Python 高阶实验九：人工智能案例实践

陈兴浩　王为　王亦辉

目录

一、 问题分析

1.1 实验目标

基于 AI 框架编写人工智能程序，掌握 AI 框架的使用方法，能够在已有代码基础上修改模型结构。

1.2 问题描述

根据第 10 章人工智能应用案例课件，以流程工业控制系统时序数据预测案例为基础，尝试修改 Step_Aware_TCN_MLP 类的代码，实现如下图所示的具有更复杂结构的带偏差块的 TCN MLP with Bias Block 类，并编写相应的模型训练和测试代码。

二、 模型原理分析（陈兴浩）

2.1 数据集情况概述

数据集中共含有 14516 条由 23 个传感器记录的数据，我们需要从中挑选相应的传感器数据，进行多步预测，得到预测值。

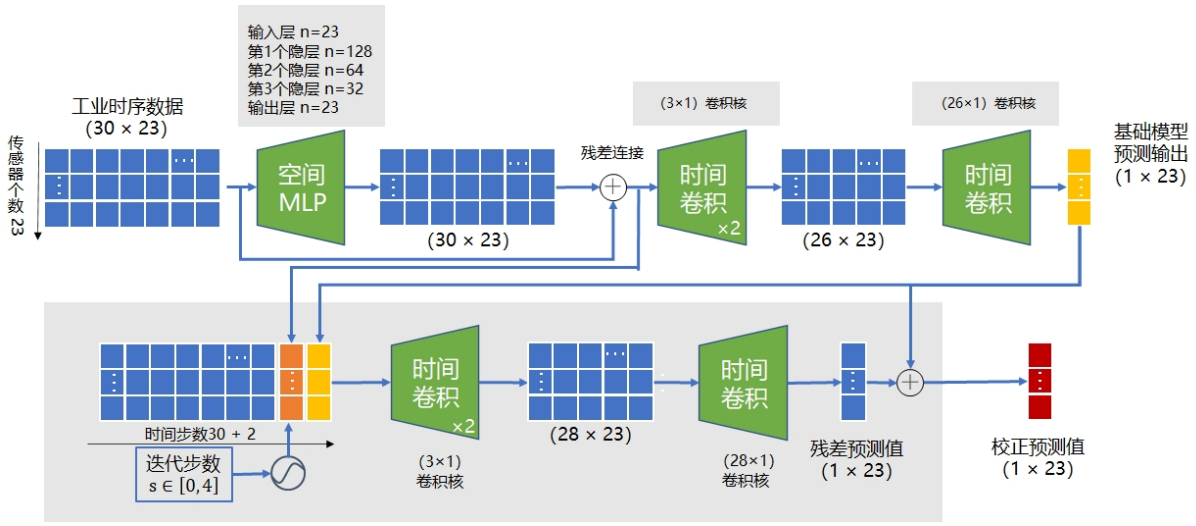

图 1　更复杂结构的带偏差块的 TCN MLP with Bias Block 类

2.2 TCN MLP with Bias Block 类基础功能概述

我们从此图片中提取整体信息，来分析我们需要实现的整体功能，同时对需要进行的代码内容做一定分析。

2.2.1 输入数据准备

输入数据的形状为 $X \in \mathbb{R}^{30\times 23}$，其中 30 表示时间步，23 表示传感器的数量。此处的 30 个时间步事实上是从 14516 条数据中进行数据分组得到的，最终有 14482 组这样的数据，因此从整个数据集处理的角度来看，输入数据的形状事实上为 $x \in \mathbb{R}^{14482\times 30\times 23}$。

2.2.2 空间特征提取（空间 MLP）

使用一个多层感知机（MLP）来提取空间特征。输入数据 $X \in \mathbb{R}^{30\times 23}$ 通过 MLP 进行处理，输出为 $H_{MLP} \in \mathbb{R}^{30\times 23}$。

残差连接操作为：

$$H_{MLP} = X + \mathrm{MLP}(X)$$

此处空间特征的提取代码与 $Step\_Aware\_TCN\_MLP$ 部分一致，即通过含有 4 个全连接层的空间 MLP 模型，提取传感器数据关联关系的特征表示，由于数据集是同一个，处理的形状也几乎相似，因此我们采用原类中的代码进行实现。

2.2.3 时间特征提取（时间卷积层）

通过两个时间卷积层（TCN）提取时间依赖关系。

第一卷积层输出形状为 $(1, 1, 28, 23)$，第二卷积层输出形状为 $(1, 1, 26, 23)$。

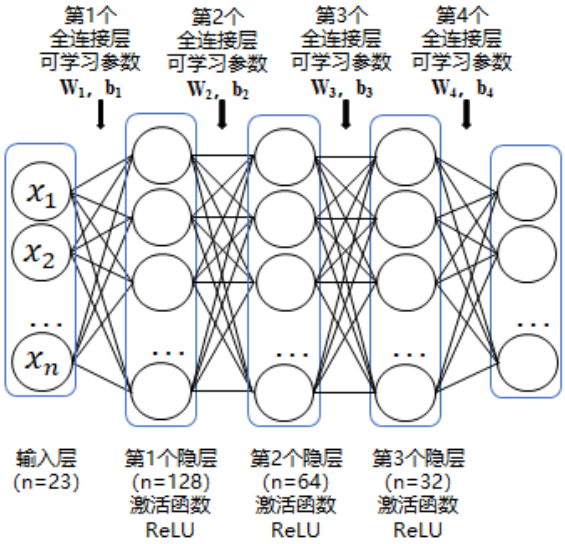

图 2　TCN MLP with Bias Block 类的 MLP 结构

数据通过卷积后使用 ReLU 激活函数。

考虑到在进行时间卷积时需要考虑通道数的大小问题，故需要提前扩展原来的输入数据结构，使之升维以适应卷积的要求。综合以上情况，卷积层如下定义，依旧遵照原有模型的结构。

卷积层定义　：

- **in_channels=1**：输入的通道数为 1，表示输入数据是单通道的。对于时间序列数据，通常每个时间步的数据是单一的通道，即 batch size $\times$ 1 $\times$ time steps $\times$ features 的四维张量。
- **out_channels=1**：输出通道数为 1，表示卷积层的输出仍然是单通道的数据。每一层卷积都会根据该通道数生成新的特征图。
- **kernel_size=(3,1)**：卷积核的大小是 $(3, 1)$，即卷积核在**时间轴**上跨越 3 个时间步，在**特征维度**上跨越 1 个特征点。这意味着该卷积核在时间序列上会在相邻的 3 个时间步内进行滑动卷积计算，并仅作用于每个时间点的一个特征。
- **pad_mode='valid'**：表示卷积操作不进行填充（即有效填充）。这意味着卷积核不能覆盖图像的边缘，因此输出的尺寸会比输入的尺寸小，具体减少的尺寸为卷积核的大小减去 1（即输出尺寸为 input_size $-$ 3 $+$ 1 $=$ input_size $-$ 2）。

此时，输出的数据结构应该如下。

输出的尺寸　：每个卷积层的 `kernel_size` 为 $(3, 1)$，且 `pad_mode='valid'`，因此每次卷积操作都会减少输出的时序长度。假设输入的形状为 (batch size, 1, time steps, features)，经过第一个卷积层后，输出的形状会变为：

$$\text{output\_shape} = (\text{batch size}, 1, \text{time steps} - 2, \text{features})$$

因为每个卷积操作会减少 2 个时间步。同样，第二个卷积层的输出会再次减少 2 个时间步，最终的输出形状为：

$$\text{final\_output\_shape} = (\text{batch size}, 1, \text{time steps} - 4, \text{features})$$

2.2.4 基础模型预测输出

通过最终卷积层将时间维度的特征汇总为一个时间步的预测。基础模型的初步预测 $Y_{\text{base}} \in \mathbb{R}^{1\times 23}$ 形状为：

$$Y_{\text{base}} \in \mathbb{R}^{1\times 23}$$

这里就是原有基础模型的输出，不再赘述。

2.3 扩展模块概述

在本模型中，比较独特的内容是加入迭代步数（step）和基础预测结果（y）作为输入，通过空间 MLP 和时间卷积进一步提取数据之间的关联性来计算残差预测值。

残差学习的核心思想是学习残差（即误差），而不是直接学习目标值。数学上，给定一个函数 $f(x)$，通过引入残差，我们可以将目标表达为：

$$y = f(x) + r(x)$$

对模型的各参数解释如下：

- y 是预测值，
- $f(x)$ 是基础的预测函数，
- $r(x)$ 是残差函数，也就是模型通过训练学到的偏差部分。

这样，模型并不是学习完全的目标值，而是学习如何修正基础预测的偏差。提高了模型的准确度。

在多步预测中，模型不仅需要基于当前时刻的数据做出预测，还需要预测多个未来时刻的值。每个时刻的预测依赖于前一个时刻的结果，因此需要引入一个“时间步数”（step）作为输入。

假设我们有一个序列 $x = [x_1, x_2, \ldots, x_T]$，目标是预测未来的 y 值。一个常见的做法是逐步预测，即：

$$\hat{y}_t = f(x_t, \hat{y}_{t-1})$$

这意味着当前的预测依赖于前一个时刻的预测结果。而不会出现当前预测反而使用了后一步数据的结果，造成数据的相互影响，削弱其关联性。

结合迭代步数和基础预测结果进行残差学习的关键点在于通过残差函数优化每个时间步的预测。具体来说，对于每一个时间步 t，模型计算出的基础预测值 y_t 之后，残差 r_t 将基于基础预测和实际值之间的差异进行调整：

$$\hat{y}_t = y_t + r_t$$

其中，基础预测 y_t 是由网络的空间和时间部分共同产生的，而残差 r_t 通过对输入数据和预测结果的进一步优化，修正模型的偏差。迭代步数 $step$ 通过调整残差预测的计算方式，帮助模型更加有效地在多步预测任务中生成更准确的预测。

将基础预测 y_t 与残差 r_t 相加的目标，是通过模型学习到当前预测的偏差并在下一步迭代时修正它。在这种结构下，模型的优化目标是：

$$L(\hat{y}_t, y_t) = L(f(x_t, \hat{y}_{t-1}, step) + r_t, y_t)$$

其中 L 表示损失函数。该损失函数通过最小化 $\hat{y}_t$ 与实际值 y_t 之间的差异，从而优化网络。

2.3.1 校正模块

时间步嵌入 引入时间步信息，通过嵌入层将当前预测的时间步映射为一个向量。输出形状为 $\mathbb{R}^{1\times 23}$。

数据拼接 将输入数据、时间步嵌入和基础模型预测输出拼接成一个新的输入张量，形状为 $(32, 23)$。拼接后的输入为：

$$\text{concatenated\_input} \in \mathbb{R}^{32\times 23}$$

2.3.2 校正卷积层

通过校正卷积层进一步提取特征。校正卷积层输出的形状为：

$$\text{校正卷积层输出} \in \mathbb{R}^{1\times 1\times 30\times 23} \quad \text{和} \quad \mathbb{R}^{1\times 1\times 28\times 23}$$

2.3.3 空间特征提取

对现有的含有基础预测值和时间步数的信息进行空间 MLP 处理，可以进一步得到预测值和迭代步数和原数据的关联关系，再对其做时间上的残差预测，可以更好地反应动态的变化关系。

因此我们对输入数据做多层感知器处理。

2.3.4 残差预测值计算

通过同样的残差卷积层计算残差预测值 $Y_{\text{residual}} \in \mathbb{R}^{1\times 23}$ 作为基础预测的校正值。由于输入数据的结构发生了改变，我们此处应用 $X \in \mathbb{R}^{28\times 1}$ 对数据做卷积操作，相应的得到 $Y_{\text{residual}} \in \mathbb{R}^{1\times 23}$ 的残差预测值。

2.3.5 校正预测输出

最终校正预测输出为：

$$Y_{\text{final}} = Y_{\text{base}} + Y_{\text{residual}}$$

其中，$Y_{\text{base}} \in \mathbb{R}^{1\times 23}$ 为基础模型的预测输出，$Y_{\text{residual}} \in \mathbb{R}^{1\times 23}$ 为残差预测值。

综合以上步骤，我们实现了更复杂结构的带偏差块的 TCN_MLP_with_Bias_Block 类。

接下来，我们将对模型实现代码做深入浅出的分析。

三、 了解残差连接（王亦辉）

在此次的 TCN_MLP 模型中，在空间 MLP 前后使用了残差连接技术。而在阅读与 SparseTSF 作对比的模型时，可以发现有许多模型使用了这个技术，因此应该了解一下这个基础技术。

3.1 提出背景

神经网络作为深度学习的核心，经历了多次波折与发展。从早期的感知机到现代的深度卷积神经网络，神经网络屡屡打破计算机视觉、语音识别等领域的性能极限。然而，随着网络层数的增加，神经网络开始遭遇到许多难以逾越的技术挑战。尤其是在深层网络中，梯度消失和信息丢失的问题愈发严重，使得训练变得异常困难。传统的神经网络在数十层以上时，网络的训练效果反而会退化，这一现象被称为模型退化问题。

3.2 残差网络的提出

为了应对这些挑战，研究者们不断探索新的解决方案，其中最具代表性的一项突破就是残差网络的提出。2015 年，微软研究院的何凯明等人提出了残差网络（ResNet）。通过在每一层之间引入残差连接，能够让信息直接绕过某些层，从而解决了深层网络中的梯度消失问题。通过这种结构，网络的训练变得更加稳定，模型的性能大幅提升。

这一突破迅速改变了深度学习的发展方向，残差网络成为了现代深度学习中一种标准的结构，并在各个领域得到了广泛应用。

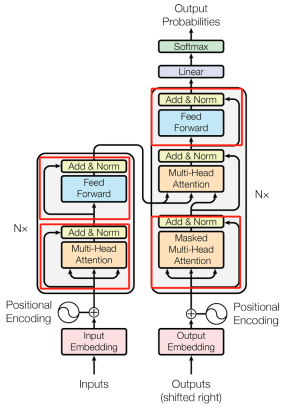

Figure 1: The Transformer - model architecture.

图 3　transformer 中的残差连接

3.3 作用与效果

残差连接允许网络将输入信息直接与输出相加，避免了信息在多层非线性变换中的丢失。传统神经网络中，数据需要经过每一层的转换才能得到最终的输出，而残差网络通过直接将输入与输出相加，实现了短路连接。

残差连接的最大优势在于，它能让网络有效缓解以下几个问题：

- **梯度消失问题**：随着层数的增加，梯度在反向传播时会变得越来越小，最终导致梯度消失。残差连接能够为梯度提供更直接的路径，使得梯度能够顺利传播到更深的层次，避免梯度消失。
- **信息流动的顺畅性**：通过跳过某些层，残差连接确保了输入信息在网络中的流动不受阻碍，使得网络能够有效捕捉输入数据中的信息，并传递到后续层。
- **模型退化问题**：传统的深层网络在增加层数时往往会出现退化现象，即网络性能下降。残差连接通过允许输入信息不经过中间层直接传递，有效避免了这个问题，反而让更深的网络表现出更好的性能。

3.4 原理

残差连接就是直接将输入与输出相加。这样的设计使得网络只需要学习输入和输出之间的残差，而不是去拟合输出。这样的设计减轻了训练难度，提高了信息传递的效率。

3.5 残差连接的普及

今天，残差连接已经成为深度学习中的一种标准技术，广泛应用于各类模型中，成为了许多深度学习模型的基础构件。例如，Transformer 结构大量使用了残差连接的设计。在时间序列预测任务中，诸如 FEDformer 和 Autoformer 等模型也广泛采用了残差连接以提升性能。使用部分如上图红框所示。

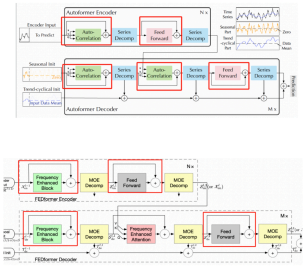

图 4　autoformer，FEDformer 中的残差连接

四、 模型的 Mindspore 实现

由于模型的基础部分与多步预测高度重合，我们直接使用多步预测的代码进行修改，且不对这一部分代码加以过多的解释而直接进行运用。而对新增功能的代码进行阐释，以保证我们实现了上文文字所述的模型。

4.1 前置准备：早停功能的实现（陈兴浩）

通过运行前文的基础代码，我们发现迭代多步（加入迭代信息）的模型在前三轮就基本完成了训练效果的提升，继续训练只能导致过拟合情况的出现。虽然模型中已经实现了训练的选择。

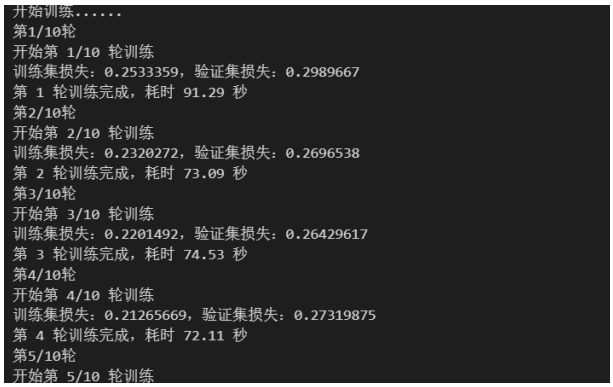

图 5　过拟合

```
if eval_loss < min_loss:
            mindspore.save_checkpoint(self.model,
   ckpt_file_path)
            min_loss = eval_loss
```

但出于运行效率的考量，我们实现早停机制。

```
class EarlyStopping:
```

```
    def __init__(self, patience=3, delta=0.002):
        self.patience = patience
        self.delta = delta
        self.best_loss = float('inf')  # 初始化为一个很大的值
        self.patience_counter = 0

    def should_stop(self, val_loss):
        if val_loss < self.best_loss - self.delta:
            self.best_loss = val_loss
            self.patience_counter = 0  # 如果验证损失改善，重置计数器
        else:
            self.patience_counter += 1
            print(f"损失已经共计有{self.patience_counter}轮未改善")
            if self.patience_counter >= self.patience:
                return True  # 如果验证损失连续'patience'轮没有改善，停止训练
        return False
```

并通过在调用函数时引入早停类对象参数，实现训练过程中的早停功能。

```
class MODEL_RUN: #定义MODEL_RUN类
    def __init__(self, model, loss_fn, optimizer=None, grad_fn=None,early_stopping = None): #构造方法
        self.model = model #设置模型
        self.loss_fn = loss_fn #设置损失函数
        self.optimizer = optimizer #设置优化器
        self.grad_fn = grad_fn #设置梯度计算函数
        self.early_stopping = EarlyStopping()

'''分割线 上方是参数定义，下方是实现'''

if self.early_stopping.should_stop(eval_loss):
                print(f"提前停止，验证集损失未改善超过 {self.early_stopping.patience} 轮")
```

```
13                  break
```

4.2 前置准备：时间计数掌握效率（陈兴浩）

在每个 Epoch 训练开始前和后加入两句计时语句即可。

```
1  def train(self, train_dataset, val_dataset, max_epoch_num,
       ckpt_file_path): #定义用于训练模型的train方法
2          min_loss = mindspore.Tensor(float('inf'), mindspore.
       float32)
3          print('开始训练......')
4          for epoch in range(1,max_epoch_num+1): #迭代训练
5              print(f'开始第 {epoch}/{max_epoch_num} 轮训练')
6              start_time = time.time()
7              self._train_one_epoch(train_dataset) #调用
       _train_one_epoch完成一轮训练
8              train_loss,_,_ = self.evaluate(train_dataset) #在
       训练集上计算模型损失值
9              eval_loss,_,_ = self.evaluate(val_dataset) #在验证
       集上计算模型损失值
10             print('训练集损失: {0}, 验证集损失: {1}'.format(
       train_loss,eval_loss))
11             if eval_loss < min_loss: #如果验证集损失值低于原来
       保存的最小损失值
12                 mindspore.save_checkpoint(self.model,
       ckpt_file_path) #更新最优模型文件
13                 min_loss = eval_loss #保存新的最小损失值
14             if self.early_stopping.should_stop(eval_loss):
15                 print(f"提前停止, 验证集损失未改善超过 {self.
       early_stopping.patience} 轮")
16                 break
17             epoch_time = time.time() - start_time
18             print(f'第 {epoch} 轮训练完成, 耗时 {epoch_time:.2
       f} 秒')
19         print('训练完成! ')
```

4.3 前置准备：动态调整学习率（王为）

在深度学习模型的训练过程中，学习率（Learning Rate）是一个关键的超参数，它决定了模型参数更新的步伐大小。合适的学习率能够加速模型收敛，提高模型性能，而不适当的学习率则可能导致训练过程缓慢甚至不收敛。为了在训练过程中动态调整学习率，本文采用了指数衰减（Exponential Decay）策略，并结合早停机制（Early Stopping）以防止过拟合。

以下是实现动态调整学习率和早停机制的关键代码片段及其详细说明：

```
class MULTI_STEP_MODEL_RUN:
    def __init__(self, model, loss_fn, optimizer=None, grad_fn=None):
        self.model = model
        self.loss_fn = loss_fn
        self.optimizer = optimizer
        self.grad_fn = grad_fn

    def train(self, train_dataset, val_dataset, max_epoch_num, ckpt_file_path, patience=5):
        min_loss = float('inf')
        no_improve_count = 0  # 记录验证集上损失未改善的次数

        # 动态学习率调整器
        lr_scheduler = mindspore.nn.exponential_decay_lr(
            learning_rate=1e-3, decay_rate=0.9, total_step=max_epoch_num, step_per_epoch=1, decay_epoch=10
        )
        print('开始训练......')
        for epoch in range(1, max_epoch_num + 1):
            # 更新学习率
            lr = lr_scheduler[epoch - 1]
            self.optimizer.learning_rate = Tensor(lr, mindspore.float32)
            print(f'开始第 {epoch}/{max_epoch_num} 轮训练，学习率：{lr}')

            start_time = time.time()
```

```
            train_loss = self._train_one_epoch(train_dataset)
            eval_loss, _, _ = self.evaluate(val_dataset)
            print(f'训练集损失：{train_loss:.6f}，验证集损失：{eval_loss:.6f}')

            # 早停机制
            if eval_loss < min_loss:
                mindspore.save_checkpoint(self.model, ckpt_file_path)
                min_loss = eval_loss
                no_improve_count = 0
                print(f'验证集损失降低，保存模型到 {ckpt_file_path}')
            else:
                no_improve_count += 1
                print(f'验证集损失未降低，连续 {no_improve_count} 次未提升')
                if no_improve_count >= patience:
                    print('验证集损失连续多次未提升，提前停止训练')
                    break

            epoch_time = time.time() - start_time
            print(f'第 {epoch} 轮训练完成，耗时 {epoch_time:.2f} 秒')
        print('训练完成！')
```

Listing 1: 动态调整学习率和早停机制的实现

上述代码实现了一个训练类 MULTI_STEP_MODEL_RUN，其中包含了动态调整学习率和早停机制的功能。下面将详细解释其中的关键部分。

4.3.1 动态学习率调整器

在 train 方法中，首先定义了一个指数衰减学习率调整器：

```
lr_scheduler = mindspore.nn.exponential_decay_lr(
```

```
    learning_rate=1e-3, decay_rate=0.9, total_step=
    max_epoch_num, step_per_epoch=1, decay_epoch=10
)
```

Listing 2: 指数衰减学习率调整器

- `learning_rate=1e-3`：初始学习率设为 0.001。
- `decay_rate=0.9`：学习率每次衰减的比例为 0.9。
- `total_step=max_epoch_num`：总的训练步数与最大训练轮数相同。
- `step_per_epoch=1` 和 `decay_epoch=10`：控制衰减的步长和频率。

在每个训练轮次开始时，获取当前轮次对应的学习率，并更新优化器的学习率参数：

```
lr = lr_scheduler[epoch - 1]
self.optimizer.learning_rate = Tensor(lr, mindspore.float32)
print(f'开始第 {epoch}/{max_epoch_num} 轮训练，学习率：{lr}')
```

Listing 3: 更新学习率

这样，随着训练的进行，学习率会按照预设的指数衰减策略逐步减小，有助于在训练后期进行更精细的参数调整，提升模型的收敛效果。

4.3.2 完整的训练流程

结合动态学习率调整和早停机制，完整的训练流程如下：

1. 初始化最小损失值 `min_loss` 为无穷大，未提升计数 `no_improve_count` 为 0。
2. 定义指数衰减学习率调整器 `lr_scheduler`。
3. 开始训练循环，每轮次执行以下步骤：
   - 根据当前轮次获取学习率，并更新优化器的学习率。
   - 执行一个训练轮次，计算训练损失 `train_loss`。
   - 在验证集上评估模型，计算验证损失 `eval_loss`。
   - 输出当前轮次的训练损失和验证损失。
   - 检查验证损失是否有改善：
     - 如果有改善，保存当前模型，更新 `min_loss`，并重置 `no_improve_count`。
     - 如果没有改善，增加 `no_improve_count`，并检查是否达到早停条件。
4. 如果达到早停条件，提前终止训练循环。
5. 训练完成后，输出训练结束信息。

通过上述方法，训练过程不仅能够动态调整学习率以适应不同训练阶段的需求，还能通过早停机制避免不必要的计算，提升整体训练效率和模型性能。

4.4 前置条件：性能优化（王为）

在提升模型性能时，我们不对原有的模型架构做出调整，也不使用学习率调度策略，而是通过以下几个方面提升训练速度：

1. ** 增大批量大小（Batch Size）**：加大批处理的规模来提高计算效率。2. ** 采用混合精度训练（Mixed Precision Training）**：使用 float16 精度来减少计算量和显存占用。3. ** 优化数据加载与处理 **：- 并行数据加载（通过设置 `num_parallel_workers`）。- 使用 `cache()` 对数据进行缓存以减少 I/O 开销。

下面的代码示例在原有基础上进行了上述优化措施的应用。

```
1
2
3  # 启用GPU和混合精度训练
4  mindspore.set_context(mode=mindspore.GRAPH_MODE, device_target
       ="GPU", precision_mode="allow_mix_precision")
5
6
7  # sensor_num, horizon, PV_index, OP_index, DV_index,
       train_X_t2, train_Y_t2, val_X_t2, val_Y_t2, test_X_t2,
       test_Y_t2
8
9  class MultiTimeSeriesDataset():
10     def __init__(self, X, Y):
11         self.X, self.Y = X, Y
12     def __len__(self):
13         return len(self.X)
14     def __getitem__(self, index):
15         return self.X[index], self.Y[index]
16
17 def generateMindsporeDataset(X, Y, batch_size):
18     # 使用并行数据加载和缓存提升数据读取性能
19     dataset = MultiTimeSeriesDataset(X.astype(np.float32), Y.
       astype(np.float32))
20     dataset = GeneratorDataset(dataset, column_names=['data','
```

```
    label'], num_parallel_workers=4)
21      dataset = dataset.batch(batch_size=batch_size,
    drop_remainder=False).cache()
22      return dataset
23
24  # 增大批量大小，提高计算效率
25  batch_size = 64
26  train_dataset_t2 = generateMindsporeDataset(train_X_t2,
    train_Y_t2, batch_size=batch_size)
27  val_dataset_t2 = generateMindsporeDataset(val_X_t2, val_Y_t2,
    batch_size=batch_size)
28  test_dataset_t2 = generateMindsporeDataset(test_X_t2,
    test_Y_t2, batch_size=batch_size)
29
30
31  model = TCN_MLP_with_Bias_Block_More()
32  loss_fn = nn.MAELoss()
33  optimizer = nn.Adam(model.trainable_params(), learning_rate=1e
    -3)
34
35
36  grad_fn = mindspore.value_and_grad(multi_step_forward_fn, None
    , optimizer.parameters, has_aux=True)
37  multi_step_model_run = MULTI_STEP_MODEL_RUN(model, loss_fn,
    optimizer, grad_fn)
38
39  # 减少不必要的打印，仅在epoch结束时打印信息
40  multi_step_model_run.train(train_dataset_t2, val_dataset_t2,
    10, 'tcn_mlp_bias_More_optimized.ckpt')
```

Listing 4: 优化后的关键代码示例

上述代码示例中：

- 增大了批量大小为 64。- 启用 GPU 与混合精度训练，加快计算速度。- 使用 `num_parallel_workers` 和 `cache()` 对数据进行并行加载与缓存，减少数据读取瓶颈。

4.5 模型适用（陈兴浩）

结合上文要求，我们在基础模型代码上进行增加。同样我们只展示改变的代码。

```
    def __init__(self): #构造方法
        super().__init__()
        self.bias_block = nn.SequentialCell(
            nn.Dense(sensor_num, 64),
            nn.ReLU(),
            nn.Dense(64, 32),
            nn.ReLU(),
            nn.Dense(32, sensor_num),
            nn.ReLU(),
            nn.Conv2d(in_channels=1, out_channels=1,
  kernel_size=(28,1), pad_mode='valid')
        )
```

由于在偏差块预测时迭代步数和基础预测值的加入，因此我们需要扩展时间卷积块的大小为 28，以适应数据的结构。

```
def construct(self, x, iter_step): #construct方法
        h = self.spatial_mlp(x)
        h = x + h
        h = h.unsqueeze(1)
        h = self.tcn(h)
        y = self.final_conv(h)
        y = y.squeeze(1)
```

以上是实现基础预测块的代码，与原有模型并无不同。

```
        #计算时间步数据的嵌入编码
        iter_step_tensor = mindspore.numpy.full((x.shape[0],
  1), iter_step, dtype=mindspore.int32)
        step_embedding = self.step_embedding(iter_step_tensor)
   #step_embedding的形状：[batch_size,1,23]
        concat_op = mindspore.ops.Concat(axis=1)
        bias_input = concat_op((x, step_embedding,y)) #
  bias_input的形状：[batch_size,32,23]
        bias_input = bias_input.unsqueeze(1)
```

```
        bias_input = self.tcn(bias_input)
        bias_input = bias_input.squeeze(1)
        bias_output = self.bias_block(bias_input.unsqueeze(1))
    #[batch_size, 1, 1, 23]
        bias_output = bias_output.squeeze(1) # [batch_size, 1,
    23]

        y = y + bias_output #y的形状: [batch_size, 1, 23]
        return y #返回计算结果
```

在该段代码中，模型处理了时间步的嵌入、通过时序卷积网络（TCN）进行处理，并加入了偏差块来修正预测结果。

1. 时间步嵌入编码 首先，生成一个张量 iter_step_tensor，它填充了一个常量 ‘iter_step‘，表示当前时间步。该张量的形状为 ‘[batch_size, 1]‘，其中 ‘batch_size‘ 是输入数据 ‘x‘ 的批次大小。接着，‘iter_step_tensor‘ 被传入嵌入层 ‘self.step_embedding‘，该层将 ‘iter_step_tensor‘ 转换为形状为 ‘[batch_size, 1, 23]‘ 的嵌入向量。这个嵌入向量用于表示当前时间步的特征。

2. 拼接输入数据、时间步嵌入和目标数据 接下来，使用 mindspore.ops.Concat(axis=1) 将输入数据 x、时间步嵌入 step_embedding 和目标数据 y 按照轴 1 拼接起来，得到 bias_input。拼接后的 bias_input 形状为 [batch_size, 32, 23]。

3. 时序卷积处理 接着，bias_input 通过 unsqueeze(1) 增加了一个维度，使得形状变为 [batch_size, 1, 32, 23]，以适应时序卷积网络（TCN）。然后，它被传入 self.tcn 层进行卷积处理，得到新的 bias_input。

4. 偏差块修正 接下来，经过时序卷积处理后的 bias_input 被传入 bias_block 进行进一步的处理。bias_input 在此之前通过 ‘unsqueeze(1)‘ 增加了一个维度。bias_block 的输出 bias_output 形状为 [batch_size, 1, 1, 23]，然后通过 squeeze(1) 移除多余的维度，最终得到形状为 [batch_size, 1, 23] 的 bias_output。

5. 加入偏差块的预测结果 最后，偏差块的输出 bias_output 被加到原始目标 y 上，以修正预测结果。最终，返回经过偏差修正后的 y。到目前为止，已经实现了对一个数据块的分析，其余数据块类似分析即可得到预测结果。

五、 实验

5.1 模型运行实验（陈兴浩）

5.1.1 训练和测试功能

点击跳转到训练和测试代码部分。执行代码阐释如下。

- **模型初始化**：创建并初始化了 TCN_MLP_with_Bias_Block_More 模型对象，定义了损失函数（平均绝对误差，MAE）和优化器（Adam）。
- **多步前向计算**：定义了 multi_step_forward_fn 函数，在每一步预测中，模型根据当前输入预测未来的时间步数据。每一步的预测结果会被加到输入数据中，控制变量（OP）始终使用真实值。最终计算多步预测的损失。
- **梯度计算和优化**：使用 mindspore.value_and_grad 计算梯度，并结合优化器进行训练。
- **模型训练**：通过 MULTI_STEP_MODEL_RUN 类执行模型训练，进行 10 个周期的训练，并保存模型。
- **模型评估**：在训练集、验证集和测试集上计算损失，并打印训练结果。

5.1.2 训练结果

训练结束图片如上。和前面几个模型进行对比，列表如下。

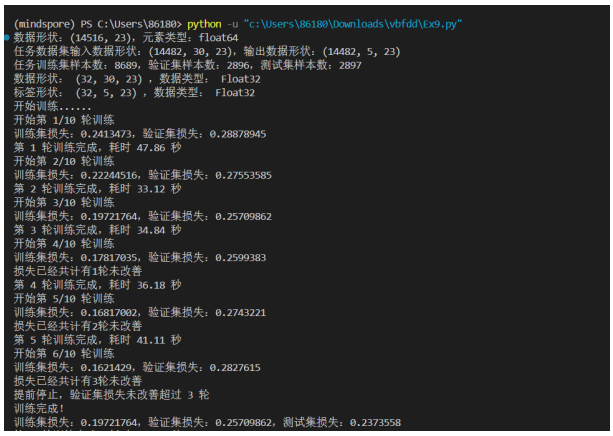

图 6　训练结束截图

数据表明，带有偏差块和迭代步数的模型在泛化性能上有所提升，能够更好地适应不同的数据模式。不仅在训练集和验证集上表现良好，且具备更强的外部验证能力，能够处理未见过的数据。

从结果可以看出，本模型在验证集和测试集都取得了相当好的优化，并且训练集损失适当，一定程度上避免了过拟合情况的出现。证明带有偏差块和迭代步数的改进模型事实上优化了预测的结果，且避免了不良情况的出现。

实验	训练集损失	验证集损失	测试集损失
实验 1: 基于任务 1 训练的 TCN_MLP 模型进行多步预测	0.184051	0.352757	0.403503
实验 2: 基于多步预测损失，重新训练 TCN_MLP 模型	0.223743	0.286676	0.259619
实验 3: 在多步预测模型中，引入迭代步数信息	0.222728	0.275755	0.246427
实验 4: 更复杂的带有偏差块结构的模型（上述代码实现的模型）	0.197218	0.257099	0.237356
实验四与最优差值	-0.013167	0.018656	0.009071

表 1　实验结果

5.2 在不同时间步的准确率（王为）

为了评估模型在不同时间步的预测准确率，我们将预测值与实际值进行了对比分析。图 7 展示了模型在第 1 至第 5 个时间步的预测结果。每个子图中，红色曲线代表模型的预测值，蓝色曲线代表实际值。

从图中可以看出，模型在各个时间步的预测结果与实际值总体上较为接近，说明模型能够较好地捕捉数据的变化趋势。然而，随着时间步的增加，预测值与实际值之间的偏差逐渐增大。这种偏差的增大可能是由于时间步的累积误差导致的，尤其在长时间步的情况下更加明显。

总体来看，模型在短时间步内的预测准确率较高，但随着时间步的增加，预测准确率有所下降。未来的改进方向可以考虑引入更多的历史信息或使用更复杂的模型结构，以提高长时间步预测的准确率。

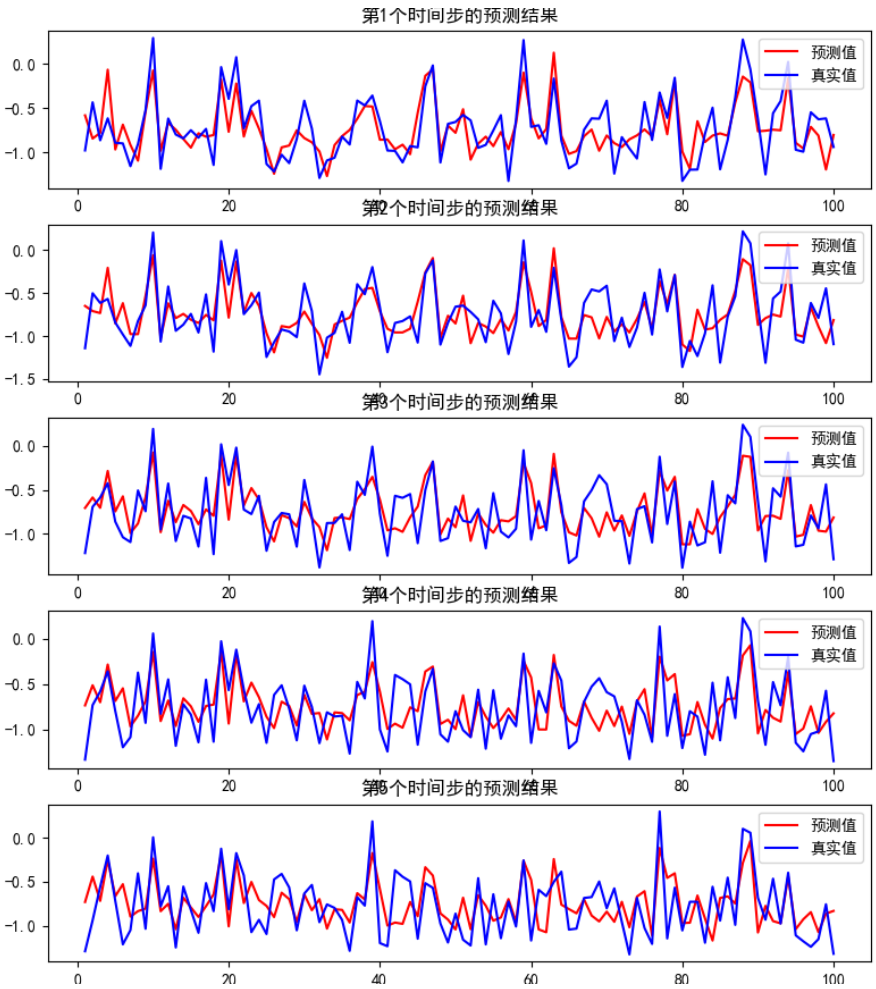

图 7　不同时间步的预测结果

5.3 消融实验（王为）

为了更加直观地理解各组件对模型性能的影响，我们对模型进行消融实验。具体而言，我们逐个移除关键模块（偏差块、时间步嵌入、空间 MLP、时序卷积层和最终卷积层），并在相同数据集条件下比较其性能。同时，我们提供一个基础模型（Basic Model）作为参考。表 2给出了不同变体在训练、验证和测试集上的损失。为便于对比，我们对每一列中的最佳（最低）值以红色标注，对次佳值以蓝色标注。

从表 2和图 8的结果可以得到以下观察与分析：

1. **偏差块（Bias Block）对预测精度的关键作用**：移除偏差块后，验证和测试集上的性能显著下降（测试集损失由 0.2045 增至 0.2317）。这说明在整合多种时空特征后，偏差块有助于微调与校正最终预测结果，是提高模型精度的核心模块之一。

2. **时间步嵌入（Step Embedding）的辅助价值**：去除时间步嵌入后，测试集损失略微上升（从 0.2045 至 0.2054），变化虽小，但仍表明对未来时间位置的嵌入可为模型提供额外信息，使预测更加准确。

3. **空间 MLP（Spatial MLP）的意外现象**：有趣的是，移除空间 MLP 后的性能不

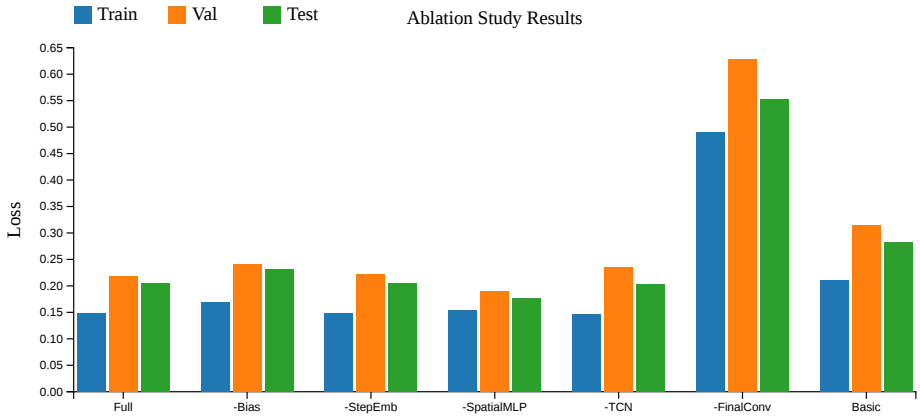

图 8　消融实验的图示说明。

仅未下降，反而显著提高（测试集损失降至 0.1772，为该列最佳）。这表明在当前数据分布下，过于复杂的空间特征提取可能导致冗余信息或过拟合问题，简化模型在此反而能提升泛化能力。该现象值得进一步深入研究。

4. **时序卷积层（TCN）对泛化能力的影响**：虽然没有 TCN 的模型在测试集上接近完整模型（0.2025 对 0.2045），但验证集损失偏高（0.2343）。这意味着 TCN 帮助模型在验证集这类近似但非完全相同分布的数据上表现更稳定。TCN 在长程时间序列特征提取与泛化方面体现了其价值。

5. **最终卷积层（Final Conv Layer）的不可或缺性**：去除最终卷积层后，模型性能显著恶化（测试集损失从 0.2045 飙升至 0.5528）。这说明最终卷积层是整合多层特征并生成高质量预测结果的关键步骤。

6. **基础模型（Basic Model）作为参照点**：与基础模型相比，各模块的增添为性能提升提供了实证支撑。基础模型的测试集损失为 0.2822，明显高于含关键模块的变体。

模型变体	训练集损失	验证集损失	测试集损失
Baseline (Full Model)	0.1492	0.2183	0.2045
− Bias Block	0.1686	0.2404	0.2317
− Step Embedding	0.1476	0.2223	0.2054
− Spatial MLP	0.1543	0.1906	0.1772
− TCN	0.1471	0.2343	0.2025
− Final Conv Layer	0.4911	0.6276	0.5528
Basic Model	0.2100	0.3140	0.2822

表 2　消融实验结果（seed=2024）

综上，消融实验清晰地展示了各组件对整体模型性能的贡献和相互作用关系。偏差块与最终卷积层是性能提升的核心因素，时间步嵌入和 TCN 在提高泛化与时间信息建模上有所助益，而空间 MLP 在本数据集上并非必要，甚至可能带来冗余与负面影响。这些发现为模型结构的设计与优化提供了实证参考，同时彰显了在实际应用中根据数据特性灵活调整模块复杂度与组合方式的重要性。

5.4 参数敏感性（王为）

为了研究学习率对模型性能的影响，我们进行了一系列实验，改变学习率的同时保持其他超参数不变（批量大小固定为 16）。测试的学习率为 1×10^{-3}、5×10^{-4}、1×10^{-4}、5×10^{-5} 和 1×10^{-5}。使用了耐心值为 3 的早停机制以防止过拟合。

实验结果如表 3所示。此外，图 10 直观展示了学习率与模型在验证集和测试集上损失之间的关系。

表 3　不同学习率下模型性能

学习率	训练损失	验证损失	测试损失
1×10^{-3}	0.2030	0.2675	0.2244
5×10^{-4}	0.1756	0.2807	0.2457
1×10^{-4}	0.1756	0.2709	0.2529
5×10^{-5}	0.2223	0.2978	0.2660
1×10^{-5}	0.3033	0.4028	0.3526

从结果可以看出，学习率对模型性能有显著影响。学习率为 1×10^{-3} 时，测试损失最低为 0.2244，表明相对较高的学习率可以更有效地找到更好的最小值。这个学习率在收敛速度和训练稳定性之间取得了平衡，从而提高了对未见数据的泛化能力。

当学习率降低到 5×10^{-4} 和 1×10^{-4} 时，训练损失略有下降，说明模型更贴合训练数据。然而，验证和测试损失增加，这可能表明模型开始过拟合训练数据，泛化能力下降。

学习率进一步减小到 5×10^{-5} 和 1×10^{-5} 时，训练和验证损失显著增加。这可能是由于优化器参数更新幅度过小，导致收敛速度慢，模型可能陷入次优解。

图 10 、 9 图示了学习率与模型性能之间的关系。曲线表明，学习率在 1×10^{-3} 附近时，模型表现最佳。偏离这个最佳学习率会导致性能下降，突显出选择合适学习率的重要性。

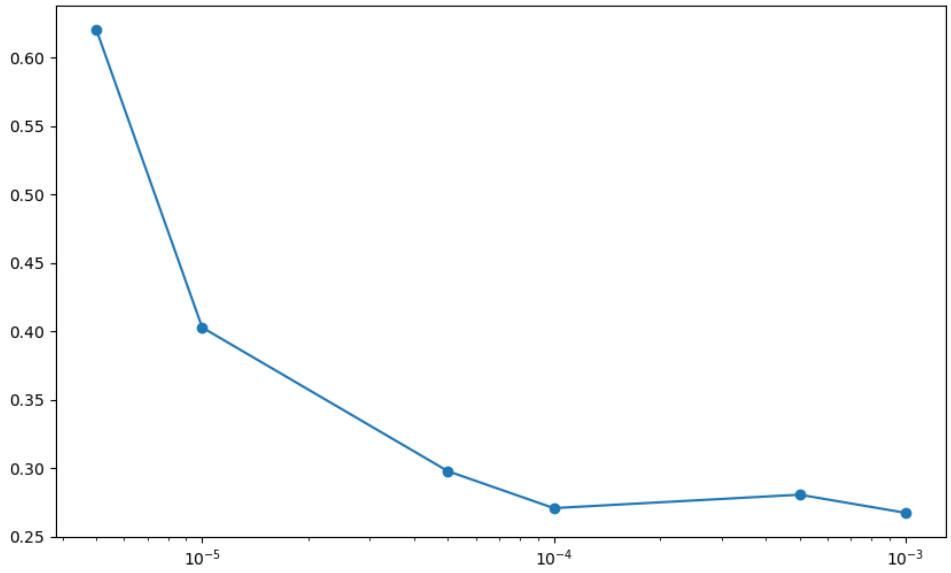

图 9　学习率对测试集损失的影响

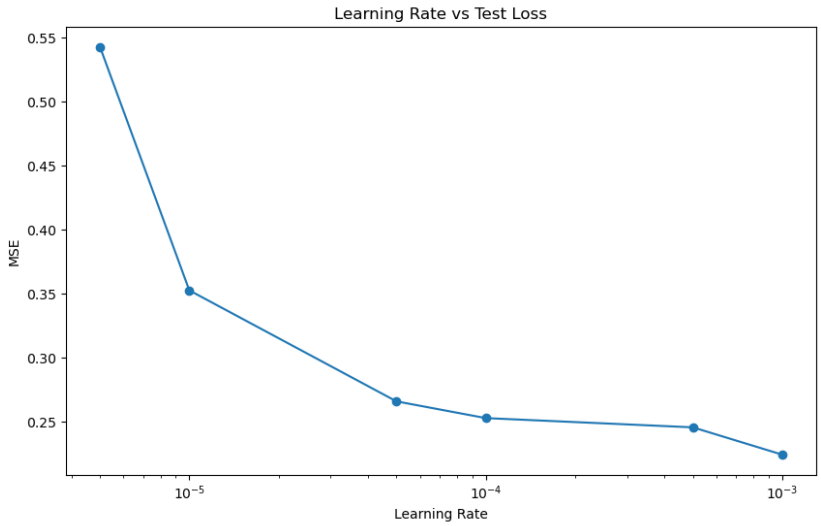

图 10　学习率对验证集损失的影响

学习率是一个对训练过程和模型最终性能有显著影响的关键超参数。合理选择学习率可以使模型高效收敛到较好的解。根据我们的实验，对于该数据集，学习率为 1×10^{-3} 提供了最佳的收敛速度和模型泛化能力之间的平衡。

六、 可能的优化方向（王为）

基于前述消融实验结果与分析，我们可以从以下几个方面对模型进行优化和改进：

1. **针对偏差块（Bias Block）的精细化设计**：偏差块在提高模型最终预测精度方面发挥了重要作用。后续可尝试：

- 引入自注意力（Self-Attention）或门控机制，对偏差输出进行动态调节，以适应不同时间步骤与传感器数据。
- 探索多偏差块级联或多层偏差结构，进一步增强模型对预测偏差的纠正能力。

2. **适度简化空间特征提取层（Spatial MLP）**：消融实验显示，移除空间 MLP 后的性能不降反升，说明过度复杂的空间特征提取可能导致冗余信息与过拟合。可考虑：
   - 减少层数或神经元数量，使网络结构更为精简。
   - 在简化的同时加入正则化（如 Dropout、L1/L2 正则）或稀疏约束，以提高模型的泛化能力。
3. **提升时间步嵌入（Step Embedding）的有效利用**：虽然移除时间步嵌入对性能影响不大，但其对时序位置的额外表征仍有价值。优化方向包括：
   - 探索更适合序列预测的时间位置编码方案（如相对位置编码、周期函数编码）。
   - 根据任务需求灵活调整时间步嵌入的维度和形式，提高模型对未来时刻的预测精度。
4. **保持并拓展最终卷积层（Final Conv Layer）的稳定性与灵活性**：最终卷积层是整合多层特征并输出高质量预测结果的关键步骤。后续可考虑：
   - 探索多分支最终卷积结构或引入可变形卷积技术，提高特征融合的灵活性与鲁棒性。
   - 针对不同传感器特性设计特定卷积核结构，更好地适应复杂的时空数据。

在未来的研究中，可以根据应用场景与数据特性，从上述方向中进行有针对性的尝试和改进。

七、 分工

在本项目中，各成员的具体分工如下：

- **王为**：
  - 完成模型的 Mindspore 实现的动态学习率调整和性能优化。
  - 对模型进行实验（消融实验、准确率实验、参数敏感性实验等）。
  - 根据实验结果探索可能的模型优化方向。
  - 辅助其它模块的书写与论文排版。
- **王亦辉**：
  - 完成了解残差连接部分。
- **陈兴浩**
  - 完成主要模型 Mindspore 实现代码。

– 运行并分析模型运行预测结果。
– 学习并撰写模型原理部分。
– 辅助其它模块的书写。

A 附录：模型代码

```
import numpy as np #导入numpy工具包
import mindspore
from mindspore import Tensor #导入Tensor类
from mindspore.dataset import GeneratorDataset #导入
   GeneratorDataset类
from mindspore import nn
import matplotlib.pyplot as plt
sensor_num = 23 #传感器数量
horizon = 5 #预测的时间步数
PV_index = [idx for idx in range(9)] #PV变量的索引值范围
OP_index = [idx for idx in range(9,18)] #OP变量的索引值范围
DV_index = [idx for idx in range(18,sensor_num)] #DV变量的索引
   值范围
data_path = 'C:\\Users\\86180\\Downloads\\vbfdd\\data_train.
   csv' #数据文件路径

data = np.loadtxt(data_path, delimiter=',', skiprows=1,
   usecols=range(1,sensor_num+1)) #读取数据（忽略第1行的标题及
   第1列的时间戳）
print('数据形状：{0}，元素类型：{1}'.format(data.shape, data.
   dtype))
def generateData(data, X_len, Y_len, sensor_num):#定义
   generateData函数
    point_num = data.shape[0] #时间点总数
    sample_num = point_num-X_len-Y_len+1 #生成的总样本数
    X = np.zeros((sample_num, X_len, sensor_num)) #用于保存输
   入数据
    Y = np.zeros((sample_num, Y_len, sensor_num)) #用于保存对
   应的输出数据
    for i in range(sample_num): #通过遍历逐一生成输入数据和对
   应的输出数据
        X[i] = data[i:i+X_len] #前X_len个时间点数据组成输入数
   据
```

```
        Y[i] = data[i+X_len:i+X_len+Y_len]#后Y_len个时间点数据组成输出数据
    return X, Y #返回所生成的模型的输入数据X和输出数据Y


X_t2, Y_t2 = generateData(data, 30, horizon, sensor_num) #生成任务2所用的数据集

print('任务数据集输入数据形状：{0}，输出数据形状：{1}'.format(X_t2.shape, Y_t2.shape))
def splitData(X, Y): #定义splitData函数
    N = X.shape[0] #样本总数
    train_X,train_Y=X[:int(N*0.6)],Y[:int(N*0.6)] #前60%的数据作为训练集
    val_X,val_Y=X[int(N*0.6):int(N*0.8)],Y[int(N*0.6):int(N*0.8)] #中间20%的数据作为验证集
    test_X,test_Y=X[int(N*0.8):],Y[int(N*0.8):] #最后20%的数据作为测试集
    return train_X,train_Y, val_X,val_Y, test_X,test_Y#返回划分好的数据集


train_X_t2, train_Y_t2, val_X_t2, val_Y_t2, test_X_t2, test_Y_t2=splitData(X_t2, Y_t2) #划分任务2的数据集
s = '训练集样本数：{0}，验证集样本数：{1}，测试集样本数：{2}'


print('任务'+s.format(train_X_t2.shape[0], val_X_t2.shape[0], test_X_t2.shape[0])) #输出任务2训练集、验证集和测试集的样本数
class MultiTimeSeriesDataset(): #定义MultiTimeSeriesDataset类
    def __init__(self, X, Y): #构造方法
        self.X, self.Y = X, Y #设置输入数据和输出数据
    def __len__(self):
        return len(self.X) #获取数据的长度
```

```
48      def __getitem__(self, index):
49          return self.X[index], self.Y[index] #根据索引值为index
        的数据
50
51  def generateMindsporeDataset(X, Y, batch_size): #定义
        generateMindsporeDataset函数
52      dataset = MultiTimeSeriesDataset(X.astype(np.float32), Y.
        astype(np.float32)) #根据X和Y创建MultiTimeSeriesDataset类对
        象
53      dataset = GeneratorDataset(dataset, column_names=['data','
        label']) #创建GeneratorDataset类对象，并指定数据集两列的列名
        称分别是data和label
54      dataset = dataset.batch(batch_size=batch_size,
        drop_remainder=False) #将数据集分成多个批次，以支持批量训练
55      return dataset #返回可用于模型训练和测试的数据集
56
57
58  train_dataset_t2 = generateMindsporeDataset(train_X_t2,
        train_Y_t2, batch_size=32)
59  val_dataset_t2 = generateMindsporeDataset(val_X_t2, val_Y_t2,
        batch_size=32)
60  test_dataset_t2 = generateMindsporeDataset(test_X_t2,
        test_Y_t2, batch_size=32)
61
62  for data, label in train_dataset_t2.create_tuple_iterator():
63      print('数据形状：', data.shape, '，数据类型：', data.dtype
        )
64      print('标签形状：', label.shape, '，数据类型：', label.
        dtype)
65      break
66  class EarlyStopping:
67      def __init__(self, patience=3, delta=0.002):
68          self.patience = patience
69          self.delta = delta
70          self.best_loss = float('inf')  # 初始化为一个很大的值
```

```
71          self.patience_counter = 0  # 计算没有改进的epoch数量
72
73      def should_stop(self, val_loss):
74          if val_loss < self.best_loss - self.delta:
75              self.best_loss = val_loss
76              self.patience_counter = 0  # 如果验证损失改善，重置计数器
77          else:
78              self.patience_counter += 1
79              print(f"损失已经共计有{self.patience_counter}轮未改善")
80              if self.patience_counter >= self.patience:
81                  return True  # 如果验证损失连续'patience'轮没有改善，停止训练
82          return False
83  class TCN_MLP(nn.Cell): #定义TCN_MLP类
84      def __init__(self): #构造方法
85          super().__init__() #调用父类的构造方法
86          #对不同传感器的数据做融合（提取传感器数据间的关联特征）
87          self.spatial_mlp = nn.SequentialCell(
88              nn.Dense(sensor_num, 128),
89              nn.ReLU(),
90              nn.Dense(128, 64),
91              nn.ReLU(),
92              nn.Dense(64, 32),
93              nn.ReLU(),
94              nn.Dense(32, sensor_num)
95          )
96          #对时间序列做卷积（提取时间点数据间的关联特征）
97          self.tcn = nn.SequentialCell(
98              nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(3,1), pad_mode='valid'),
99              nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(3,1), pad_mode='valid'),
```

```
        )
        #通过一个卷积层得到最后的预测结果
        self.final_conv = nn.Conv2d(in_channels=1,
    out_channels=1, kernel_size=(26, 1), pad_mode='valid') #使用
    26*1卷积核，不补边
     def construct(self, x, step=None): #construct方法
        #输入数据x的形状：[batch_size, 30, 23]
        h = self.spatial_mlp(x) #经过spatial_mlp空间处理后，得
    到的数据h的形状：[batch_size, 30, 23]
        x = x + h #残差连接，将x和h对应元素相加，得到的数据x的
    形状：[batch_size, 30, 23]
        x = x.unsqueeze(1) #根据卷积操作需要，将3维数据升为4维
    数据：[batch_size, 1, 30, 23]
        x = self.tcn(x) #经过tcn时间卷积后，得到的数据x的形
    状：[batch_size, 1, 26, 23]
        y = self.final_conv(x) #通过26*1的卷积操作后，得到的数
    据y的形状：[batch_size, 1, 1, 23]
        y = y.squeeze(1) #将前面增加的维度去掉，得到的数据y的
    形状：[batch_size, 1, 23]
        return y #返回计算结果
from mindspore import nn
import time
mindspore.set_context(mode=mindspore.GRAPH_MODE) #设置为静态图
    模式
class MODEL_RUN: #定义MODEL_RUN类
     def __init__(self, model, loss_fn, optimizer=None, grad_fn
    =None,early_stopping = None): #构造方法
        self.model = model #设置模型
        self.loss_fn = loss_fn #设置损失函数
        self.optimizer = optimizer #设置优化器
        self.grad_fn = grad_fn #设置梯度计算函数
        self.early_stopping = EarlyStopping()
     def _train_one_step(self, data, label): #定义用于单步训练
    的_train_one_step方法
        (loss, _), grads = self.grad_fn(data, label) #根据数据
```

```
和标签计算损失和梯度
124         self.optimizer(grads) #根据梯度进行模型优化
125         return loss #返回损失值
126     def _train_one_epoch(self, train_dataset): #定义用于一轮训
练的_train_one_epoch方法
127         self.model.set_train(True) #设置为训练模式
128         for data, label in train_dataset.create_tuple_iterator
(): #取出每一批数据
129             self._train_one_step(data, label) #调用
_train_one_step方法进行模型参数优化
130     def evaluate(self, dataset, step=None): #定义用于评估模型
的evaluate方法
131         self.model.set_train(False) #设置为测试模式
132         ls_pred,ls_label=[],[] #分别用于保存预测结果和标签
133         for data, label in dataset.create_tuple_iterator(): #
遍历每批数据
134             pred = self.model(data) #使用模型对一批数据进行预
测
135             ls_pred += list(pred[:,:,PV_index].asnumpy()) #保
存预测结果
136             ls_label += list(label[:,:,PV_index].asnumpy()) #
保存标签
137         return loss_fn(Tensor(ls_pred), Tensor(ls_label)), np.
array(ls_pred), np.array(ls_label)
138     def train(self, train_dataset, val_dataset, max_epoch_num,
 ckpt_file_path): #定义用于训练模型的train方法
139         min_loss = mindspore.Tensor(float('inf'), mindspore.
float32)
140         print('开始训练......')
141         for epoch in range(1,max_epoch_num+1): #迭代训练
142             print(f'开始第 {epoch}/{max_epoch_num} 轮训练')
143             start_time = time.time()
144             self._train_one_epoch(train_dataset) #调用
_train_one_epoch完成一轮训练
145             train_loss,_,_ = self.evaluate(train_dataset) #在
```

```
    训练集上计算模型损失值
146             eval_loss,_,_ = self.evaluate(val_dataset) #在验证
    集上计算模型损失值
147             print('训练集损失：{0}，验证集损失：{1}'.format(
    train_loss,eval_loss))
148             if eval_loss < min_loss: #如果验证集损失值低于原来
    保存的最小损失值
149                 mindspore.save_checkpoint(self.model,
    ckpt_file_path) #更新最优模型文件
150                 min_loss = eval_loss #保存新的最小损失值
151             if self.early_stopping.should_stop(eval_loss):
152                 print(f"提前停止，验证集损失未改善超过 {self.
    early_stopping.patience} 轮")
153                 break
154             epoch_time = time.time() - start_time
155             print(f'第 {epoch} 轮训练完成，耗时 {epoch_time:.2
    f} 秒')
156         print('训练完成！ ')
157     def test(self, test_dataset, ckpt_file_path): #定义用于测
    试模型的test方法
158         mindspore.load_checkpoint(ckpt_file_path, net=self.
    model) #从文件中加载模型
159         loss,preds,labels = self.evaluate(test_dataset) #在测
    试集上计算模型损失值
160         return loss,preds,labels #返回损失值
161 class MULTI_STEP_MODEL_RUN(MODEL_RUN): #定义
    MULTI_STEP_MODEL_RUN类
162     def __init__(self, model, loss_fn, optimizer=None, grad_fn
    =None,early_stopping=None): #构造方法
163         early_stopping = EarlyStopping()
164         super().__init__(model, loss_fn, optimizer, grad_fn,
    early_stopping)
165     def evaluate(self, dataset): #重定义evaluate方法
166         self.model.set_train(False) #设置为测试模式
167         ls_pred,ls_label=[],[] #分别用于保存预测结果和标签
```

```
168         for data, label in dataset.create_tuple_iterator(): #遍历每批数据
169             muti_step_pred = mindspore.numpy.zeros_like(label[:,:,PV_index])
170             x = data
171             for step in range(horizon):
172                 pred = self.model(x, step) #使用sa_tcn_mlp模型进行预测
173                 muti_step_pred[:,step:step+1,:] = pred[:,:,PV_index] #将当前时间步的预测结果保存到multi_step_pred中
174                 concat_op = mindspore.ops.Concat(axis=1)
175                 x = concat_op((x[:,1:,:], pred)) #将预测结果加到输入中
176                 x[:,-1:,OP_index] = label[:,step:step+1,OP_index] #OP控制变量无法预测、始终使用真实值
177             ls_pred += list(muti_step_pred.asnumpy()) #保存预测结果
178             ls_label += list(label[:,:,PV_index].asnumpy()) #保存标签
179         return loss_fn(Tensor(ls_pred), Tensor(ls_label)),np.array(ls_pred),np.array(ls_label)
180
181
182 class TCN_MLP_with_Bias_Block_More(nn.Cell): #定义TCN_MLP_with_Bias_Block类
183     def __init__(self): #构造方法
184         super().__init__() #调用父类的构造方法
185         #比MULTI_STEP_TCN_MLP类增加一个偏差块
186         self.bias_block = nn.SequentialCell(
187             nn.Dense(sensor_num, 64),
188             nn.ReLU(),
189             nn.Dense(64, 32),
190             nn.ReLU(),
191             nn.Dense(32, sensor_num),
192             nn.ReLU(),
```

```
193             nn.Conv2d(in_channels=1, out_channels=1,
    kernel_size=(28,1), pad_mode='valid')
194         )
195         #对预测时间步数据的嵌入编码操作
196         self.step_embedding = nn.Embedding(horizon, sensor_num
    )
197         #对不同传感器的数据做融合（提取传感器数据间的关联特
    征）
198         self.spatial_mlp = nn.SequentialCell(
199             nn.Dense(sensor_num, 128),
200             nn.ReLU(),
201             nn.Dense(128, 64),
202             nn.ReLU(),
203             nn.Dense(64, 32),
204             nn.ReLU(),
205             nn.Dense(32, sensor_num)
206         )
207         #对时间序列做卷积（提取时间点数据间的关联特征）
208         self.tcn = nn.SequentialCell(
209             nn.Conv2d(in_channels=1, out_channels=1,
    kernel_size=(3,1), pad_mode='valid'),
210             nn.Conv2d(in_channels=1, out_channels=1,
    kernel_size=(3,1), pad_mode='valid'),
211         )
212         #通过一个卷积层得到最后的预测结果
213         self.final_conv = nn.Conv2d(in_channels=1,
    out_channels=1, kernel_size=(26, 1), pad_mode='valid') #使用
    26*1卷积核，不补边
214
215     def construct(self, x, iter_step): #construct方法
216         h = self.spatial_mlp(x) #经过spatial_mlp空间处理后，得
    到的数据h的形状：[batch_size, 30, 23]
217         #输入数据x的形状：[batch_size, 30, 23]
218         h = x + h #残差连接，将x和h对应元素相加，得到的数据h的
    形状：[batch_size, 30, 23]
```

```
        h = h.unsqueeze(1) #根据卷积操作需要，将3维数据升为4维数据：[batch_size, 1, 30, 23]
        h = self.tcn(h) #经过tcn时间卷积后，得到的数据x的形状：[batch_size, 1, 26, 23]
        y = self.final_conv(h) #通过26*1的卷积操作后，得到的数据y的形状：[batch_size, 1, 1, 23]
        y = y.squeeze(1) #将前面增加的维度去掉，得到的数据y的形状：[batch_size, 1, 23]
        #计算时间步数据的嵌入编码
        iter_step_tensor = mindspore.numpy.full((x.shape[0], 1), iter_step, dtype=mindspore.int32)
        step_embedding = self.step_embedding(iter_step_tensor) #step_embedding的形状：[batch_size,1,23]


        concat_op = mindspore.ops.Concat(axis=1)
        bias_input = concat_op((x, step_embedding,y)) #bias_input的形状：[batch_size,32,23]
        bias_input = bias_input.unsqueeze(1)
        bias_input = self.tcn(bias_input)


        bias_input = bias_input.squeeze(1)


        bias_output = self.bias_block(bias_input.unsqueeze(1)) #[batch_size, 1, 1, 23]
        bias_output = bias_output.squeeze(1) # [batch_size, 1, 23]
        #加上偏差块的预测结果
        y = y + bias_output #y的形状：[batch_size, 1, 23]
        return y #返回计算结果
```

B 训练和测试功能代码

```
1
2  tcn_mlp_bias_more = TCN_MLP_with_Bias_Block_More() #创建
      TCN_MLP_with_Bias_Block类对象tcn_mlp_bias
3  loss_fn = nn.MAELoss() #定义损失函数
4  multi_step_optimizer = nn.Adam(tcn_mlp_bias_more.
      trainable_params(), 1e-3) #使用Adam优化器
5  def multi_step_forward_fn(data, label): #定义多步预测前向计算
      的multi_step_forward_fn方法
6      muti_step_pred = mindspore.numpy.zeros_like(label[:,:,
      PV_index+DV_index])
7      x = data
8      for step in range(horizon):
9          pred = tcn_mlp_bias_more(x, step) #使用tcn_mlp_bias模
      型进行预测
10         muti_step_pred[:,step:step+1,:] = pred[:,:,PV_index+
      DV_index] #将当前时间步的预测结果保存到multi_step_pred中
11         concat_op = mindspore.ops.Concat(axis=1)
12         x = concat_op((x[:,1:,:], pred)) #将预测结果加到输入中
13         x[:,-1:,OP_index] = label[:,step:step+1,OP_index] #OP
      控制变量无法预测、始终使用真实值
14     loss = loss_fn(muti_step_pred, label[:,:,PV_index+DV_index
      ]) #根据损失函数计算PV和DV变量的损失值
15     return loss, muti_step_pred #返回损失值和预测结果
16 multi_step_grad_fn = mindspore.value_and_grad(
      multi_step_forward_fn, None, multi_step_optimizer.parameters
      , has_aux=True) #获取用于计算梯度的函数
17 multi_step_model_run = MULTI_STEP_MODEL_RUN(tcn_mlp_bias_more,
       loss_fn, multi_step_optimizer, multi_step_grad_fn) #创建
      MODEL_RUN类对象model_run
18 multi_step_model_run.train(train_dataset_t2, val_dataset_t2,
      10, 'tcn_mlp_bias_More.ckpt') #调用model_run.train方法完成训
      练
19
20 tcn_mlp_bias = TCN_MLP_with_Bias_Block_More() #创建
```

```
    TCN_MLP_with_Bias_Block类对象tcn_mlp_bias
loss_fn = nn.MAELoss() #定义损失函数
multi_step_model_run = MULTI_STEP_MODEL_RUN(tcn_mlp_bias,
    loss_fn) #创建MULTI_STEP_MODEL_RUN类对象multi_step_model_run
train_loss,_,_ = multi_step_model_run.test(train_dataset_t2, '
    tcn_mlp_bias_More.ckpt') #计算训练集损失
val_loss,_,_ = multi_step_model_run.test(val_dataset_t2, '
    tcn_mlp_bias_More.ckpt') #计算验证集损失
test_loss,preds,labels = multi_step_model_run.test(
    test_dataset_t2, 'tcn_mlp_bias_More.ckpt') #计算测试集损失
print('训练集损失：{0}，验证集损失：{1}，测试集损失：{2}'.
    format(train_loss,val_loss,test_loss))
plt.rcParams['font.family'] = 'SimHei'
plt.rcParams['axes.unicode_minus'] = False
_,axes = plt.subplots(5,1,figsize=(8, 16))
interval = int(horizon/5)
for step in range(5):
    axes[step].set_title('第%d个时间步的预测结果'%(step*
    interval+1))
    axes[step].plot(range(1,101), preds[:100,step*interval,0],
    color='Red') # 绘制第1个传感器的前100条数据的预测结果
    axes[step].plot(range(1,101), labels[:100,step*interval
    ,0], color='Blue') # 绘制第1个传感器的前100条数据的标签
```

C 消融实验代码

```
class TCN_MLP_with_Bias_Block_More(nn.Cell): #定义
    TCN_MLP_with_Bias_Block类
    def __init__(self): #构造方法
        super().__init__() #调用父类的构造方法
        #比MULTI_STEP_TCN_MLP类增加一个偏差块
        self.bias_block = nn.SequentialCell(
            nn.Dense(sensor_num, 64),
            nn.ReLU(),
            nn.Dense(64, 32),
```

```
            nn.ReLU(),
            nn.Dense(32, sensor_num),
            nn.ReLU(),
            nn.Conv2d(in_channels=1, out_channels=1,
kernel_size=(28,1), pad_mode='valid')
        )
        #对预测时间步数据的嵌入编码操作
        self.step_embedding = nn.Embedding(horizon, sensor_num
)
        #对不同传感器的数据做融合（提取传感器数据间的关联特
征）
        self.spatial_mlp = nn.SequentialCell(
            nn.Dense(sensor_num, 128),
            nn.ReLU(),
            nn.Dense(128, 64),
            nn.ReLU(),
            nn.Dense(64, 32),
            nn.ReLU(),
            nn.Dense(32, sensor_num)
        )
        #对时间序列做卷积（提取时间点数据间的关联特征）
        self.tcn = nn.SequentialCell(
            nn.Conv2d(in_channels=1, out_channels=1,
kernel_size=(3,1), pad_mode='valid'),
            nn.Conv2d(in_channels=1, out_channels=1,
kernel_size=(3,1), pad_mode='valid'),
        )
        #通过一个卷积层得到最后的预测结果
        self.final_conv = nn.Conv2d(in_channels=1,
out_channels=1, kernel_size=(26, 1), pad_mode='valid') #使用
26*1卷积核，不补边

    def construct(self, x, iter_step): #construct方法
        h = self.spatial_mlp(x) #经过spatial_mlp空间处理后，得
到的数据h的形状：[batch_size, 30, 23]
```

```
        #输入数据x的形状：[batch_size, 30, 23]
        h = x + h #残差连接，将x和h对应元素相加，得到的数据h的形状：[batch_size, 30, 23]
        h = h.unsqueeze(1) #根据卷积操作需要，将3维数据升为4维数据：[batch_size, 1, 30, 23]
        h = self.tcn(h) #经过tcn时间卷积后，得到的数据x的形状：[batch_size, 1, 26, 23]
        y = self.final_conv(h) #通过26*1的卷积操作后，得到的数据y的形状：[batch_size, 1, 1, 23]
        y = y.squeeze(1) #将前面增加的维度去掉，得到的数据y的形状：[batch_size, 1, 23]
        #计算时间步数据的嵌入编码
        iter_step_tensor = mindspore.numpy.full((x.shape[0], 1), iter_step, dtype=mindspore.int32)
        step_embedding = self.step_embedding(iter_step_tensor) #step_embedding的形状：[batch_size,1,23]


        concat_op = mindspore.ops.Concat(axis=1)
        bias_input = concat_op((x, step_embedding,y)) #bias_input的形状：[batch_size,32,23]
        bias_input = bias_input.unsqueeze(1)
        bias_input = self.tcn(bias_input)


        bias_input = bias_input.squeeze(1)


        bias_output = self.bias_block(bias_input.unsqueeze(1)) #[batch_size, 1, 1, 23]
        bias_output = bias_output.squeeze(1) # [batch_size, 1, 23]
        #加上偏差块的预测结果
        y = y + bias_output #y的形状：[batch_size, 1, 23]
        return y #返回计算结果
```

```
class TCN_MLP_Without_Bias_Block(nn.Cell):
    def __init__(self):
        super().__init__()
        # 移除了 bias_block

        # 时间步嵌入
        self.step_embedding = nn.Embedding(horizon, sensor_num)

        # 空间MLP层
        self.spatial_mlp = nn.SequentialCell(
            nn.Dense(sensor_num, 128),
            nn.ReLU(),
            nn.Dense(128, 64),
            nn.ReLU(),
            nn.Dense(64, 32),
            nn.ReLU(),
            nn.Dense(32, sensor_num)
        )

        # 时序卷积层
        self.tcn = nn.SequentialCell(
            nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(3, 1), pad_mode='valid'),
            nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(3, 1), pad_mode='valid'),
        )

        # 最终卷积层
        self.final_conv = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(26, 1), pad_mode='valid')

    def construct(self, x, iter_step):
        h = self.spatial_mlp(x)
        h = x + h  # 残差连接
```

```
92          h = h.unsqueeze(1)  # [batch_size, 1, seq_len,
    sensor_num]
93          h = self.tcn(h)
94          y = self.final_conv(h)
95          y = y.squeeze(1)  # [batch_size, 1, sensor_num]
96
97          # 时间步嵌入编码
98          iter_step_tensor = mindspore.numpy.full((x.shape[0],
    1), iter_step, dtype=mindspore.int32)
99          step_embedding = self.step_embedding(iter_step_tensor)
100
101         # 在没有 bias_block 的情况下，直接将 y 与
    step_embedding 相加
102         y = y + step_embedding
103         return y
104 class TCN_MLP_Without_Step_Embedding(nn.Cell):
105     def __init__(self): #构造方法
106         super().__init__() #调用父类的构造方法
107         # 增加偏差块
108         self.bias_block = nn.SequentialCell(
109             nn.Dense(sensor_num, 64),
110             nn.ReLU(),
111             nn.Dense(64, 32),
112             nn.ReLU(),
113             nn.Dense(32, sensor_num),
114             nn.ReLU(),
115             nn.Conv2d(in_channels=1, out_channels=1,
    kernel_size=(28,1), pad_mode='valid')
116         )
117         # 移除 step_embedding 部分，不再对预测时间步进行嵌入编
    码
118         # self.step_embedding = nn.Embedding(horizon,
    sensor_num) # 已移除
119
120         # 对不同传感器的数据做融合（提取传感器数据间的关联特
```

```
征）
121         self.spatial_mlp = nn.SequentialCell(
122             nn.Dense(sensor_num, 128),
123             nn.ReLU(),
124             nn.Dense(128, 64),
125             nn.ReLU(),
126             nn.Dense(64, 32),
127             nn.ReLU(),
128             nn.Dense(32, sensor_num)
129         )
130         # 对时间序列做卷积（提取时间点数据间的关联特征）
131         self.tcn = nn.SequentialCell(
132             nn.Conv2d(in_channels=1, out_channels=1,
    kernel_size=(3,1), pad_mode='valid'),
133             nn.Conv2d(in_channels=1, out_channels=1,
    kernel_size=(3,1), pad_mode='valid'),
134         )
135         # 通过一个卷积层得到最后的预测结果
136         self.final_conv = nn.Conv2d(in_channels=1,
    out_channels=1, kernel_size=(26, 1), pad_mode='valid') #使用
    26*1卷积核，不补边
137
138     def construct(self, x, iter_step): #construct方法
139         # 输入数据x的形状：[batch_size, 30, 23]
140         h = self.spatial_mlp(x) #经过spatial_mlp空间处理后，h
    的形状与x相同：[batch_size, 30, 23]
141         h = x + h # 残差连接
142         # h形状仍为：[batch_size, 30, 23]
143
144         h = h.unsqueeze(1) # 升维为4D：[batch_size, 1, 30, 23]
145         h = self.tcn(h) # 经过tcn后，假设输出形状：[batch_size
    , 1, 26, 23]
146         y = self.final_conv(h) # 通过26*1的卷积后，y的形状：[
    batch_size, 1, 1, 23]
147
```

```
148         y = y.squeeze(1) # 去掉多余的维度，y形状：[batch_size, 1, 23]
149 
150         # 移除对时间步的嵌入编码操作
151         # iter_step_tensor = mindspore.numpy.full((x.shape[0], 1), iter_step, dtype=mindspore.int32)
152         # step_embedding = self.step_embedding(iter_step_tensor) # 已移除
153 
154         # 拼接时仅使用 x 与 y
155         # 原为 concat_op((x, step_embedding, y))，去掉 step_embedding 后为 concat_op((x, y))
156         concat_op = mindspore.ops.Concat(axis=1)
157         bias_input = concat_op((x, y)) # bias_input的形状：[batch_size, 31, 23] (原为32, 现在少了一步)
158 
159         bias_input = bias_input.unsqueeze(1) # [batch_size, 1, 31, 23]
160         bias_input = self.tcn(bias_input)     # 经过tcn后形状可能为：[batch_size, 1, reduced_time, 23]
161 
162         bias_input = bias_input.squeeze(1)    # [batch_size, reduced_time, 23]
163 
164         bias_output = self.bias_block(bias_input.unsqueeze(1)) # [batch_size, 1, 1, 23]
165         bias_output = bias_output.squeeze(1) # [batch_size, 1, 23]
166 
167         # 加上偏差块的预测结果
168         y = y + bias_output # [batch_size, 1, 23]
169 
170         return y # 返回计算结果
171 # 定义模型类
172 class TCN_MLP_Without_Spatial_MLP(nn.Cell):
```

```
 def __init__(self): # 构造方法
     super().__init__()
     # 偏差块与时间步嵌入保持不变
     self.bias_block = nn.SequentialCell(
         nn.Dense(sensor_num, 64),
         nn.ReLU(),
         nn.Dense(64, 32),
         nn.ReLU(),
         nn.Dense(32, sensor_num),
         nn.ReLU(),
         nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(28,1), pad_mode='valid')
     )
     # 对预测时间步数据的嵌入编码操作保持不变
     self.step_embedding = nn.Embedding(horizon, sensor_num)

     # 移除空间 MLP 层，不再对 x 进行传感器间关联特征提取
     # self.spatial_mlp = nn.SequentialCell(
     #     nn.Dense(sensor_num, 128),
     #     nn.ReLU(),
     #     nn.Dense(128, 64),
     #     nn.ReLU(),
     #     nn.Dense(64, 32),
     #     nn.ReLU(),
     #     nn.Dense(32, sensor_num)
     # )

     # 时间卷积网络保持不变
     self.tcn = nn.SequentialCell(
         nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(3,1), pad_mode='valid'),
         nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(3,1), pad_mode='valid'),
     )
```

```
204
205         # 最终卷积保持不变
206         self.final_conv = nn.Conv2d(in_channels=1,
    out_channels=1, kernel_size=(26, 1), pad_mode='valid')
207
208     def construct(self, x, iter_step):
209         # 不再调用 spatial_mlp(x)，直接使用 x
210         # 原逻辑: h = self.spatial_mlp(x)
211         # h = x + h
212         # 移除上述两行，直接使用 h = x
213         h = x
214
215         h = h.unsqueeze(1)  # [batch_size, 1, 30, 23]
216         h = self.tcn(h)      # [batch_size, 1, 26, 23]
217         y = self.final_conv(h) # [batch_size, 1, 1, 23]
218         y = y.squeeze(1)     # [batch_size, 1, 23]
219
220         # 计算时间步数据的嵌入编码
221         iter_step_tensor = mindspore.numpy.full((x.shape[0],
    1), iter_step, dtype=mindspore.int32)
222         step_embedding = self.step_embedding(iter_step_tensor)
     # [batch_size,1,23]
223
224         concat_op = mindspore.ops.Concat(axis=1)
225         # 拼接时仍使用 x, step_embedding, y
226         # 这里 x 的形状为 [batch_size, 30, 23]
227         # step_embedding 的形状 [batch_size, 1, 23]
228         # y 的形状 [batch_size, 1, 23]
229         # 拼接结果 [batch_size, 32, 23]
230         bias_input = concat_op((x, step_embedding, y))
231         bias_input = bias_input.unsqueeze(1) # [batch_size, 1,
     32, 23]
232         bias_input = self.tcn(bias_input)     # 经过tcn处理
233         bias_input = bias_input.squeeze(1)    # 回到 [
    batch_size, reduced_time, 23]
```

```
234
235         bias_output = self.bias_block(bias_input.unsqueeze(1))
    # [batch_size, 1, 1, 23]
236         bias_output = bias_output.squeeze(1) # [batch_size, 1,
    23]
237
238         y = y + bias_output # [batch_size, 1, 23]
239
240         return y
241 class TCN_MLP_Without_TCN(nn.Cell):
242
243     def __init__(self): #构造方法
244         super().__init__() #调用父类的构造方法
245         # 偏差块保持不变，但修改最后卷积核大小为(32,1)，因为不
    再有tcn降维
246         self.bias_block = nn.SequentialCell(
247             nn.Dense(sensor_num, 64),
248             nn.ReLU(),
249             nn.Dense(64, 32),
250             nn.ReLU(),
251             nn.Dense(32, sensor_num),
252             nn.ReLU(),
253             nn.Conv2d(in_channels=1, out_channels=1,
    kernel_size=(32,1), pad_mode='valid')
254         )
255         # 对预测时间步数据的嵌入编码操作保持不变
256         self.step_embedding = nn.Embedding(horizon, sensor_num
    )
257
258         # 保留空间 MLP 层（spatial_mlp）
259         self.spatial_mlp = nn.SequentialCell(
260             nn.Dense(sensor_num, 128),
261             nn.ReLU(),
262             nn.Dense(128, 64),
263             nn.ReLU(),
```

```
            nn.Dense(64, 32),
            nn.ReLU(),
            nn.Dense(32, sensor_num)
        )

        # 移除时序卷积层 tcn
        # self.tcn = nn.SequentialCell(
        #     nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(3,1), pad_mode='valid'),
        #     nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(3,1), pad_mode='valid'),
        # )

        # final_conv使用(30,1)卷积核代替原本的(26,1)，直接将30步压缩到1步
        self.final_conv = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(30, 1), pad_mode='valid')

    def construct(self, x, iter_step): #construct方法
        h = self.spatial_mlp(x) # 经过spatial_mlp提取空间特征 [batch_size,30,23]
        h = x + h  # 残差连接 [batch_size,30,23]

        h = h.unsqueeze(1)  # 升维为4D [batch_size,1,30,23]
        # 去掉对 h 的 tcn 调用，直接使用 h 进入 final_conv
        y = self.final_conv(h) # 使用30x1卷积核将30步降到1步 [batch_size,1,1,23]
        y = y.squeeze(1)       # [batch_size,1,23]

        # 计算时间步数据的嵌入编码
        iter_step_tensor = mindspore.numpy.full((x.shape[0], 1), iter_step, dtype=mindspore.int32)
        step_embedding = self.step_embedding(iter_step_tensor)  # [batch_size,1,23]

```

```
        concat_op = mindspore.ops.Concat(axis=1)
        bias_input = concat_op((x, step_embedding, y)) # [batch_size,32,23]
        bias_input = bias_input.unsqueeze(1)           # [batch_size,1,32,23]

        # 去掉对 bias_input 的 tcn 调用，直接进入 bias_block
        bias_output = self.bias_block(bias_input)      # 卷积核(32,1)，输出 [batch_size,1,1,23]
        bias_output = bias_output.squeeze(1)           # [batch_size,1,23]

        # 加上偏差块的预测结果
        y = y + bias_output # [batch_size,1,23]

        return y
class TCN_MLP_Without_Final_Conv(nn.Cell): #定义TCN_MLP_with_Bias_Block类
    def __init__(self): #构造方法
        super().__init__() #调用父类的构造方法
        self.bias_block = nn.SequentialCell(
            nn.Dense(sensor_num, 64),
            nn.ReLU(),
            nn.Dense(64, 32),
            nn.ReLU(),
            nn.Dense(32, sensor_num),
            nn.ReLU(),
            nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(28,1), pad_mode='valid')
        )
        self.step_embedding = nn.Embedding(horizon, sensor_num)

        self.spatial_mlp = nn.SequentialCell(
            nn.Dense(sensor_num, 128),
```

```
            nn.ReLU(),
            nn.Dense(128, 64),
            nn.ReLU(),
            nn.Dense(64, 32),
            nn.ReLU(),
            nn.Dense(32, sensor_num)
        )

        self.tcn = nn.SequentialCell(
            nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(3,1), pad_mode='valid'),
            nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(3,1), pad_mode='valid'),
        )

        # 移除 final_conv，不再使用最终卷积层
        # self.final_conv = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(26, 1), pad_mode='valid')

    def construct(self, x, iter_step):
        # 经过空间MLP和残差连接
        h = self.spatial_mlp(x) # [batch_size,30,23]
        h = x + h               # [batch_size,30,23]

        # 升维后通过tcn提取时间特征
        h = h.unsqueeze(1)      # [batch_size,1,30,23]
        h = self.tcn(h)         # [batch_size,1,26,23]

        # 不再通过final_conv进行降维，这里从最后一个时间步提取特征
        # h[:, :, -1, :]会选取最后一个时间步的特征，形状为[batch_size,1,23]
        y = h[:, :, -1, :]  # [batch_size,1,23]

        # 计算时间步的嵌入编码
```

```
349         iter_step_tensor = mindspore.numpy.full((x.shape[0],
    1), iter_step, dtype=mindspore.int32)
350         step_embedding = self.step_embedding(iter_step_tensor)
     # [batch_size,1,23]
351 
352         concat_op = mindspore.ops.Concat(axis=1)
353         # 拼接 x, step_embedding, y
354         # x:[batch_size,30,23], step_embedding:[batch_size
    ,1,23], y:[batch_size,1,23]
355         # 拼接后为[batch_size,32,23]
356         bias_input = concat_op((x, step_embedding, y))
357         bias_input = bias_input.unsqueeze(1) # [batch_size
    ,1,32,23]
358 
359         # 通过tcn后再进入bias_block
360         bias_input = self.tcn(bias_input)     # [batch_size
    ,1,26,23]
361         bias_input = bias_input.squeeze(1)    # [batch_size
    ,26,23]
362 
363         bias_output = self.bias_block(bias_input.unsqueeze(1))
     # [batch_size,1,1,23]
364         bias_output = bias_output.squeeze(1) # [batch_size
    ,1,23]
365 
366         # 最终输出 y + bias_output
367         y = y + bias_output # [batch_size,1,23]
368         return y
369 class TCN_MLP_Basic(nn.Cell): #定义TCN_MLP_with_Bias_Block类
370     def __init__(self):
371         super().__init__()
372         sensor_num = 23   # 假设原先定义在外部
373         horizon = 5       # 假设原先定义在外部
374 
375         # 原先的bias_block组件被移除，不再定义bias_block
```

```
376         # self.bias_block = nn.SequentialCell(...)
377 
378         # 移除step_embedding，不再对时间步进行嵌入编码
379         # self.step_embedding = nn.Embedding(horizon,
    sensor_num)
380 
381         # 空间MLP保留
382         self.spatial_mlp = nn.SequentialCell(
383             nn.Dense(sensor_num, 128),
384             nn.ReLU(),
385             nn.Dense(128, 64),
386             nn.ReLU(),
387             nn.Dense(64, 32),
388             nn.ReLU(),
389             nn.Dense(32, sensor_num)
390         )
391         # 时间卷积层保留
392         self.tcn = nn.SequentialCell(
393             nn.Conv2d(in_channels=1, out_channels=1,
    kernel_size=(3,1), pad_mode='valid'),
394             nn.Conv2d(in_channels=1, out_channels=1,
    kernel_size=(3,1), pad_mode='valid'),
395         )
396         # 最终卷积层保留
397         self.final_conv = nn.Conv2d(in_channels=1,
    out_channels=1, kernel_size=(26, 1), pad_mode='valid')
398 
399     def construct(self, x, iter_step):
400         # 空间特征提取并残差连接
401         h = self.spatial_mlp(x) # [batch_size,30,23]
402         h = x + h               # [batch_size,30,23]
403 
404         # 升维后进行时序卷积特征提取
405         h = h.unsqueeze(1)      # [batch_size,1,30,23]
406         h = self.tcn(h)         # [batch_size,1,26,23]
```

```
407
408         # 使用final_conv得到最终预测结果
409         y = self.final_conv(h)  # [batch_size,1,1,23]
410         y = y.squeeze(1)        # [batch_size,1,23]
411
412         # 移除 step_embedding，不计算iter_step_tensor和
    step_embedding
413         # 移除 bias_block，不进行拼接 x、step_embedding、y ，
    不进行第二次 tcn 和不经过bias_block修正
414
415         # 因为已经没有bias_input和bias_block的修正过程，此时 y
     就是最终输出
416         return y
417
418     def construct(self, x):  # construct 方法
419         h = self.spatial_mlp(x)  # 经过 spatial_mlp 处理，得到
     h，形状：[batch_size, 30, sensor_num]
420         h = x + h  # 残差连接，形状保持不变
421         h = h.unsqueeze(1)  # 扩展维度，形状变为：[batch_size,
     1, 30, sensor_num]
422         h = self.tcn(h)  # 经过时序卷积层，形状：[batch_size,
    1, 26, sensor_num]
423         y = self.final_conv(h)  # 最终卷积层，形状：[
    batch_size, 1, 1, sensor_num]
424         y = y.squeeze(1)  # 去掉第一个维度，形状：[batch_size,
     1, sensor_num]
425         return y  # 返回预测结果 y，形状：[batch_size, 1,
    sensor_num]
```