DAM - Final Report

数据预处理

本部分详细描述了数据预处理的过程，以便为后续模型训练做好准备。我们处理的数据包括上海的1型糖尿病（T1DM）和2型糖尿病（T2DM）患者的血糖测量数据，具体包括CGM（连续血糖监测）和CBG（毛细血管血糖）。以下是每个步骤的详细说明：

1.1 导入必要的库

```
import numpy as np
import pandas as pd
import os
from sklearn.metrics import mean_absolute_error
```

1.2 定义数据处理函数

1.2.1 创建特征和目标数据集

函数 `createXY` 用于根据过去和未来的时间步长，从数据集中创建特征（X）和目标（Y）数据。

```
def createXY(dataset, npast, n_future, column_target):
    dataX, dataY = [], []
    for i in range(npast, dataset.shape[0] - n_future + 1):
        dataX.append(dataset.iloc[i - npast:i].values)
        dataY.append(dataset.iloc[i:i + n_future][column_target].values)
    return np.array(dataX), np.array(dataY)
```

1.2.2 处理多个数据文件

函数 `processFiles` 遍历所有数据文件，并应用 `createXY` 函数生成所有的特征和目标数据集。

```
def processFiles(need_columns, target_column, all_df, n_past=1, n_future=1):
    all_x, all_y = np.array([]), np.array([])
    for df in all_df:
        df = df[need_columns]
        x, y = createXY(df, n_past, n_future, target_column)
        all_x = np.vstack([all_x, x]) if all_x.size else x
```

```
        all_y = np.vstack([all_y, y]) if all_y.size else y
    return all_x, all_y
```

1.2.3 处理汇总文件

函数 `processSummary` 读取文件夹中的所有Excel文件，并根据患者编号将其与汇总信息合并。

```
def processSummary(summary_columns, summary, folder_path):
    all_df = []
    for f in os.listdir(folder_path)[:10]:
        if f.endswith('.xlsx') or f.endswith('.xls'):
            df = pd.read_excel(folder_path + f)
            df['Patient Number'] = f.split('.')[0]
            df = pd.merge(df, summary[summary_columns], on='Patient Number')
            all_df.append(df)
    return all_df
```

1.3 数据读取和初步处理

1.3.1 读取汇总文件

读取一型糖尿病和二型糖尿病的病人摘要数据。

```
summary1 = pd.read_excel('./data/Shanghai_T1DM_Summary.xlsx')
summary2 = pd.read_excel('./data/Shanghai_T2DM_Summary.xlsx')
```

1.3.2 编码分类变量

将数据集中的分类变量进行数值编码。

```
summary1['Type of Diabetes'] = summary1['Type of Diabetes'].apply(lambda x: 1 if x == 'T1DM' else 2)
summary2['Type of Diabetes'] = summary2['Type of Diabetes'].apply(lambda x: 1 if x == 'T1DM' else 2)

summary1['Alcohol Drinking History (drinker/non-drinker)'] = summary1['Alcohol Drinking History (drinker/non-drinker)'].apply(lambda x: 1 if x == 'drinker' else 0)
summary2['Alcohol Drinking History (drinker/non-drinker)'] = summary2['Alcohol Drinking History (drinker/non-drinker)'].apply(lambda x: 1 if x == 'drinker' else 0)
```

1.3.3 处理数据文件

```
T1df_all = processSummary(summary_columns, summary1, './data/Shanghai_T1DM/')
T2df_all = processSummary(summary_columns, summary2, './data/Shanghai_T2DM/')
```

1.4 特征和目标数据集创建

```
PAST_HOURS = 6  # 使用过去6小时的数据输入
FUTURE_HOURS = 1  # 预测未来1小时的数据
NPAST, NFUTURE = 4 * PAST_HOURS, 4 * FUTURE_HOURS
all_x, all_y = processFiles(need_columns, target_column, T1df_all + T2df_all,
n_past=NPAST, n_future=NFUTURE)
```

1.5 数据集转换和特征工程

1.5.1 将数据转换为DataFrame

```
def to_df(x):
    return pd.DataFrame(x, columns=need_columns)
all_x_df = list(map(to_df, all_x))
```

1.5.2 删除无关属性

```
for i in range(len(all_x_df)):
    all_x_df[i] = all_x_df[i].drop(columns=['Patient Number'])

need_columns.remove('Patient Number')
```

1.5.3 时间特征提取

将 `Date` 转换为 `Minute` ，并使用正余弦函数将 `Minute` 转换为周期性特征。

```
# 将Date转换为一天的第几分钟
for i in range(len(all_x_df)):
    all_x_df[i]['Minute']=all_x_df[i]['Date'].apply(lambda x:
x.hour*60+x.minute)
    # 使用正余弦函数将Minute转换为周期性特征
```

```
    all_x_df[i]['Minute_sin'] = np.sin(2 * np.pi * all_x_df[i]['Minute'] /
1440)
    # 删除Hour列和Date列
    all_x_df[i]=all_x_df[i].drop(columns=['Date','Minute'])

need_columns.remove('Date')
need_columns.append('Minute_sin')
```

1.5.4 数据类型转换

将每个元素的类型设置为float类型

```
all_x_df = [df.astype(float) for df in all_x_df]
```

1.5.5 处理缺失值

```
_all_x_df = [df.fillna(0) for df in all_x_df]
```

对于目标数据 `all_y`，使用前一个值填充缺失值：

```
_all_y = pd.DataFrame(all_y).fillna(method='ffill').values
print(pd.Series(_all_y.flatten()).isnull().sum())
```

1.6 数据归一化

在进行模型训练之前，通常需要对数据进行归一化处理，以便模型能够更有效地学习数据中的模式。

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler
# 选择归一化方法（MinMaxScaler 或 StandardScaler）
scaler = MinMaxScaler()
# 将所有 DataFrame 合并成一个 DataFrame
all_x_combined = pd.concat(_all_x_df, ignore_index=True)
print(all_x_combined)

# 进行归一化
all_x_combined_scaled =
pd.DataFrame(scaler.fit_transform(all_x_combined.iloc[:,1:-1]),
columns=all_x_combined.columns[1:-1])
```

```
all_x_combined_scaled.insert(loc=0,column='CGM (mg / dl)',value=all_x_combined['CGM (mg / dl)'])
all_x_combined_scaled.insert(column = 'Minute_sin', value = all_x_combined['Minute_sin'], loc = all_x_combined_scaled.shape[1])

# 将归一化后的 DataFrame 拆分回原来的形状
index_splits = np.cumsum([len(df) for df in _all_x_df])
_all_x_df_scaled = np.split(all_x_combined_scaled, index_splits[:-1])

# 验证归一化后的结果
print(_all_x_df_scaled[0].head())
```

1.7 特征重要性评估

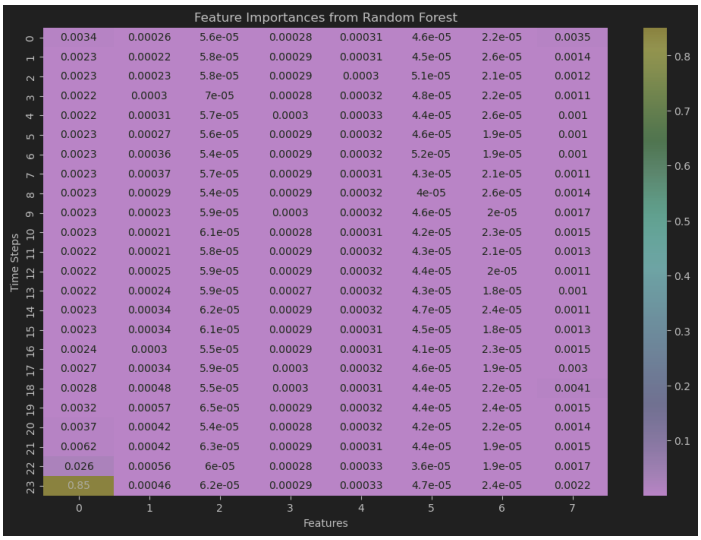
时间序列中特征重要性的确定有多种方法，包括统计特征提取与分析、机器学习特征选择、训练预测模型并通过性能指标来评估等方法。本项目我们采用机器学习的方法进行特征选择，分别使用随机森林与XGBoost来分析上述处理后的特征的重要性。

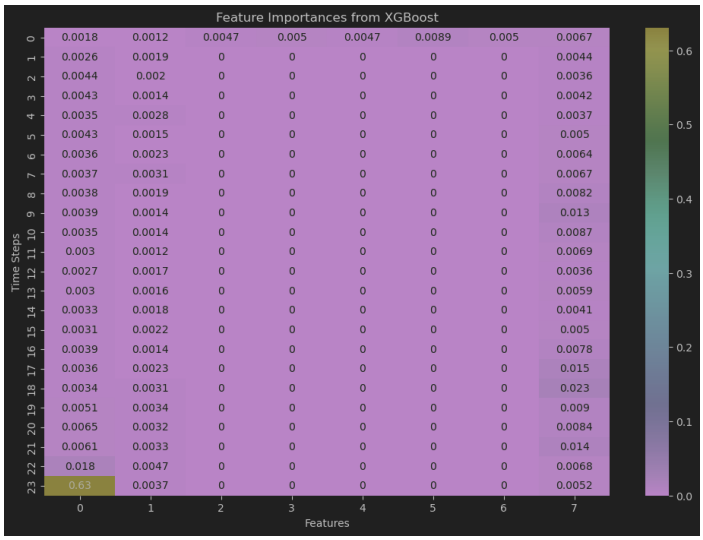
```
# 分割训练集与测试集
from sklearn.model_selection import train_test_split
SpiltRatio=0.7
X_train, X_test, y_train, y_test = train_test_split(all_x_df, all_y, test_size=1-SpiltRatio, random_state=42)
```

```
# 使用随机森林与XGBoost进行特征选择
from sklearn.ensemble import RandomForestRegressor
import xgboost as xgb
import matplotlib.pyplot as plt
import seaborn as sns

rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
xgb_model = xgb.XGBRegressor(n_estimators=100, random_state=42)


rf_model.fit(X_train_flattened, _y_train)
xgb_model.fit(X_train_flattened, _y_train)
rf_importance = rf_model.feature_importances_
xgb_importance = xgb_model.feature_importances_

# 将重要性重新排列成原始特征的形式 (24, 8)
rf_importance_reshaped = rf_importance.reshape((24, 8))
xgb_importance_reshaped = xgb_importance.reshape((24, 8))

```

```
plt.figure(figsize=(12, 8))
sns.heatmap(rf_importance_reshaped, annot=True, cmap='viridis')
plt.title('Feature Importances from Random Forest')
plt.xlabel('Features')
plt.ylabel('Time Steps')
plt.show()

plt.figure(figsize=(12, 8))
sns.heatmap(xgb_importance_reshaped, annot=True, cmap='viridis')
plt.title('Feature Importances from XGBoost')
plt.xlabel('Features')
plt.ylabel('Time Steps')
plt.show()
```

通过运行代码，得到一组数据中各时间步中各个特征的重要性热力图：





可以看到：

- 两个模型都显示出对特征0在最后一个时间步（时间步23）的高度依赖，这表明这个特征在预测未来4个时间步的值时最为关键。
- 随机森林模型的特征重要性更集中在时间步23，而XGBoost模型则在时间步22和23都有较高的重要性。

为了兼顾模型的可解释性与复杂度，我们这里最终选取的特征为：

CGM (mg / dl)、CBG (mg / dl)、Age (years)、BMI (kg/m2)、Type of Diabetes和Minute_sin

经过上述处理，

输入数据的维度为：[16 * 6] (4小时 * 4刻钟 * 6特征)

目标变量的维度为：[1 * 4]

模型实现

本次实验选择了三种主流经典的机器学习模型，分别是LSTM长短期记忆网络、GRU门控循环神经网络与TCN时间卷积神经网络。接下来将详细介绍项目的实施步骤、实验结果与最终实验结果的分析。

1. LSTM

LSTM（Long Short-Term Memory，长短期记忆网络）是一种特殊的递归神经网络（RNN），在处理时间序列预测任务时表现非常优异。与传统的RNN相比，LSTM解决了长期依赖问题，使得它可以记住长时间跨度的信息，这对于时间序列数据的预测尤为重要。该模型属于隐变量自回归模型，即通过隐变量来保留一些对过去观测的总结 h_t ，并且同时更新预测 $\hat{x}_t$ 与总结 h_{t_0} 。 这就产生了基于 $\hat{x}_t = P(x_t \mid h_t)$ 估计 x_t ，以及公式 $h_t = f(h_{t-1}, x_{t-1})$ 的更新模型。

1.1 LSTM的基本结构

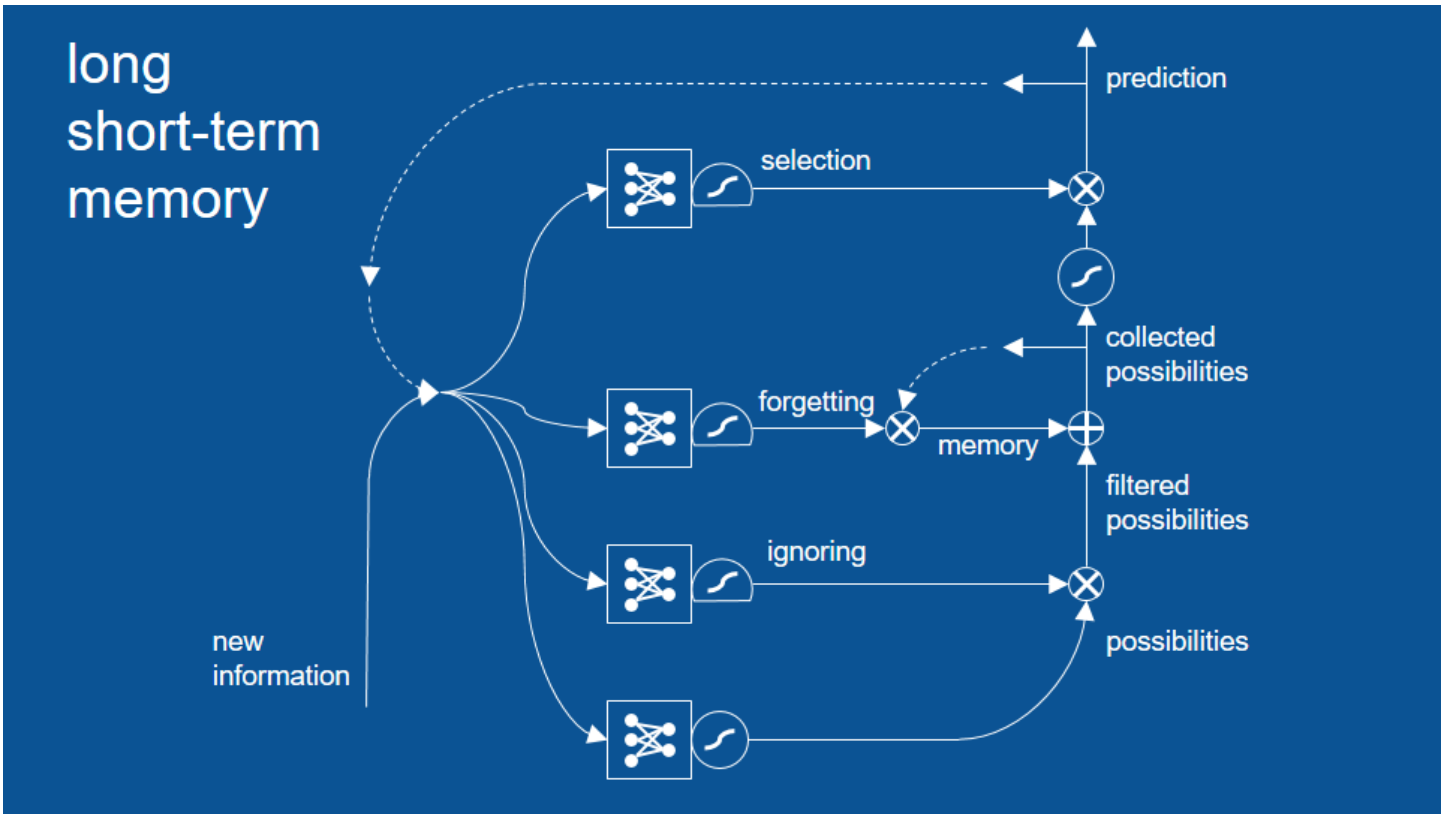
LSTM由一系列互联的“细胞”（cells）组成，它是一种特殊的隐状态设计，用于记录附加的信息。它与隐状态具有相同的形状，旨在帮助网络捕获序列中的长距离依赖关系。每个细胞有三个关键的门控单元：输入门（Input Gate）、遗忘门（Forget Gate）和输出门（Output Gate）。这些门控单元帮助LSTM在处理序列数据时，决定是否保存、更新或输出某些信息。

- **输入门（Input Gate）**：决定当前输入信息的重要性，并且需要多少信息能够流入细胞状态。
- **遗忘门（Forget Gate）**：决定细胞状态中的哪些部分应该被遗忘。
- **输出门（Output Gate）**：决定当前细胞状态的哪些部分对下一时刻有用，并生成输出。

$$
\begin{aligned}
I_t &= \sigma(X_t W_{xi} + H_{t-1} W_{hi} + b_i),\\
F_t &= \sigma(X_t W_{xf} + H_{t-1} W_{hf} + b_f),\\
O_t &= \sigma(X_t W_{xo} + H_{t-1} W_{ho} + b_o),\\
\mathbf{C}_t &= \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t, \qquad ,\\
\tilde{C}_t &= \tanh(X_t W_{xc} + H_{t-1} W_{hc} + b_c),\\
\mathbf{H}_t &= \mathbf{O}_t \odot \tanh(\mathbf{C}_t),
\end{aligned}
$$

9.2. 长短期记忆网络（LSTM） — 动手学深度学习

网络架构图：



1.2 实验步骤

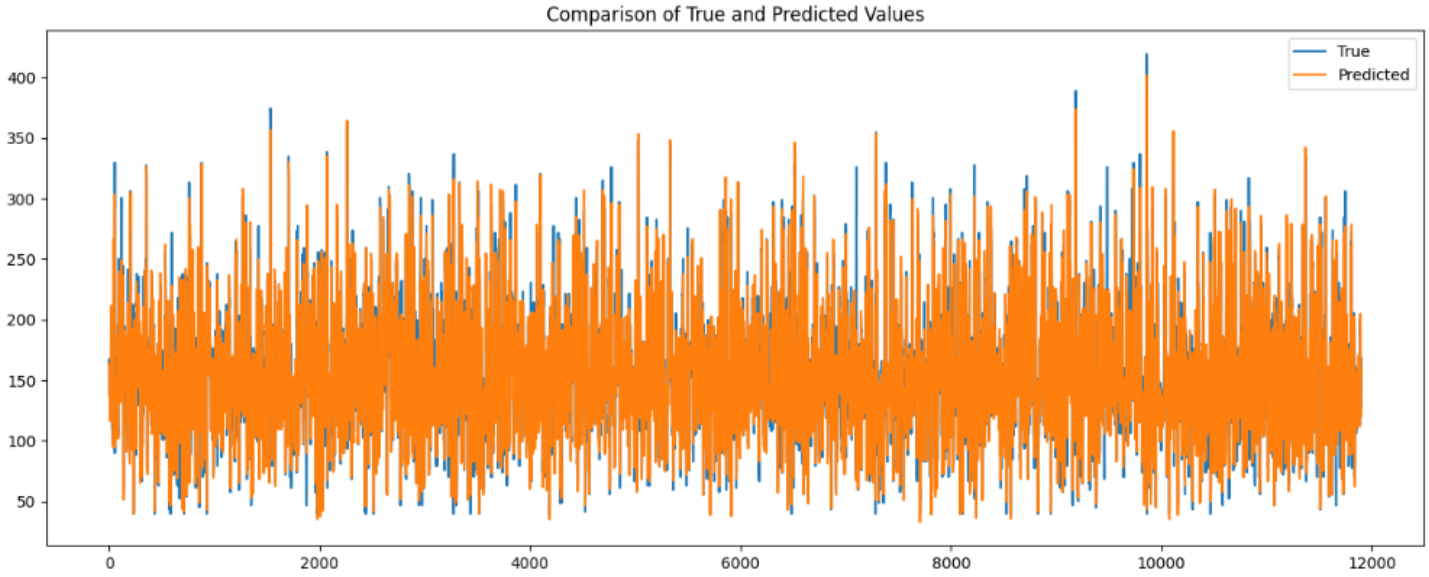
构建LSTM模型

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
class LSTM(nn.Module):
    def __init__(self,_input_size,_hidden_size,_num_layers,_output_size):
        super(LSTM,self).__init__()
        self.hidden_size=_hidden_size
        self.num_layers=_num_layers
        self.lstm=nn.LSTM(_input_size,_hidden_size,_num_layers,batch_first=True)
        self.fc=nn.Linear(_hidden_size,_output_size)
    def forward(self,x):
        # 使用relu激活函数
        h0=torch.zeros(self.num_layers,x.size(0),self.hidden_size).to(device)
        c0=torch.zeros(self.num_layers,x.size(0),self.hidden_size).to(device)
        out,_=self.lstm(x,(h0,c0))
        out=self.fc(out[:,-1,:])
        return out
input_size=len(need_columns)
hidden_size=8
num_layers=3
output_size=NFUTURE
lstm_model = LSTM(input_size, hidden_size, num_layers, output_size).to(device)
```

```
# 训练模型
from torch.utils.data import Dataset, DataLoader, TensorDataset
# 定义超参数
num_epochs = 600
initial_learning_rate = 0.01
batch_size = 128

# 定义损失函数
criterion = nn.MSELoss()

# 准备数据
train_dataset = TensorDataset(X_train_t, y_train_t)
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True, pin_memory=True, num_workers=8)

# 定义模型、优化器和学习率调度器
optimizer = torch.optim.Adam(lstm_model.parameters(), lr=initial_learning_rate)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.1)

# 训练循环
for epoch in range(num_epochs):
    lstm_model.train()
    for x_batch, y_batch in train_loader:
        x_batch = x_batch.to(device, non_blocking=True)
        y_batch = y_batch.to(device, non_blocking=True)
        outputs = lstm_model(x_batch)
        optimizer.zero_grad()
        loss = criterion(outputs, y_batch)
        loss.backward()
        # 梯度裁剪（可选）
        # torch.nn.utils.clip_grad_norm_(lstm_model.parameters(), max_norm=1.0)
        optimizer.step()

    # 每个epoch结束后更新学习率
    scheduler.step()

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}, Learning Rate: {scheduler.get_last_lr()[0]:.6f}')

# 保存模型
model_path = 'lstm_model.pth'
torch.save(lstm_model.state_dict(), model_path)
print(f'Model parameters saved to {model_path}')
```

运行结果

```
Test loss: 120.2895
LSTM MAE: 7.4900
```



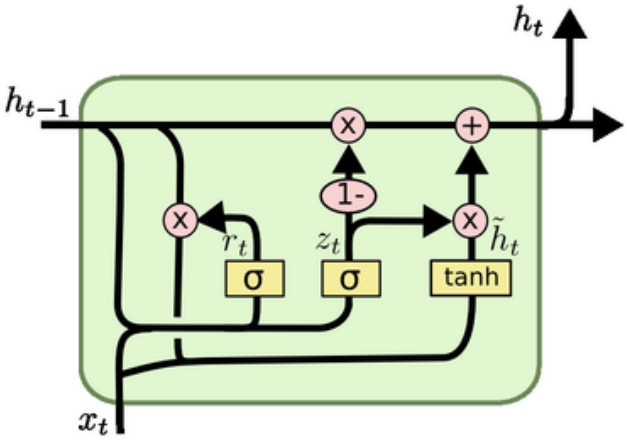
可以看到，LSTM最终的MAE较低，效果较好。

2. GRU

GRU是一种改进的循环神经网络（RNN）架构，旨在解决传统RNN在长序列数据处理中的梯度消失和梯度爆炸问题。与LSTM（长短期记忆网络）相比，GRU具有更简洁的结构，计算效率更高。GRU主要由两个门（重置门和更新门）组成，用于控制信息的流动：

- **重置门（Reset Gate）**：控制当前输入信息和先前记忆的结合方式。
- **更新门（Update Gate）**：决定了多少先前的记忆需要保留以及多少新的记忆需要添加。

2.1 GRU基本结构



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- GRU (门控循环单元) 组件
- 更新门 h_t ：
  - 功能：用于控制前一时刻的隐藏状态 $h_t - 1$ 如何影响当前时刻的隐藏状态 h_t 。

- 更新公式：$h_t = \sigma(W_z \cdot [h_t - 1, x_t])$
- 其中 σ 是 sigmoid 激活函数
- W_z 是需要学习的权重矩阵
- $[\cdot, \cdot]$ 表示向量的拼接

- 重置门 r_t：
  - 功能：用于控制如何结合前一时刻的隐藏状态 $h_t - 1$ 和当前输入 x_t。
  - 数学表达式：r_t = σ (W_r · [$h_t - 1$, x_t])

- 当前候选隐藏状态 $\tilde{h}_t$：
  - 功能：基于重置门 r_t 计算的候选隐藏状态
  - 数学表达式：$\tilde{h}_t$ = tanh $(W \cdot [r_t * h_t - 1, x_t])$
  - 其中 tanh 是激活函数
  - W 是需要学习的权重矩阵
  - $*$ 表示逐元素相乘

- 隐藏状态 h_t：
  - 功能：更新后的最终隐藏状态，通过更新门 z_t 和当前候选隐藏状态 $\tilde{h}_t$ 进行计算。
  - 数学表达式：h_t = (1− z_t) $*$ h_t −1 + z_t $*$ $\tilde{h}_t$

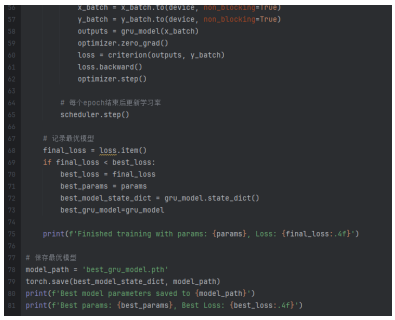
总结：

GRU 相比传统的 LSTM 更为简洁，因为它只有两个门（更新门和重置门），而 LSTM 有三个门（输入门、遗忘门和输出门）。这种简化使得 GRU 在许多任务中更容易训练，同时表现并不逊色于 LSTM。

2.2 实验步骤

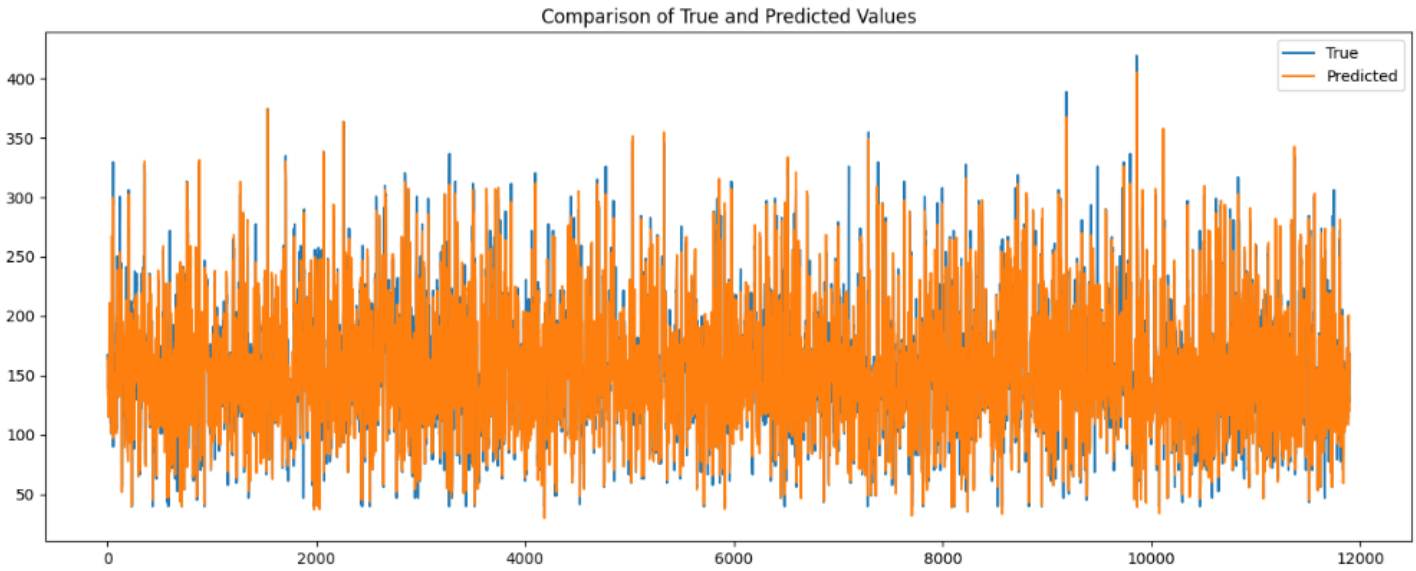
构建GRU模型

```python
# 使用GRU进行4步长的预测
class GRU(nn.Module):
    def __init__(self,_input_size,_hidden_size,_num_layers,_output_size):
        super(GRU,self).__init__()
        self.hidden_size=_hidden_size
        self.num_layers=_num_layers
        self.gru=nn.GRU(_input_size,_hidden_size,_num_layers,batch_first=True)
        self.fc=nn.Linear(_hidden_size,_output_size)
    def forward(self,x):
        h0=torch.zeros(self.num_layers,x.size(0),self.hidden_size).to(device)
        out,_=self.gru(x,h0)
        out=self.fc(out[:,-1,:])
        return out
```



运行结果

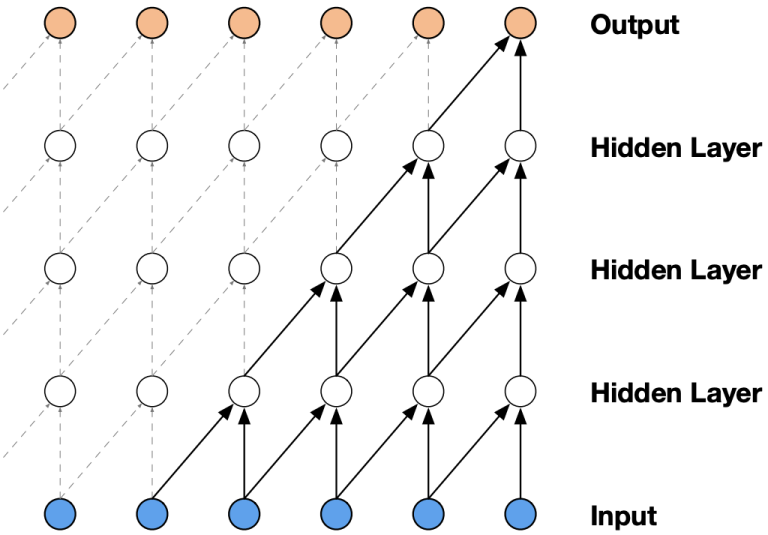
```
Test loss: 119.3730
GRU MAE: 7.4805
```



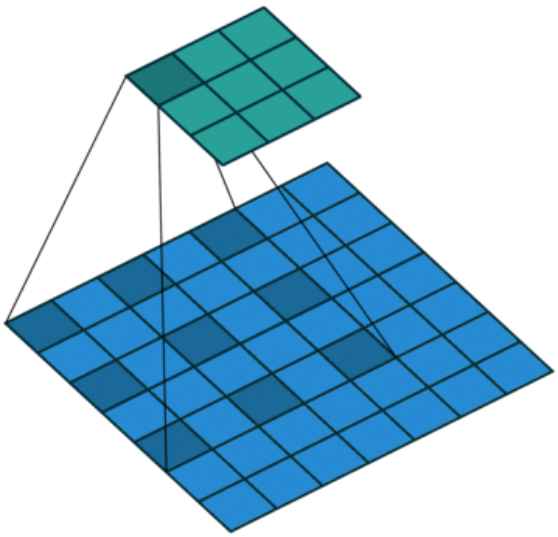
3. TCN

时间卷积网络（Temporal Convolutional Network，TCN）是一种基于卷积神经网络（CNN）的架构，专门用于处理序列数据，特别是在时间序列预测方面。TCN 在处理时间序列数据时具有许多优点，例如并行化计算、灵活的接收域和长程依赖处理等。它主要架构特点如下：

- **因果卷积**：TCN确保在预测当前时刻的值时只使用当前时刻及之前的数据，保证了模型的因果性。
- **膨胀卷积**：在卷积核中引入空洞（即在卷积核的每两个相邻元素之间插入若干个空洞），使得在不增加卷积核大小的情况下扩大接收域，从而捕获长程依赖关系。

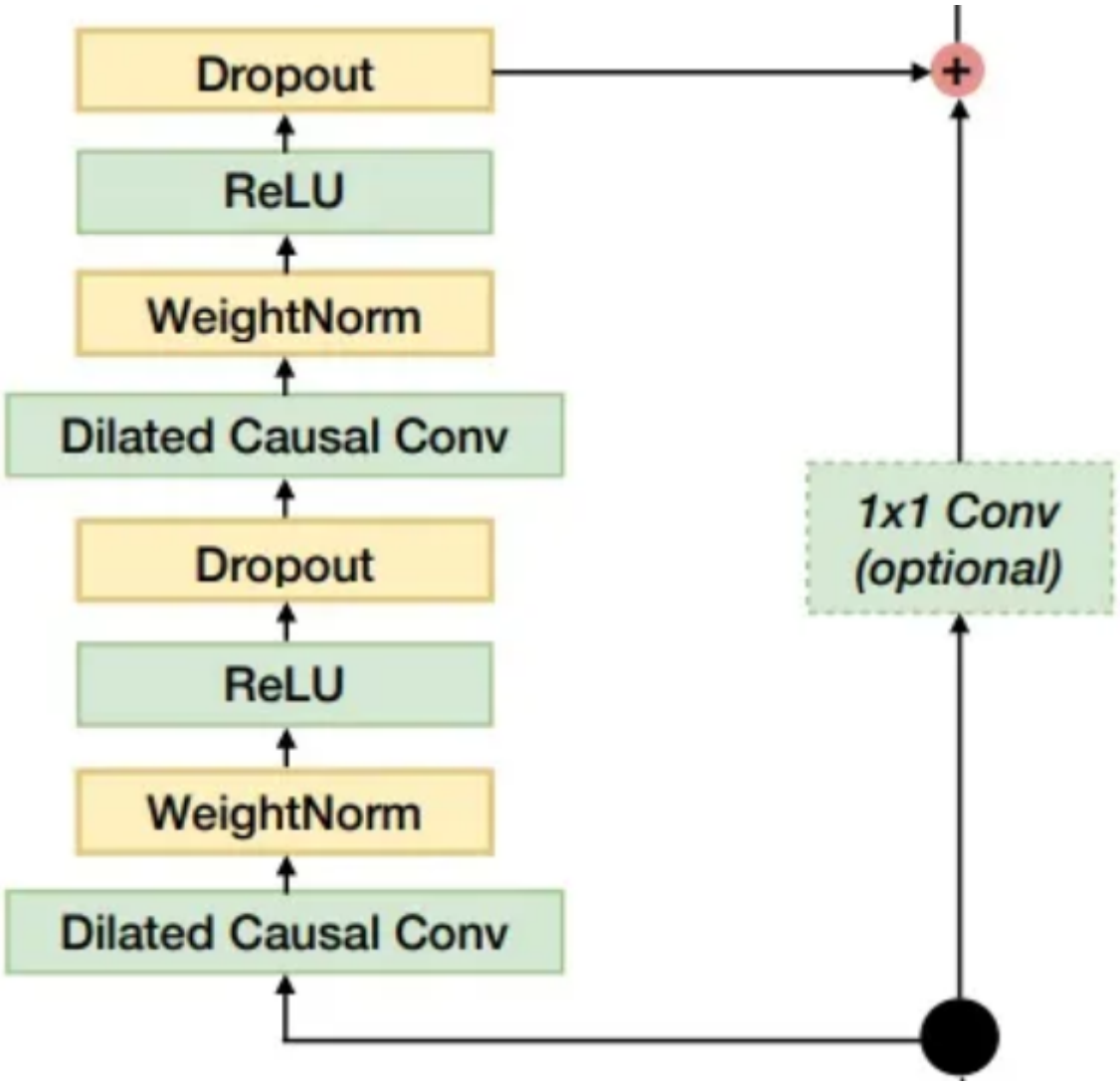


因果卷积



膨胀卷积

3.1 TCN基本结构



TCN的基本模块TemporalBlock

1. **Dilated Causal Convolution (扩张因果卷积)**:

- **因果卷积 (Causal Convolution)**: 保持时间序列数据的因果关系，即模型在时间步 t 只使用来自于时间步 $\leqslant t$ 的信息，不会泄漏未来信息。
- **扩张卷积 (Dilated Convolution)**: 使用扩张率（dilation rate）插入的零来扩大感受野，从而可以在不增加计算量的情况下捕捉更长的历史信息。

2. **Weight Normalization (权重归一化)**:
   - 在每层卷积后应用权重归一化，有助于加速训练并稳定网络的收敛。
3. **ReLU 激活函数**:
   - 在每层卷积权重归一化后，应用 ReLU 激活函数，引入非线性。
4. **Dropout**:
   - 在 ReLU 激活函数后应用 dropout，防止模型过拟合。

这些步骤形成一个“层”，并且这个基本块通常会堆叠多个这样的层。

5. **残差连接（Residual Connection）**：
   - 残差连接帮助解决深层网络中的梯度消失问题。残差连接的输入直接相加到经过两层Dilated Causal Conv后的输出。
   - 如果输入输出的通道数不一致，会通过一个 `1x1` 卷积来调整输入的形状使其与输出对齐。

3.2 实验步骤

构建TCN模型

```python
class Chomp1d(nn.Module):
    def __init__(self, chomp_size):
        super(Chomp1d, self).__init__()
        self.chomp_size = chomp_size

    def forward(self, x):
        return x[:, :, :-self.chomp_size].contiguous()


class TemporalBlock(nn.Module):
    def __init__(self, n_inputs, n_outputs, kernel_size, stride, dilation, padding, dropout=0.2):
        super(TemporalBlock, self).__init__()
        self.conv1 = weight_norm(nn.Conv1d(n_inputs, n_outputs, kernel_size,
                                           stride=stride, padding=padding, dilation=dilation))
        self.chomp1 = Chomp1d(padding)
        self.relu1 = nn.ReLU()
        self.dropout1 = nn.Dropout(dropout)

        self.conv2 = weight_norm(nn.Conv1d(n_outputs, n_outputs, kernel_size,
                                           stride=stride, padding=padding, dilation=dilation))
        self.chomp2 = Chomp1d(padding)
        self.relu2 = nn.ReLU()
        self.dropout2 = nn.Dropout(dropout)

        self.net = nn.Sequential(self.conv1, self.chomp1, self.relu1, self.dropout1,
                                 self.conv2, self.chomp2, self.relu2, self.dropout2)
        self.downsample = nn.Conv1d(n_inputs, n_outputs, 1) if n_inputs != n_outputs else None
        self.relu = nn.ReLU()
        self.init_weights()

    def init_weights(self):
        self.conv1.weight.data.normal_(0, 0.01)
        self.conv2.weight.data.normal_(0, 0.01)
        if self.downsample is not None:
            self.downsample.weight.data.normal_(0, 0.01)

    def forward(self, x):
        out = self.net(x)
        res = x if self.downsample is None else self.downsample(x)
        return self.relu(out + res)
```

```python
class TemporalConvNet(nn.Module):
    def __init__(self, num_inputs, outputs, pre_len, num_channels, n_layers, kernel_size=2, dropout=0.2):
        super(TemporalConvNet, self).__init__()
        layers = []
        self.pre_len = pre_len
        self.n_layers = n_layers
        self.hidden_size = num_channels[-2]
        self.hidden = nn.Linear(num_channels[-1], num_channels[-2])
        self.relu = nn.ReLU()
        self.lstm = nn.LSTM(self.hidden_size, self.hidden_size, n_layers, bias=True,
                            batch_first=True)  # output (batch_size, obs_len, hidden_size)
        num_levels = len(num_channels)
        for i in range(num_levels):
            dilation_size = 2 ** i
            in_channels = num_inputs if i == 0 else num_channels[i - 1]
            out_channels = num_channels[i]
            layers += [TemporalBlock(in_channels, out_channels, kernel_size, stride=1, dilation=dilation_size,
                                     padding=(kernel_size - 1) * dilation_size, dropout=dropout)]

        self.network = nn.Sequential(*layers)
        self.linear = nn.Linear(num_channels[-2], outputs)

    def forward(self, x):
        x = x.permute(0, 2, 1)
        x = self.network(x)
        x = x.permute(0, 2, 1)

        batch_size, obs_len, features_size = x.shape  # (batch_size, obs_len, features_size)
        xconcat = self.hidden(x)  # (batch_size, obs_len, hidden_size)
        H = torch.zeros(batch_size, obs_len - 1, self.hidden_size).to(  (function) to: Any
        ht = torch.zeros(self.n_layers, batch_size, self.hidden_size).to(
            device)  # (num_layers, batch_size, hidden_size)
        ct = ht.clone()
        for t in range(obs_len):
            xt = xconcat[:, t, :].view(batch_size, 1, -1)  # (batch_size, 1, hidden_size)
            out, (ht, ct) = self.lstm(xt, (ht, ct))  # ht size (num_layers, batch_size, hidden_size)
            htt = ht[-1, :, :]  # (batch_size, hidden_size)
            if t != obs_len - 1:
                H[:, t, :] = htt
        H = self.relu(H)  # (batch_size, obs_len-1, hidden_size)

        x = self.linear(H)
        return x[:, -self.pre_len:, :]
```

自行构建网络模型较为复杂并且需要网络输入需要符合特定格式，后来团队发现了pypi上已有相应的包：pytorch-tcn 1.1.0，它将TCN模型封装成了一个类，可以直接调用，因此团队决定使用这个包来实现TCN模型。

Search projects

Help Sponsors Log in Register

pytorch-tcn 1.1.0

```
pip install pytorch-tcn
```

Latest version

Released: Mar 4, 2024

Pytorch TCN

Navigation

- Project description
- Release history
- Download files

Verified details

These details have been verified by PyPI

Maintainers

PaKr

Unverified details

*These details have **not** been verified by PyPI*

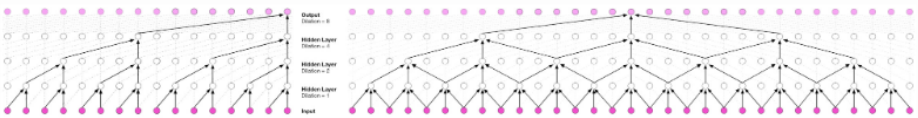
Project description

PyTorch-TCN

Streamable (Real-Time) Temporal Convolutional Networks in PyTorch

This python package provides a flexible and comprehensive implementation of temporal convolutional neural networks (TCN) in PyTorch analogous to the popular tensorflow/keras package keras-tcn. Like keras-tcn, the implementation of pytorch-tcn is based on the TCN architecture presented by Bai et al., while also including some features of the original WaveNet architecture (e.g. skip connections) and the option for automatic reset of dilation sizes to allow training of very deep TCN structures.

Additionally, this package offers a streaming inference option for causal networks (with and without lookahead). This allows to process data in small blocks instead of the whole sequence, which is essential for real-time applications. See section Streaming Inference for more details.



Dilated causal (left) and non-causal convolutions (right).

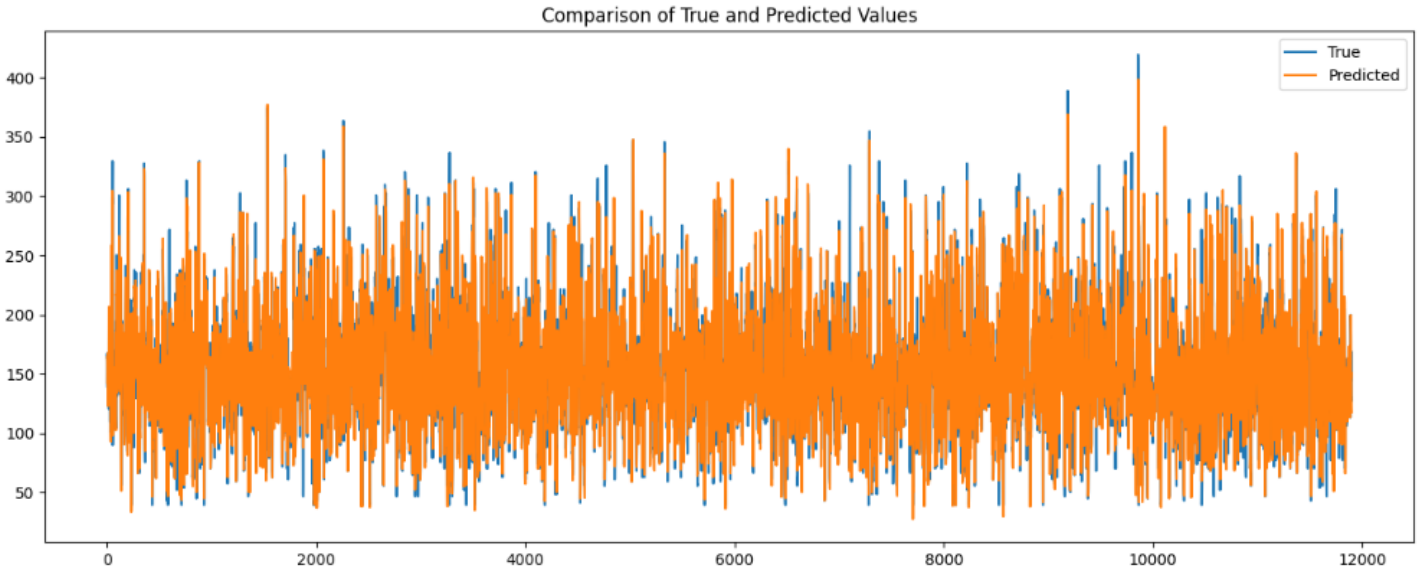
具体代码如下：

```python
from torch import nn
from pytorch_tcn import TCN

class TCNModel(nn.Module):
    def __init__(self,_input_size,_output_size,num_steps):
        super(TCNModel,self).__init__()
        self.num_steps = num_steps
        self.tcn=TCN(
            num_inputs=_input_size,
            kernel_size=4,
            dropout=0.1,
            num_channels=[16,32,64],
            use_skip_connections=True,
            causal=True,
            dilation_reset = 16,
        )
        self.fc=nn.Linear(6,_output_size)
    def forward(self,x):
        out=self.tcn(x)
        out=self.fc(out[:,-1,:])
        return out
tcn_model = TCNModel(NPAST, output_size, num_steps=NFUTURE).to(device)
```

```python
num_epochs = 100
learning_rate = 0.001
batch_size = 64
criterion = nn.MSELoss()

train_dataset = TensorDataset(X_train_t, y_train_t)
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True,pin_memory=True,num_workers=8)

optimizer = torch.optim.Adam(tcn_model.parameters(), lr=learning_rate)
for epoch in range(num_epochs):
    tcn_model.train()
    for x_batch, y_batch in train_loader:
        x_batch = x_batch.to(device,non_blocking=True)
        y_batch = y_batch.to(device,non_blocking=True)
        outputs = tcn_model(x_batch)
        optimizer.zero_grad()
        loss = criterion(outputs, y_batch)
        loss.backward()

        optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')
```

运行结果

```
Test loss: 137.7269
TCN MAE: 7.9930
```



4. 实验过程的困难

4.1 各个模型参数较多，难以确定最佳超参数

通过查阅资料得知，可以采用**网格搜索（Grid Search）**或**随机搜索（Random Search）**，找到最佳超参数组合。

在项目中，我们使用随机搜索，确定了三种模型的最佳超参，代码如图：

```python
from torch.utils.data import TensorDataset, DataLoader
from sklearn.model_selection import ParameterSampler

param_grid = {
    'initial_learning_rate': [0.001, 0.002, 0.0005, 0.0006, 0.0008],
    'step_size': [50, 100, 150],
    'gamma': [0.1, 0.5, 0.9],
    'dropout': [0.1, 0.15, 0.2],
    'kernel_size' : [2, 3, 4],
    'num_channels' : [[16,32,64],[32,64,128],[64,128,256]],
}
# 生成随机超参数组合
n_iter_search = 10  # 进行随机搜索的次数
param_list = list(ParameterSampler(param_grid, n_iter=n_iter_search, random_state=42))


input_size = len(need_columns) # 6
output_size = NFUTURE # 4 * 1 = 4

# 定义损失函数
criterion = nn.MSELoss()

# 记录最佳模型和超参数
best_loss = float('inf')
best_model_state_dict = None
best_params = None
best_model = None

# 开始随机搜索最佳超参数
for params in param_list:
    initial_learning_rate = params['initial_learning_rate']
    step_size = params['step_size']
    gamma = params['gamma']
    dropout = params['dropout']
    kernel_size = params['kernel_size']
    num_channels = params['num_channels']

    # 准备数据
    train_dataset = TensorDataset(X_train_t, y_train_t)
    train_loader = DataLoader(dataset=train_dataset, batch_size=128, shuffle=True)
```

4.2 长期依赖问题

问题描述：尽管 LSTM 设计用来处理长期依赖问题，但在实际应用中仍可能因时间跨度增大而难以捕捉到远期的依赖关系。

解决方法：

- 使用 TCN：TCN 通过其特殊的膨胀卷积设计，能够有效捕获长期依赖，为处理长序列数据提供了另一种有效的解决方案。
- 序列分割：将长序列数据分割成较短的片段进行训练，有助于缓解因序列长度过长引起的问题。

4.3 梯度消失和梯度爆炸

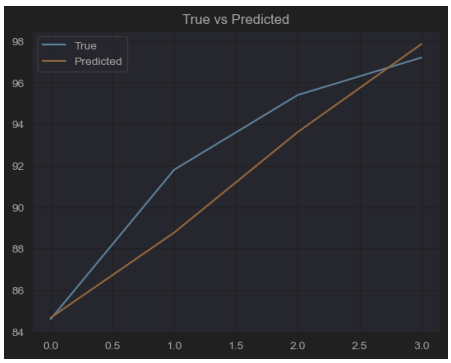
问题描述：尤其是在深层网络或长序列数据中，传统的 RNN 及其变体如 LSTM 和 GRU 可能会遇到梯度消失或爆炸的问题。

解决方法：

- 梯度裁剪：在训练过程中对梯度进行裁剪，限制梯度的最大值，防止梯度爆炸。
- 使用门控机制：GRU 和 LSTM 的门控机制本身就是为了解决梯度消失问题，确保模型中的关键信息可以长时间保留。

结果比较分析

1. 模型的预测准确性和泛化能力



通过比较的最终的MAE与预测曲线分析，我们发现：

- 第0时间点显示，真实值和预测值非常接近，模型在短时间内（15分钟）预测血糖浓度的准确性较高。
- 第1时间点显示，真实值和预测值虽然仍然相差不大，但误差开始略微上涨。这表明在30分钟预测的情况下，模型依然有良好的预测性能，但已经开始出现一些偏差。
- 在第60分钟时（3.0时间点），尤其明显，模型的误差更大，说明随着预测时间增加，模型的准确性降低。

这表明由于错误的累计，预测可能会相当快地偏离真实的观测结果。

```
MSE for 15 minute: 89.7302993698049
MSE for 30 minute: 180.3712628625243
MSE for 45 minute: 283.7302274737272
MSE for 60 minute: 388.28344116190135
MSE for one hour: 235.52880771698946
MAE for each time period:

MAE for 15 minute: 6.78191670613812
MAE for 30 minute: 9.62389103142641
MAE for 45 minute: 12.17818432070719
MAE for 60 minute: 14.458671298514098
MAE for one hour: 10.760665839196454
```

2. 不同因素是如何影响血糖预测的

我们分别将性别、BMI 等通用变量删除后，所得的预测结果（CGM）和不去掉差不多，主要是因为这些特征可能对数据集中的血糖预测影响较小。首先，我们的使用的模型结构较为简单，对特征的重要性分辨能力较低，因此一些不太显著的特征关注的较少，所以去掉一些不太显著的特征对预测结果的影响不大。其次，对于血糖类型，我们的数据集属于不平衡数据集，即1型和2型的比例类别分布不均衡（2型与1型的数量差别很大），所以该属性对血糖预测影响很小。