合作为模型的输入变量。

## 四. 模型选择和实现

在模型选择的部分，我们使用并且比较了若干算法，最终确定使用 Seq2Seq-LSTM 的方法来训练模型。

## Seq2Seq 的基本原理

Seq2Seq (Sequence-to-Sequence) 模型是一种深度学习模型，最初用于处理自然语言处理任务，如机器翻译和文本摘要。它的核心思想是将一个序列转换成另一个序列（如输出文本），并且这两个序列的长度可以不同。

主要组成部分和原理：

### 1. 编码器-解码器架构：

- **编码器 (Encoder)：**负责将输入序列编码为一个上下文向量 (context vector)，这个向量捕捉了输入序列的重要信息。通常使用循环神经网络 (RNN) 如 LSTM 或 GRU 来处理变长输入序列，将其转换成固定长度的上下文向量。
- **解码器 (Decoder)：**利用编码器生成的上下文向量，解码器逐步生成目标序列的输出。在生成过程中，解码器同样使用 RNN 来处理和预测输出序列的下一个元素。

### 2. 循环神经网络 (RNN)：

- 在传统的 Seq2Seq 模型中，编码器和解码器通常使用 RNN 作为基本单元。RNN 可以处理序列数据，并在其隐藏状态中保持序列的状态信息，有助于处理时间依赖关系。

### 3. 注意力机制 (Attention)：

- 注意力机制是对 Seq2Seq 模型的一个重要改进，特别是处理长文本时。它允许模型在生成每个目标词时动态地“关注”输入序列的不同部分，以提高模型的性能和准确性。

#### 4. 长短期记忆网络（LSTM）/门控循环单元（GRU）：

- 为了解决传统 RNN 中的长期依赖问题，引入了 LSTM 和 GRU 等门控循环单元。这些模型结构能够更好地捕捉长期依赖关系，从而改善序列到序列任务的性能。

○

#### 编码器-解码器在时间序列领域的应用

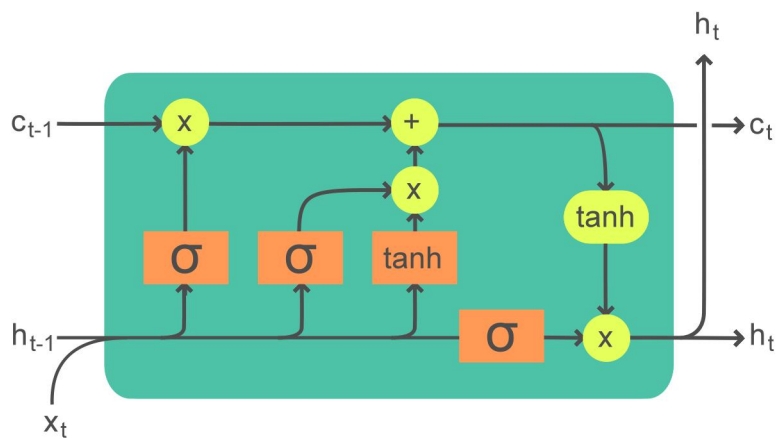
Seq2Seq 模型在时间序列领域的应用是将输入的历史时间序列数据编码为上下文向量，然后使用解码器生成未来的时间序列数据。具体流程如下：

- **编码器：**
  - 处理时间序列输入：接收历史数据序列，例如过去几天的股票价格或气温记录。这些数据通常是连续的数值。
  - 特征提取：通过 RNN 网络（如 GRU），编码器可以捕捉时间序列的特征和模式。
  - 上下文向量：编码器将历史数据编码为一个上下文向量，这个向量包含了对过去数据的理解和总结。
- **解码器：**
  - 初始状态和输入：解码器的初始状态通常由编码器生成的最终状态设置。解码器的第一个输入可以是序列的最后一个观测值或特殊标记。
  - 逐步预测：解码器基于当前状态和前一步的预测输出（或初始输入），生成未来时间步的预测值。
  - 迭代更新：解码器的输出用于更新其状态，并作为下一个时间步的输入，直到生成整个预测序列。

这种编码器-解码器结构使得 Seq2Seq 模型能够有效地处理具有复杂时间依赖关系的序列数据，例如血糖预测、股票价格预测、天气预报等任务。

#### 编码器-解码器结构图示例

下面是一个简化的编码器-解码器结构示意图，展示了其在序列到序列任务中的基本流程：



Legend:

Layer Componentwise Copy Concatenate

- **编码器部分:**
  - 嵌入层 (Embedding): 将输入的 token 转换为嵌入向量。
  - RNN 层 (如 GRU): 处理输入序列, 更新隐藏状态。
  - 时间步和位置标记: 表示序列的处理进程和位置信息。
- **解码器部分:**
  - 嵌入层: 将输出 token 转换为嵌入向量。
  - RNN 层 (如 GRU): 根据当前状态和输入, 生成序列的下一个 token。
  - 全连接层: 将 RNN 的输出转换为目标词汇表大小的向量, 表示下一个 token 的概率分布。

这个结构展示了如何使用编码器-解码器模型进行序列到序列的预测和生成, 如机器翻译中的源语言到目标语言的转换。

## 滚动预测介绍

滚动预测是时间序列分析中常见的一种策略, 特别适用于预测未来多个时间步的值。其基本思想是连续预测未来一段时间内的值, 并将这些预测作为新的输入, 反复进行预测。

### 主要特点和应用:

- **连续预测和填充:** 首先, 模型预测未来一段时间内的值, 然后将这些预测值作为新的输入, 用于预测接下来的时间步。这种方法可以连续地填充和预测未来时间序列数据。
- **实时数据更新:** 滚动预测策略可以与实时数据更新结合, 例如在股票市场分析中, 每天预测未来一周的股票价格, 然后根据新的数据更新预测模型。
- **模型评估和调整:** 通过滚动预测, 可以评估模型在长期预测任务中的表现, 同时根据实际情况调整模型参数和预测策略。

滚动预测的实施需要考虑数据的周期性、趋势和季节性, 以及模型的性能和实时性要求。这种策略对于长期预测和实时决策具有重要意义。

## 模型优势

1. **捕捉短期和长期依赖：**TCN 的扩张卷积和 LSTM 的记忆单元能够有效结合，捕捉序列中的短期和长期依赖关系。
2. **提高模型的鲁棒性：**通过结合两种不同类型的网络结构，模型能够更好地适应不同类型的时间序列数据。
3. **增强预测性能：**在一些复杂的序列任务中，TCN\_LSTM 的混合结构能够提供比单独使用 TCN 或 LSTM 更好的预测性能。

### 在血糖预测中的应用

TCN\_LSTM 在血糖预测中的应用主要是通过结合时间卷积网络和长短期记忆网络的优点，来实现对血糖水平的高精度预测。这种方法在处理具有时间依赖性和非线性特征的生物学信号数据（如血糖水平）时表现尤为出色。

### 模型训练

1. **模型设计：**构建 TCN\_LSTM 网络，设置模型参数如层数、每层神经元数量、卷积核大小和扩张系数等。
2. **模型训练：**使用历史数据训练模型，优化损失函数，调整模型参数以提高预测精度。
3. **模型验证：**使用验证集评估模型性能，防止过拟合。

### 模型评估

1. **性能指标：**采用均方误差（MSE）、均绝对误差（MAE）、决定系数（ $R^2$ ）等指标评估模型性能。
2. **对比实验：**与其他常用的时间序列预测模型进行对比实验，得出 TCN\_LSTM 的 loss 更低，预测精度更高，以此验证 TCN\_LSTM 的优越性。

## 算法参数：

```
parser.add_argument(
    "-window_size",
    type=int,
    default=126,
    help="时间窗口大小, window_size > pre_len",
)
parser.add_argument("-pre_len", type=int, default=96, help="预测未来数据长度")
# data
```

#### -window\_size:

- **作用：**时间窗口大小，决定了每次输入到模型中的时间序列长度。
- **高值：**模型可以看到更长的时间序列，适用于需要长时间依赖的信息，但可能增加计算复杂度。
- **低值：**适用于短时间依赖的信息，但可能忽略长时间依赖的特征。
- **数据量大：**可选较大的 window\_size，充分利用数据中的长时间依赖信息。
- **数据量小：**可选较小的 window\_size，减少过拟合的风险。

#### -pre\_len:

- **作用:** 预测未来数据的长度。
- **高值:** 预测更长时间的未来, 但可能增加预测的不确定性。
- **低值:** 预测短时间的未来, 通常准确性较高。
- **数据量大:** 可选较大的 `pre_len`, 充分利用数据进行长时间预测。
- **数据量小:** 可选较小的 `pre_len`, 提高预测的准确性。

```
# learning
parser.add_argument("-lr", type=float, default=0.0001, help="学习率")
parser.add_argument(
    "-drop_out", type=float, default=0.25, help="随机丢弃概率,防止过拟合"
)
parser.add_argument("-epochs", type=int, default=15, help="训练轮次")
parser.add_argument("-batch_size", type=int, default=128, help="批次大小")
parser.add_argument("-save_path", type=str, default="models")

# model
parser.add_argument("-hidden_size", type=int, default=128, help="隐藏层单元数")
parser.add_argument("-kernel_sizes", type=int, default=3)
parser.add_argument("-laryer_num", type=int, default=3)

# device
parser.add_argument("-use_gpu", type=bool, default=True)
parser.add_argument(
    "-device", type=int, default=0, help="只设置最多支持单个gpu训练"
)
```

#### **-lr (学习率):**

- **作用:** 控制模型权重更新的步长。
- **高值:** 学习速度快, 但可能导致模型不稳定或发散。
- **低值:** 学习速度慢, 训练时间长, 但更有可能收敛到较好的结果。
- **数据量大:** 可适当降低 `lr`, 提高模型训练的稳定性。
- **数据量小:** 可适当提高 `lr`, 加快收敛速度。

#### **-drop\_out (随机丢弃率):**

- **作用:** 防止过拟合, 通过随机丢弃一部分神经元。
- **高值:** 强正则化效果, 防止过拟合, 但可能导致欠拟合。
- **低值:** 正则化效果弱, 模型更容易过拟合。
- **数据量大:** 可选较高的 `drop_out`, 防止过拟合。
- **数据量小:** 可选较低的 `drop_out`, 避免欠拟合。

#### **-epochs (训练轮次):**

- **作用:** 模型训练的完整周期数。
- **高值:** 模型有更多机会学习, 但可能导致过拟合。
- **低值:** 模型训练不充分, 可能欠拟合。
- **数据量大:** 可选较多的 `epochs`, 充分训练模型。
- **数据量小:** 可选较少的 `epochs`, 防止过拟合。

#### **-batch\_size (批次大小):**

- **作用:** 每次更新模型时使用的样本数量。
- **高值:** 计算效率高, 但需要更多内存, 梯度更新更稳定。
- **低值:** 计算效率低, 但需要更少内存, 梯度更新更噪声。
- **数据量大:** 可选较大的 `batch_size`, 提高计算效率。

- 数据量小: 可选较小的 `batch_size`, 更有效利用数据。
- save\_path:**
  - 作用: 模型保存路径, 用于保存训练后的模型。
- hidden\_size (隐藏层单元数):**
  - 作用: LSTM 隐藏层的神经元数量。
  - 高值: 模型更复杂, 能学习更复杂的模式, 但计算复杂度和过拟合风险增加。
  - 低值: 模型简单, 计算效率高, 但可能欠拟合。
  - 数据量大: 可选较大的 `hidden_size`, 捕捉复杂的模式。
  - 数据量小: 可选较小的 `hidden_size`, 防止过拟合。
- kernel\_sizes (卷积核大小):**
  - 作用: 卷积层的核大小 (若使用卷积层)。
  - 高值: 能捕捉更长的依赖关系, 但计算复杂度高。
  - 低值: 计算效率高, 但只能捕捉短时间依赖。
  - 数据量大: 可选较大的 `kernel_sizes`, 捕捉长时间依赖。
  - 数据量小: 可选较小的 `kernel_sizes`, 提高计算效率。
- layer\_num (层数):**
  - 作用: LSTM 的层数。
  - 高值: 模型更深, 能学习更复杂的模式, 但计算复杂度和过拟合风险增加。
  - 低值: 模型较浅, 计算效率高, 但可能欠拟合。
  - 数据量大: 可选较多的 `layer_num`, 学习复杂模式。
  - 数据量小: 可选较少的 `layer_num`, 防止过拟合。
- use\_gpu (是否使用 GPU):**
  - 作用: 是否使用 GPU 进行训练。
  - **True:** 使用 GPU, 计算速度快。
  - **False:** 使用 CPU, 计算速度慢。
- device (设备号):**
  - 作用: 指定使用哪一个 GPU 进行训练 (如果有多个 GPU)。

## 标准化

标准化过程是将数据转换为具有零均值和单位方差的分佈。这对机器学习模型训练很重要, 因为它可以提高模型收敛速度和性能。

标准化过程包括两个步骤:

**拟合 (fit):** 计算训练数据的均值和标准差。

**转换 (transform):** 使用计算的均值和标准差将数据标准化。

```

class StandardScaler:
    def __init__(self):
        self.mean = 0.0
        self.std = 1.0

    def fit(self, data):
        self.mean = data.mean(0)
        self.std = data.std(0)

    def transform(self, data):
        mean = (
            torch.from_numpy(self.mean).type_as(data).to(data.device)
            if torch.is_tensor(data)
            else self.mean
        )
        std = (
            torch.from_numpy(self.std).type_as(data).to(data.device)
            if torch.is_tensor(data)
            else self.std
        )
        transformed_data = (data - mean) / std
        return transformed_data

    def inverse_transform(self, data):
        mean = (
            torch.from_numpy(self.mean).type_as(data).to(data.device)
            if torch.is_tensor(data)
            else self.mean
        )
        std = (
            torch.from_numpy(self.std).type_as(data).to(data.device)
            if torch.is_tensor(data)
            else self.std
        )
        if data.shape[-1] != mean.shape[-1]:
            mean = mean[-1:]
            std = std[-1:]
        return (data * std) + mean
        # return data

```

**fit 方法:** 计算输入数据的均值和标准差。

- `data.mean(0)` 计算每个特征的均值。
- `data.std(0)` 计算每个特征的标准差。
- 这些计算结果分别保存在 `self.mean` 和 `self.std` 中。

**transform 方法:** 标准化输入数据。

- 将存储的均值和标准差转换为与输入数据类型和设备一致的 `Tensor`。
- 标准化公式为  $(data - mean) / std$ ，将每个特征值减去均值，再除以标准差。
- 返回标准化后的数据。

**inverse\_transform 方法:** 将标准化后的数据还原为原始数据。

- 同样将存储的均值和标准差转换为与输入数据类型和设备一致的 `Tensor`。



- 如果输入数据的最后一个维度与均值的维度不一致，取均值和标准差的最后一个元素（适用于处理单一特征的情况）。
- 逆标准化公式为  $(\text{data} * \text{std}) + \text{mean}$ ，将每个特征值乘以标准差，再加上均值。
- 返回还原后的数据。

## 创建数据加载器

[illegible]

**create\_dataloader** 函数的目的是准备和加载时间序列预测模型的训练、验证和测试数据。

## 文件读取和合并

```
dataframes = []

# config.data_path = "data/T2DM/csv/2000_0_20201230out_put.csv"

for index in range(2000, 2100): # 这个循环会执行多次
    file_pattern = os.path.join("data/T2DM/csv/", f"{index}*.csv")
    all_files = glob.glob(file_pattern)

    # 打印文件名
    for file in all_files:
        filename = os.path.basename(file)
        # 读取 CSV 文件并添加到 dataframes 列表中
        df_single = pd.read_csv(file)
        dataframes.append(df_single)

# 将所有 DataFrame 合并为一个
df = pd.concat(dataframes, ignore_index=True)
```

首先，通过循环读取位于 `data/T2DM/csv/` 目录下的从 2000 到 2099 的多个 CSV 文件。将每个 CSV 文件读取为 `DataFrame`，并将它们存储在 `dataframes` 列表中。最后，使用 `pd.concat` 将所有 `DataFrame` 连接成一个大的 `DataFrame` `df`。

## 数据预处理

```

# 删除原始时间戳列
pre_len = config.pre_len # 预测未来数据的长度
train_window = config.window_size # 观测窗口

# 编码时间信息为数值特征
df[["Hour", "Minute"]] = df["Time"].str.split(":", expand=True)
df.drop(columns=["Time"], inplace=True)
df["Hour"] = pd.to_numeric(df["Hour"])
df["Minute"] = pd.to_numeric(df["Minute"])

# 保留特征列
feature_columns = [
    "Hour",
    "Minute",
    "Gender (Female=1, Male=2)",
    "Age (years)",
    "Height (m)",
    "Weight (kg)",
    "CSII - basal insulin (Novolin R, IU / H)",
]

# 将特征列和目标列分开
feature_data = df[feature_columns].values
target_data = df[[config.target]].values
full_data = np.hstack((feature_data, target_data))

```

删除原始数据中的时间戳列，并将时间信息分解为 Hour 和 Minute 列。  
将特定的特征列和目标列提取出来，并将它们合并成 full\_data 数组。

## 数据标准化

```

scaler = StandardScaler()
scaler.fit(full_data)
full_data_normalized = scaler.transform(full_data)

```

使用 StandardScaler 对 full\_data 进行标准化处理，确保数据在相同的尺度上进行训练。

## 数据集划分和 Tensor 转换

```
# 划分数据集
train_data = full_data_normalized[int(0.3 * len(full_data)) :]
valid_data = full_data_normalized[
    int(0.15 * len(full_data)) : int(0.30 * len(full_data))
]
test_data = full_data_normalized[: int(0.15 * len(full_data))]
print(
    "训练集尺寸:",
    len(train_data),
    "测试集尺寸:",
    len(test_data),
    "验证集尺寸:",
    len(valid_data),
)

# 转化为深度学习模型需要的类型Tensor
train_data_normalized = torch.FloatTensor(train_data).to(device)
test_data_normalized = torch.FloatTensor(test_data).to(device)
valid_data_normalized = torch.FloatTensor(valid_data).to(device)
```

将标准化后的数据划分为训练集、验证集和测试集。

将这些数据转换为 PyTorch 张量，并将它们移动到指定的设备上（GPU）。

## 创建数据加载器

```

# 定义训练器的输入
train_inout_seq = create_inout_sequences(
    train_data_normalized, train_window, pre_len, config
)
test_inout_seq = create_inout_sequences(
    test_data_normalized, train_window, pre_len, config
)
valid_inout_seq = create_inout_sequences(
    valid_data_normalized, train_window, pre_len, config
)

# 创建数据集
train_dataset = TimeSeriesDataset(train_inout_seq)
test_dataset = TimeSeriesDataset(test_inout_seq)
valid_dataset = TimeSeriesDataset(valid_inout_seq)

# 创建 DataLoader
train_loader = DataLoader(
    train_dataset, batch_size=args.batch_size, shuffle=True, drop_last=True
)
test_loader = DataLoader(
    test_dataset, batch_size=args.batch_size, shuffle=False, drop_last=True
)
valid_loader = DataLoader(
    valid_dataset, batch_size=args.batch_size, shuffle=False, drop_last=True
)

```

使用 `create_inout_sequences` 函数将数据转换为输入-输出序列。

将转换后的序列创建为 `TimeSeriesDataset` 对象。

使用 `DataLoader` 创建训练、验证和测试数据加载器，以便于模型训练和评估。

## LSTMEncoder 详解

初始化方法 `__init__`

```

class LSTMEncoder(nn.Module):
    def __init__(
        self,
        rnn_num_layers=1,
        input_feature_len=1,
        sequence_len=168,
        hidden_size=100,
        bidirectional=False,
    ):
        super().__init__()
        self.sequence_len = sequence_len
        self.hidden_size = hidden_size
        self.input_feature_len = input_feature_len
        self.num_layers = rnn_num_layers
        self.rnn_directions = 2 if bidirectional else 1
        self.lstm = nn.LSTM(
            num_layers=rnn_num_layers,
            input_size=input_feature_len,
            hidden_size=hidden_size,
            batch_first=True,
            bidirectional=bidirectional,
        )

```

#### 参数解释:

- **rnn\_num\_layers**: LSTM 层的堆叠数量。
- **input\_feature\_len**: 输入特征的长度。
- **sequence\_len**: 输入序列的长度。
- **hidden\_size**: LSTM 隐藏状态的大小。
- **bidirectional**: 是否使用双向 LSTM。

#### 操作:

- `super().__init__()` 调用父类 `nn.Module` 的初始化方法。
- 设置了类成员变量 `sequence_len`、`hidden_size`、`input_feature_len`、`num_layers` 和 `rnn_directions`，这些参数将在后续的前向传播方法中使用。
- 创建了一个 `nn.LSTM` 实例 `self.lstm`，配置了 LSTM 的层数、输入特征大小、隐藏状态大小、是否批处理优先和是否双向。

#### 前向传播方法 (forward)



```

def forward(self, input_seq):

    ht = torch.zeros(
        self.num_layers * self.rnn_directions,
        input_seq.size(0),
        self.hidden_size,
        device="cuda",
    )
    ct = ht.clone()
    if input_seq.ndim < 3:
        input_seq.unsqueeze_(2)
    lstm_out, (ht, ct) = self.lstm(input_seq, (ht, ct))
    if self.rnn_directions > 1:
        lstm_out = lstm_out.view(
            input_seq.size(0),
            self.sequence_len,
            self.rnn_directions,
            self.hidden_size,
        )
        lstm_out = torch.sum(lstm_out, axis=2)
    return lstm_out, ht.squeeze(0)

```

参数解释：

- **input\_seq**: 输入的序列数据。

操作：

- ht 和 ct 初始化为全零张量，表示 LSTM 的初始隐藏状态和细胞状态。
- 如果 input\_seq 的维度小于 3，则扩展维度为 (batch\_size, sequence\_len, 1)，以适应 LSTM 的输入要求。
- 将 input\_seq 和初始的 (ht, ct) 输入到 self.lstm 中，得到 lstm\_out 是 LSTM 的输出和最终的 (ht, ct)。
- 如果使用双向 LSTM (bidirectional=True)，则对 lstm\_out 进行视图重塑，将其从 (batch\_size, sequence\_len, rnn\_directions \* hidden\_size) 转换为 (batch\_size, sequence\_len, rnn\_directions, hidden\_size)，然后对第二个维度进行求和，得到合并后的输出。
- 返回 lstm\_out 和最后一个时间步的隐藏状态 ht.squeeze(0)，其中 squeeze(0) 是为了去除批次维度，得到形状为 (batch\_size, hidden\_size) 的隐藏状态。

# 训练和评估

## train 函数

训练模型的函数，使用均方误差损失函数（MSELoss）和 Adam 优化器。在每个 epoch 中迭代 DataLoader，计算损失并反向传播优化模型参数。

```
def train(model, args, scaler, device):
    start_time = time.time() # 计算起始时间
    model = model
    loss_function = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
    epochs = args.epochs
    model.train() # 训练模式
    results_loss = []
    for i in tqdm(range(epochs)):
        loss = []
        for seq, labels in train_loader:
            optimizer.zero_grad()

            y_pred = model(seq)

            single_loss = loss_function(y_pred, labels)

            single_loss.backward()

            optimizer.step()
            loss.append(single_loss.detach().cpu().numpy())
        tqdm.write(f"\t Epoch {i + 1} / {epochs}, Loss: {sum(loss) / len(loss)}")
        results_loss.append(sum(loss) / len(loss))

    torch.save(model.state_dict(), "save_model.pth")
    time.sleep(0.1)
```

## 初始化

- **start\_time**: 记录训练开始时间，用于计算总训练时长。
- **model**: 接收输入模型参数。
- **loss\_function**: 定义均方误差损失函数，用于衡量模型预测值与真实标签之间的差异。
- **optimizer = torch.optim.Adam(model.parameters(), lr=0.005)**: 使用 Adam 优化器，用于更新模型的参数，学习率设置为 0.005。
- **epochs = args.epochs**: 从参数中获取训练的总轮数。
- **model.train()**: 将模型设置为训练模式，这会影响到某些层（如 Dropout、BatchNorm 等），使其在训练和评估阶段的行为不同。

## 训练循环

- **for i in tqdm(range(epochs))**: 遍历每个训练轮次。
  - **loss = []**: 用于存储每个批次的损失值。
  - **for seq, labels in train\_loader**: 遍历训练数据加载器，每次加载一个批次的数据和对应的标签。
  - **tqdm.write(f"\t Epoch {i + 1} / {epochs}, Loss: {sum(loss) / len(loss)}")**: 每个



epoch 结束后，计算并打印平均损失。

- `results_loss.append(sum(losss) / len(losss))`: 将每个 epoch 的平均损失存储到 `results_loss` 列表中。
- `torch.save(model.state_dict(), "save_model.pth")`: 每个 epoch 结束后，保存模型的状态字典到文件 "save\_model.pth"。

### valid 函数

在验证集上评估模型的函数，加载已保存的模型，计算预测值与真实值的平均绝对误差（MAE）。

```
def valid(model, args, scaler, valid_loader):
    lstm_model = model
    # 加载模型进行预测
    lstm_model.load_state_dict(torch.load("save_model.pth"))
    lstm_model.eval() # 评估模式
    loss = []

    for seq, labels in valid_loader:
        pred = lstm_model(seq)
        mae = calculate_mae(
            pred.detach().numpy().cpu(), np.array(labels.detach().cpu())
        ) # MAE误差计算绝对值(预测值 - 真实值)
        loss.append(mae)

    print("验证集误差MAE:", loss)
    return sum(loss) / len(loss)
```

### 加载模型和设置

- `lstm_model` : 使用传入的模型参数初始化 `lstm_model`。
- `lstm_model.load_state_dict(torch.load("save_model.pth"))`: 加载训练好的模型参数
- `lstm_model.eval()`: 将模型设置为评估模式，这会影响到某些层（如 Dropout、BatchNorm 等），使其在训练和评估阶段的行为不同。

### 验证过程

- `loss` 用于存储每个批次的评估指标，这里是平均绝对误差（MAE）。
- `for seq, labels in valid_loader`: 遍历验证数据加载器，每次加载一个批次的数据和对应的标签。

### test 函数

在测试集上评估模型的函数，加载已保存的模型，进行预测，并计算整体 MAE，并绘制真实值与预测值的对比图。

```

def test(model, args, test_loader, scaler):
    # 加载模型进行预测
    losss = []
    model = model
    model.load_state_dict(torch.load("save_model.pth"))
    model.eval() # 评估模式
    results = []
    labels = []
    for seq, label in test_loader:
        pred = model(seq)
        mae = calculate_mae(
            pred.detach().cpu().numpy(), np.array(label.detach().cpu())
        ) # MAE误差计算绝对值(预测值 - 真实值)
        losss.append(mae)
        pred = pred[:, 0, :]
        label = label[:, 0, :]
        pred = scaler.inverse_transform(pred.detach().cpu().numpy())
        label = scaler.inverse_transform(label.detach().cpu().numpy())
        for i in range(len(pred)):
            results.append(pred[i][-1])
            labels.append(label[i][-1])
    plt.figure(figsize=(10, 5))
    print("测试集误差MAE:", losss)
    print("overallmae:", mean_absolute_error(results, labels))

```

### inspect\_model\_fit 函数

检验模型在验证集上的拟合情况，计算整体 MAE，并绘制真实值与预测值的对比图。

```
def inspect_model_fit(model, args, valid_loader, scaler):
    # Load the saved model state
    model.load_state_dict(torch.load("save_model.pth"))
    model.eval() # Set model to evaluation mode
    results = []
    labels = []

    # Iterate over the validation data
    for seq, label in valid_loader:
        # Move tensors to the correct device
        seq, label = seq.to(device), label.to(device)

        with torch.no_grad():
            pred = model(seq)

        pred = pred[:, 0, :]
        label = label[:, 0, :]

        # Inverse transform the predictions and labels
        pred = scaler.inverse_transform(pred.detach().cpu().numpy())
        label = scaler.inverse_transform(label.detach().cpu().numpy())
        for i in range(len(pred)):
            results.append(pred[i][-1])
            labels.append(label[i][-1])
```

## 五. 模型评估和验证

在模型评估和验证阶段，我们采用了以下指标对所训练的模型进行了全面的评估和验证：

### 1. 性能评估指标：

均方误差（MSE）：MSE 是衡量模型预测结果与真实值之间差异的平方的平均值，其公式为：

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

其中  $y_i$  是真实值， $\hat{y}_i$  是模型预测值， $n$  是样本数量。MSE 值越小表示模型的预测精度越高。

均绝对误差（MAE）：MAE 是模型预测结果与真实值之间绝对差的平均值，其公式为：

$$MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

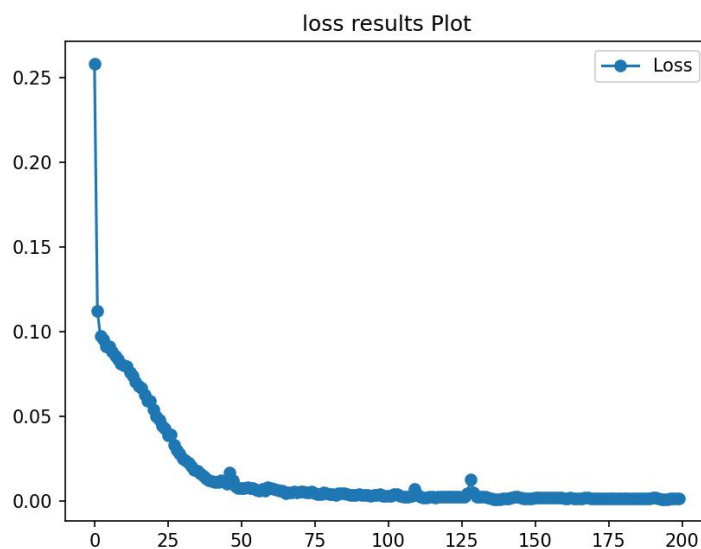
其中  $y_i$  是真实值， $\hat{y}_i$  是模型预测值， $n$  是样本数量。与  $MSE$  不同的是， $MAE$  不考虑差异的平方，因此  $MAE$  对异常值的敏感性较低。 $MAE$  值越小表示模型的预测精度越高。

**决定系数 ( $R^2$ )**： $R^2$  是衡量模型对数据变异性解释程度的统计指标，其取值范围在 0 到 1 之间，表示模型对总变异的解释程度。 $R^2$  值越接近 1 表示模型拟合效果越好。

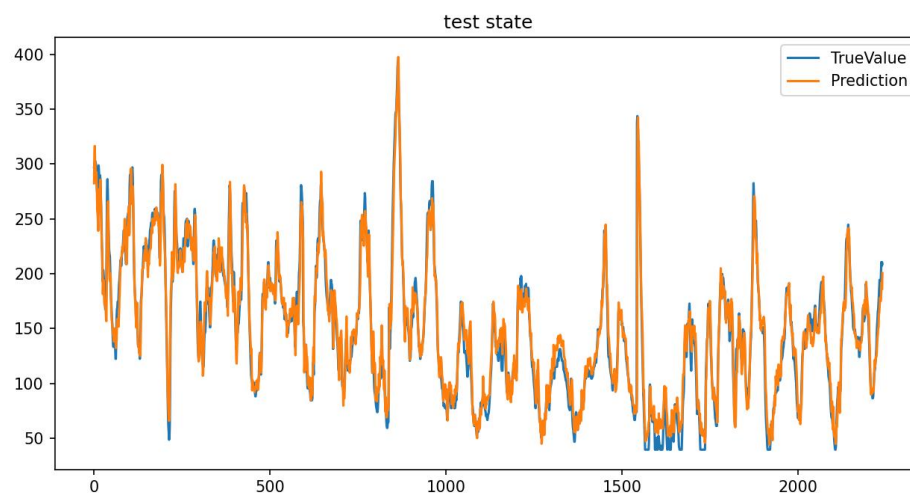
## 2. 测试集与验证集验证

我们将训练好的模型应用于测试集与验证集数据，并将模型预测的血糖水平与真实血糖水平进行对比，通过比较预测值与真实值之间的差异来评估模型的泛化能力和预测准确性。同时，我们还观察模型是否能够准确地捕捉血糖水平的变化趋势，以及对高血糖和低血糖事件的预测表现。

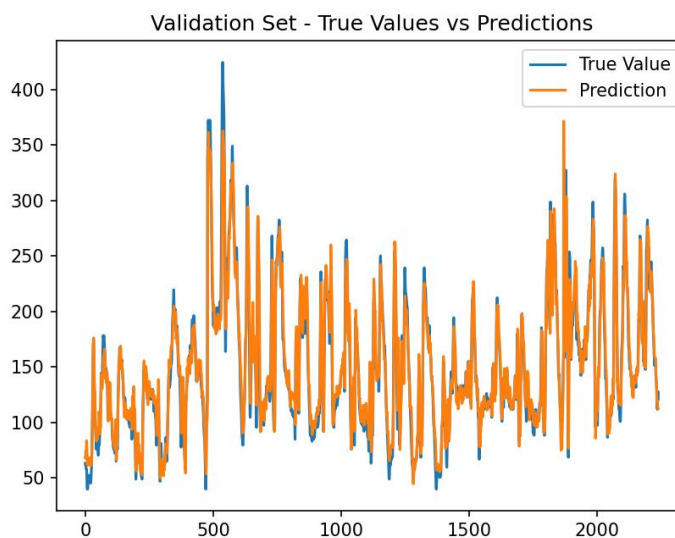
**T1DM:**



损失函数是用来衡量模型预测值与实际值之间的差异的指标，它是一个标量值，表示模型在训练集上的预测误差。在训练过程中，我们通过不断调整参数来降低  $loss$ ，从而优化算法，使得模型能够更准确地预测目标值。可以看到，在经过 50 个左右的  $epoch$  之后， $loss$  降低到了较为理想的水平。

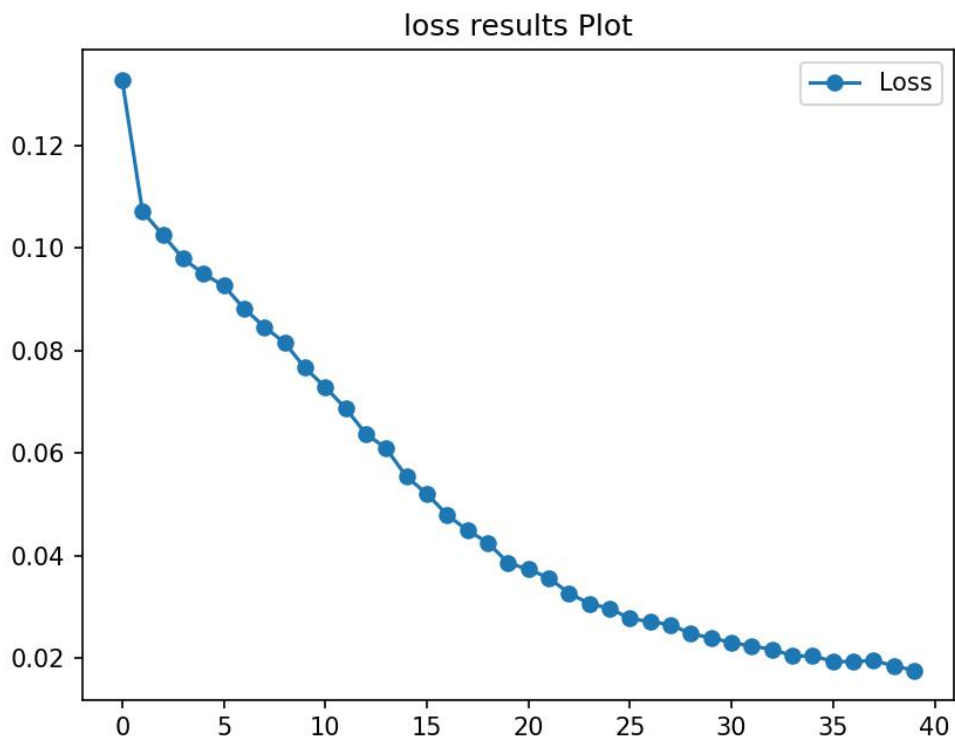
[illegible]

我们在测试集上对模型拟合效果进行了初步的验证，从效果图中可以看出，模型对测试数据集的拟合效果较好，能够较为准确地反映出血糖随时间序列变化的趋势。另外，MAE、MSE 等性能指标也都表现出了较好的效果。

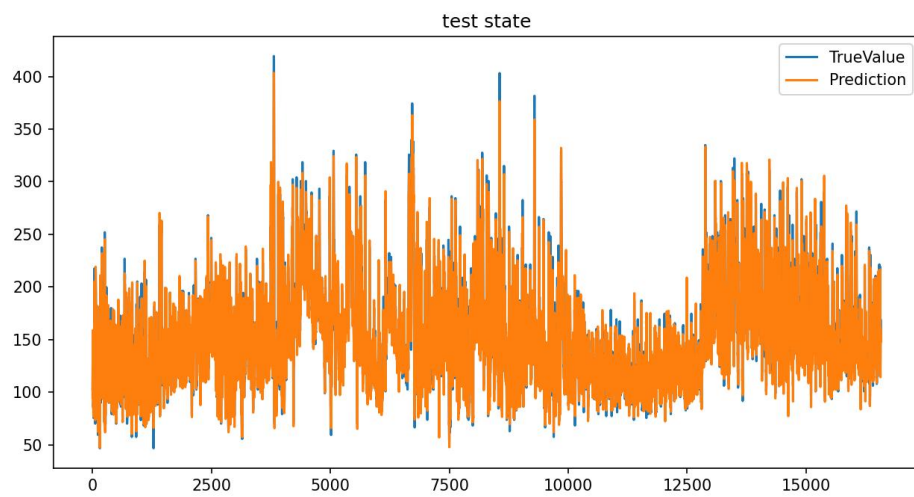
[illegible]

接下来，我们在验证集上对模型预测效果进行验证。从效果图中可以看出，相较于测试集上的效果，模型在验证集上的效果稍显逊色，但表现仍然不错，**MAE** 和 **MSE** 也都在较为合理的范围之内。

T2DM:



T2DM 分析与 T1DM 类似，在 40 个 epoch 之后，loss 降低到了理想的水平。









于饮食指标难以量化，在处理过程中较为棘手，为简化处理过程，我们舍去了这一特征，相应地也牺牲了部分准确度。后续过程中，我们可以考虑使用 SeqToSeq 模型将每餐的饮食拆分为脂肪、蛋白质、碳水等各占的重量，从而将饮食量化，然后再进行特征选取与训练，使得模型取得更好的预测准确度。

2.调整参数对结果的影响：

```
# learning
parser.add_argument("-lr", type=float, default=0.001, help="学习率")
parser.add_argument(
    "-drop_out", type=float, default=0.1, help="随机丢弃概率,防止过拟合"
)
parser.add_argument("-epochs", type=int, default=50, help="训练轮次")
parser.add_argument("-batch_size", type=int, default=256, help="批次大小")
parser.add_argument("-save_path", type=str, default="models")

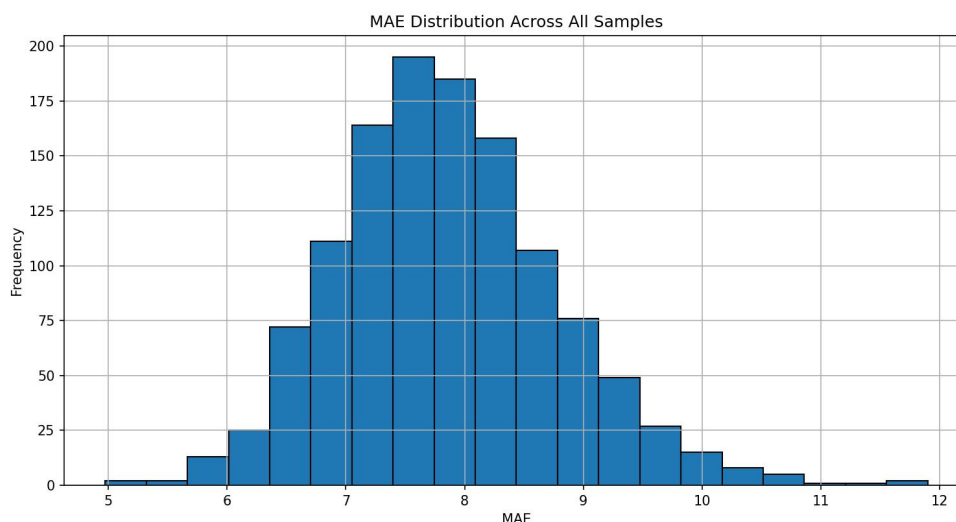
# model
parser.add_argument("-hidden_size", type=int, default=256, help="隐藏层单元数")
parser.add_argument("-kernel_sizes", type=int, default=3)
parser.add_argument("-laryer_num", type=int, default=6)
```

模型训练过程中，为了让模型能够取得更好的预测结果，我们不断优化上述的参数来降低 loss 值，力求模型能够有一个较好的拟合效果。在经过多次调参后，我们逐渐逼近了最小化损失函数，从而使得预测精确度得到了提升。

## 六．结果分析和可视化

1.预测集上 MAE 频率分布图：

我们对数据进行了随机取样，并以 15 分钟为间隔对血糖进行预测，然后计算其 MAE，并得出了如下的 MAE 频次分布直方图。可以看到 MAE 很符合正态分布，且都在一个很理想的范围内，这表明我们的模型有很不错的预测效果。



2.整体评价

我们综合考虑模型的预测准确性、稳定性和实用性，对模型的整体表现进行评价。

模型的优点在于能够基于真实临床数据支撑，较好地预测患者随时间变化的血糖水平，且基于多领域应用，具有一定的可扩展性；不足之处在于模型的学习和表现受限于训练数据的质量和覆盖范围，如果训练数据不全面或存在偏差，模型的性能可能会受到影响。此外，模型可能会受到输入数据的偏差或错误，导致输出结果存在一定的误差，容错性可能因此降低。

### 3.遇到的困难：

在项目中，我们遇到了模型选择上的困难。在最初的尝试中，我们选择了 TCN-LSTM（Temporal Convolutional Network - Long Short-Term Memory）模型。TCN-LSTM 结合了时间卷积网络和长短期记忆网络的优点，理论上能够处理序列数据中的长期依赖关系。然而，在实际应用中，我们发现该模型的效果并没有达到预期，模型的预测性能较差，无法满足项目的需求。

鉴于此，我们决定尝试另一种模型架构，即 Seq2Seq (Sequence to Sequence)。Seq2Seq 模型最初是为自然语言处理任务而设计的，特别是用于机器翻译任务。它通过编码器-解码器结构来处理输入和输出的序列数据，能够很好地捕捉输入序列中的信息并生成相应的输出序列。经过一系列的实验和调优后，Seq2Seq 模型在我们的应用场景中表现出了显著的优势，预测效果明显优于 TCN-LSTM 模型。

通过这次模型选择的过程，我们深刻认识到，不同模型在不同应用场景中的适配性可能存在很大的差异。虽然 TCN-LSTM 在某些任务中表现优异，但在我们的项目中，Seq2Seq 模型更能胜任。这一经历不仅提升了我们的模型选择能力，也加深了我们对不同序列模型的理解，为未来的项目提供了宝贵的经验。

### 4.未来展望：

基于时间序列的血糖预测模型在临床应用中具有潜在意义和一些局限性，其潜在意义包括个性化管理、预防性干预和提高生命质量等方面。然而，模型的准确性受到数据质量、个体差异、多因素影响和预测准确性等局限性的限制。为了提高模型的性能，未来的研究和应用应注重优化数据质量、个体化模型、多因素综合和临床实践探索，这些指导和建议有助于推动基于时间序列的血糖预测模型在临床应用中的发展和应用。