SparseTSF 代码分析与复现

高景珩　　丛方昊　　章昊

Abstract

通过对 *SparseTSF* 模型源代码整体架构、各个模块、关键算法和其他模型相关代码的细致分析，我们成功搭建了代码的运行环境，对给定的数据集进行了训练和测试，得出了论文中相对应实验的参数、误差，并绘制了训练损失曲线。进一步，我们也根据代码对 *SparseTSF* 与其他模型进行了比较。另外，我们依据生成的误差图像和训练损失图像评估了 *SparseTSF* 模型的优势与劣势，提出了一些改进建议。我们还尝试将代码迁移到其他数据集，编写了对应的模块和脚本，得出了预期结果，我们在迁移过程中积累了深度学习模型的使用经验，对模型的代码产生了更深的理解。

小组分工：高景珩：整体架构分析，环境搭建，复现实验，迁移模型；从方昊：整体架构分析，各个模块代码分析，环境搭建，改进代码；章昊：处理模型生成数据，分析模块优缺点，迁移模型，整合实验报告。

1. 对代码整体架构的分析

完成人：丛方昊，高景珩

如图1，我们对模型的整体架构进行了细致的分析，总结到了流程图当中，其中包含各个模块的功能，以及控制权之间的转移方向。

2. 模块代码分析

完成人：丛方昊，章昊

2.1. 代码实现逻辑与关键算法

1）Shell 脚本是文件的入口，进行对文件根路径、数据路径、预测序列长度等内容的设置操作。通过 do - done 中的命令块，反复执行 run_longExp.py 文件，将 shell 文件中设置的模型构建数据传入 run_longExp.py 文件并将控制权流转给 run_longExp.py。

2）Run_longExp.py 文件中配置有主解析器及模型，为 parser 添加了大量命令行参数，并将所有参数传入到 args 中等待下一步进行的操作。在正式搭建模型之前，设置种子，检查 gpu 和 cuda 的配置，并调用 exp_main.py，初始化实验的主类 Exp。之后，run_longExp 还进行了对当前模式的判断，为训练模式时，设置训练的参数并调用 train()、test()、predict() 等开始训练（多次），否则只执行一次。自调用 train 函数开始，run_longExp.py 函数将控制权流转给 exp_main.py。

3）Exp_main.py 是主操作部分，调用了多个模块中的方法。Exp_Main 类继承于 exp_basic.py 中的 Exp_Basic 类，这个父类只为子类提供了配置/获取设备、模型迁移等十分基础的工作，起到操作的占位作用，并无其他关键功能。此外，Exp_Main 类提供了选择优化器、选择损失函数等小规模待调用函数，在使用时再进行说明，不在此单独指出。

4）首先收到控制权的是 train 函数，它首先调用了本类的 _get_data 函数，用来为各个迭代器配置数据，这些迭代器相关内容是存储在 data_loader.py 中的，而 _get_data 函数获得返回值需要用到与 loader 同文件夹中的 data_factory.py 文件中的 data_provider 函数，控制权交予 data_factory.py。

5）Data_factory.py 获得控制权后，首先要找 data_loader.py 获得数据集，值得一提的是，这里的数据集的地位与 shell 脚本文件是相当的，都是由外部输入直接控制的。loader 为各数据创建了一个类，人工搭配了相应的参数集，并提供了获取数据划分、特征选择、逆标准化等多个方法。在数据集的创建之前，首先调用了 ___read_data___ 函数，控制权交给

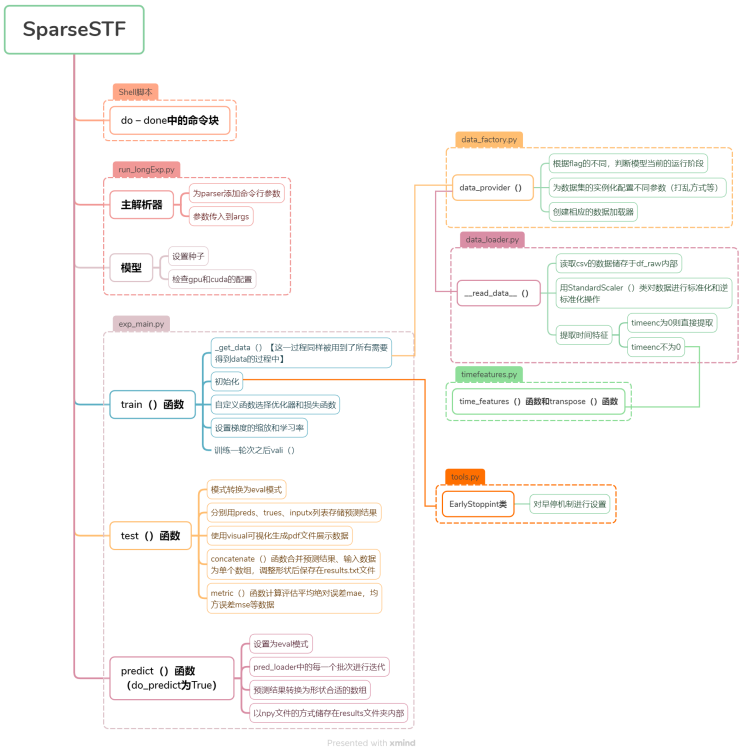

图 1. 整体架构分析图

data_loader.py。

6）___read_data___ 中，读取 csv 的数据储存于 df_raw 内部，使用了 sklearn 中（值得注意，并不是 tools 中实现的）设置的 StandardScaler 类，进行对数据的标准化和逆标准化操作，而若 sacle 为 False，则直接使用原始数据进行操作；随后根据 timeenc 的值，若为 0 则直接提取时间特征，反之使用 timefeatures.py 中的 time_features 函数和 transpose 函数生成时间编码特征，并储存起来，至此，数据集的搭建工作完成，控制权回交给 data_factory.py 中的 data_provider 函数。

7）Provider() 首先需要根据 flag 的不同，判断模型当前的运行阶段，而后为数据集的实例化配置不同参数（打乱方式等），使其可以根据需求自适应工作，创建相应的数据加载器，把这些传递给上一级——train 函数，配置模型存储路径，至此，数据集和加载器的工作顺利完成。

8）随后，train() 需要使用已有包中的函数对一些训练设置进行初始化，包括训练时间、训练步，这两点可以方便查询模型训练的进程；使用 tools.py 中的 EarlyStoppint 类对早停机制进行设置；使用 Exp_Main 自定义函数选择优化器和损失函数；以及设置梯度的缩放和学习率，随即正式进入训练过程。

9）训练轮次由 train_epochs 决定，训练次数存储于 iter_count 中，批次损失记录于 loss 中。随后，值得注意的是，self.model.train 函数是 torch 中的将模式设置为训练模式的函数，而不是对自身的调用，处理好每个 batch 数据。前向中，如果启用了混合精度训练，则使用 torch.cuda.amp.autocast() 加速计算，否则直接输出 outputs，并根据 criterion 计算损失；随即进行反向传播，根据损失计算梯度，并提供时间戳等信息，根据训练结果调整学习率。这一部分集中调用的是 torch 中的函数，并未将控制权交予其他模块。

10）在每个 epoch 的最后阶段，记录每个 epoch 的平均损失，随之调用 vali 函数进行评估，这是每个 epoch 内部的自验证，若在指定数量的 epoch 内验证损失没有显著下降，则触发早停，提前结束，并保存最优模型。反之，一直训练到规定的 epoch 结束，得到最优

模型，train() 的工作顺利完成。

11）在 train() 调用 vali() 时，控制权交予 vali()，vali() 的需要进行的基础操作不超出 train() 的操作之外，因此这里只进行简单描述，vali() 遍历验证集后，迁移到设备上，构建解码器输入进行前向传播操作。随即，对 outputs 和 batch_y 进行删减，根据 feature 获得所需要的特征，并在尽可能减少梯度运算的情况下计算损失、恢复训练模式、并最终获得总的平均损失。

12）Train() 任务执行完毕后，控制权返回 run_longExo.py，再到下一步流转给 exp_main.py 的 test 函数，test() 禁用梯度计算，相同地进行导入数据、配置容器、构建解码器等操作，并将模式转换为 eval 模式，这里不再赘述。因为已经训练完成，因此在前向传播过程中不再计算损失，除此之外，执行操作相同。将得到的输入裁剪过后，分别用 preds、trues、inputx 列表存储预测结果，而后每处理 20 个批次，使用 visual 可视化生成 pdf 文件展示数据。随后，调用 concatenate 函数将所有批次的预测结果、输入数据合并为单个数组，调整形状后保存在 results.txt 文件内。最后，使用 metric 函数计算评估平均绝对误差 mae，均方误差 mse 等数据，完成最后的分析。至此，训练部分结束。

13）控制权再次回归到 run_longExp.py 手中，若 do_predict 为 True，最后一步执行预测程序，否则清除缓存后结束程序。若执行 exp_main.py 中的 predict 函数——predict 使用 _get_data() 获取预测数据和对应的数据加载器，读取训练过程中存储于.pth 文件中的模型，获得参数加载到当前模型中，设置为 eval 模式，对 pred_loader 中的每一个批次进行迭代，进行前向传播后将预测结果转换为形状合适的数组，最终以 npy 文件的方式储存在 results 文件夹内部。模型预测完成。

2.2. 各个模块功能分析

1. *Checkpoints* 保存.pth 文件做为检查点，在程序运行过程中生成；主要用于保存和加载模型中的各个参数，避免数据丢失和重复训练，方便进行迁移学习。

2. *Data_provider* 包含两个 py 文件，其中 data_factory 为模型提供构建所适配的数据集和数据加载器；data_loader 负责读取 csv 文件，将其内部的数据标准化处理，并提供了逆标准化方法，还承担着数据分割、查找时间信息等操作。

3. *Dataset* 以.csv 文件方式存储数据集。

4. *Exp* 定义了一个父类 Exp_Basic 作为框架，可以初始化设备、迁移模型；子类 Exp_Main 是整个程序的主控区，它对父类的一些占位方法实现出来，train、vali、test 和 pred 四个模型构建从训练到预测的核心步骤均在这里实现。

5. *Models* SparseTSF.py 整合了归一化、1D 卷积、线性层、上下采样相关模块，作为一个完整的模型。

6. *Results* 存储预测结果

7. *Scripts* 存放 shell 脚本文件，这里是程序运行的起点，用来导入数据集，规定 w、频率等内容，将信息传递给 run_longExp 开始构建模型。

8. *Test_results* 存储测试结果

9. *Utils* 主要有 3 个模块在 SparseTSF 模型中使用：

   - Metrics 定义了相关系数、均方误差等一系列模型评估标准，用于评估模型的表现。
   - Timefeatures 生成时间特征，让给定的时间频率标准化（秒数、分钟数等），将字符串转换为 offset 类型，以便于读取过程中的进一步操作。在 data_loader.py 中被调用。
   - Tools 定义了几个工具——调整学习率（衰减倍数），早停机制（防止过拟合）、字典访问方式的补充、标准化和逆标准化、绘图、计算模型复杂度等方法。在 exp_main.py 中被调用。

3. 环境的搭建与运行结果

完成人：高景珩，丛方昊，章昊

3.1. 环境搭建

我们根据 GitHub 上作者的 Guidelines，在 anaconda 上创建了 SparseTSF 虚拟环境，并由 requirements.txt 安装了相应配置文件，完成了环境的配置。

由于小组成员都使用 Windows 系统，运行模型时我们也对代码进行了一定修改。

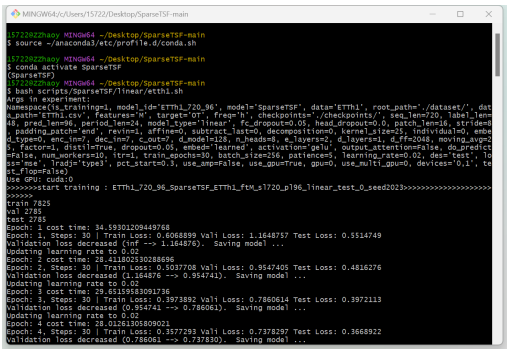

图 2. 运行开始截图

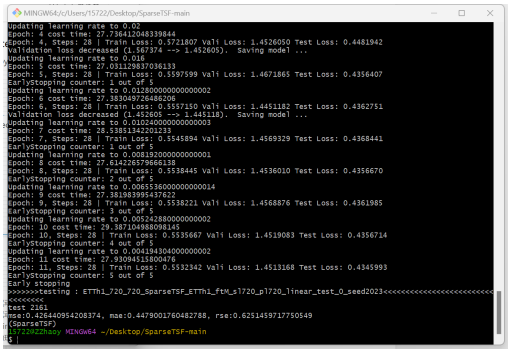

图 3. 运行结束截图

1. 修改 run_longExp，将主程序部分定义在“__main__”中，以适合 Windows 系统运行。
2. .sh 文件是 Linux 系统 Shell 脚本的命令，我们利用 GitBash 的 bash 操作，使其能够在 Windows 环境下运行。
3. 部分组员电脑没有 GPU，我们修改了相应配置（如删去 cuda 的相关需求），使模型能够使用 CPU 运行。

Listing 1. run_longExp.py 中的修改

```
1  self.model.load_state_dict(torch.load(
       best_model_path, map_location=torch.device(
       'cpu')))
2  # 将这里的device改成cpu，让模型使用CPU运行
```

3.2. 运行结果展示

通过 run_all.sh 命令，我们成功对所有给定的数据集进行了处理，部分结果（ETTh2 数据集结果图4，Electricity 数据集结果图5）如图。

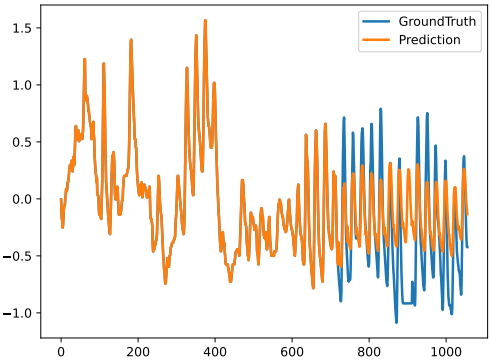

图 4. ETTh2 数据集 720-336 的预测结果

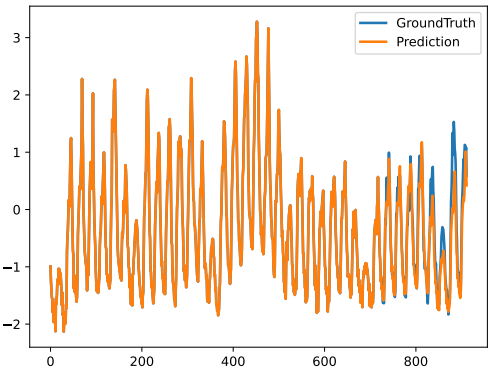

图 5. Electricity 数据集 720-192 的预测结果

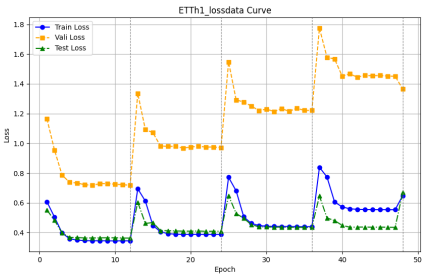

图 6. 四种预测长度下 ETTh1 数据集的损失曲线

3.3. 损失曲线绘制

对于模型运行中的损失，除了 checkpoints 中的.pth 二进制文件，并没有具体显示的文件输出。于是我们编写了 Python 脚本，在控制台的输出信息中提取出每个 Epoch 的 Train_loss 等信息，转换成 csv 数据，然后利用 Python 的 matplotlib 库绘制损失曲线，图中集中绘制了四个预测长度对应的数据。部分结果（ETTh1 损失曲线图6，weather 损失曲线图7）如图。

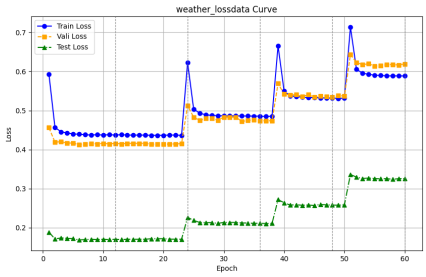

图 7. 四种预测长度下 weather 数据集的损失曲线

曲线分为四段，代表四个不同的预测序列长度。通过对比，四段损失曲线的相对关系正确。观察损失曲线发现，Vali_loss 的值回升后立即下降，体现了模型的自调整功能。对于设定的最大 30Epoch，所有预测均提前结束，并且最终的损失均比较低，说明拟合效果好，模型确实提取了数据的内在趋势，形成了优秀的预测效果。

4. 模块优缺点分析与建议

完成人：章昊，丛方昊，高景珩

4.1. 优点

1. **模块化设计：**模型的各个功能被分成不同的模块，如数据加载（data_loader.py）、时间特征处理（timefeatures.py）、早停机制（tools.py）等。这种分层设计便于维护、调试和扩展，调试方便，容易扩展。

2. **主程序分离：**主解析器（run_longExp.py）、主程序（exp_main.py）和模型配置（模型）分离，使得不同的任务（如参数解析、训练、测试、预测）各自独立，降低了耦合性，提高了代码的复用性。

3. **数据处理流程：**data_factory.py 和 data_provider() 函数提供了动态的数据构建方式，可以根据标志位进行数据处理，增加了模型适应不同数据源的灵活性。

4. **支持早停机制：**通过 tools.py 中的 EarlyStopping() 函数引入早停机制，模型可以捕获 Vali_loss 的变化做出调整，避免模型过拟合，也避免了计算资源的浪费。

5. **滑动聚合：**为了捕获一个周期内数据的内在联系，作者利用一维卷积，在 SparseTSF.py 中将卷积核 conv1d 长度设为 $1 + \lfloor (w/2) \rfloor$，对一个周期的数据聚合，利用数据的内在联系，提高训练效果。

6. **残差的利用：**恰当的借用了残差的思想，很多情况下，面对跨度很大的数据集，并不是预测效果的问题，而是现有的物理设备能不能预测的问题，残差的思想减少每一层之间的连接点甚至跨层连接。而周期性数据天然具备良好的相关性的性质，本模块敏锐地抓住了这一点。

7. **下采样与上采样的利用：**在 SparseTSF.py 中，作者定义了 forward 函数，其中对输入进行下采样和上采样，将周期趋势解耦，并且使用同一组参数训练，在保证准确性的同时，有效减少了参数的使用量，大幅度减少计算成本、提高运行效率。

4.2. 缺点及建议

1. **新数据集的支持：**更换数据集尝试时，boarder 等参数需要重新选取，让泛化性受到挑战；而且，面对一个全新的数据集时，超参数 period_len 的选取需要符合数据集性质，常常很难保证模型的效果最佳。建议让模型能够自主学习数据的周期属性，在提高模型准确性的同时提高模型的泛化性。

2. **周期选择问题：**面对长序列预测时，w 选取较小（2,4）时效果更好，但是这并不符合自然周期，对这个模型的可信力而言，是一个重大挑战。

3. **模型的入口：**模块中只提供了一种 shell 脚本文件的入口，当运行者的设备内存不足以支撑单批次运行结束时，往往需要分批次运行。目前的代码结构支持上一次运行数据保存至.pth 文件中供使用，但是并不支持两次运行结果的合并或在已有的基础上再进行重新训练。建议可以添加调用已有.pth 文件开始训练的脚本，方便模型的复位调试。

4. **预测结果展示：**模型的结果都是依据数据集得出的，并没有对模型使用者未知数据的预测功能。建议在模型拟合完毕后，继续向后生成完全预测数据和图像，以供模型使用者参考。

5. **全线性层的局限性**：模型中，作者只利用了线性层对数据进行预测，对于某些特定数据集可能难以把握特征与目标间的函数关系。

5. 与其他模型的对比

完成人：丛方昊，章昊

5.1. Autoformer

实现原理分析

1）AutoCorrelation.py 实现了自相关机制，用来代替自注意力机制，主要用来分析数据的周期性关系并从中提取出来，这也是本论文优化代码的核心思想。AutoCorrelation 类通过 period-based dependencies discovery 和 time delay aggregation 来获得序列中的依赖关系。其中，对于时延聚合操作，它给出了训练阶段、推理阶段和标准版本的操作。前两个专门针对各个过程的特点进行了一些效率上的优化，在结果上没有差别。

此外，在这里对其实现的 forward 方法进行初步说明，后面的类基本都实现了前向传播方法，不再单独指出。Forward 接收到的是注意力机制中标准的查询、键、值，以及一个用来去除 padding 填充等部分以减少计算的 attn_mask。首先，函数对查询和键执行快速傅里叶变换，得到周期性依赖关系；而后对 queries 和 keys 进行乘法操作，得到自相关的权重；最后，根据这个权重进行时延聚合操作，最终获得加权后的 values。

AutoCorrelationLayer 将这个方法推广到多头注意力的情况，使这个方法更加完备。

2）Autoformer_EncDec.py 这个文件实现了 Encoder 和 Decoder 层，它应用自相关机制对序列进行分解为趋势和周期两个部分，从而以较少的消耗获得进行预测所需要的是趋势的部分。

其中 series_decomp 以及后面的 multi（根据卷积核的区别进行加权）负责分解这个序列，用 moving_avg 这个移动平均层 padding 后获得趋势部分，将周期性部分交给 my_Layernorm 进行处理。

EncoderLayer 就是结合上述部分的一层编码器，在上面的分解提取的基础上还带有 drowout 操作，以防止过拟合。Encoder 类则是将多个编码器层堆叠在一起，加入卷积层处理后根据需要得到相应的注意力权重。相应的，有 DecoderLayer 解码器层，在编码器层的基础上有结合了交叉注意力，以及堆叠起来的 Decoder，在此不再赘述。

Model	Parameters	MACs	Max Mem.(MB)	Epoch Time(s)
Informer (2021)	12.53 M	3.97 G	969.7	70.1
Autoformer (2021)	12.22 M	4.41 G	2631.2	107.7
FEDformer (2022b)	17.98 M	4.41 G	1102.5	238.7
FiLM (2022a)	12.22 M	4.41 G	1773.9	78.3
PatchTST (2023)	6.31 M	11.21 G	10882.3	290.3
DLinear (2023)	485.3 K	156.0 M	123.8	25.4
FITS (2024)	10.5 K	79.9 M	496.7	35.0
SparseTSF (Ours)	**0.92 K**	**12.71 M**	125.2	31.3

图 8. 各模型对比

与 SparseTSF 对比

Autoformer 基于 Transformer 架构，采用了“自相关机制”代替自注意力，以减少计算复杂度并提高在长序列上的效率。它主要关注对序列中的周期性和趋势性变化进行解耦，适合更复杂的时间序列建模。而 SparseTSF 采用极简的线性层和稀疏预测策略，核心是“交叉周期稀疏预测”技术，通过降采样来分离周期性和趋势特征，从而减少模型参数至 1k 以下。其设计注重模型轻量化和长周期的捕获。

根据实验数据，对于 Electricity 数据集的拟合，Autoformer 使用了 12.22M 变量，平均每个 Epoch 花费 107s。而 SparseTSF 只使用了 0.92K 变量，平均每 Epoch 花费 31.3s。

因此，对比而言，Autoformer 更适合需要强大建模能力和多维度时序数据的应用，比如金融数据或更复杂的物联网数据场景。SparseTSF 适合资源受限的环境或小样本数据，尤其是周期性明显的数据（如电力或交通流量）。

5.2. PatchTST

实现原理分析

1）PatchTST_backbone.py 基于 Transformer 架构，实现了 PatchTST_backbone 的神经网络模型，它结合了时序数据的分段机制、位置编码和残差机制，重点用于处理预测任务。

Flatten_Head 负责编码器的输出转换为所需的预期形式（这里提供了共享头部或是独立头部的两种方式），据此，转换过后，这个 py 文件提供了两种不同的处理方式：其一是使用 TSTiEncoder 编码器，它采用通道独立的设计，每个变量会独立地经过编码。其二是由多层 TSTEncoderLayer 类构成的 TSTEncoder 编码器，它使用多头注意力机制（MultiheadAttention），将输

入通过多个编码器层进行处理，和普通的 transformer 相仿。并且可以选择进行残差连接，来进一步减少计算量。另外，本文件中还有一个相对独立的 _SScaleDotProductAttention，它能将注意力进行缩放，并提供注意力查询的操作。

2）PatchTST_layers.py 此文件中提供一些运算操作，包括转置、获得激活函数、移动平均、分解序列获得趋势和残差。

以上操作比较简单，在此单独介绍 Positional Encoding。它和 transformer 一样，使用正余弦函数计算位置编码，对位置编码进行正则化，后面由数据的位置编码和它的维度得到二维的位置编码，帮助模型获得序列中的顺序关系。

与 SparseTSF 对比

PatchTST 基于 Transformer 架构，使用“patch embedding”技术，将时间序列数据切片为小块并处理，能有效捕获局部模式和长程依赖关系。PatchTST 相当于在时序数据中引入了类似图像处理中“patch”的概念，以提升建模效果。而 SparseTSF 通过周期性降采样和稀疏预测实现参数压缩，将预测任务转化为不同周期的趋势预测，模型整体较轻量。

根据实验数据，对于 Electricity 数据集的拟合，PatchTST 使用了 6.31M 变量，虽然少于 Autoformer，但还是远远多于 SparseTSF 的 0.92K。另外，相比 Autoformer，PatchTST 降低了参数量，但每轮平均花费 290s，在本次实验中不及 SparseTSF 模型。

因此，分析得出，PatchTST 适合有复杂模式或多尺度特征的长序列预测任务，尤其在高维和异构数据上表现良好。SparseTSF 更适用于周期性明显且计算资源有限的任务。

6. 代码迁移的尝试

完成人：高景珩，章昊，丛方昊

我们尝试将模型迁移到 *exchange_rate* 数据集上。主要经历了代码改写，数据分析，训练集、验证集、测试集划分，相应参数设置等过程。

6.1. 实现方法

主要内容仿照源代码的格式内容，为新的数据集编写加载数据的类以及运行脚本，目的是为新的数据集增加参数描述以及训练划分，具体如下：

Listing 2. DataLoader.py 中补充的类

```
class Dataset_exchange_rate(Dataset):
    def __init__(self, root_path, flag='train',
        size=None,
                 features='S', data_path='
                     exchange_rate.csv',
                 target='OT', scale=True, timeenc
                     =0, freq='d'):
                 # ...

    def __read_data__(self):
        # ...
        border1s = [0, 15 * 4 * 90 - self.seq_len
            , 15 * 4 * 90 + 2 * 4 * 90 - self.
            seq_len]
        border2s = [15 * 4 * 90, 15 * 4 * 90 + 2
            * 4* 90, 15 * 4 * 90 + 4 * 4 * 90]
        # ...
```

参数含义：因为 exchange_rate.csv 为一个一天一次记录，长达近 20 年的汇率变换数据，我们将 15 年、2 年、2 年的数据划分为训练集、验证集、测试集，在 borders 参数中对应改写。其中，一年 4 个季度，一季度（91 天左右）可看作一个最短周期。

Listing 3. 新增 exchange_rate.sh 脚本文件

```
……
seq_len=600
for pred_len in 60 150 300 600
……
python -u run_longExp.py \
--period_len 30 \
--enc_in 8 \
……
--itr 1 --batch_size 128 --learning_rate 0.02
……
```

参数含义：因为 exchange_rate.csv 有 8 个输入特征，我们将 enc_in 设置为 8；根据数据性质，推测具有 30 天的周期，period_len 设置为 30；seq_len 和 pred_len 两个序列长度相应设置为 period_len 的整数倍。

6.2. 迁移结果

对于 exchange_rate 数据集，SparseTSF 迁移的效果比较优秀。每一次运行均在设置的限制 30Epoch 内完成，且 Train_loss,Vali_loss 和 Test_loss 三项关键数据都较低，预测结果较好。

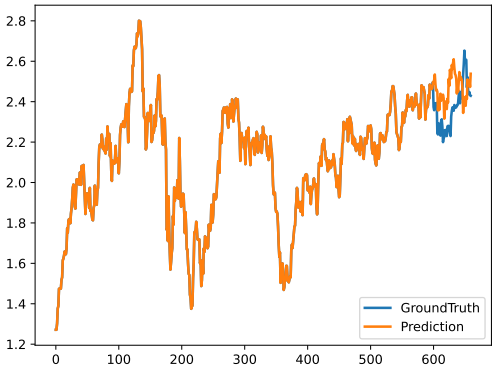

图 9. 迁移：pred_len 为 60 的结果

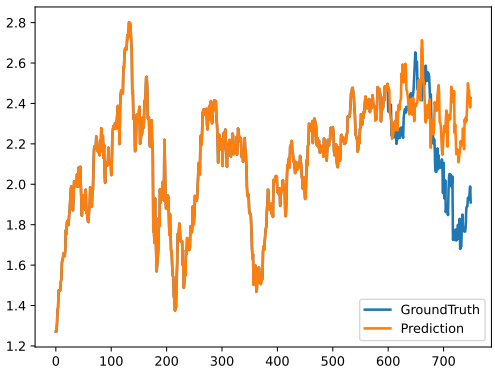

图 10. 迁移：pred_len 为 150 的结果

以下是两个预测结果图 (图9，图10) 与训练损失曲线（四次训练合为一张图11）。

在结果图中可以看出，模型对数据趋势的把握很准确，虽然具体数值有一定偏差，但误差值较小，预测效果总体令人满意。

在损失曲线中，可以看出，各类损失随着训练轮次的增加都有所下降，并且模型把握到了“过拟合”的趋势，在 Vali_loss 上升后对模型进行了正确调整。

以上结果说明模型具有良好的泛化能力，并且具有参数少、运行快的优势。但模型的效果需要建立在数据集具有周期性的前提下，这也是 SparseTSF 的局限性。

7. 分析代码遇到的问题与解决方式

完成人：章昊，丛方昊，高景珩

7.1. 不熟悉 **PyTorch** 的基本使用

解决过程：

- **学习 PyTorch 基础教程：**学习 torch.tensor、torch.optim 优化器、torch.autograd 自动微分、loss.backward() 反向传播等基础操作。
- **参考示例代码：**在 PyTorch 官方文档或开源代码库中查找类似的训练代码示例，分析结构和逻辑。
- **注释分析：**在阅读代码时，为关键步骤添加注释，标注汉语解释，并分解理解每一步的作用。例如，将前向传播、损失计算、反向传播和参数更新分别标记出来。

7.2. 各代码间的相互关系混乱

解决过程：

- **绘制流程图：**在总览代码各个部分后，我们绘制了各个.py 文件的关系（即图1），以及各个重要的类、函数之间的调用关系，绘制成流程图，便于理解。
- **跟踪调试：**在代码中插入 print() ，逐步跟踪数据从原始输入到模型输入的转换过程，观察数据的形状和内容变化，帮助理解每一步的处理作用。例如，我们在 data_loader 中 print 出 borders 数据，以理解模型对数据集的划分方式与调用位置。
- **模块化理解代码：**我们逐块分析代码，如从 data_loader 到 data_factory，理解数据集在 loader 中的处理方式，以及在 factory 中的传递过程。

7.3. 对 **EarlyStopping** 机制的理解

解决过程：

- **理解定义和作用：**通过查阅资料，我们了解到早停机制是一种防止模型过拟合的策略。它监测验证集的损失（或准确度）变化情况，当验证损失在若干个训练轮次内不再下降时，提前停止训练。这样可以避免模型过度拟合训练数据，提高在测试数据上的泛化能力。
- **理解早停的代码：**查阅文档，学习到 patience 参数可以设置一个“耐心值”，即当验证损失连续 patience 个轮次未改善时，触发早停机制。
- **理解模型输出：**模型在运行中会不断输出 EarlyStopping 的相关参数，通过观察发现，模型

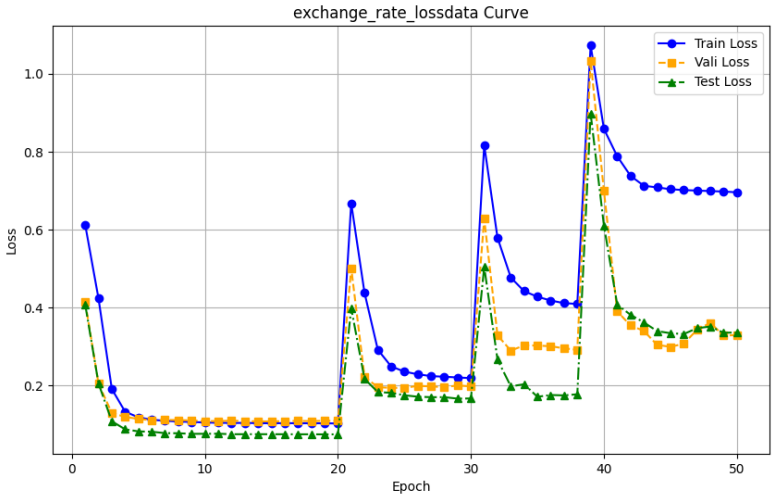

图 11. 四种预测长度下迁移训练的损失曲线

确实捕获了 Vali_loss 的上升趋势，并对早停实现了合理控制，能正确修正模型。

8. 遗留的问题

对于输出到 test_result 的 pdf 图片中，我们还不理解 GroundTruth 和 Prediction 的关系图是如何画出的，例如图5，0-719 拟合效果好，720-911 拟合效果差。可以看出应该是利用前 720 个数据点对后 192 个数据进行了预测。但是测试集的原理是利用已生成的模型对新数据进行预测，并不会进行重新拟合，这使我们很疑惑。

另外，run_longExp 中使用了随机种子 torch.manual_seed，但在模型中并未发现具体的用途。