研究性学习三：基于 MindSpore 国产 AI 框架的人工智能开源代码迁移

陈兴浩、王为、王亦辉

目录

A. 准备工作（王亦辉）

首先通过华为云创建一个带有 MindSpore 框架的 Notebook 实例，并启用 SSH 远程开发。然后，生成密钥并将私钥保存到本地 C 盘当前用户的目录下。在 Notebook 环境中，点击打开 VS Code，安装 ModelArts 插件并连接到云端服务器。接着，使用将论文的 GitHub 仓库克隆到服务器上，并将数据集的压缩包从本地上传至服务器，使用 unzip 命令解压到指定的文件夹中。这样我们就有了用来迁移代码的环境以及所需要的资源。接着需要熟悉 Mindspore 框架，通过教程和文档了解 PyTorch 中的接口在 MindSpore 中的对应关系，以及是否有差异，有哪些差异，如可以查阅 MindSpore 文档中的 PyTorch 与 MindSpore API 映射表。

B. 代码迁移过程（王为）

B.1. 引言

在深度学习的快速发展中，时序预测模型在许多应用领域（如金融预测、气象预报、设备故障检测等）扮演着重要角色。随着不同深度学习框架的不断演进，开发者面临着如何有效地迁移和优化现有模型的挑战。特别是将时序预测模型的代码从 PyTorch 迁移到 MindSpore，不仅可以利用 MindSpore 在特定硬件上的优化性能，还能借助其动静态图的灵活性，提升模型的训练和推理效率。

本章节将详细探讨这一迁移过程的重要性，首先分析 PyTorch 和 MindSpore 的核心特性和设计理念，接着通过可视化工具展示两者的模块依赖关系，最后评估在迁移过程中可能遇到的挑战和解决方案。通过对比分析，旨在为开发者提供有效的迁移策略和实践指导，以便更好地适应不断变化的深度学习生态系统。

B.2. 框架整体分析

B.2.1 框架概述

在深度学习的快速发展中，各种深度学习框架如雨后春笋般涌现，PyTorch 和 MindSpore 是当前最受关注的两个框架。它们各自具有独特的设计理念和优势，适用于不同的应用场景。对这两个框架的整体分析不仅有助于理解它们的核心特性，还能为开发者在选择合适的工具时提供指导。

PyTorch 以其动态计算图和灵活的 API 设计受到广泛欢迎，尤其在学术界。它的主要优势在于：

- **易于调试和开发**：动态计算图使得模型的构建和调试变得直观，开发者可以即时查看中间结果，快速迭代。
- **广泛的生态系统**：强大的社区支持和丰富的第三方库（如 torchvision、torchtext 等）使得 PyTorch 在处理计算机视觉和自然语言处理等任务时具有极大的便利性。

MindSpore 则专注于高效的计算和灵活的开发体验，尤其是在华为的硬件平台上，MindSpore 展现出优越的性能。其主要特点包括：

- **动静态图的灵活性**：支持动态图和静态图的切换，能够根据不同的需求进行优化，适应多样化的开发环境。
- **针对硬件的深度优化**：MindSpore 针对华为自有的 Ascend 处理器进行了深度优化，使得在这些硬

件上运行时性能显著提升，尤其是在大规模并行计算时。

综上所述，PyTorch 和 MindSpore 各有千秋，开发者在选择框架时应考虑具体的应用需求、硬件环境以及团队的技术栈。通过对这两个框架的深入分析，开发者可以更好地利用它们的优势，提升模型的开发效率和性能。

B.2.2 框架比较

为了深入分析 MindSpore、PyTorch 和 TensorFlow 三个深度学习框架，我们从易用性、性能、灵活性、社区支持等多个维度进行了比较。图 1 展示了各框架在不同维度的综合表现。

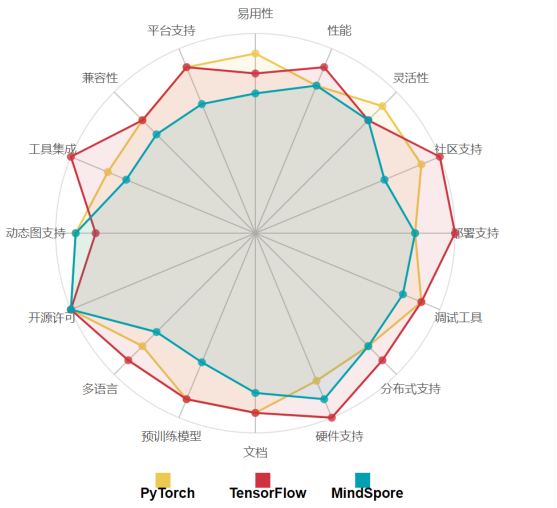


图 1. 深度学习框架比较雷达图

从雷达图可以看出：

- **易用性**：PyTorch 以直观的 API 设计和符合 Python 习惯的动态计算图机制，使新手能够迅速上手并方便地调试和开发。MindSpore 同样提供了友好的接口，支持动态图和静态图，易用性不断提升。
- **性能**：TensorFlow 凭借 XLA 编译器对计算图的深度优化，在性能上表现出色，适用于大规模分布式训练。PyTorch 支持 JIT 编译和对 CUDA 的良好支持，性能优异。MindSpore 针对 Ascend 芯片等硬件进行了优化，在特定硬件上具有优势。
- **灵活性**：PyTorch 的动态图机制允许开发者在运行时动态改变网络结构，具有高度的灵活性。TensorFlow 和 MindSpore 既支持动态图也支持静态图，可以根据需求进行选择，但在某些高级定制操作上可能需要额外的工作。
- **生态系统**：TensorFlow 拥有庞大的社区和丰富的生态系统，包含大量预训练模型和工具。PyTorch 社区活跃，第三方库和资源丰富。MindSpore 的社区正在快速发展，生态系统逐步完善。

综合来看，PyTorch 在易用性和灵活性方面表现突出，适合研究和快速原型开发；TensorFlow 在性能优化、部署和生态系统上具有优势，适合需要跨平台部署和生产环境的项目；MindSpore 针对全场景 AI 进行了优化，特别适合在华为软硬件生态中构建和部署 AI 应用。

以上比较为后续的框架 API 比较分析和代码迁移提供了基础.

B.2.3 模块依赖分析

为了深入了解 PyTorch 和 MindSpore 两个框架的模块设计差异，我们对它们的内部模块依赖关系进行了分析，并将结果进行了可视化。如图 2 和图 3 所示，分别展示了 PyTorch 和 MindSpore 的模块依赖图。

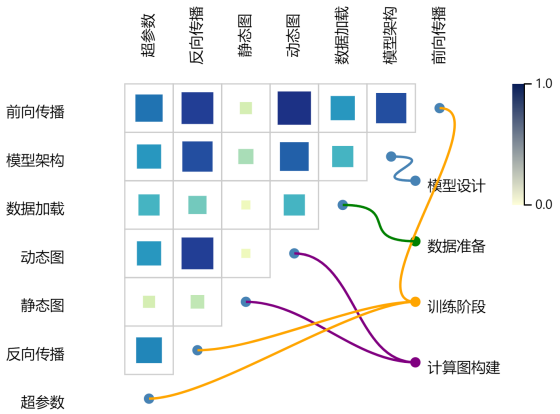


图 2. PyTorch 模块依赖图

模块依赖分析的重要性

在代码迁移过程中，模块依赖分析是至关重要的一步。通过对源框架和目标框架的模块依赖关系进行比较，可以：

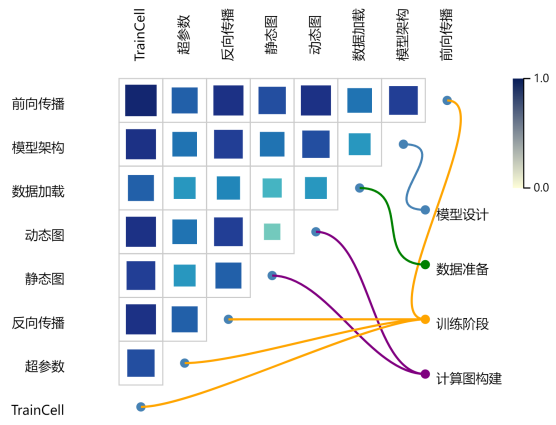


图 3. MindSpore 模块依赖图

- 理解各模块之间的耦合程度，识别关键路径，确保在迁移过程中重点关注高依赖性的模块。
- 发现模块功能的对应关系，帮助确定哪些模块需要重构、替换或调整，以适应目标框架的设计。
- 评估可能的迁移难点和风险，从而制定更有效的迁移策略，降低开发成本和时间。

PyTorch 模块依赖分析

从图 2 可以看出，PyTorch 的模块设计相对松散，各模块之间的依赖关系较为清晰：

- **动态图**是 PyTorch 的核心特性，直接影响 **模型训练**和 **反向传播**，与其他模块的交互也十分紧密。
- **模型架构**在整个框架中起到支柱作用，既影响模型的训练，又与数据加载、超参数等模块有联系。
- **静态图**在 PyTorch 中使用较少，因而与其他模块的依赖性较低。

这种模块设计使得开发者在进行模型构建和训练时具有较高的灵活性，可以根据需要自由调整各模块而不受过多限制。

MindSpore 模块依赖分析

从图 3 可以观察到，MindSpore 的模块之间依赖关系更为紧密：

- **TrainOneStepCell** 是 MindSpore 中高度封装的训练单元，与 **模型训练**、**模型架构**、**反向传播**等核心模块都有强烈的依赖关系。
- **动态图**和 **静态图**两种模式在 MindSpore 中均得到支持，且对其他模块有较大的影响，体现了框架对动静态图融合的重视。
- 各模块之间的高依赖性反映了框架的高封装性和一体化设计理念，有助于提升性能和简化开发过程。

然而，高度的模块依赖也意味着在进行代码迁移时，需要更加注意各模块的兼容性和协调性。

小结

模块依赖分析为代码迁移提供了全局视角，帮助我们识别关键模块和潜在的迁移挑战。理解源框架和目标框架的设计理念和模块组织方式，是成功完成代码迁移的关键步骤。

B.3. 框架 API 分析与映射

在代码迁移过程中，深入理解源框架和目标框架的 API 差异，对于高效、准确地完成迁移至关重要。本节将对 PyTorch 和 MindSpore 的 API 进行分析和对比，找出关键模块的对应关系，为代码迁移提供指导。

B.3.1 API 总览

为了直观地展示两个框架的 API 结构，我们绘制了 PyTorch 和 MindSpore 的 API 径向树图，如图 4 和图 5 所示。

从以上图示可以看出，两个框架的 API 设计虽有不同，但在核心功能上存在相似之处。例如，它们都提供了张量操作、神经网络模块、优化器、数据加载等基础组件，这为代码迁移提供了可能性。

B.3.2 核心模块 API 对比

为了更详细地了解两个框架之间的 API 差异，我们针对关键模块进行了分析，并整理出对应的 API 映射关系，如表 1 所示。

张量操作

两者都使用 Tensor 作为基本的数值计算单元，但在创建和操作方法上可能存在细微差异。例如，torch.Tensor 支持大部分的 NumPy 风格操作，而 mindspore.Tensor 在使用上与之类似，但需要注意接口是否完全一致。

自动求导机制

表 1. PyTorch 与 MindSpore 核心 API 映射表

模块	PyTorch API	MindSpore API
张量定义	torch.Tensor	mindspore.Tensor
自动求导	torch.autograd	mindspore.ops.GradOperation
神经网络模块	torch.nn.Module	mindspore.nn.Cell
常用层	torch.nn.Linear	mindspore.nn.Dense
激活函数	torch.nn.ReLU	mindspore.nn.ReLU
损失函数	torch.nn.CrossEntropyLoss	mindspore.nn.SoftmaxCrossEntropyWithLogits
优化器	torch.optim.SGD	mindspore.nn.optim.SGD
学习率调度	torch.optim.lr_scheduler	mindspore.nn.optim.lr_scheduler
数据集	torch.utils.data.Dataset	mindspore.dataset.Dataset
数据加载器	torch.utils.data.DataLoader	mindspore.dataset 模块的相关方法
设备管理	torch.device	context.set_context(device_target=...)
模型保存	torch.save	mindspore.save_checkpoint
模型加载	torch.load	mindspore.load_checkpoint
计算图模式	动态图（默认）	动态/静态图（PYNATIVE_MODE / GRAPH_MODE）

图 4. PyTorch API 径向树图

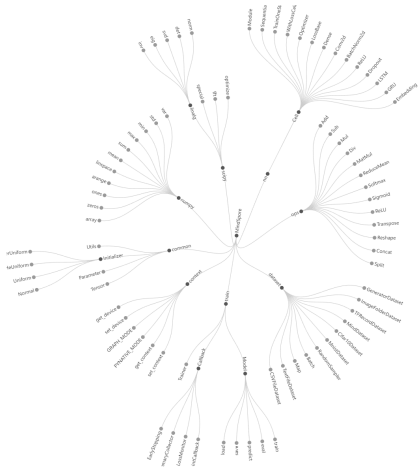


图 5. MindSpore API 径向树图

PyTorch 的自动求导由 torch.autograd 实现，通过动态图机制，计算过程会被即时记录，反向传播时根据计算图自动求导。

MindSpore 提供了 mindspore.ops.GradOperation 用于求导，但在实际使用中，更常通过重写 construct 方法，并使用装饰器 @ms_function 来实现自动求导。需要注意 MindSpore 中动、静态图模式下自动求导的差异。

神经网络模块

PyTorch 使用 torch.nn.Module 作为神经网络的基类，所有自定义的神经网络都需要继承自这个类。

MindSpore 则使用 mindspore.nn.Cell 作为基类，

其设计理念与 PyTorch 类似，但在方法命名和实现上可能有所不同。例如，PyTorch 通常会重写 forward 方法，而 MindSpore 则需要重写 construct 方法。

优化器和损失函数

优化器和损失函数在两个框架中都有类似的实现，但需要注意参数的命名和用法可能有所不同。在迁移时，需要仔细对照 API 文档，确保参数传递正确。

数据处理

PyTorch 提供了 Dataset 和 DataLoader 用于数据加载和批处理。数据集可以通过继承 torch.utils.data.Dataset 并实现 __len__ 和 __getitem__ 方法来自定义。

MindSpore 的数据处理模块更加丰富，提供了专门的 mindspore.dataset 模块，用于构建从数据源到数据处理的完整管道。在迁移过程中，需要将 PyTorch 的数据加载流程转换为 MindSpore 的数据处理管道，并注意数据格式和类型的匹配。

小结

通过对 PyTorch 和 MindSpore 的 API 进行深入分析和映射，我们可以更清晰地了解两者之间的差异，为代码迁移制定合理的策略。充分利用两者的相似性，关注关键差异点，能够有效地降低迁移难度，提高开发效率。

B.4. 核心代码迁移

在本节中，我们将详细介绍将时序预测模型从 PyTorch 迁移到 MindSpore 的具体过程。我们需要充分理解了源代码的功能和目标框架的特点，以确保迁移后的代码在性能和功能上都能满足预期。

根据模型的结构，我们将迁移过程分为以下几个核心部分：

1. 模型定义的迁移 原始模型在 PyTorch 中定义为一个继承自 nn.Module 的类，我们需要将其转换为继承自 mindspore.nn.Cell 的类。

- 将 torch.nn.Module 替换为 mindspore.nn.Cell。
- 将 forward 方法重命名为 construct 方法，这是 MindSpore 的要求。
- 替换 PyTorch 的运算函数为 MindSpore 对应的运算符，例如使用 mindspore.ops 中的函数。

示例代码：

Listing 1. SparseTSF 模型的迁移

```python
from mindspore import nn, ops

class SparseTSF(nn.Cell):
    def __init__(self, configs):
        super(SparseTSF, self).__init__()
        # 初始化参数和层
        # ...

    def construct(self, x):
        # 定义前向计算过程
        # 使用 MindSpore 的操作符替换 PyTorch 的函数
        # ...
        return y
```

2. 损失函数和优化器的迁移 MindSpore 提供了与 PyTorch 类似的损失函数和优化器，但在使用方式上有所区别。

- 使用 mindspore.nn.MSELoss 替代 torch.nn.MSELoss。
- 优化器的定义需要传入模型的可训练参数，可以通过 model.trainable_params() 获取。

示例代码：

Listing 2. 损失函数和优化器的迁移

```python
# 定义损失函数
self.criterion = nn.MSELoss()

# 定义优化器
self.optimizer = nn.Adam(self.model.trainable_params
    (), learning_rate=self.configs.learning_rate)
```

3. 训练步骤的迁移 在 MindSpore 中，训练步骤通常使用 TrainOneStepCell 封装，这与 PyTorch 中手动编写训练循环有所不同。

- 使用 MyWithLossCell 将模型和损失函数封装在一起。
- 使用 TrainOneStepCell 封装训练步骤，传入网络和优化器。
- 调用 train_network 对输入数据进行训练。

4. 数据加载的迁移 由于原始代码使用了 PyTorch 的数据加载器，我们需要将其替换为 MindSpore 的数据处理方式。

- 将 torch.utils.data.Dataset 替换为自定义的数据集类，或使用 MindSpore 提供的 mindspore.dataset 模块。
- 读取数据并转换为 MindSpore 的 Tensor 对象。

示例代码:

```
dataset = ds.NumpySlicesDataset(
        {"batch_x": batch_x, "batch_y": batch_y, "
            batch_x_mark": batch_x_mark, "
            batch_y_mark": batch_y_mark},
        shuffle=shuffle_flag
    ).batch(batch_size, drop_remainder=drop_last)
```

B.4.1 小结

通过上述步骤，我们成功地将时序预测模型从 PyTorch 迁移到了 MindSpore。在迁移过程中，我们充分利用了两者之间的相似性，同时针对差异点进行了相应的调整和优化。

B.5. 迁移结果

经过对模型代码的迁移和优化，我们从多个方面对迁移结果进行了评估，具体如下：

- **混合精度训练**：代码实现了混合精度训练，加速了模型训练过程，同时降低了显存占用。
- **GPU 支持**：通过设置 device_target="GPU"，模型可以在 GPU 环境下运行，充分利用 GPU 的计算能力，提高了训练速度。
- **动态图模式**：在代码中使用了 PYNATIVE_MODE，实现了动态图模式，方便了代码的调试和开发。
- **静态图模式**：代码中提供了根据配置切换到 GRAPH_MODE 的功能，可在训练时使用静态图模式提升性能。
- **内存优化**：通过优化数据加载和模型的内存分配，减少了内存的占用，提高了模型的运行效率。
- **模型保存**：使用 mindspore.save_checkpoint 函数，实现了模型的保存功能，便于后续的模型加载和推理。
- **异常处理**：在训练和测试过程中，添加了必要的异常捕获和处理机制，增强了代码的稳定性。
- **学习率调整**：实现了动态调整学习率的功能，根据不同的策略在训练过程中调整学习率，提高了模型的收敛速度。
- **早停机制**：引入了早停机制，当验证集损失不再下降时，提前停止训练，避免过拟合。
- **多设备支持**：代码中考虑了在 CPU 和 GPU 环境下的运行，具有良好的兼容性。

为直观展示迁移后模型的功能实现情况，表 2 列出了常用功能及其实现情况。

从上表可以看出，迁移后的模型在各个常用功能上都得到了很好的支持，满足了性能优化和开发便捷性的要求。

B.6. 关键问题与解决方案

在迁移过程中，我们遇到了以下关键问题，并针对性地提出了解决方案。在模型迁移过程中，我们遇到了若干需要重点解决的问题，以下将对其中部分进行详细阐述。

B.6.1 TrainOneStepCell 封装

问题描述 在 PyTorch 中，我们通常使用自定义的训练循环来控制模型的前向传播、损失计算、反向传播和参数更新。然而，MindSpore 提供了封装好的训练单元 TrainOneStepCell，用于将网络和优化器组合在一起，以提高训练效率。

在迁移过程中，我们发现直接使用 TrainOneStepCell 存在以下挑战：

表 2. 迁移后模型功能实现情况一览表

功能	混合精度训练	GPU 支持	动态图	静态图	内存优化
实现情况	✓	✓	✓	✓	✓
功能	模型保存	异常处理	学习率调整	早停机制	多设备支持
实现情况	✓	✓	✓	✓	✓

- **灵活性不足**：封装后的训练步骤可能不易定制，无法满足一些特殊的训练需求，例如多任务学习或自定义的梯度更新策略。
- **调试困难**：由于训练过程被高度封装，难以及时获取中间变量，增加了调试难度。
- **动态图兼容性**：在动态图模式下，需要确保 `TrainOneStepCell` 的使用方式与动态图机制相兼容，避免出现运行时错误。

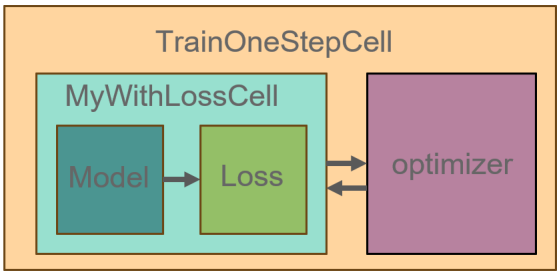


图 6. `TrainOneStepCell` 结构图

如图 6 所示，`TrainOneStepCell` 将模型、损失函数和优化器紧密结合在一起。这种封装提高了训练效率，但也带来了灵活性和调试上的挑战。

解决方案　针对上述问题，我们采取了以下措施：

1. **使用自定义的 WithLossCell**：为了增加训练的灵活性，我们定义了一个自定义的 `MyWithLossCell`，将模型的前向传播和损失计算封装在一起。这使得我们可以在不修改 `TrainOneStepCell` 的情况下，自由地调整损失函数和前向计算。

Listing 3. 自定义的 `MyWithLossCell` 封装

```
class MyWithLossCell(nn.Cell):
    def __init__(self, backbone, loss_fn, configs):
        super(MyWithLossCell, self).__init__(auto_prefix=False)
        self.backbone = backbone
        self.loss_fn = loss_fn
        ...

    def construct(self, data, label):
        outputs = self.backbone(data)
        ...
        loss = self.loss_fn(outputs, label)
        return loss
```

2. **封装训练过程**：利用 `TrainOneStepCell`，我们将自定义的 `MyWithLossCell` 和优化器封装在一起。这使得训练步骤简洁明了，同时保留了必要的灵活性。

Listing 4. 封装训练网络

```
# 将模型和损失函数封装
self.net_with_loss = MyWithLossCell(self.model, self.criterion, self.configs)

# 使用 TrainOneStepCell 封装网络和优化器
self.train_network = nn.TrainOneStepCell(self.net_with_loss, self.optimizer)
```

3. **提高调试便捷性**：在动态图模式（`PYNATIVE_MODE`）下，我们可以在 `MyWithLossCell` 和训练循环中插入断点或打印中间结果，以便进行调试。

4. **确保动态图兼容性**：我们在训练函数中根据配置选择运行模式，确保在动态图模式下，`TrainOneStepCell` 能够正常工作。

Listing 5. 设置运行模式

```
if self.configs.use_GRAPH_MODE:
    context.set_context(mode=context.GRAPH_MODE)
else:
    context.set_context(mode=context.
        PYNATIVE_MODE)
```

效果与优势 通过上述方法，我们成功解决了 TrainOneStepCell 使用中的问题，带来了以下优势：

- **灵活性增强**：自定义的 MyWithLossCell 提供了更高的灵活性，便于调整模型的前向和损失计算方式。
- **便于调试**：在动态图模式下，可以方便地查看中间结果，定位和解决问题变得更加容易。
- **性能提升**：TrainOneStepCell 的封装提高了训练效率，充分发挥了 MindSpore 的性能优势。
- **代码简洁**：封装后的训练过程简洁明了，降低了代码的复杂性，提升了可读性。

B.6.2 动态图与静态图

PyTorch 默认使用动态图机制，而 MindSpore 支持动态图（PYNATIVE_MODE）和静态图（GRAPH_MODE）模式。

解决方案：

- 在调试阶段，将 MindSpore 设置为动态图模式，方便调试和问题定位。

```
    import mindspore as ms
    ms.context.set_context(mode=ms.context.
        PYNATIVE_MODE)
```

- 在正式训练时，可以切换到静态图模式以提高性能。

```
    ms.context.set_context(mode=ms.context.
        GRAPH_MODE)
```

B.6.3 操作函数的替换

PyTorch 中的许多操作函数需要替换为 MindSpore 的对应操作符，这涉及大量的代码修改。

解决方案：

- 使用 mindspore.ops 中的操作符替换 PyTorch 函数，确保功能等价。
- 注意操作符的参数顺序和返回值可能有所不同，需仔细核对。

B.6.4 数据类型和维度差异

在数据处理过程中，可能会遇到数据类型不匹配或维度不一致的问题。

解决方案：

- 及时检查和打印张量的形状，确保在每一步操作后数据维度正确。
- 使用 mindspore.Tensor 明确指定数据类型，例如 ms.float32。

B.6.5 验证与测试

完成代码迁移后，我们对模型进行了验证和测试（具体内容见下一节）：

- **训练过程**：模型能够正常训练，损失函数逐渐收敛。
- **性能评估**：在测试集上计算了 MSE、RSE、MAE 等指标，结果符合预期。
- **模型保存**：使用 mindspore.save_checkpoint 成功保存了训练好的模型。

C. 迁移后代码运行结果分析（陈兴浩）

在对代码进行上述迁移之后，我们在华为云平台上使用 cuda10.1 的 mindspore 进行运行，并得出结果。示例并逐一分析如下。在开始分析前，首先阐明一个事实：在运行较大数据集（如 traffic 数据集）时，在进行数据加载时，内存占用会在形成 numpy 矩阵时变得特别大。但同时真实 CPU 的内存是有限的，故我们不能直接加载所有的数据，而是采用分批加载并上传、处理

数据的方式，降低同时处理数据的个数。于是编写如下函数。

```
def data_generator(data_set):
    for i in range(len(data_set)):
        yield data_set[i][0], data_set[i][1],
            data_set[i][2], data_set[i][3]
```

通过 yield 函数，在处理数据时进行分批加载，防止占用过多内存。那么，在过程当中，我们也需要去相应地加载数据。

```
for epoch in range(self.configs.train_epochs):
        epoch_time = time.time()
        iter_count = 0
        train_loss_list = []  # Reset at the
            beginning of each epoch
        epoch_time = time.time()
        train_loss = 0
        batches = 0
        train_data, train_loader = self.
            _get_data(flag='train')
        vali_data, vali_loader = self._get_data(
            flag='val')
        test_data, test_loader = self._get_data(
            flag='test')
        # Training phase
        for i, (batch_x, batch_y, batch_x_mark,
            batch_y_mark) in enumerate(
            train_loader):
            iter_count += 1
            batches += 1
            # Convert to MindSpore Tensors
            batch_x = ms.Tensor(batch_x, ms.
                float32)
            batch_y = ms.Tensor(batch_y, ms.
                float32)

            # Compute loss
            loss = self.train_network(batch_x,
                batch_y)
            loss_value = loss.asnumpy()
            train_loss_list.append(loss_value)
            train_loss += loss_value

        avg_train_loss = train_loss / batches
```

通过这样的方式加载数据，可以保证每一个 Epoch 都可以训练全部的数据，且在数据加载时占用较少的内存。

在此基础上，我们进行后续的讨论。

C.1. *ETTh*1 数据集分析

运行之后，对各个情况的损失曲线绘图，展示如下，可以发现都能很好地收敛到对应的值，且是向一个方向收敛的，说明判断正确。

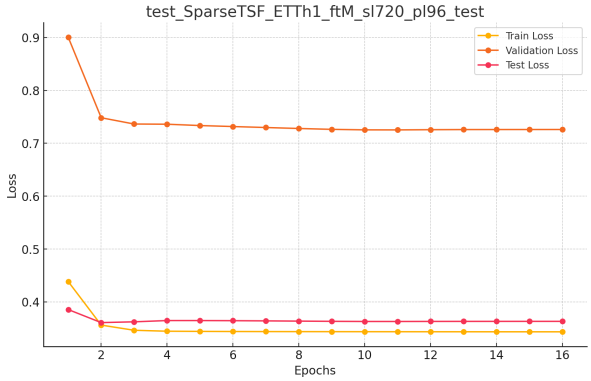


图 7. pl96_ETTh1

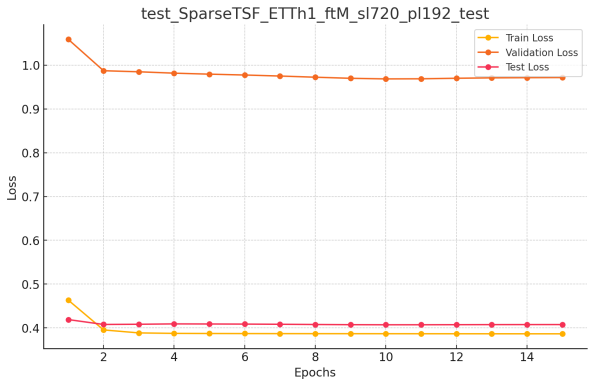


图 8. pl192_ETTh1

接下来，对 pytorch 中和 mindspore 中运行的结果进行对比分析，各种衡量指标绘制表格对比如下。

对其其进行分析如下：

- **总体性能**: Mindspore 在所有三个评估指标 (MSE、MAE、RSE) 上均表现优于 PyTorch，尤其是在长预测范围（例如 $pl = 720$）时，其性能差距更加明显，这说明 mindspore 框架在训练较长周期数据时更具优势。

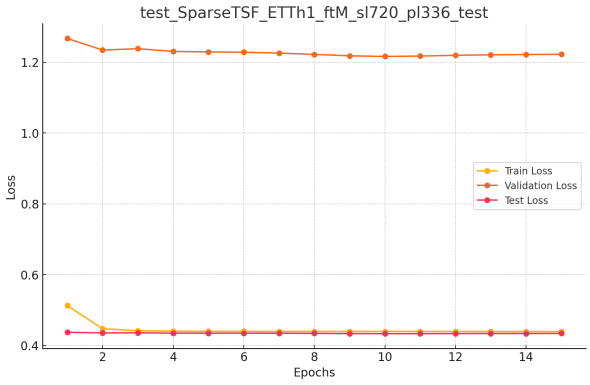


图 9. pl336_ETTh1

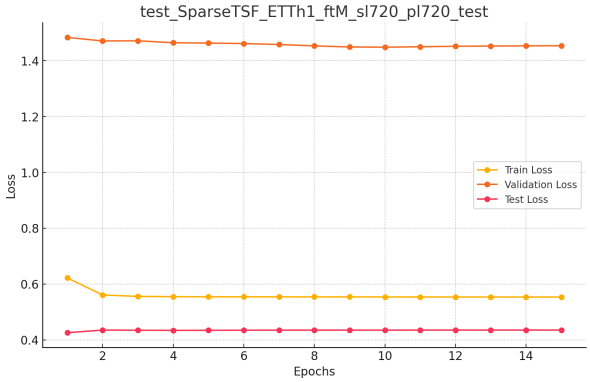


图 10. pl720_ETTh1

pl	Framework	MSE	MAE	RSE
96	PyTorch	0.36227	0.38859	0.57171
	Mindspore	**0.36220**	**0.38825**	**0.57125**
192	PyTorch	0.40382	0.41180	0.60346
	Mindspore	**0.40232**	**0.41111**	**0.60174**
336	PyTorch	0.43452	0.42837	0.62757
	Mindspore	**0.43229**	**0.42733**	**0.62493**
720	PyTorch	0.42644	0.44790	0.62515
	Mindspore	**0.42502**	**0.44540**	**0.62238**

表 3. Comparison of PyTorch and Mindspore performance on ETTh1 dataset.

- **指标细化**:
  - **MSE (Mean Squared Error)**: Mindspore 的 MSE 在所有预测范围内都略低，反映其对数据点预测的偏差更小。

Framework	Time/Epoch (s)
PyTorch	**0.88**
Mindspore	6.26

表 4. Time per epoch comparison of PyTorch and Mindspore on ETTh1 dataset.

  - **MAE (Mean Absolute Error)**: MAE 的对比进一步支持了 Mindspore 的优势，其绝对误差在不同预测范围内都保持较低值。
  - **RSE (Relative Squared Error)**: Mindspore 的相对误差比 PyTorch 更低，意味着其整体预测波动性更小。

- **适用性建议**: 对于高精度需求的场景，尤其是长预测范围（如 336 或 720 步）的应用，Mindspore 是更佳选择。若对计算效率要求较高且容忍少量精度损失，PyTorch 仍然是一个稳健的选择。

C.2. *ETTh2* 数据集分析

为了节省篇幅，我把四个折线图绘制到一个图表款内，展示如下。

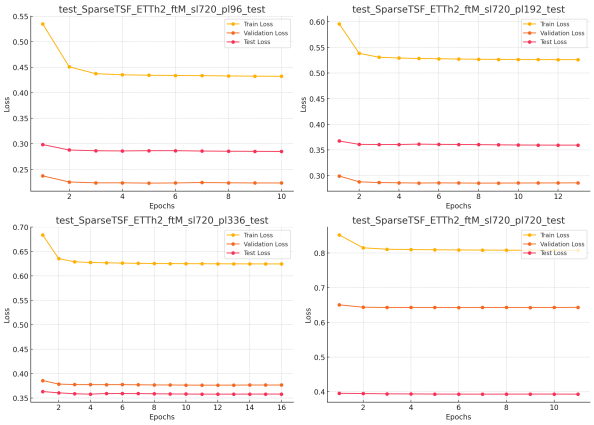


图 11. ETTh2

对以上情况作如下分析:

- **总体性能**: Mindspore 在所有预测范围 (pl) 的 MSE、MAE 和 RSE 均优于 PyTorch，特别是在短预测范围 ($pl = 96$) 时表现尤为显著。

- **指标细化**:

pl	Framework	MSE	MAE	RSE
96	PyTorch	0.29445	0.34633	0.43731
	Mindspore	**0.28254**	**0.34144**	**0.29920**
192	PyTorch	0.33987	0.37732	0.46751
	Mindspore	**0.33688**	**0.37608**	**0.32636**
336	PyTorch	0.35953	0.39784	0.47941
	Mindspore	**0.35552**	**0.39508**	**0.33615**
720	PyTorch	0.38313	0.42488	0.49474
	Mindspore	**0.37935**	**0.42230**	**0.34910**

表 5. Comparison of PyTorch and Mindspore performance on ETTh2 dataset.

Framework	Time/Epoch (s)
PyTorch	**0.82**
Mindspore	6.24

表 6. Time per epoch comparison of PyTorch and Mindspore on ETTh2 dataset.

  - **MSE (Mean Squared Error)**: Mindspore 在短预测范围 (96) 下的 MSE 明显低于 PyTorch，表明其误差较小。
  - **MAE (Mean Absolute Error)**: Mindspore 的 MAE 在各预测范围内均低于 PyTorch，误差幅度更小。
  - **RSE (Relative Squared Error)**: Mindspore 在每个预测范围下的 RSE 均表现出更稳定的趋势，波动性更小。

- **适用性建议**:
  - 在高精度要求的短预测范围场景（如 $pl = 96, 192$），建议使用 Mindspore。
  - 对于长预测范围（如 $pl = 336, 720$），两者差异缩小，但 Mindspore 仍有略微优势。

C.3. $ETTm1$ 数据集分析

分析得到结论如下：

- **总体性能**: 在短预测范围（如 $pl = 96$ 和 $pl = 192$）中，PyTorch 的 MSE 和 MAE 略优于 Mindspore。

pl	Framework	MSE	MAE	RSE
96	PyTorch	**0.31297**	**0.35414**	**0.53233**
	Mindspore	0.33531	0.36908	0.55078
192	PyTorch	**0.34774**	**0.37646**	**0.56134**
	Mindspore	0.35276	0.37747	0.56513
336	PyTorch	**0.36750**	**0.38607**	**0.57687**
	Mindspore	0.37552	0.39024	0.58284
720	PyTorch	**0.41923**	**0.41325**	**0.61602**
	Mindspore	0.42007	0.41407	0.61616

表 7. Comparison of PyTorch and Mindspore performance on ETTm1 dataset.

Framework	Time/Epoch (s)
PyTorch	**2.26**
Mindspore	28.79

表 8. Time per epoch comparison of PyTorch and Mindspore on ETTm1 dataset.

- **指标细化**:
  - **MSE (Mean Squared Error)**: PyTorch 在大多数情况下 MSE 稍低，但差距并不显著。
  - **MAE (Mean Absolute Error)**: 两种方法在 MAE 上的表现非常接近，Mindspore 稍有劣势。
  - **RSE (Relative Squared Error)**: RSE 值显示 PyTorch 的整体预测波动性稍小，但差异微弱。

- **适用性建议**:
  - 对于短期预测场景，PyTorch 是稍好的选择。
  - 对于长预测范围，Mindspore 和 PyTorch 的表现非常接近，取决于具体应用需求。

C.4. $ETTm2$ 数据集分析

- **总体性能**: 在短期预测（$pl = 96$）中，PyTorch 的表现稍优，而在长期预测（$pl = 720$）中两者性能相近。

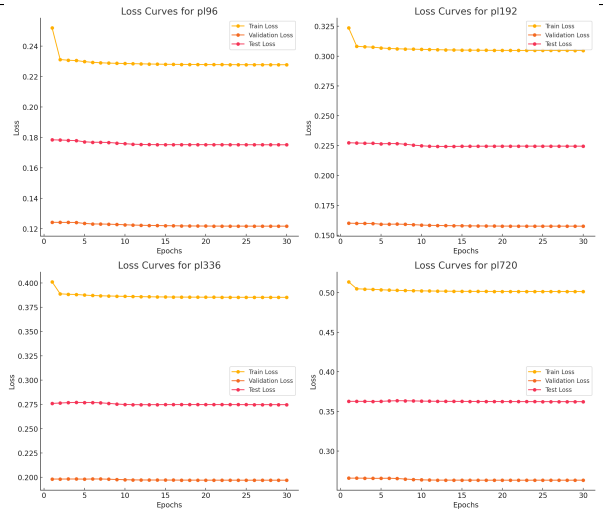


图 12. ETTm2

pl	Framework	MSE	MAE	RSE
96	PyTorch	**0.16294**	**0.25341**	0.32728
	Mindspore	0.17584	0.26454	**0.23697**
192	PyTorch	**0.21539**	**0.28932**	0.37567
	Mindspore	0.22625	0.29848	**0.26860**
336	PyTorch	**0.26824**	**0.32537**	0.41833
	Mindspore	0.27542	0.33092	**0.29611**
720	PyTorch	**0.35166**	**0.37851**	0.47666
	Mindspore	0.35366	0.38093	**0.33520**

表 9. Performance comparison of PyTorch and Mindspore on ETTm2 dataset.

Framework	Time/Epoch (s)
PyTorch	**2.28**
Mindspore	28.82

表 10. Time per epoch comparison of PyTorch and Mindspore on ETTm2 dataset.

- **性能细化**:
  - **MSE (Mean Squared Error)**: PyTorch 的 MSE 增长较快，而 Mindspore 增长相对平缓。
  - **MAE (Mean Absolute Error)**: 短期预测时差距较小，长期预测时差距略增。
  - **RSE (Relative Squared Error)**: Mindspore 在短期预测中的 RSE 显著低于 PyTorch，长期预测时表现趋于一致。
- **适用性建议**:
  - 短期预测任务（$pl = 96, 192$），建议选择 PyTorch。
  - 长期预测任务（$pl = 336, 720$），两者性能接近，可根据需求选择。

C.5. *weather* 数据集分析

损失折线图绘制如下。

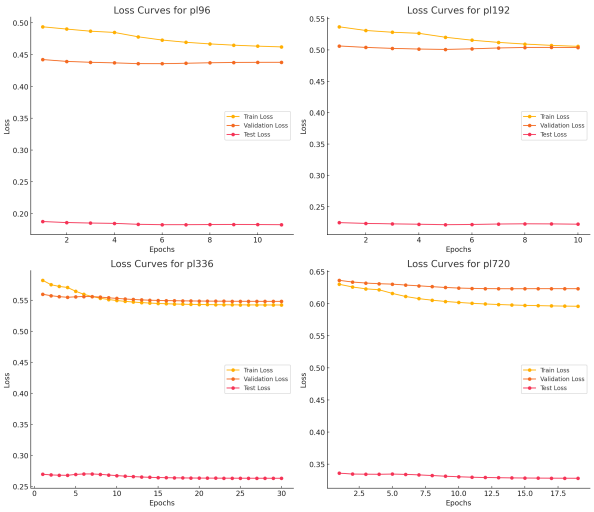


图 13. weather

误差对比情况如下。

- **总体性能**: 在短期预测 ($pl = 96$) 中，PyTorch 表现更优；而在长期预测 ($pl = 720$) 中，Mindspore 的稳定性更好。
- **性能细化**:
  - **MSE**: PyTorch 在短期预测上 MSE 更低，Mindspore 增长较平稳。
  - **MAE**: Mindspore 在所有范围内的 MAE 均稍高于 PyTorch。
  - **RSE**: Mindspore 在长期预测中 RSE 表现更优。

pl	Framework	MSE	MAE	RSE
96	PyTorch	**0.16937**	**0.22309**	0.54233
	Mindspore	0.18326	0.23939	**0.53655**
192	PyTorch	**0.21441**	**0.26216**	0.60953
	Mindspore	0.22452	0.27213	**0.59348**
336	PyTorch	**0.25737**	**0.29358**	0.66629
	Mindspore	0.26361	0.29996	**0.64193**
720	PyTorch	**0.32169**	**0.34040**	0.74636
	Mindspore	0.32409	0.34251	**0.71314**

表 11. Performance comparison of PyTorch and Mindspore on Weather dataset.

Framework	Time/Epoch (s)
PyTorch	**3.93**
Mindspore	339.27

表 12. Time per epoch comparison of PyTorch and Mindspore on Weather dataset.

- **适用性建议**:
  - 短期预测场景 (如 $pl = 96$): 推荐使用 PyTorch。
  - 长期预测场景 (如 $pl = 720$): 推荐使用 Mindspore。

C.6. *traffic* 数据集

pl	Framework	MSE	MAE	RSE
96	PyTorch	0.38920	0.26813	0.51658
	Mindspore	**0.38909**	**0.26637**	**0.51552**
192	PyTorch	0.39898	0.27047	0.52132
	Mindspore	**0.39864**	**0.26983**	**0.52001**
336	PyTorch	**0.41116**	**0.27590**	0.52699
	Mindspore	0.41127	0.27612	**0.52583**
720	PyTorch	**0.44874**	**0.29720**	**0.54777**
	Mindspore	0.44924	0.29790	0.54867

表 13. Performance comparison of PyTorch and Mindspore on Traffic dataset.

Framework	Time/Epoch (s)
PyTorch	**38.29**
Mindspore	669.33

表 14. Time per epoch comparison of PyTorch and Mindspore on Traffic dataset.

C.7. *electricity* 数据集

pl	Framework	MSE	MAE	RSE
96	PyTorch	0.13854	0.23323	0.36995
	Mindspore	**0.13845**	**0.23288**	**0.36979**
192	PyTorch	0.15101	0.24472	0.38637
	Mindspore	**0.15101**	**0.24445**	**0.38631**
336	PyTorch	**0.16605**	**0.25996**	**0.40557**
	Mindspore	0.16679	0.26090	0.40637
720	PyTorch	**0.20650**	0.29693	**0.45330**
	Mindspore	0.20744	**0.29624**	0.45413

表 15. Performance comparison of PyTorch and Mindspore on Electricity dataset.

Framework	Time
PyTorch	**84.25**
Mindspore	318.36

表 16. Time per epoch comparison of PyTorch and Mindspore on Electricity dataset.

C.8. 总结

我们可以发现，在大多数预测任务数据集上，Pytorch 都是短周期任务较有优势，但在少部分数据集上，Pytorch 在长周期任务上更具优势。但事实上，在所有模型中，这两者的误差指标差距均非常小，说更具优势的一方事实上高出的水平不多，因此可以大概地认为两者性能相近（这点从数据上是很容易看出的，两者的差距基本都小于 0.1 甚至 0.01）。

所以我们在选取使用时，如果标准不高，则按照自身意愿选择即可。但事实上，我们在运行时发现，

mindspore 很多运行时长都远远参与 pytorch 上的时长，因此可以考虑在能使用 pytorch 的时候，出于效率的考虑，均使用 pytorch，毕竟两者性能相差无几。综合上述所有数据分析，绘制出三个衡量指标的雷达图。

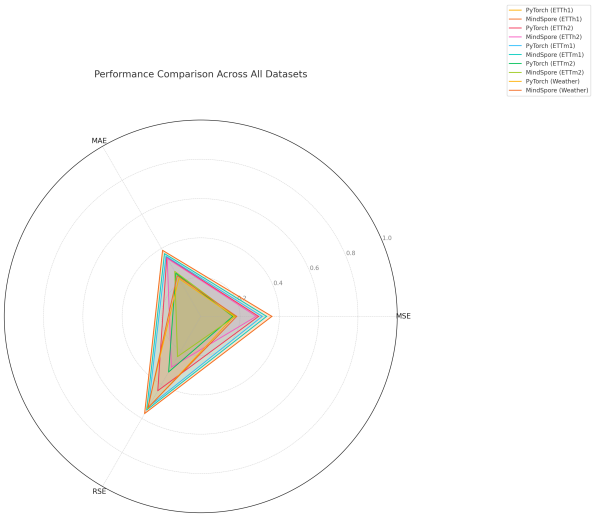


图 14. 性能雷达图

这是整合了所有数据集(ETTh1、ETTh2、ETTm1、ETTm2、Weather）的 PyTorch 与 MindSpore 的性能雷达图。

每个框架在不同数据集的三项指标（MSE、MAE、RSE）表现都被展示。可以清晰看到，某些数据集（如 ETTh2 和 Weather 数据集）的 MindSpore 在 RSE 上稍具优势，而 PyTorch 在部分短期预测（如 ETTh1 和 Weather 数据集的 MSE）表现更优。

```
Epoch: 19 cost time: 77.11709380149841
Epoch: 19, Steps: 138 | Train Loss: 0.5959068 Vali Loss: 0.6232185 Test Loss: 0.328154
1
Updating learning rate to 0.000141
Validation loss did not improve.
EarlyStopping counter: 5 out of 5
Early stopping triggered. Stopping training.
Training complete.
>>>>>>>testing : test_SparseTSF_custom_ftM_sl720_pl720_test<<<<<<<<<<<<<<<<<<<<<<<<<<
<<<<<<
Average Test Loss (MSE): 0.328154
MSE: 0.324093, RSE: 0.713137, MAE: 0.342507
(MindSpore) [ma-user SparseTSF-MS]$
```

图 15. 其中一张运行截图

D. 使用独热编码进行参数可视化对比分析（王亦辉）

通过论文作者提供的可视化方法，我们可以通过独热编码得到 SparseTSF 的等效于 Linear 的权重。作者的想法是，既然 Linear 对对角矩阵进行一次前向传播（也就是用权重矩阵与对角矩阵相乘）得到的是 Linear 的权重，那么可以认为 SparseTSF 对对角矩阵进行一次前向传播得到的是可以与 Linear 进行对比的等效权重，如图 16。

$$weight' = SparseTSF(\begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix})^\top$$

图 16. 得到等效权重的方法

D.1. 可视化分析的好处

通过可视化权重可以看出模型如何对周期内的不同时间点赋予不同的重要性，从而直观地理解周期性特征对模型的贡献，提高模型的可解释性。通过独热编码，我们可以将 SparseTSF 这个并非单纯线性的模型与 Linear 进行可视化的对比，从而可以直观地看出 SparseTSF 相对于 Linear 模型的提升。

D.2. 可视化过程

对于 SparseTSF，首先加载已经训练好的模型（在 PyTorch 下与在 MindSpore 下略有不同），然后利用这个模型，对一个对角矩阵中的每个行向量做一次前向传播，将输出收集起来拼成一个矩阵，这个矩阵即是 SparseTSF 的等效权重，代码见下。对于 Linear 模型，我们可以直接得到其权重矩阵。之后利用 `matplotlib.pyplot` 库中的函数对权重归一化为 0 到 1 并绘制热图，代码见下。

```
    net = SparseTSF()
    net.load_state_dict(torch.load('
        your_checkpoint_path', map_location='cpu'))
    weights = np.zeros((96, 96))

    for i in range(96):
        x = np.zeros((1, 96, 1), dtype=np.float32)
        x[0, i, 0] = 1
        x_tensor = torch.tensor(x)
        output = net(x_tensor)
        weights[i] = output.detach().numpy().flatten
            ()
        weights_matrix = np.array(weights)
```

```
    plot_weights(weights_matrix,'
        weights_plot_no_norm_xxxx.png',norm=True)
```

```
def plot_weights(array,picture_name,norm):


    if norm == True:
        array = (array - np.min(array)) / (np.max(
            array) - np.min(array))


    fig, ax = plt.subplots(figsize=(6, 6))

    im = ax.imshow(array, cmap='Reds', aspect='auto'
        )


    fig.colorbar(im, ax=ax)

    plt.tight_layout()
    plt.show()
    plt.savefig(picture_name)
```

D.3. 可视化结果与分析

我们对 PyTorch 以及 MindSpore 框架下训练的 SparseTSF 模型绘制了等效权重图，以及绘制了 PyTorch 下训练的 Linear 模型作为参照，见图17, 图18, 图19。模型在 ETTH1 数据集下训练，回溯长度（x 轴）与预测长度（y 轴）都是 96。

总览这三张图可以发现，各图中有较为明显的深色斜线，说明各模型在训练时成功捕捉到周期性特征，在预测时加以利用，从而在预测特定点的时候，随步数不同，输入中被赋予高权重的数据的位置在不断发生改变。因此从图中可以看出数据的周期大约是 24。

对三张权重图进行互相对比，可以发现，SparseTSF 对周期性特征的捕捉能力明显比 Linear 更强，表现为深色斜线的颜色更深，即对应的权重更大。此外，令人感到惊奇的是，在 MindSpore 下训练的 SparseTSF 模型的效果比 PyTorch 下的效果更好，表现为 MindSpore 的权重图中的斜线的深色部分更多。

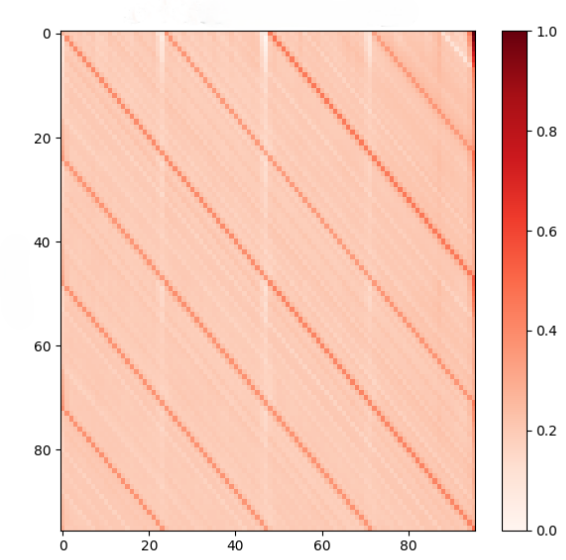


图 17. Linear

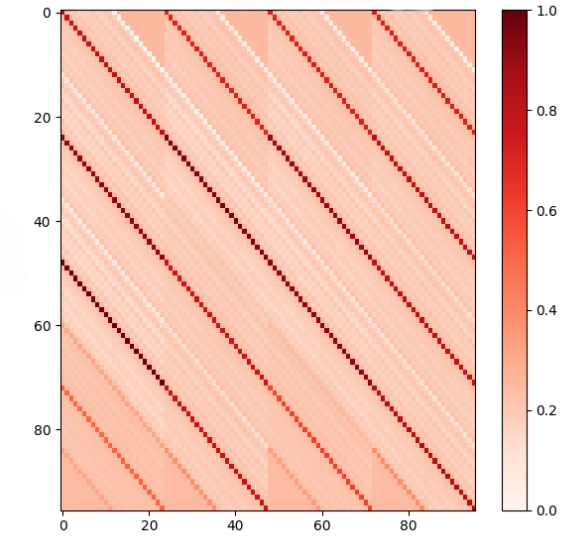


图 18. SparseTSF in MindSpore

E. 模型改进（王为）

E.1. 引言

SparseTSF 模型在实现低参数量和高准确率的时间序列预测方面表现出色。然而，该模型对超参数周期的依赖性较强，而在大多数数据中，我们难以直接观察到其周期性。因此，我们的模型 MyModel 通过傅里叶

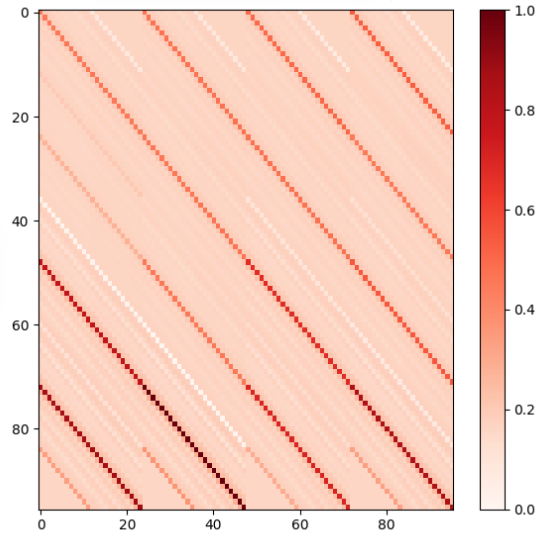


图 19. SparseTSF in PyTorch

变换将时域数据转换为频域，以提取数据周期，从而替代超参数。这一方法不仅保留了原模型的低参数量特性，还实现了端到端的处理。

E.2. 模型架构

MyModel 模型在 SparseTSF 的基础上进行了改进，其核心思想是通过傅里叶变换自动提取时间序列的主要周期，从而消除对超参数周期的依赖。模型的整体架构如图 20 所示，主要由以下几个模块组成：

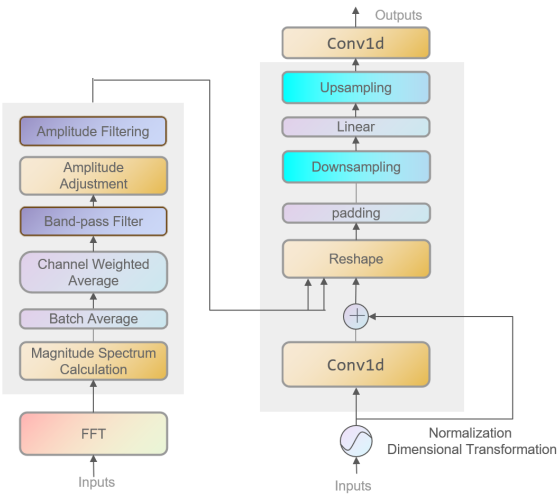


图 20. AutoPeriodTSF 模型整体架构

如图 21 所示，模型的细节包括数据预处理模块、傅里叶变换模块、特征提取模块和预测模块。每个模块在模型中扮演着重要的角色，协同工作以提高时间序列预测的准确性。

E.2.1 周期提取模块

该模块的主要功能是从输入的时间序列数据中自动提取主要周期。具体步骤如下：

1. 对输入序列 $x \in \mathbb{R}^{B\times T\times C}$ 进行傅里叶变换，将时域数据转换到频域：

$$X = \mathcal{F}(x)$$

其中，B 为批次大小，T 为时间序列长度，C 为特征通道数。

2. 计算幅度谱 $|X|$，并对批次维度求平均，得到平均幅度谱 $\overline{|X|}$。

3. 增加主要特征通道的权重，强调对主要特征的关注：

$$\overline{|X|}_c = \begin{cases} w \cdot \overline{|X|}_c, & \text{if } c = c_{\text{main}} \\ \overline{|X|}_c, & \text{otherwise} \end{cases}$$

其中，w 为主要特征的权重，c_{main} 为主要特征的索引。

4. 根据幅度谱找出幅度最大的 k 个频率分量，对应的频率为 f_i，计算对应的周期 $T_i = \frac{1}{f_i}$，并对周期进行筛选，确保 T_i 在预设的范围内。

5. 将筛选后的主要周期 $\{T_i\}$ 作为后续模块的参数。

通过该模块，模型能够自动识别时间序列中的主要周期，无需手动设定，使模型更加自适应。

E.2.2 卷积模块

在获得主要周期后，模型对输入的时间序列进行卷积操作，以捕获周期性的模式。具体操作包括：

1. 对输入序列进行归一化处理，减去序列的均值：

$$\tilde{x} = x - \mu_x$$

其中，μ_x 为输入序列在时间维度上的均值。

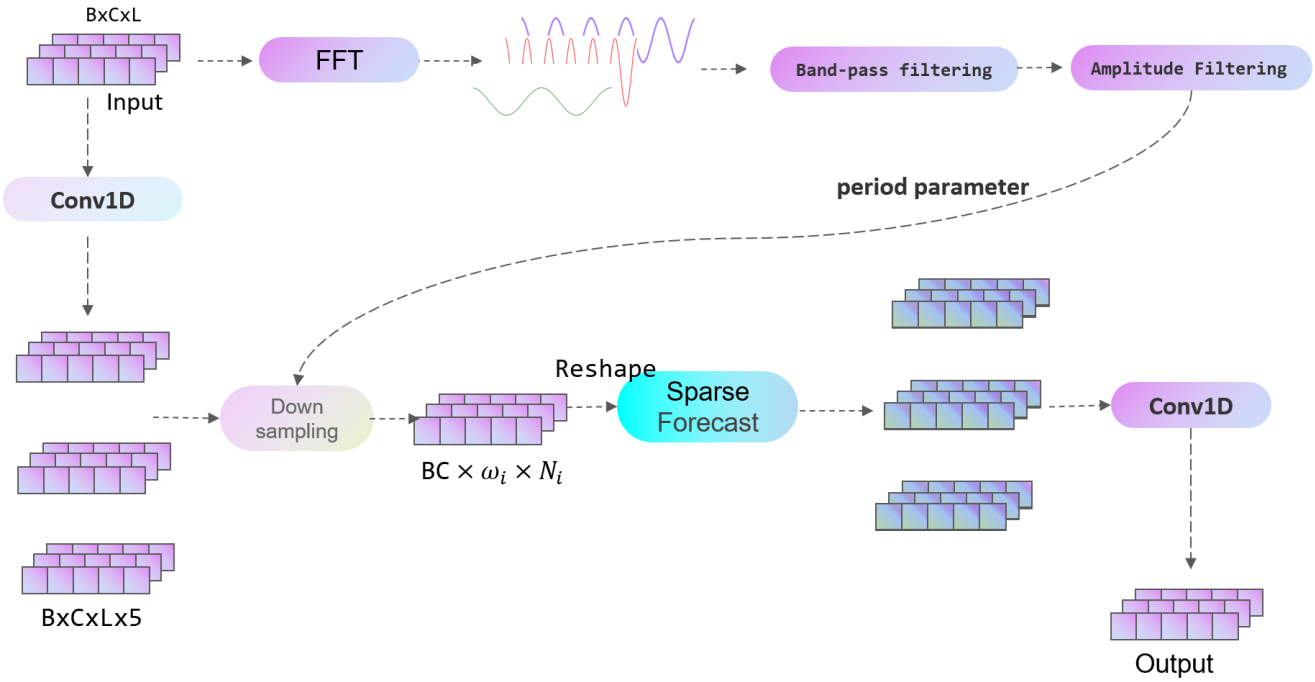


图 21. 模型细节图示

2. 对归一化后的序列应用一维卷积操作，卷积核的大小与主要周期相关：

$$h = \text{Conv1D}(\tilde{x})$$

卷积核的大小为 $k = 1 + 2\left\lfloor \frac{T_{\text{main}}}{2} \right\rfloor$，其中 T_{main} 为主要周期。

3. 将卷积输出与输入序列进行残差连接，获得特征表示：

$$z = h + \tilde{x}$$

通过卷积操作，模型能够提取与主要周期相关的特征，捕获时间序列中的周期性模式。

E.2.3 分段与线性映射

由于时间序列的长度可能不是周期的整数倍，模型对序列进行分段处理：

1. 根据主要周期 T_i，将序列划分为若干段，每段长度为 T_i：

$$z = [z_1, z_2, \ldots, z_{N_i}]$$

其中，$N_i = \left\lfloor \frac{T}{T_i} \right\rfloor$。

2. 对每一段进行线性映射，预测未来的序列片段：

$$y_i = \text{Linear}(z_i)$$

线性层的输入维度为分段数量，输出维度为预测序列分段数量。

该过程使模型能够根据不同的主要周期，对序列进行灵活的预测。

E.2.4 多周期融合模块

由于模型提取了多个主要周期，需将基于不同周期的预测结果进行融合：

1. 将不同周期的预测结果 $\{y_i\}$ 进行堆叠和拼接，形成一个多通道的预测张量。

2. 应用一维卷积 Conv1D2 对拼接的结果进行融合：

$$\hat{y} = \text{Conv1D2}([y_1, y_2, \ldots, y_k])$$

卷积核大小与参与融合的周期数量相关。

3. 对融合后的结果进行形状调整，恢复到预测序列的原始形状，并加回均值进行反归一化：

$$\hat{y} = \hat{y} + \mu_x$$

多周期融合提高了模型的预测准确性，充分利用了不同周期的信息。

E.2.5 组件功能总结

- **周期提取模块**：自动识别主要周期，消除对超参数的依赖，提高模型的自适应性。
- **卷积预测模块**：利用周期相关的卷积核，提取时间序列中的周期性特征。
- **分段与线性映射**：根据主要周期对序列进行分段，使用线性层实现从历史到未来的映射。
- **多周期融合模块**：融合基于不同周期的预测结果，增强模型的预测能力。

E.2.6 模型特点

AutoPeriodTSF 模型具有以下特点：

- **低参数量**：通过使用卷积和线性层，模型参数量小，计算效率高。
- **高准确率**：自动提取主要周期并融合多周期信息，提高了预测的准确性。
- **端到端训练**：模型各部分可通过梯度下降联合训练，简化了训练流程。

E.3. 参数量分析

在本节中，我们对 AutoPeriodTSF 模型的参数量进行详细分析。模型主要由以下三个组件构成：

- 第一层一维卷积层（`self.conv1d`）
- 第二层一维卷积层（`self.conv1d2`）
- 线性映射层（`self.linear`）

E.3.1 第一层一维卷积层

参数量计算公式为：

$$\text{Conv1d 参数量} = \text{in\_channels} \times \text{kernel\_size} \times \text{out\_channels}$$

结合代码中的参数，有：

- $\text{in\_channels} = 1$
- $\text{out\_channels} = 1$
- $\text{kernel\_size} = 1 + 2 \times \left\lfloor \frac{\text{max\_period}}{2} \right\rfloor$

因此，该卷积层的参数量为：

$$\text{Conv1d 参数量} = 1 + 2 \times \left\lfloor \frac{\text{max\_period}}{2} \right\rfloor$$

E.3.2 第二层一维卷积层

该层的参数量计算如下：

$$\begin{aligned}\text{Conv1d2 参数量} &= \text{in\_channels} \times \text{kernel\_size} \times \text{out\_channels} \\ &\quad + \text{out\_channels} \\ &= 5 \times 5 \times 1 + 1 = 26\end{aligned}$$

E.3.3 线性映射层

线性层的参数量为：

$$\text{Linear 参数量} = \left\lfloor \frac{\text{seq\_len}}{\text{max\_period}} \right\rfloor \times \left\lfloor \frac{\text{pred\_len}}{\text{max\_period}} \right\rfloor$$

E.3.4 模型总参数量

模型的总参数量为：

$$\begin{aligned}\text{总参数量} &= \text{Conv1d 参数量} + \text{Conv1d2 参数量} \\ &\quad + \text{Linear 参数量} \\ &= \left(1 + 2 \times \left\lfloor \frac{\text{max\_period}}{2} \right\rfloor\right) + 26 \\ &\quad + \left(\left\lfloor \frac{\text{seq\_len}}{\text{max\_period}} \right\rfloor \times \left\lfloor \frac{\text{pred\_len}}{\text{max\_period}} \right\rfloor\right)\end{aligned}$$

E.3.5 参数分析总结

AutoPeriodTSF 模型在保持低参数量的同时，利用自动提取的主要周期进行高效的时间序列预测。

- **第一层卷积层**：参数量与 max_period 成线性关系。
- **第二层卷积层**：参数量固定为 26。

- **线性层**：参数量受 seq_len、pred_len 和 max_period 的影响。

通过精心设计，模型在参数量和预测性能之间取得了良好的平衡。

E.4. 性能分析

E.4.1 周期提取能力

为了评估模型的周期提取能力，我们在 ETTh1 数据集上设置了不同的最小周期长度（`Min_Period`）进行实验。具体地，我们将 `Min_Period` 分别设定为 1、6、12 和 24，观察模型在不同最小周期限制下的周期提取效果。实验结果如图 22 所示。

从图中可以看出，提取出的频率主要集中在真实周期附近或其因数附近。出现周期因数的情况，可能是由于对较高周期的惩罚稍大，导致模型更倾向于选择较小的周期长度。

E.4.2 与 SparseTSF 的性能比较

为了评估我们提出的模型（MyModel）的预测性能，我们在相同的实验条件下，与 SparseTSF 模型进行了对比。在本次实验中，SparseTSF 需要提供真实的周期作为其超参数，而 MyModel 只需要一个可能的最小周期，然后自动寻找真实的周期。实验在六个广泛使用的时间序列数据集上进行：ETTh1、ETTh2、ETTm1、ETTm2、Weather 和 Traffic。我们在不同的预测长度（即，预测范围）96、192、336 和 720 上进行了实验。两种模型的均方误差（MSE）结果汇总在表 17 中。

从表 17 可以看出，SparseTSF 模型在大多数数据集和预测长度上取得了较低的 MSE 值。这是因为 SparseTSF 在实验中被提供了真实的周期作为其超参数，这使得它能够最大程度地利用时间序列中的周期性信息，从而提高预测精度。

相比之下，MyModel 并不需要真实的周期作为超参数，仅需设定一个可能的最小周期，模型会自动搜索并识别真实的周期。因此，在不知道真实周期的情况下，MyModel 能够适应更多的实际应用场景。然而，由于没有直接提供真实的周期信息，MyModel 可能在周期的识别和利用上存在一定的误差，导致预测的 MSE 较高。

值得注意的是，SparseTSF 在实验中表现出了其模型的性能上限，因为实际应用中通常无法获得时间序列的真实周期。在这种情况下，SparseTSF 的性能可能会下降，因为需要对周期进行估计或设置默认值。

相反，MyModel 的优势在于无需提前知道真实周期，能够通过模型自身自动寻找周期性，这使其在实际应用中更具优势。虽然在本次实验中 MSE 略高于 SparseTSF，但在无法获取真实周期的情况下，MyModel 可能会有更好的性能。

E.5. 消融实验

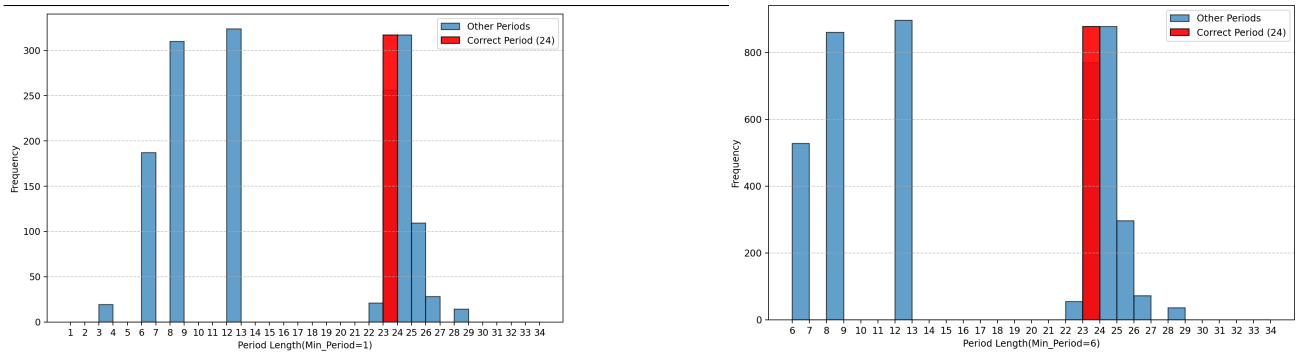
为了进一步评估 MyModel 和 SparseTSF 在未知真实周期情况下的性能，我们进行了消融实验。在本次实验中，我们假设真实周期未知，使用一个较小的值分别作为 MyModel 的最小周期超参数和 SparseTSF 的周期超参数。实验仍然在四个数据集（ETTh1、ETTh2、ETTm1、ETTm2）上进行，预测长度为 96、192、336 和 720。两种模型的均方误差（MSE）结果汇总在表 18 中。

从表 18 可以看出，当真实周期未知且使用较小的周期超参数时，两种模型的性能表现出了不同的趋势。

对于周期较短的 ETTm1 和 ETTm2 数据集，其真实周期为 4，与我们使用的较小周期超参数接近，因此 SparseTSF 的性能没有明显下降。以 **ETTm2** 为例，SparseTSF 在预测长度为 96 时的 MSE 为 0.1629，性能与已知真实周期时相近。而 MyModel 在这些数据集上，由于能够自动适应周期变化，性能也保持较为稳定。在预测长度为 96 时，MSE 为 0.2866，与 SparseTSF 相比略高，但仍在可接受范围内。

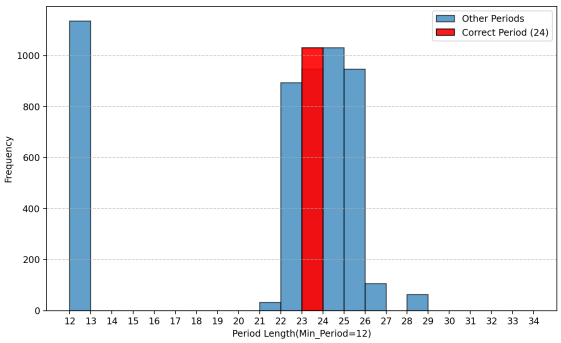
然而，对于周期较长的 ETTh1 和 ETTh2 数据集，使用较小的周期超参数会导致模型无法充分捕捉数据中的周期性特征。SparseTSF 的性能受到一定影响，尤其在较长的预测长度下。以 **ETTh2** 数据集为例，SparseTSF 的 MSE 从预测长度为 96 时的 0.3157 上升到预测长度为 720 时的 0.4943，增幅明显。相比之下，MyModel 在短预测长度下表现出一定的优势。在 **ETTh1** 数据集上，预测长度为 96 时，MyModel 的 MSE 为 0.3811，略优于 SparseTSF 的 0.3902。这表明 MyModel 在周期未知且周期较长的数据上，具有更好的短期预测能力。

总体来看，MyModel 在周期未知的情况下，通过

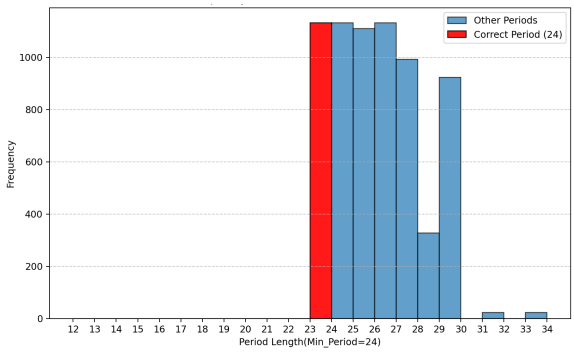


(a) Min_Period = 1

(b) Min_Period = 6



(c) Min_Period = 12



(d) Min_Period = 24

图 22. 不同 Min_Period 设置下模型的周期提取结果

数据集	MyModel				SparseTSF			
	96	192	336	720	96	192	336	720
ETTh1	0.3672	0.4203	0.4525	0.4881	0.362	0.403	0.434	0.426
ETTh2	0.3313	0.4194	0.5080	0.8228	0.294	0.339	0.359	0.312
ETTm1	0.4002	0.4140	0.4301	0.4759	0.312	0.347	0.367	0.419
ETTm2	0.3424	0.3509	0.3750	0.4556	0.165	0.218	0.272	0.352
Weather	0.3066	0.2984	0.3233	0.3626	0.169	0.214	0.257	0.321
Traffic	0.3914	0.3989	0.4125	0.4520	0.389	0.398	0.411	0.448

表 17. 在已知真实周期情况下 MyModel 和 SparseTSF 的 MSE 比较

使用较小的最小周期超参数，依然能够适应不同数据的周期性，表现出较好的鲁棒性，尤其在短预测长度和周期较长的数据集上，具有一定优势。而 SparseTSF 对周期超参数较为敏感，当周期设定偏离真实值时，性能可能受到较大影响，特别是在长预测长度的情况下。

这次消融实验表明，在实际应用中，如果无法准确获得时间序列的真实周期，MyModel 可能提供更稳定的性能，特别是在处理周期较长且需要短期预测的任务时。未来的工作可以进一步优化 MyModel 的周期识别机制，以增强其在不同场景下的适应性和预测精度。

F. 问题与解决方法

在将模型从 PyTorch 迁移到 MindSpore 时，可能会遇到多种问题，以下是详细说明及解决方法。

F.1. API 差异

问题：

- PyTorch 和 MindSpore 在 API 上有差异，比如模

数据集	MyModel				SparseTSF			
	96	192	336	720	96	192	336	720
ETTh1	0.3811	0.4215	0.4574	0.5221	0.3902	0.4459	0.4647	0.4510
ETTh2	0.3177	0.3969	0.5199	0.8058	0.3157	0.3479	0.4083	0.4943
ETTm1	0.4144	0.4160	0.4561	0.6581	0.3112	0.3398	0.3687	0.4158
ETTm2	0.2866	0.3214	0.3597	0.4316	0.1629	0.2154	0.2682	0.3517

表 18. 在未知真实周期情况下，MyModel 和 SparseTSF 的 MSE 比较

型定义、优化器使用等。直接迁移时，代码可能会因为调用不同的 API 而出错。

解决方法：

- 学习 MindSpore 的 API 文档，替换 PyTorch 的 API 调用，特别是模型结构（如 `nn.Module` 转为 `nn.Cell`）和优化器部分。

F.2. 张量操作差异

问题：

- PyTorch 和 MindSpore 的张量操作不同，可能会导致维度、广播等方面的错误。

解决方法：

- 对照两者的张量操作，找到对应的函数并替换。
- 注意检查维度和形状的兼容性。

F.3. 数据加载

问题：

- PyTorch 使用 `DataLoader` 加载数据，而 MindSpore 有自己的数据加载模块，可能导致加载方式不兼容。

解决方法：

- 使用 MindSpore 的 `mindspore.dataset` 模块替代 PyTorch 的 `DataLoader`，并调整数据预处理方式。

F.4. 模型保存与加载

问题：

- PyTorch 的 `torch.save()` 和 `torch.load()` 与 MindSpore 的存储方式不同，直接迁移时可能无法加载权重。

解决方法：

- 使用 MindSpore 的 `save_checkpoint()` 和 `load_checkpoint()` 方法来保存和加载模型。

F.5. 使用 **AI** 工具辅助迁移

问题：

- 在迁移过程中，人工分析和修改代码可能会耗费大量时间，尤其是对于大型项目或复杂模型。

解决方法：

- 在代码分析过程中，可以使用 ChatGPT 等 AI 工具来帮助识别代码中的潜在问题或提出建议。
- 具体使用过程：
  1. 将迁移中遇到的 PyTorch 代码片段输入 AI 工具，询问如何将其转换为 MindSpore 的等效实现。
  2. 当转换出的代码有问题时，询问 AI 哪里有问题。
  3. AI 工具可以帮助识别常见的代码迁移问题，如数据加载、模型构建等方面的差异，并提供相应的代码示例或建议。
  4. 结合 AI 工具的帮助，可以更高效地解决 API 差异、函数调用问题和性能调优，提升迁移过程的效率。

A. 附录 1：模型迁移关键代码（王为；陈兴浩完善）

```
class MyWithLossCell(nn.Cell):
        def __init__(self, backbone, loss_fn, configs):
            super(MyWithLossCell, self).__init__(auto_prefix=False)
            self.backbone = backbone
            self.loss_fn = loss_fn
            self.configs = configs
            self.pred_len=self.configs.pred_len
            self.features=self.configs.features

        def construct(self, data, label):
            outputs = self.backbone(data)
            f_dim = -1 if self.features == 'MS' else 0
            outputs = outputs[:, -self.pred_len:, f_dim:]
            label = label[:, -self.pred_len:, f_dim:]
            loss = self.loss_fn(outputs, label)
            return loss
class Exp_Main:
    def __init__(self, args):
        self.configs=args
        self.model = SparseTSF(self.configs)
        self.set_random_seed(args.seed)

        # 定义损失函数和优化器
        self.criterion = nn.MSELoss()

        self.optimizer = nn.Adam(self.model.trainable_params(), learning_rate=self.configs.learning_rate)

        # 将模型和损失函数封装到 WithLossCell 中

        self.net_with_loss = MyWithLossCell(self.model, self.criterion, self.configs)

        # 使用 TrainOneStepCell 封装网络和优化器
        self.train_network = nn.TrainOneStepCell(self.net_with_loss, self.optimizer)
    def adjust_learning_rate(self, epoch):
        """根据当前的 epoch 调整学习率"""
        if self.configs.lradj == 'type1':
            lr_adjust = {epoch: self.configs.learning_rate * (0.5 ** ((epoch - 1) // 1))}
        elif self.configs.lradj == 'type2':
            lr_adjust = {
                2: 5e-5, 4: 1e-5, 6: 5e-6, 8: 1e-6,
                10: 5e-7, 15: 1e-7, 20: 5e-8
            }
        elif self.configs.lradj == 'type3':
            lr_adjust = {epoch: self.configs.learning_rate if epoch < 3 else self.configs.learning_rate * (0.8
```

```
                ** ((epoch - 3) // 1))}
        elif self.configs.lradj == 'constant':
            lr_adjust = {epoch: self.configs.learning_rate}
        elif self.configs.lradj == '3':
            lr_adjust = {epoch: self.configs.learning_rate if epoch < 10 else self.configs.learning_rate *
                0.1}
        elif self.configs.lradj == '4':
            lr_adjust = {epoch: self.configs.learning_rate if epoch < 15 else self.configs.learning_rate *
                0.1}
        elif self.configs.lradj == '5':
            lr_adjust = {epoch: self.configs.learning_rate if epoch < 25 else self.configs.learning_rate *
                0.1}
        elif self.configs.lradj == '6':
            lr_adjust = {epoch: self.configs.learning_rate if epoch < 5 else self.configs.learning_rate * 0.1}
        elif self.configs.lradj == 'TST':
            lr_adjust = {epoch: self.optimizer.learning_rate}  # Assuming this is how you access the learning
                rate

        if epoch in lr_adjust.keys():
            lr = lr_adjust[epoch]
            self.optimizer.learning_rate = ms.Tensor(lr, ms.float32)  # Update learning rate
            print(f'Updating learning rate to {lr:.6f}')

    def set_random_seed(self,seed):
        np.random.seed(seed)
        ms.set_seed(seed)
    def _get_data(self, flag):
            data_set, data_loader = get_numpy_data(self.configs, flag)
            return data_set, data_loader


    def train(self):
        # Load data
        if self.configs.use_GRAPH_MODE:
            context.set_context(mode=context.GRAPH_MODE)
        else:
            context.set_context(mode=context.PYNATIVE_MODE)

        if self.configs.use_gpu and ms.context.get_context("device_target") == "GPU" :
            context.set_context(device_target="GPU")
            print("Using GPU for training.")

        else:
            context.set_context(device_target="CPU")
            print("Using CPU for training.")
```

```
        if self.configs.use_amp:
            context.set_context(enable_auto_mixed_precision=True)
        train_data, train_loader = self._get_data(flag='train')
        vali_data, vali_loader = self._get_data(flag='val')
        test_data, test_loader = self._get_data(flag='test')

        self.train_network.set_train()

        best_vali_loss = float('inf')  # Initialize best validation loss
        learning_rate = self.configs.learning_rate  # Initial learning rate

        # Initialize early stopping
        patience = self.configs.patience if hasattr(self.configs, 'patience') else 7  # Default patience is 7
        early_stopping = EarlyStopping(patience=patience, min_delta=0)

        # Create directory for saving models
        model_save_dir = self.configs.checkpoints  # Change the directory name to 'checkpoint'
        os.makedirs(model_save_dir, exist_ok=True)  # Create directory if it doesn't exist

        # Set context for mixed precision if enabled




        for epoch in range(self.configs.train_epochs):
            epoch_time = time.time()
            iter_count = 0
            train_loss_list = []  # Reset at the beginning of each epoch
            epoch_time = time.time()
            train_loss = 0
            batches = 0
            train_data, train_loader = self._get_data(flag='train')
            vali_data, vali_loader = self._get_data(flag='val')
            test_data, test_loader = self._get_data(flag='test')
            # Training phase
            for i, (batch_x, batch_y, batch_x_mark, batch_y_mark) in enumerate(train_loader):
                iter_count += 1
                batches += 1
                # Convert to MindSpore Tensors
                batch_x = ms.Tensor(batch_x, ms.float32)
                batch_y = ms.Tensor(batch_y, ms.float32)

                # Compute loss
                loss = self.train_network(batch_x, batch_y)
                loss_value = loss.asnumpy()
```

```
130                 train_loss_list.append(loss_value)
131                 train_loss += loss_value
132 
133             avg_train_loss = train_loss / batches
134 
135             # Validation phase
136             vali_loss = 0
137             vali_batches = 0
138             for j, (vali_x, vali_y, vali_x_mark, vali_y_mark) in enumerate(vali_loader):
139                 vali_batches += 1
140                 vali_x = ms.Tensor(vali_x, ms.float32)
141                 vali_y = ms.Tensor(vali_y, ms.float32)
142 
143                 # Compute validation loss
144                 vali_loss_value = self.net_with_loss(vali_x, vali_y)
145                 vali_loss += vali_loss_value.asnumpy()
146 
147             avg_vali_loss = vali_loss / vali_batches
148 
149             # Test phase (optional)
150             test_loss = 0
151             test_batches = 0
152             for k, (test_x, test_y, test_x_mark, test_y_mark) in enumerate(test_loader):
153                 test_batches += 1
154                 test_x = ms.Tensor(test_x, ms.float32)
155                 test_y = ms.Tensor(test_y, ms.float32)
156 
157                 # Compute test loss
158                 test_loss_value = self.net_with_loss(test_x, test_y)
159                 test_loss += test_loss_value.asnumpy()
160 
161             avg_test_loss = test_loss / test_batches
162             print("Epoch: {} cost time: {}".format(epoch + 1, time.time() - epoch_time))
163             # Print epoch summary
164             f = open("result.txt", 'a')
165             print(f'Epoch: {epoch + 1}, Steps: {batches} | '
166                   f'Train Loss: {avg_train_loss:.7f} Vali Loss: {avg_vali_loss:.7f} Test Loss: {avg_test_loss
                          :.7f}')
167             f.write(f'Epoch: {epoch + 1}, Steps: {batches} | '
168                   f'Train Loss: {avg_train_loss:.7f} Vali Loss: {avg_vali_loss:.7f} Test Loss: {avg_test_loss
                          :.7f}')
169             f.write('\n')
170             self.adjust_learning_rate(epoch + 1)
171 
172             # Check for improvement
```

```
173             if avg_vali_loss < best_vali_loss:
174                 print(f'Validation loss decreased ({best_vali_loss:.6f} --> {avg_vali_loss:.6f}). Saving model
                        ...')
175                 f.write(f'Validation loss decreased ({best_vali_loss:.6f} --> {avg_vali_loss:.6f}). Saving
                        model ...')
176                 f.write('\n')
177                 best_vali_loss = avg_vali_loss
178                 # Save the model here, overwriting the previous one
179                 model_name = f"{self.configs.model}_{self.configs.data}_{self.configs.pred_len}_best_model.
                        ckpt"  # 拼接 model 字符串
180                 model_path = os.path.join(model_save_dir, model_name)
181
182                 ms.save_checkpoint(self.train_network, model_path)
183             else:
184                 print(f'Validation loss did not improve.')
185                 f.write(f'Validation loss did not improve.')
186                 f.write('\n')
187             # Call early stopping
188             if early_stopping(avg_vali_loss):
189                 break  # Exit the training loop
190
191         print("Training complete.")
192         f.write("Training complete.")
193         f.write('\n')
194         f.close()
195
196
197     def test(self):
198         test_data,test_loader=self._get_data(flag='test')
199         test_loss = 0
200         test_batches = 0
201         all_outputs = []
202         all_labels = []
203
204         for k, (test_x, test_y, test_x_mark, test_y_mark) in enumerate(test_loader):
205             test_batches += 1
206             test_x = ms.Tensor(test_x, ms.float32)
207             test_y = ms.Tensor(test_y, ms.float32)
208
209             # 计算测试损失
210             test_loss_value = self.net_with_loss(test_x, test_y)
211             test_loss += test_loss_value.asnumpy()
212
213             # 记录输出和标签
214             outputs = self.model(test_x)
```

```
            all_outputs.append(outputs.asnumpy())
            all_labels.append(test_y.asnumpy())

        # 计算平均测试损失
        avg_test_loss = test_loss / test_batches
        print(f'Average Test Loss (MSE): {avg_test_loss:.6f}')

        # 计算 MSE, RSE, MAE
        all_outputs = np.concatenate(all_outputs, axis=0)
        all_labels = np.concatenate(all_labels, axis=0)


        all_labels = all_labels[:, -self.configs.pred_len:, :]
        # 计算 MSE
        mse = np.mean((all_outputs - all_labels) ** 2)

        # 计算 RSE
        rse = np.sqrt(np.sum((all_outputs - all_labels) ** 2)) / np.sqrt(np.sum(all_labels ** 2))

        # 计算 MAE
        mae = np.mean(np.abs(all_outputs - all_labels))

        f = open("result.txt", 'a')
        f.write(setting + "  \n")
        f.write('mse:{}, mae:{}, rse:{}'.format(mse, mae, rse))
        f.write('\n')
        f.write('\n')
        f.close()
        print(f'MSE: {mse:.6f}, RSE: {rse:.6f}, MAE: {mae:.6f}')
```

B. 附录 2:MyModel 模型代码（王为）

```
def get_main_periods(
    data_batch,
    sample_rate=1.0,
    min_period=24,
    max_period=96,
    main_feature_index=0,
    main_feature_weight=2.0,
    penalty_exponent=1.2,  # 调整此参数来控制幂指数
    alpha=0.035,           # 可选的指数衰减参数，默认不使用指数衰减
    device='cuda'
):
    """
    该函数接受一个时间序列批次，计算所有特征的傅里叶变换，
```

```
    找到幅度最大的 5 个频率分量，并返回对应的周期（整数形式），
    优先选取更接近最小周期的值，并确保周期在指定的最小和最大范围内。

    参数:
    - data_batch: 输入数据，形状为 (batch_size, T, channels)
    - sample_rate: 采样率，默认为 1.0
    - min_period: 最小周期，默认为 24
    - max_period: 最大周期，默认为 96
    - main_feature_index: 主特征的索引（从 0 开始）
    - main_feature_weight: 主特征的权重，默认为 2.0
    - penalty_exponent: 惩罚的幂指数，默认为 1.2，可调整
    - alpha: 指数衰减参数，如果为 None，则不使用指数衰减，默认为 None
    - device: 设备，默认为 'cuda'
    """
    # 检查是否有可用的 GPU，如果没有则回退到 CPU
    if device == 'cuda' and not torch.cuda.is_available():
        print("GPU 不可用，切换到 CPU。")
        device = 'cpu'

    # 确保数据在正确的设备上
    data_batch = data_batch.to(device)
    batch_size, T, channels = data_batch.shape

    # 对时间维度进行傅里叶变换
    fft_result = torch.fft.fft(data_batch, n=T, dim=1)  # 形状：(batch_size, T, channels)

    # 计算频率序列
    freqs = torch.fft.fftfreq(T, d=1 / sample_rate, device=device)  # 形状：(T,)
    positive_freqs = freqs[:T // 2]  # 只考虑正频率部分，形状：(T//2,)

    # 提取正频率部分的傅里叶变换结果
    fft_positive = fft_result[:, :T // 2, :]  # 形状：(batch_size, T//2, channels)

    # 计算幅度谱并对批次维度取平均，形状：(T//2, channels)
    magnitudes = torch.abs(fft_positive).mean(dim=0)

    # 增加主特征的权重
    magnitudes[:, main_feature_index] *= main_feature_weight

    # 聚合所有特征的幅度谱，形状：(T//2,)
    aggregated_magnitudes = magnitudes.mean(dim=1)

    # 计算对应的周期，并取整数，形状：(T//2,)
    periods = (1.0 / positive_freqs).round().int()

```

```
    # 处理可能的无穷大和 NaN 值（零频率会导致无穷大的周期）
    periods = torch.where(
        torch.isinf(periods) | torch.isnan(periods),
        torch.tensor(T, device=device, dtype=periods.dtype),
        periods
    )

    # 筛选周期在指定范围内的频率分量
    mask = (periods >= min_period) & (periods <= max_period)
    periods = periods[mask]
    aggregated_magnitudes = aggregated_magnitudes[mask]

    # 如果没有符合条件的频率分量，抛出异常或返回默认值
    if aggregated_magnitudes.numel() == 0:
        raise ValueError("在指定的周期范围内没有频率分量，请调整 min_period 和 max_period。")

    # 调整幅度以惩罚较长的周期，控制惩罚力度
    if alpha is not None:
        # 使用指数衰减函数来调整幅度
        adjusted_magnitudes = aggregated_magnitudes * torch.exp(-alpha * (periods.float() - min_period))
    else:
        # 使用幂指数来调整幅度，降低惩罚力度
        adjusted_magnitudes = aggregated_magnitudes / (periods.float() ** penalty_exponent)

    # 找到调整后幅度最大的 5 个频率分量
    topk = torch.topk(adjusted_magnitudes, k=5, largest=True)
    top_indices = topk.indices  # 形状：(5,)

    # 对应的周期
    top_periods = periods[top_indices]  # 形状：(5,)

    # 将周期按从小到大排序
    top_periods, _ = torch.sort(top_periods)

    # 返回形状为 (5,) 的张量，包含更接近最小周期的 5 个主要周期（整数）
    return top_periods
class Model(nn.Module):
    def __init__(self, configs):
        super(Model, self).__init__()

        # 获取参数
        self.seq_len = configs.seq_len
        self.pred_len = configs.pred_len
        self.enc_in = configs.enc_in
        self.period_len = configs.period_len
```

```
104         self.data = configs.data
105 
106         self.seg_num_x = self.seq_len // self.period_len
107         self.seg_num_y = self.pred_len // self.period_len
108 
109         self.conv1d = nn.Conv1d(
110             in_channels=1,
111             out_channels=1,
112             kernel_size=1 + 2 * (self.period_len // 2),
113             stride=1,
114             padding=self.period_len // 2,
115             padding_mode="zeros",
116             bias=False
117         )
118         self.conv1d2 = nn.Conv1d(
119             in_channels=5,
120             out_channels=1,
121             kernel_size=5,
122             stride=1,
123             padding=2
124         )
125         self.linear = nn.Linear(self.seg_num_x, self.seg_num_y, bias=False)
126 
127     def forward(self, x):
128         batch_size = x.shape[0]
129 
130         # 获取主要周期
131         main_periods = get_main_periods(
132             x,
133             min_period=self.period_len,
134             main_feature_index=self.enc_in - 1
135         )
136 
137         # 归一化和维度变换
138         seq_mean = torch.mean(x, dim=1, keepdim=True)  # (batch_size, 1, channels)
139         x = (x - seq_mean).permute(0, 2, 1)  # (batch_size, channels, seq_len)
140         x = self.conv1d(x.reshape(-1, 1, self.seq_len)).reshape(-1, self.enc_in, self.seq_len) + x
141 
142         z = []  # 用于存储计算得到的张量 y
143 
144         for i in range(5):
145             # 提取主要周期的标量值
146             period = int(main_periods[i].item())
147 
148             # 计算分段数量
```

```
            seg_num_x1 = self.seq_len // period

            # 计算余数并确保为整数
            remainder = int(self.seq_len % period)

            if remainder != 0:
                x_i = x[:, :, :-remainder]  # 截断多余的元素
            else:
                x_i = x

            x_i = x_i.reshape(-1, seg_num_x1, period).permute(0, 2, 1)
            x_i = x_i.to(torch.float32)

            # 截断第二维度
            if x_i.shape[1] > self.period_len:
                x_i = x_i[:, :self.period_len, :]

            # 填充第三维度
            if x_i.shape[2] < self.seg_num_x:
                padding_size = self.seg_num_x - x_i.shape[2]
                x_i = torch.nn.functional.pad(x_i, (0, padding_size))

            x_i = x_i.reshape(batch_size * self.enc_in, self.period_len, self.seg_num_x)
            # 稀疏预测
            y = self.linear(x_i)  # (batch_size * channels, period_len, m)

            # 上采样并恢复形状
            y = y.permute(0, 2, 1).reshape(batch_size, self.enc_in, self.pred_len)

            # 反归一化
            y = y.permute(0, 2, 1) + seq_mean  # (batch_size, self.pred_len, channels)
            z.append(y)

        # 将结果堆叠并调整维度
        z = torch.stack(z)  # (5, batch_size, self.pred_len, channels)
        z = z.permute(1, 3, 2, 0)  # (batch_size, channels, self.pred_len, 5)

        # 合并维度并应用卷积层
        z = z.contiguous().view(batch_size, self.enc_in * self.pred_len, 5)
        z = z.permute(0, 2, 1)  # (batch_size, 5, enc_in * pred_len)
        z = z.to(torch.float32)

        output = self.conv1d2(z)

        # 恢复输出形状
```

```
194         output = output.squeeze(1)  # (batch_size, enc_in * pred_len)
195 
196         output = output.view(batch_size, self.enc_in, self.pred_len)  # (batch_size, channels, pred_len)
197         output = output.permute(0, 2, 1)
198         return output
```