软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(2):584−600 [doi: 10.13328/j.cnki.jos.007451] [CSTR: 32375.14.jos.007451]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于领域知识图谱的框架间 AI 源码自动迁移*

丁 嵘[1], 刘屹洲[2], 王雨倩[2], 李一錡[1]

[1](北京航空航天大学 人工智能研究院, 北京 100191)

[2](北京航空航天大学 计算机学院, 北京 100191)

通信作者: 丁嵘, E-mail: dingr@buaa.edu.cn

摘　要: 作为人工智能的基础设施, 深度学习框架已经成为人工智能实现跨越发展的重要突破口. 但是由于缺乏统一标准, 不同框架的兼容水平较差. 忠实模型转换通过将源模型迁移为另一种目标框架下的等价模型, 来增强框架间的互操作性. 然而, 深度学习框架数量较多且相互间差异较大, 并且自主框架的需求逐渐增多, 互相转换成本较高. 因此, 提出基于领域知识图谱的框架间 AI 源码自动迁移方法. 该方法基于领域知识图谱和抽象语法树来系统地处理迁移挑战, 首先将源代码转换为特定的抽象语法树, 提取通用依赖信息和特定算子信息, 然后再利用存储在领域知识图谱中的框架间算子及参数映射关系来迁移到目标框架下, 形成目标框架下的目标模型代码, 大大降低了工程复杂度. 对比同类型的代码迁移工具, 所提方法可以在国内外流行深度学习框架如 PyTorch、PaddlePaddle 和 MindSpore 之间进行互相迁移, 达到了较好的成熟度和质量, 部分成果已经开源到百度官方迁移工具 PaConvert 中.

关键词: 深度学习框架; 代码迁移; 知识图谱; 抽象语法树

中图法分类号: TP311

中文引用格式: 丁嵘, 刘屹洲, 王雨倩, 李一錡. 基于领域知识图谱的框架间AI源码自动迁移. 软件学报, 2026, 37(2): 584–600. http://www.jos.org.cn/1000-9825/7451.htm

英文引用格式: Ding R, Liu YZ, Wang YQ, Li YQ. Automatic Migration of AI Source Code Between Frameworks Based on Domain Knowledge Graph. Ruan Jian Xue Bao/Journal of Software, 2026, 37(2): 584–600 (in Chinese). http://www.jos.org.cn/1000-9825/7451.htm

Automatic Migration of AI Source Code Between Frameworks Based on Domain Knowledge Graph

DING Rong[1], LIU Yi-Zhou[2], WANG Yu-Qian[2], LI Yi-Qi[1]

[1](Institute of Artificial Intelligence, Beihang University, Beijing 100191, China)

[2](School of Computer Science and Engineering, Beihang University, Beijing 100191, China)

Abstract: As the foundation of AI, deep learning frameworks play a vital role in driving the rapid progress of AI technologies. However, due to the lack of unified standards, compatibility across different frameworks remains limited. Faithful model transformation enhances interoperability by converting a source model into an equivalent model in the target framework. However, the large number and diversity of deep learning frameworks, combined with the increasing demand for custom frameworks, lead to high conversion costs. To address this issue, this study proposes an automatic AI source code migration method between frameworks based on a domain knowledge graph. The method integrates domain knowledge graphs and abstract syntax trees to systematically manage migration challenges. First, the source code is transformed into a framework-specific abstract syntax tree, from which general dependency information and operator-specific details are extracted. By applying the operator and parameter mappings stored in the domain knowledge graph, the code is migrated to the target framework, generating equivalent target model code while significantly reducing engineering complexity. Compared with existing code migration tools, the proposed method supports mutual migration among widely used deep learning frameworks, such as PyTorch,

* 基金项目: 科技创新 2030—“新一代人工智能”重大项目 (2021ZD0110600); 百度合作项目 (Z231100010323007); 华为合作项目 (22081500805582B)
收稿时间: 2024-06-06; 修改时间: 2024-08-07, 2024-10-19, 2025-03-25; 采用时间: 2025-04-23; jos 在线出版时间: 2025-09-24
CNKI 网络首发时间: 2025-09-25

PaddlePaddle, and MindSpore. The approach has proven to be both mature and reliable, with part of its implementation open-sourced in Baidu's official migration tool, PaConvert.

Key words: deep learning framework; code migration; knowledge graph; abstract syntax tree (AST)

1 引 言

人工智能是新一轮科技革命和产业变革的重要驱动力量, 正在引领全球经济发展. 其中, 深度学习框架是新一轮人工智能跨越发展的核心引擎, 也是构建人工智能全栈生态的关键. 因此, 深度学习框架引起了学术界和工业界的高度重视, 全球科技巨头纷纷布局, 各种开源深度学习框架如雨后春笋般层出不穷, 如 PyTorch、MindSpore 和 PaddlePaddle[1,2]等.

然而, 各种深度学习框架种类繁多、接口多样, 生态环境极其复杂. 一方面, 各种框架间的异构设计难以形成协同发展、群智服务的良好态势, 造成了一个个深度学习框架的生态孤岛, 给深度学习框架技术进步、协同发展、应用深化带来了巨大的挑战. 另一方面, 国外企业在深度学习框架方面凭借先发优势已经建立了强而有力的产业生态, 打造了众多行业事实标准. 而现阶段我国相关算法和创新型应用多以国外底层软硬件平台构建, 自主产业生态薄弱, "卡脖子"问题突出. 特别是国产厂商技术路线多样, 自主深度学习框架应用不够丰富, 相互间的兼容适配水平较差, 缺乏有效的互操作性与互联互通机制, 严重制约了我国自主技术生态体系的构建. 例如, 百度的昆仑芯片只支持飞桨自身以及国外 PyTorch 和 TensorFlow 等顶层框架的模型部署, 但不支持华为 MindSpore 框架的模型部署; MindSpore 外的其他深度学习框架下的模型同样不能直接在昇腾处理器上进行训练. 为了充分利用昆仑和昇腾等国内 AI 芯片的算力来提升训练性能, 研究人员需要将源模型进行迁移, 使迁移后的模型能够高效运行在目标处理器上.

为了解决上述问题, 并推动我国自主深度学习框架的快速发展, 寻求一种实现深度学习框架间模型代码自动迁移的方法就变得至关重要.

本文针对常见的神经网络模型代码在不同的深度学习框架间设计并实现出一种有效可行的代码自动转换方案, 也即使用某一个深度学习框架编写的模型代码能够迁移到另一种深度学习框架上继续进行编写或运行. 主要设计并实现了不同深度学习框架间算子差异信息的领域知识图谱、神经网络模型代码解析技术、模型代码迁移的中间件、神经网络模型代码生成技术以及不同深度学习框架间的模型代码自动转换的程序及测试.

本文主要贡献如下: (1) 提出了基于领域知识图谱的深度学习框架间模型代码迁移方法, 极大地降低了代码迁移的成本与复杂度. (2) 提出了深度学习框架知识的抽取方法, 从不同数据来源中抽取出各深度学习框架的算子信息, 特别是不同框架间的算子及参数差异的映射信息, 最终构建出领域知识图谱, 以辅助模型迁移程序进行模型代码的迁移. (3) 提出了一种基于抽象语法树 (abstract syntax tree, AST) 的模型代码迁移中间件, 简化了不同框架间的模型代码迁移流程, 提高模型代码迁移的性能. (4) 本文在单语和并行语料库上进行了 3 种深度学习框架间模型代码互相迁移的对比实验, 并将迁移前后的模型通过深度学习训练过程的对比实验来进一步验证模型代码迁移的有效性. 实验结果表明, 本文提出的方法具有较好的迁移效果, 满足功能正确性和精度要求, 部分成果已经开源到百度的迁移工具 PaConvert (https://github.com/PaddlePaddle/PaConvert) 中.

本文第 2 节介绍已有的相关工作. 第 3 节分析关键问题以及本文方法的整体框架. 第 4 节介绍本文方法的具体实现细节. 第 5 节介绍为了验证方法有效性而开展的实验以及实验设置和实验结果. 最后, 第 6 节对本文工作进行总结.

2 相关工作

微软等提出深度学习模型标准"开放神经网络交换 (ONNX)"格式, 推动了算法模型在不同机器学习框架间的应用迁移[3]. ONNX 定义了一组与环境、平台均无关的标准格式来增强各种 AI 模型的可交互性. 它使模型能够在一个框架中进行训练, 然后转移到另一个框架进行推理. 因此源框架的深度学习模型可以先转换成 ONNX 模型, 然后再将 ONNX 模型转换为目标框架的深度学习模型.

2018 年, 微软开源了 MMdnn[4], 一个开源的、综合的、可靠的深度学习模型转换工具. 它从编译的角度实现了一个可扩展的转换架构, 目的是支持 Caffe、Keras、MXNet、TensorFlow、CNTK、PyTorch 和 CoreML 等框架之间进行模型转换[5−7]. MMdnn 参考 ONNX 框架采用了一种新颖的基于统一中间表示 (IR) 的方法来系统地处理转换, 首先将源模型转换为基于简单图的 IR 表示的中间计算图, 然后再转换为目标框架格式, 大大降低了工程复杂度.

2019 年, 百度发布了飞桨生态下的模型转换工具 X2Paddle. 不同于上述的 ONNX 和 MMdnn 可以将不同深度学习框架的模型进行互相转换, X2Paddle 是单向地将 Caffe、TensorFlow、PyTorch 等主流深度学习框架的模型转换为 PaddlePaddle 模型, 实现模型的跨框架迁移和部署. 它还支持将 PyTorch 项目中的 Python 代码 (包括训练、预测) 一键转为基于飞桨框架的项目代码, 帮助开发者快速迁移项目.

X2Paddle 的核心原理是将模型转换为 PaddlePaddle 框架的计算图和参数, 利用 PaddlePaddle 框架的高效计算能力和自动优化技术, 实现模型的高效推理和部署. 具体可以分为以下几个步骤: 第一, X2Paddle 首先加载原始模型, 并根据模型的类型和结构构建出对应的计算图. 第二, 根据原始模型的计算图, X2Paddle 会自动将其转换为 PaddlePaddle 框架的计算图, 并将原始模型的参数转换为 PaddlePaddle 格式. 第三, 执行动转静操作, 保存静态图模型到磁盘中.

但是, 基于 X2Paddle 进行代码转换需要使用者手动进行转换前的代码预处理以及转换后的代码后处理. 例如, 由于部分 PyTorch 操作是目前 PaddlePaddle 暂不支持的操作, 因此在转换前需要使用者手动将这部分操作去除或者修改, 例如不支持 TensorBoard、自动下载模型等. 此外, PaddlePaddle 在使用上有部分限制, 使用者需要在转换后手动修改代码, 例如自定义 Dataset 必须继承自 paddle.io.Dataset、部分情况下 DataLoader 的 num_worker 只能为 0 等. 这些需要调用者手动进行预处理和后处理的操作也极大地增加了开发者的负担.

2020 年, MindSpore 推出了 MindConverter, 它是一款深度学习模型迁移工具, 借助华为 ModelArts 深度学习平台的高效计算能力和自动化工具, 可以快速迁移和部署 PyTorch (ONNX) 或 TensorFlow (PB) 模型到 MindSpore 框架下. 模型文件 (ONNX/PB) 包含网络模型结构 (network) 与权重信息 (weights), 迁移后将生成 MindSpore 框架下的模型定义脚本 (model.py) 与权重文件 (ckpt). MindConverter 的架构与 X2Paddle 类似, 也是单向地将 PyTorch (ONNX) 或 TensorFlow (PB) 模型快速迁移到 MindSpore 下使用, 同时提供基于抽象语法树的 PyTorch 脚本迁移.

MindConverter 的实现原理主要分为以下几个步骤: 第一, MindConverter 首先加载原始模型, 并根据模型的类型和结构构建出对应的计算图. 第二, MindConverter 将原始模型转换为中间表示格式如抽象语法树或图结构, 该格式可以方便地在不同深度学习框架之间进行转换. 第三, MindConverter 将中间表示格式的模型转换为 MindSpore 深度学习框架的模型. 最后, MindConverter 将转换后的模型导出为 MindSpore 深度学习框架的模型文件, 并可以直接在 MindSpore 框架上进行推理和部署.

但是, MindConverter 存在以下问题: 第一, 由于 MindConverter 模型转换工具本质为算子驱动, 所以在导出 ONNX 模型文件时, 需要 PyTorch 算子支持相应的 ONNX 算子, 并且对于 MindConverter 未维护的 ONNX 算子与 MindSpore 算子映射, 将会出现相应的算子无法转换的问题. 对于该类算子, 需要用户手动进行修改. 第二, 由于模型转换工具以推理模式加载 ONNX 文件, 可能会导致转换后算子丢失问题如 Dropout 算子丢失, 需要用户手动补齐. 第三, MindConverter 当前仅维护 1.6.0 和 1.7.0 两个版本, 后续维护工作也将逐步向 1.7.0 倾斜, 并且 MindConverter 从 1.9.0 开始将不再演进, 官网文档及代码也将逐步下架.

2022 年, 百度开源了新的代码转换工具 PaConvert, 该工具可以自动将其他深度学习框架下训练或推理的代码, 迁移到 Paddlepaddle 框架下的目标代码, 方便快速地进行模型代码迁移. 与 X2Paddle 不同, PaConvert 使用抽象语法树作为中间表示进行代码迁移, 首先将输入的模型代码解析为抽象语法树, 对其进行解析、遍历、匹配、编辑、替换和插入等各种操作, 然后得到基于 PaddlePaddle 的抽象语法树, 最后生成 PaddlePaddle 框架下的代码.

不同之处在于, PaConvert 使用人工将框架间算子差异信息存储在文本文件中, 极大地增加了开发人员的编写成本, 并且不容易进行版本的更新迭代. 此外, PaConvert 目前处于内测阶段, 仍然有若干算子暂时不支持迁移如 pad 算子、is_available 算子和 is_initialized 算子等.

综上所述, 大多数工具只支持将有限框架单向迁移至单个框架下, 例如 PaConvert 只支持将 PyTorch 框架下

的模型代码单向迁移到 PaddlePaddle 框架下, MindConverter 同样只支持将 PyTorch 或 TensorFlow 的模型转换成 MindSpore 下的模型, 无法形成框架间模型代码互相转换的良好态势. 此外, X2Paddle 等迁移工具针对每一对源框架和目标框架都需要手工维护相应的映射信息, 假设源框架数量为 M, 目标框架数量为 N, 则构建映射信息的复杂度为 O(MN), 成本高且扩展性差, 给深度学习框架技术进步和协同发展带来了巨大的阻碍和挑战.

3 方法框架

3.1 关键问题

定义深度学习模型为 M, 在源深度学习框架下代码为 C_1, 目标深度学习框架下代码为 C_2. 由于深度学习框架均会将代码所表征的模型形式化为有向无环图 (directed acyclic graph, DAG), 则该模型可以定义为:

$$M=\left\{V=\{u_i\}_{i=1}^{N},\ E=\{(u_i,u_j)\}_{i\neq j},\ P=\{p_i\}_{i=1}^{K}\right\} \tag{1}$$

其中, V 代表节点集合, 每个节点 u_i 表示调用深度学习框架中的某个算子, 定义为 $u_i.op$. E 代表有向边集合, 具体的, 一条有向边表示将 u_i 的输出张量作为输入张量传递给 u_j 节点, 并遵从只有在 u_i 节点运行完毕之后 u_j 节点才能开始执行. 模型 M 有一些没有父节点的节点, 例如从磁盘读取输入数据的节点, 这类节点提供初始值来激活深度学习的计算, 并将它们的输出张量的元组作为模型 M 的输入. 类似的, 模型 M 的输出由无后继节点的输出张量元组表示. P 代表超参数集合, 每个 p 都表示一个超参数, 如批量大小 (batch size)、学习率 (learning rate)、丢失率 (dropout rate) 等. 如前所述, 模型 M 本质上是一个可以被表示为 $\mathcal{F}_M$ 的数学函数.

假设 $\mathcal{X}_{i\in[1,m]}$, $\mathcal{Z}_{i\in[1,n]}$ 都是张量, 则张量元组 $\mathcal{X}=<\mathcal{X}_1,\mathcal{X}_2,\ldots,\mathcal{X}_m>$ 表示节点或算子的输入, 张量元组 $\mathcal{Z}=<\mathcal{Z}_1,\mathcal{Z}_2,\ldots,\mathcal{Z}_m>$ 表示节点或算子的输出. 令 $\mathcal{F}$ 为接收 m 维输入张量并返回 n 维输出张量的算子 op 的数学函数, 则 op 定义为:

$$<\mathcal{Z}_1,\mathcal{Z}_2,\ldots,\mathcal{Z}_m>=\mathcal{F}_{op}(<\mathcal{X}_1,\mathcal{X}_2,\ldots,\mathcal{X}_m>) \tag{2}$$

根据上述符号和参数的定义, 将深度学习模型代码迁移算法记为 $\mathcal{A}$. 采用 MMdnn 的定义, 如果 $\mathcal{A}$ 满足以下两个条件, 则是准确忠实的深度学习模型代码迁移算法.

(1) 语法合法性: 给定任意源框架深度学习代码 C_1 所表征的模型 $M_1=<V_1,E_1,P_1>$, 应该通过模型代码自动迁移算法 $\mathcal{A}$ 生成一个目标框架下的代码 C_2 所表征的模型 $M_2=<V_2,E_2,P_2>$:

$$M_1\vdash_{\mathcal{A}}M_2\wedge(P_2=P_1) \tag{3}$$

(2) 语义等价性: 给定任意有效输入的张量元组 $\mathcal{X}$, 原框架深度学习代码 C_1 所表征的模型 M_1 和经过准确忠实的模型代码自动迁移算法 $\mathcal{A}$ 生成的目标框架代码 C_2 所表征的模型 M_2 应该总是返回相同的结果 $\mathcal{Z}$:

$$\mathcal{Z}=\mathcal{F}_{M_1}(\mathcal{X})\vdash_{\mathcal{A}}\mathcal{Z}=\mathcal{F}_{M_2}(\mathcal{X}) \tag{4}$$

综上所述, 本文的目标就是设计出准确忠实的不同深度学习框架间模型代码的自动迁移方案. 如下代码 1–代码 3 所示是一些深度学习模型代码的示例, 通过对比不同深度学习框架下相同模型对应的神经网络代码, 可以对代码差异有更直观的认识, 并更好地理解代码迁移的思路.

代码 1. PyTorch 框架下的简单模型代码.

```
import torch
from torch import nn
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv_layer = nn.Conv2d(in_channels = 3, out_channels = 64,
            kernel_size = 3, padding = 1)
```

```
    def forward(self, x):
        o = self.conv_layer(x)
        return o
```

代码 2. PaddlePaddle 框架下的简单模型代码.

```
import paddle
from paddle import nn
class Model(nn.Layer):
    def __init__(self):
        super(Model, self).__init__()
        self.conv_layer = nn.Conv2d(in_channels = 3, out_channels = 64,
            kernel_size = 3, padding = 1)
    def forward(self, x):
        o = self.conv_layer(x)
        return o
```

代码 3. MindSpore 框架下的简单模型代码.

```
import mindspore
from mindspore import nn
class Model(nn.Cell):
    def __init__(self):
        super(Model, self).__init__()
        self.conv_layer = nn.Conv2d(in_channels = 3, out_channels = 64,
            kernel_size = 3, padding = 1, pad_mode = "pad")
    def construct(self, x):
        o = self.conv_layer(x)
        return o
```

通过对比上述不同框架下的模型代码可知, 本文的根本挑战在于以下几点.

第一, 神经网络模型的组网代码通常通过组装若干深度学习框架的算子来实现, 因此若要实现不同深度学习框架间模型代码的迁移, 则需要对这些算子进行转换. 然而, 不同深度学习框架的编程范式与组网方式不尽相同, 算子名称及其参数也具有一定的差异性, 如针对顺序的线性网络结构可以使用 Sequential 方式组网, 而针对一些比较复杂的网络结构, 可以使用 Layer 子类定义的方式来进行模型代码的编写等. 因此, 如果直接在字符串层面进行转换则需要考虑多种情况, 从而使得迁移程序十分复杂.

第二, 由第 2 节的分析可知, 目前工业界的深度学习神经网络模型代码迁移程序如 MindConverter 等将不同框架间的差异信息存储在表格或代码中, 导致知识的表达能力较弱, 知识与代码之间耦合性过高, 而深度学习框架更新换代较快的特性会使得上述方法难以及时迭代且更新成本过高, 实际使用起来较为困难. 具体来说, 存储在表格中的映射信息关系较为单一, 较难表达一对多或者多对多的映射关系, 例如 PyTorch 在进行数据处理时, 需要将数据集和采样器对象, 以及分批、混洗和并行等参数传入 DataLoader 接口, 以实现具有对应功能的数据迭代, 而 MindSpore 的采样器对象以及分批、混洗和并行等参数可以在创建数据集对象时传入, 也可以通过数据集内方法来定义, 不需要类似 PyTorch 一样采用额外的加载器, 类似的复杂信息适合使用表达能力更强的图结构进行存储.

此外, 存储在代码中的映射信息与迁移程序耦合程度过高, 只适用于特定场景和有限框架下的代码迁移, 无法适应后续对框架数量进行扩展.

此外, 模型构建步骤通常可以分为模型设计、准备数据、训练设置、应用部署和模型评估这 5 个阶段, 其中最核心的即为模型设计步骤. 该步骤决定了模型在解决问题方面的性能表现, 并且可以根据开发者需求来自定义选择数据集和训练配置参数等. 模型设计步骤由建模人员负责的部分主要包括网络结构设计、损失函数的指定以及优化算法的指定, 上述部分在不同深度学习框架下的差异主要体现在所调用的算子及其参数的不同, 且主要为一对一映射关系的算子, 所以本文的方法主要针对一对一关系的算子及参数节点进行迁移.

3.2 方法框架

基于上述分析, 本文拟采用知识图谱来存储框架间映射关系. 由于知识图谱以图的形式表示信息, 可以表示各种类型的实体和复杂的关系, 并且可以随着新知识的加入而动态更新以及进行知识融合; 同时知识图谱的结构化特性可以减少信息的冗余和噪声, 支持基于属性和关系等的复杂查询, 使得检索结果更加准确和高效, 具备较好的信息表达能力和检索能力. 同时, 知识图谱的图结构也适合与后续进行深度学习框架间知识融合, 提升映射信息的获取效率. 因此, 本文拟采用知识与代码相分离的思想, 将框架间算子差异信息存储在领域知识图谱中, 以降低迁移算法与映射信息之间的耦合性, 提高存储算子信息的灵活性及查询效率. 并在此基础上实现模型代码的自动迁移程序, 从而达到不同深度学习框架间模型代码自动迁移的目的.

此外, 考虑到代码翻译中的抽象语法树是语言相关且接近源代码语法结构的高级数据结构, 适合于快速的算子检测与修改, 因此本文利用抽象语法树作为中间数据结构来实现模型的自动迁移, 从而解决现有技术体系依赖人工进行代码迁移或依赖人工编写接口映射规则而导致成本居高不下的问题, 提高代码迁移的性能, 最终实现国外深度学习框架上的模型代码低成本自动化迁移至国内主流框架, 本文的方法框架如图 1 所示.

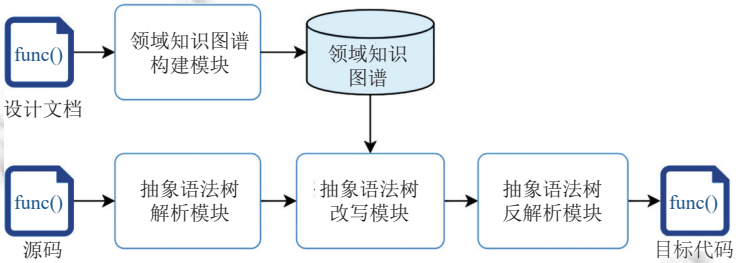

图 1 基于领域知识图谱的框架间 AI 源码自动迁移方法框架

本文的方法框架由 4 个部分组成, 包括领域知识图谱构建模块、抽象语法树解析模块、抽象语法树改写模块和抽象语法树反解析模块. 其中, 领域知识图谱构建模块负责基于设计好的节点与关系构建不同深度学习框架间的算子映射关系, 为后续进行代码迁移提供知识基础; 抽象语法树解析模块将源框架下的模型代码中表征的相关信息解析成抽象语法树, 并与目标框架一起传递到抽象语法树改写模块; 抽象语法树改写模块负责将上一步得到的抽象语法树基于知识图谱进行针对性地遍历和改写操作, 得到目标深度学习框架下的新抽象语法树; 最后, 抽象语法树反解析模块负责将新抽象语法树反解析为目标框架下的模型代码.

4 关键模块

4.1 领域知识图谱构建模块

由上述分析可知, 将不同框架间的差异信息存储在表格或代码中会导致知识的表达能力较弱. 例如, 不同深度学习框架间的算子间往往会存在多对多映射关系, 且同一框架内需要考虑算子调用顺序如 PyTorch 的数据集加载器算子需要调用具体的数据集算子; 不同框架间需要考虑算子依赖关系如 PyTorch 框架的采样器算子需要依赖数据集算子, 而 MindSpore 框架的数据集算子需要反向依赖采样器算子. 这些信息无法较好地在表格或代码中进行展现, 而知识图谱的图结构特性适合存储复杂信息.

本文的领域知识图谱采取自顶向下的构建方法, 首先需要根据研究目标确定知识图谱的数据模型. 本文的模型代码迁移程序需要用到源与目标框架下算子间的映射关系, 因此知识图谱中需要存储 PyTorch、MindSpore、PaddlePaddle 等国内外深度学习框架内的算子知识、框架间算子映射关系知识等. 由此, 本文的领域知识图谱中存放的节点就包括: 深度学习框架的模块、算子、算子参数、算子输入值、算子返回值, 存放的关系则分为框架内关系与框架间关系. 其中, 框架内关系包含模块与算子的关系、算子及其参数的关系、参数及其数据类型的关系以及算子及其输入值的关系, 框架间关系包含算子间映射关系、参数间映射关系等. 部分节点和关系设计如表 1 和表 2 所示.

表 1 领域知识图谱节点设计

节点名称	标签 (label)	属性 (properties)
框架节点	Framework	name: 深度学习框架名称 version: 深度学习框架版本
类节点	Module	framework: 所属的深度学习框架名称 name: 类名称
算子节点	Operator	framework: 所属的深度学习框架名称 name: 算子名称 full_name: 算子的完整调用路径 version: 深度学习框架版本
参数节点	Parameter	framework: 所属的深度学习框架名称 operator: 所属的算子名称 parameter_order: 该参数在所处参数中的顺序序号值 name: 参数名称 dtype: 参数的类型名称/参数可选类型的数目 default: 参数的默认值 optional: 参数的可选性

表 2 领域知识图谱关系设计

关系名称	类型 (type)	属性 (properties)
框架节点与类节点间的关系	classOfFramework	name: 所连接的类节点的名称
类节点与类节点间的关系	subClassOfClass	name: 所连接的类节点的名称
类节点与算子节点间的关系	operatorOfClass	name: 所连接的算子节点的名称
算子节点与参数节点间的关系	parameterOfOperator	parameter_order: 所连接的参数节点在当前算子的参数列表中所处的顺序值 name: 所连接的参数节点的名称
算子节点与输入节点间的关系	inputOfOperatpr	input_order: 所连接的输入节点在当前算子的输入列表中所处的顺序值 name: 所连接的输入节点的名称
不同框架的算子节点间的关系	equivalentOperator	framework_name: 目标框架名称
不同框架的参数节点间的关系	equivalentParameter	framework_name: 目标框架名称

目前, 深度学习框架如 PyTorch 和 MindSpore 等官方平台都提供了详细的 API 文档, MindSpore 和 PaddlePaddle 等官方社区也都提供了其与 PyTorch 框架的算子映射关系文档, 为本文的研究提供了材料支撑. 然而, 仅依靠深度学习框架官方提供的映射文档直接获取的框架间关系知识不理想, 有效的算子以及参数映射关系数量较少. 因此本文提出了一种深度学习框架知识抽取方法, 把抽取出的深度学习框架知识存入基于图的知识图谱中, 以更好地突出算子及参数的层次信息, 获取从关系的角度去分析问题的能力, 为代码迁移提供数据基础.

具体来说, 本文基于网络爬虫技术以基于规则的知识抽取方法, 从主流深度学习框架提供的官方文档网站中爬取出深度学习框架所有算子的信息以及官方社区提供的部分深度学习框架间算子的映射关系信息, 然后解析网页内容, 获取有效信息元素, 同时对获取到的信息进行数据清洗, 包括解析、过滤和统一重组等操作. 数据清洗的

方法包括正则表达式匹配、字符串操作和规则过滤等, 旨在对爬取的文本数据按规则进行分割、检索和分类, 以获取所需信息. 其次对文本中的重复数据、空值和异常值等进行删除、填充等操作, 保证数据的有效性和一致性. 然后按规则汇集信息, 将不同途径、不同结构的信息格式化后统一存储至结构清晰的 JSON 文件中.

在得到上述知识后, 就需要将其存入到图数据库中. 本文采取自顶向下的知识图谱构建方法来构建领域知识图谱, 首先确定知识图谱的顶层概念作为图谱的基础结构, 然后针对顶层概念定义相关的关系和属性, 然后识别出与顶层概念相关的子概念, 并建立它们之间的层次结构关系. 具体来说, 本文将框架作为领域知识图谱的顶层节点, 并继续向下依次细分为类节点、算子节点和参数节点等, 最后建立起上述节点间的关系. 本文选用 Neo4j 来存储上述定义的算子知识, 依据设计的数据模式生成 Cypher 语句, 并编写自动化脚本, 通过爬虫并解析提取出的算子信息, 将算子知识存入 Neo4j 图数据库中形成领域知识图谱. 该图谱并将知识集成为领域知识图谱供抽象语法树改写模块使用, 领域知识图谱构建模块的框架如图 2 所示.

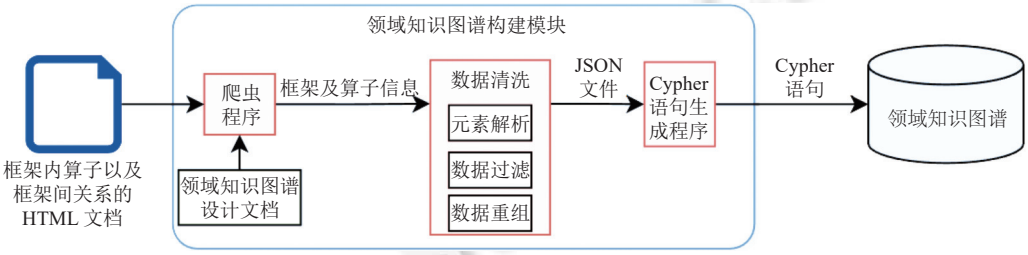

图 2 领域知识图谱构建模块的框架

由于深度学习框架更新速度较快, 因此本文在第 1 次执行深度学习框架知识抽取脚本来抽取全量基本数据后, 会通过程序定时检查各个深度学习框架的版本更新情况, 判断是否需要爬取新的数据. 如果程序判断远程深度学习框架知识与本地知识图谱知识存在差异, 则可以启动脚本来抽取新的知识并再次构建领域知识图谱供迁移程序使用.

4.2 抽象语法树解析模块

由第 2 节可知, 为了减少源框架与目标框架模型代码迁移的复杂性, 本文将跳脱代码层面的直接转换. 对于构造深度学习模型方式较为一致且编程范式相似的 PyTorch、PaddlePaddle 和 MindSpore 框架, 本文将使用抽象语法树解析代码来进行迁移工作.

对于一棵抽象语法树 T, T 是节点的集合 $T=t1,t2,\ldots,tn$, 抽象语法树 T 有且只有一个根节点, 用 $root(t)$ 表示. 对于 T 中每个节点 $t\in T$, 都有一个父节点 $p\in(T\cup\varnothing)$, 节点 t 的父节点用 $parent(t)$ 表示, 根节点 $root(t)$ 的父节点为 $\varnothing$. 每个节点 $t\in T$ 都有一个可以为空的子节点序列, 用 $children(t)$ 表示, 常见的 Python 抽象语法树中的节点及其说明如表 3 所示.

表 3 常见的 Python 抽象语法树节点说明

Python AST节点	说明
Module	代表整个Python模块, 包含该模块的所有语句和定义
FunctionDef	函数定义, 包含函数名、参数、函数体等信息
ClassDef	类定义, 包含类名、基类、类体等信息
Assign	赋值语句, 包含被赋值的对象和值
For/While	循环语句
If	条件语句
Import/ImportFrom	导入模块, 包含被导入的模块名、导入的名称和导入方式
Call	函数调用, 包含函数名、关键字参数列表、位置参数列表等信息
Constant	常量, 包含常量的值和类型信息
Attribute	属性访问, 包含访问属性的对象或模块的名称以及访问的属性名称
Name	变量或函数名

抽象语法树解析模块的框架如图 3 所示. 抽象语法树解析模块中的代码解析器使用 Python 标准库中的 AST 模块进行代码解析, 首先会将源深度学习框架搭建的神经网络模型代码中的所有字符串, 通过词法分析分割成若干标记 (tokens), 每个标记代表代码中的一个单词、操作符或标点符号等. 这些标记通常是以空格或其他特定字符进行分割. 然后再通过语法分析将标记按照语法规则组成语句和表达式, 并构建成一棵抽象语法树, 来表示代码的结构和逻辑. 该模块的底层实现基于 Python 的解释器架构 CPython, 其语法规则遵循 Python 语言标准, 直接支持 Python 语言的最新特性, 最终生成的语法树节点类型包括表 3 中的示例. 此时, 神经网络模型的代码字符串已经解析成承载所有代码信息的抽象语法树.

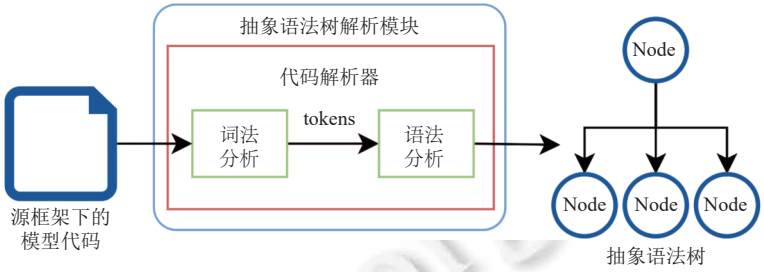

图 3 抽象语法树解析模块的框架

代码 1 的作用是定义一个简单的卷积神经网络模型, 可以用于图像分类、目标检测等任务. 在训练时, 可以将训练数据传入该模型的 forward 方法中进行前向传播, 根据预测结果与真实标签计算损失并反向传播更新模型参数. 代码中主要包含两条导入模块语句和一个面向对象的类, 且类中包含两个方法. 导入的模块是 PyTorch 框架专门为神经网络设计的模块化接口 torch.nn, 这段代码定义一个继承自 nn.Module 的类 Model, 表示了一个简单的神经网络模型. 该模型包含一个卷积层 conv, 卷积层的输入通道数为 3, 输出通道数为 64, 卷积核大小为 7×7, 步长为 2, 填充宽度为 3. 方法接收一个输入 x, 将其传入卷积层 conv, 计算输出 o 并返回.

将代码 net1 = torch.nn.Conv2d(120,240,4,stride = 2) 传递到抽象语法树解析模块后可以得到对应的抽象语法树, 如图 4 所示. 代码中等号左右的变量名称和调用语句分别转换成抽象语法树中的 Name 节点和 Call 节点, 算子内部的变量名称及其变量值也都转换成抽象语法树中对应的节点.

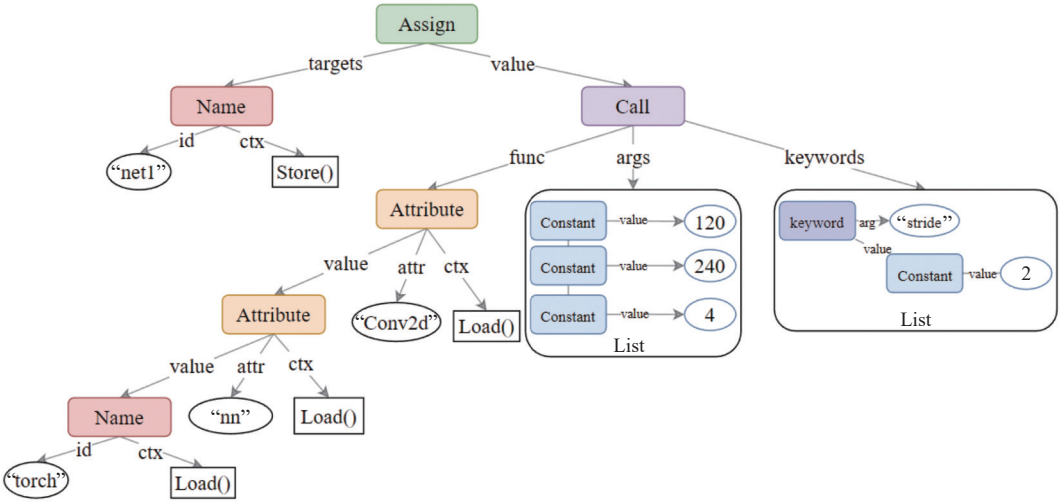

图 4 抽象语法树示例

4.3 抽象语法树改写模块

在得到抽象语法树解析模块输出的抽象语法树后, 将其作为迁移程序的中间件, 对树中相应的节点进行针对

性地改写即可得到一棵目标框架下的抽象语法树. 已知模型代码迁移的主体是神经网络模型组网代码, 由代码 1 PyTorch 版本的简单模型代码、代码 2 PaddlePaddle 版本的简单模型代码以及代码 3 MindSpore 版本的简单模型代码可知, 这几种深度学习框架组网模型代码的不同之处主要在于导入模块、类名称、继承类名称、方法名称、算子名称及其参数等.

需要注意的是, 并非代码中所有不同之处都需要进行迁移, 比如赋值语句中等号左边的变量名称则不需要进行修改, 而模型代码需要迁移的主体部分即为上述的框架间差异部分, 这些差异信息及其映射关系已经存储在第 4.1 节形成的领域知识图谱中. 因此, 抽象语法树改写模块的主要思想就是改写抽象语法树中不同框架间存在差异的节点来完成迁移. 例如, PyTorch 框架下的 torch.nn.Conv2d 算子与 MindSpore 框架下的 mindspore.nn.Conv2d 算子分别是各自框架下的二维卷积算子, 两者在语法和语义上相对应, 通过改写抽象语法树中对应的算子及其参数节点, 就可以达到迁移的目的, 从而将 PyTorch 框架下的卷积算子转换为 MindSpore 框架下的卷积算子.

然而, 大多数模型代码中的算子通常无法独立地表达自身的算子信息, 这通常由以下原因导致: 一是开发人员通常会通过引用相关模块来将算子进行简写, 如引入 torch.nn 模块将 torch.nn.Conv2d 算子简写为 Conv2d; 二是开发人员通常会将算子赋值给变量, 并在后续代码中直接使用该变量来表示对应的算子, 如将 Conv2d 算子赋值给变量 self.conv1 并在后续直接调用 self.conv1 等. 上述情况都会导致迁移程序在遍历 AST 时无法根据树中的算子节点直接到领域知识图谱中获取到对应的知识. 因此, 抽象语法树改写模块中的代码依赖信息生成器会在改写 AST 之前做一些预处理工作. 具体来说, 代码依赖信息生成器会根据模型代码的引用模块信息构建通用依赖信息 UDepInfo (universal dependency information), 包括存储导入模块第 1 级模块名称的 from、存储导入模块第 1 级或以下级别模块名称的 import 以及存储开发人员对导入模块的重命名名称的 as, 如表 4 所示. 例如, 模型代码的引用信息为 from torch.nn import Conv2d as conv2, 则通用依赖信息中的 from 为 torch.nn, import 为 Conv2d, as 为 conv2. 得到上述通用依赖信息后, AST 中的算子节点就可以根据该信息将算子名称 conv2 扩展为完整的算子名称 torch.nn.Conv2d, 从而在领域知识图谱中得到目标框架中对应算子的相关信息.

表 4　通用依赖信息说明

UDepInfo单元	说明
from	导入模块第1级名称
import	第1级或以下级别名称
as	导入模块重命名名称

在得到完整算子名称之后, 抽象语法树改写模块就可以逐级进行 AST 的改写工作了, 抽象语法树改写模块的框架如图 5 所示.

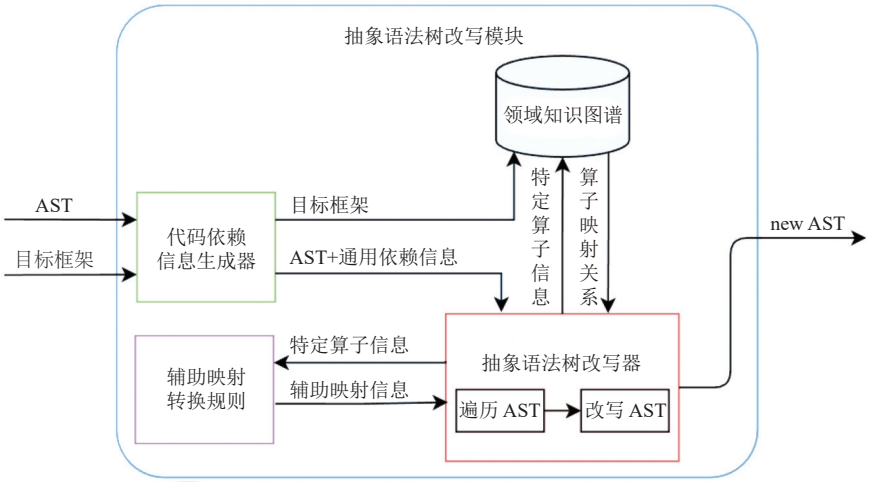

图 5　抽象语法树改写模块的框架

改写模块中的抽象语法树改写器会递归遍历抽象语法树解析模块传递过来的源框架下的抽象语法树, 在此过程中迁移程序中的 VISIT 函数会根据当前遍历到的节点类型来动态地调用相应的访问方法来处理该节点, 这些节点如表所示, 包括 Import 节点、FunctionDef 节点、Call 节点等, 分别对应本节上文所述的框架间差异信息的导入模块、方法名称、算子名称及其参数等.

在处理 Import 节点和 ImportFrom 节点时, 代码依赖生成器会根据 AST 生成前述的通用依赖信息 UDepInfo; 在处理其他节点如 FunctionDef 和 Call 等节点时, 抽象语法树改写器会根据 UDepInfo 生成当前遍历算子的完整算子名称, 并构建该算子的特定算子信息 SOpInfo (specific operator information), 包括存储完整算子名称的 name、存储算子位置和缩进信息的 lineno 和 col_offset、存储算子参数信息的 args 和 keywords 等, 此外, 若该算子赋值给了变量, 则还需要存储该算子赋值变量名称的 var, 如表 5 所示. 通过特定算子信息, 抽象语法树改写器就可以利用其中的完整算子名称及参数信息到领域知识图谱中获取目标框架下对应的算子名称及参数信息, 通过构建新节点并覆盖原节点来改写抽象语法树.

表 5 特定算子信息说明

SOpInfo单元	说明
name	算子的完整名称
lineno/end_lineno	算子位置信息
col_offset/end_col_offset	算子缩进信息
var	算子赋值的变量名称
args	按位置传入的参数组成的列表
keywords	代表以关键字传入的参数的keyword对象的列表

此外, 可能存在源框架算子与目标框架下对应算子的参数用法不一致的问题, 例如所支持的参数类型不一致、参数含义不一致等, 因此需要针对某些特定算子存储相关映射信息并进行针对性的改写. 在本文中, 辅助映射转换规则中存储了上述特定信息, 抽象语法树改写器会利用反射及特定算子信息, 到辅助映射转换规则中获取当前算子的辅助映射函数, 并据此获得目标算子及其特殊参数值. 综上所述, 自顶向下的抽象语法树改写算法如算法 1 所示.

算法 1. 基于抽象语法树的模型代码迁移算法.

Input: 源抽象语法树 *T*, 目标框架名称 TARGET_FRAMEWORK.

```
1. /* 初始化通用依赖信息和特定算子信息 */
2. init UDepInfo and SOpInfo;
3. /* 获取 T 的根节点 */
4. root ← getRoot(T);
5. VISIT(root);
6. Function VISIT(node)
7.     for value in ast.iter_fields(node) do
8.         /* 遍历节点 node 的子节点, 根据子节点 value 的分类进行对应的处理 */
9.         if value is list then
10.            /* 新建临时列表来存储 value 列表中经过递归改写后的元素 */
11.            new_values = []
12.            for child_value in value do
13.                child_value = VISIT(child_value)
14.                new_values.add(child_value)
15.            end
```

```
16.                 value = new_values
17.             end
18.             if value is ast.Import or ast.ImportFrom then
19.                 /* 访问 Import 或 ImportFrom 节点, 构建通用信赖信息 */
20.                 for alias in node.names do
21.                     UDepInfo.add(from = node.module, import = alias.name, as = aslias.asname)
22.             else if value is ast.ClassDef or ast.FunctionDef then
23.                 /* 需要递归访问其子节点 */
24.                 VISIT(children(value));
25.             else if value is ast.Call or ast.Attribute then
26.                 /* 根据 UEdepInfo 生成当前遍历到算子的特定算子信息 SOpInfo */
27.                 SOpInfo ← get_sopinfo(value, UDepInfo);
28.                 /* 根据特定算子信息及目标框架名称, 到领域知识图谱中获取映射关系 */
29.                 mapping_info ← get_mapping_info(SOpInfo.get(value), TARGET_FRAMEWORK);
30.                 if mapping_info is not None then
31.                     /* 获取辅助映射信息 */
32.                     special_para ← mapping_rules(SOpInfo.get(value));
33.                     /* 根据映射关系和辅助映射信息修改算子节点 */
34.                     change_ast(node, mapping_info, special_para);
35.                 end
36.             end
37.         end
38. end
```

4.4 抽象语法树反解析模块

在完成抽象语法树的改写获得新的抽象语法树之后, 深度学习模型代码在不同深度学习框架上的基于抽象语法树的自动转换的异构适配技术的重难点部分就已经完成了. 接下来就是使用模型代码反解析模块将新抽象语法树重构成目标深度学习框架的模型代码.

与抽象语法树解析模块不同, 反解析模块需兼顾 Python 语言标准与代码可读性. 反解析模块通过借助 astor 库的 to_source 函数, 可以将树结构转换成 Python 代码, 其步骤为使用自顶向下的方法, 遍历树中的每个节点, 当进入特定的抽象语法树节点时, 根据指定的规则将其转换为程序代码. 这些规则通常是根据目标深度学习框架所遵循的语法规则来定义的. 在遍历的过程中, 将还原出来的代码按照节点遍历的顺序进行组装和拼接, 最终得到的就是目标深度学习框架的神经网络模型代码. 此外, 采用抽象语法树作为中间件进行转换的一个好处是, 抽象语法树会将源代码的缩进情况存储起来, 在将抽象语法树恢复成代码后, 这些缩进信息仍然保留, 不会给代码结构造成严重破坏. 对于反解析生成的目标代码, AST 重构可能生成长代码行如复杂表达式和嵌套公式, 因此还需要添加分行符并遵循 Python 语言标准. 此外, AST 不保留注释信息, 因此需要人工添加注释, 我们也考虑在未来的工作中通过后处理进行补充.

5 实验分析

5.1 实验设计

为了验证本文提出的模型代码自动迁移技术的有效性, 即测试基于领域知识图谱和抽象语法树的模型代码技

术在 PyTorch、PaddlePaddle 和 MindSpore 框架之间的转换是否满足语法性和语义等价性. 本节将在构建领域知识图谱的基础上, 对源深度学习框架下的模型代码进行迁移, 并对生成的目标框架下的模型代码进行功能性测试和精度测试.

代码翻译领域的评估主要是通过将样本与参考解决方案相匹配来进行基准测试, 但是 Ren 等人[8]发现, 基于匹配的指标无法解释在功能上等同于参考解决方案的程序所构成的空间. 因此, 在无监督翻译和伪代码到代码翻译方面的工作转向了功能正确性, 即如果生成代码与参考代码在经过一组单元测试均产生了相似的结果, 那么它就被认为是翻译正确的代码. 因此, 本文采用功能性测试来对深度学习模型代码迁移程序进行测试.

更进一步的, 本文将对生成的目标框架下的模型代码进行训练, 通过对比目标模型基于指定数据集得到的精度与业界或复现论文精度的差异, 来进行代码迁移程序得到模型的精度测试.

5.2 领域知识图谱构建结果

根据图 2 的领域知识图谱构建模块的框架, 本文抽取了 PyTorch、MindSpore 和 PaddlePaddle 这 3 个深度学习框架的算子信息并形成领域知识图谱, 其中表 1 设计的节点以及表 2 设计的关系在图谱中的数量分别如表 6 和表 7 所示.

表 6 领域知识图谱中的算子数量

节点名称	数量
框架节点	6
类节点	487
算子节点	9018
参数节点	24781
合计	34292

表 7 领域知识图谱中的关系数量

关系名称	数量
框架节点与类节点间的关系	6
类节点与类节点间的关系	544
类节点与算子节点间的关系	9018
算子节点与参数节点间的关系	24781
算子节点与输入节点间的关系	952
不同框架的算子节点间的关系	3340
不同框架的参数节点间的关系	9320
合计	47961

同时, 各个框架的算子及其参数的节点数量、框架间算子及其参数映射关系数量如表 8 和表 9 所示.

通过表 6–表 9 可知, 深度学习框架的算子信息数量很大, 算子及其参数映射关系的数量相应也十分巨大, 所以人工抽取映射信息需要以数月来进行计算, 且后期必须定时安排人力维护, 否则难以跟上框架变化的发展. 而本文的抽取脚本可以设置定时爬取深度学习框架信息并进行更新, 且时间可以控制在小时级别内, 相比于 MindConverter 等工具进行人工编写映射信息的方式, 效率有极大的提升.

表 8 深度学习框架的算子及其参数的节点数量

框架名称	版本	算子数量	参数数量
PyTorch	1.8.1	1085	2764
PyTorch	2.1	1548	4641
MindSpore	2.2	1996	5169
MindSpore	2.3	2033	5296
PaddlePaddle	2.4	1720	5149
PaddlePaddle	2.6	1482	3700
合计	—	9864	26719

表 9 框架间算子及其参数映射关系数量

对应框架	对应版本	算子间映射关系数量	参数间映射关系数量
PyTorch-MindSpore	1.8.1-2.2	473	1595
PyTorch-MindSpore	1.8.1-2.3	478	1134
PyTorch-PaddlePaddle	1.8.1-2.4	1281	4326
PyTorch-PaddlePaddle	2.1-2.6	1108	2265
合计	—	3340	9320

5.3 模型功能测试

本文针对不同深度学习框架间的模型代码自动迁移的功能性测试需要满足 3 个前提条件: 框架相同; 随机种子相同; 输入张量元组相同. 此外, 本文采取 PaConvert 的测试标准来进行张量的设置, 即每个张量中的元素个数必须大于 100, 却不能全为零等无效输入. 针对并行语料库, 在随机种子设为一定值后, 将源框架的神经网络模型代码生成的目标框架模型代码与并行语料库中的目标框架参考代码输入同一张量元组后, 检测两者输出张量的差

异, 来验证功能正确性.

但是, 深度学习模型代码转换的并行语料库很少, 因此, 本文借鉴了 TransCoder 的回译功能来生成同一框架的模型代码, 也即先用源框架的模型代码转换到另一种目标框架的模型代码, 再由生成的目标框架的模型组网代码回译到源框架上, 获取源框架的新的模型代码[9]. 此时在满足上述的 3 个条件下, 将同一框架下的原始模型代码与回译后生成的新的模型代码之间检测模型代码翻译的功能正确性. 综上, 本文按照不同的深度学习框架, 分别测试并行语料库和单语言语料库的神经网络模型代码转换的异构适配技术, 语料库的数量及其代码规模如表 10 所示.

表 10　语料库数量

语料库名称	数量	代码行数
PyTorch模型代码	98	93 786
PaddlePaddle模型代码	76	71 820
MindSpore模型代码	54	50 544

本文采用回归问题的常用评价指标来衡量代码迁移前后的输出张量的差异性. 由于深度学习模型输出张量元组, 因此采用平均绝对误差 (mean absolute error, *MAE*) 和均方误差 (mean squared error, *MSE*) 两种评价指标. 假设 t 是输入张量元组的数量, k 是输出张量元组的数量, $\mathcal{X}_{i\in[1,k]}$、$\mathcal{Y}_{i\in[1,k]}$、$\mathcal{Z}_{i\in[1,k]}$ 都是张量. $\mathcal{X}=<\mathcal{X}_1,\mathcal{X}_2,\ldots,\mathcal{X}_t>$ 表示输入张量元组, $\mathcal{Y}=<\mathcal{Y}_1,\mathcal{Y}_2,\ldots,\mathcal{Y}_t>$ 表示参考模型代码在输入 $\mathcal{X}$ 后计算得出的输出张量元组, $\mathcal{Z}=<\mathcal{Z}_1,\mathcal{Z}_2,\ldots,\mathcal{Z}_t>$ 表示通过本文迁移后生成的目标模型代码在输入同一个张量元组 $\mathcal{X}$ 后计算得出的输出张量元组. $\mathcal{Y}_i$ 和 $\mathcal{Z}_i$ 是欧几里得空间 R^n 的两个结果向量, 并且 $\mathcal{Y}_{ij}$ 和 $\mathcal{Z}_{ij}$ 分别表示输出张量和的第 j 个分量.

$$\begin{cases} MAE_i=\dfrac{1}{n}\displaystyle\sum_{j=1}^{n}\left|y_{ij}-z_{ij}\right| \\ MAE_c=\dfrac{\displaystyle\sum_{i=1}^{k}MAE_i}{k} \\ MAE=\dfrac{\displaystyle\sum_{c=1}^{C}MAE_c}{C} \end{cases} \tag{5}$$

平均绝对误差如公式 (5) 所示, 其中 MAE_i 表示 $\mathcal{Y}_i$ 和 $\mathcal{Z}_i$ 两个张量的平均绝对误差, n 表示张量的维度, MAE_c 表示 $\mathcal{Y}$ 和 $\mathcal{Z}$ 两个张量元组的平均绝对误差, 其值也代表在单个模型代码上迁移的平均相对误差, MAE 则表示最终的平均相对误差, C 表示经过测试的模型代码的总数量.

$$\begin{cases} MSE_i=\dfrac{1}{n}\displaystyle\sum_{j=1}^{n}(y_{ij}-z_{ij})^2 \\ MSE_c=\dfrac{\displaystyle\sum_{i=1}^{k}MSE_i}{k} \\ MSE=\dfrac{\displaystyle\sum_{c=1}^{C}MSE_c}{C} \end{cases} \tag{6}$$

均方误差如公式 (6) 所示, 其中 MSE_i 表示 $\mathcal{Y}_i$ 和 $\mathcal{Z}_i$ 两个张量的均方误差, n 表示张量的维度, MSE_c 表示 $\mathcal{Y}$ 和 $\mathcal{Z}$ 两个张量元组的均方误差, 其值也代表在单个模型代码上迁移的均方误差, MSE 则表示最终的均方误差, C 表示经过测试的模型代码的总数量.

本文分别对比了 PyTorch、PaddlePaddle 和 MindSpore 这 3 种深度学习框架的神经网络模型代码的迁移效果,

在将随机种子设置为相同值之后, 随机生成相同的张量元组并输入模型, 3 种框架间互相迁移的 *MAE* 和 *MSE* 分别如表 11 所示. 由于将源框架下的模型代码迁移到目标框架下再进行回译的过程中会遇到某些算子的参数不是完全一一对应的关系, 例如 PyTorch 与 PaddlePaddle 框架下的模型代码在互相迁移的过程中会存在参数用法不一致、参数缺失等问题, PyTorch 的若干算子拥有的 inplace 参数在 PaddlePaddle 对应的算子下却没有相应的映射参数, 因此可能会对输出结果存在影响, PaddlePaddle 官网也提到这一差异可能会对网络训练结果产生影响, 所以会导致输出张量与输入张量不一致的差异.

同时, 本文将迁移程序分别与 MindConverter 以及 PaConvert 工具在上述数据集中进行对比, 对比结果如表 12 所示. 可以看到, 在将 PyTorch 框架下的模型代码迁移至 PaddlePaddle 框架下的评价指标下, 本文方法的 MAE 略高于 PaConvert, 但是 *MSE* 值低于 PaConvert; 在将 PyTorch 框架下的模型代码迁移至 MindSpore 框架下的评价指标下, 本文方法的 *MAE* 和 *MSE* 值均低于 MindConverter. 此外, MindConverter 工具只能将 PyTorch 框架下的模型设计代码单向迁移至 MindSpore 框架下, 而既不支持反向迁移, 也不支持与 PaddlePaddle 之间的互相迁移; PaConvert 同样只能将 PyTorch 框架下的模型代码单向迁移至 PaddlePaddle 框架下, 而既不支持反向迁移, 也不支持与 MindConverter 之间的互相迁移, 无法形成互联互通的良好态势. 最后, 本文针对单语语料库以及并行语料库的结果进行了区分, 结果如表 13 所示, 可以看到在单语语料库上, 由于在同一框架下模型的一致性较高, 所以结果的 *MAE* 和 *MSE* 值均相对较小.

表 11 模型功能测试结果

源框架	目标框架		
	PyTorch	PaddlePaddle	MindSpore
PyTorch	—	*MAE*: 0.152 *MSE*: 0.085	*MAE*: 0.143 *MSE*: 0.079
PaddlePaddle	*MAE*: 0.182 *MSE*: 0.144	—	*MAE*: 0.186 *MSE*: 0.148
MindSpore	*MAE*: 0.091 *MSE*: 0.089	*MAE*: 0.095 *MSE*: 0.093	—

表 12 模型功能测试的对比结果

方法	框架	
	PyTorch→PaddlePaddle	PyTorch→MindSpore
MindConverter	不支持	*MAE*: 0.149 *MSE*: 0.081
PaConvert	*MAE*: 0.096 *MSE*: 0.091	不支持
本文方法	*MAE*: 0.152 *MSE*: 0.085	*MAE*: 0.143 *MSE*: 0.079

表 13 模型功能测试在不同语料库上的对比结果

语料库	*MAE*	*MSE*
并行语料库	0.185	0.137
单语语料库	0.143	0.082

综上, 通过模型功能测试的实验结果和对比结果, 我们发现本文的迁移程序达到了较低的 *MAE* 和 *MSE*, 并且结果向量之间的总体差异较小, 验证了本文方法的功能正确性.

5.4 模型精度测试

更进一步的, 本文将对若干经典论文中提出的模型进行迁移, 并对迁移前后的神经网络模型代码进行深度学习训练, 通过对比迁移前后模型在数据集上的精度差异来进一步验证转换的有效性. 其中, ResNet50[10]、MobileNetV2[11]、DenseNet121[12]、ViT[13]、ShuffleNet V2[14]、HarDNet[15]、EfficientNet[16]和 Gather-excite[17]属于分类模型, U-Net[18]和 PSPNet_ResNet50[19]属于分割模型. 本文采用 CIFAR10 作为数据集, 该数据集由 60 000 张 32×32 彩色图片组成, 总共有 10 个类别, 每类 6 000 张图片, 有 50 000 个训练样本和 10 000 个测试样本, 10 个类别包含飞机、汽车、鸟类、猫、鹿、狗、青蛙、马、船和卡车.

表 14 列出了官方 PyTorch 版本的各模型代码和转换后得到的 PaddlePaddle 以及 MindSpore 版本的模型代码在训练后进行测试的精度对比. 可以看到, 迁移前后的精度非常接近, 无较大差异. 因此, 可以验证在本文自动迁移技术的转换下, 转换前的模型代码与转换后的模型代码在精度上无明显损失.

表 14 模型精度测试结果 (%)

网络名称	PyTorch	PaddlePaddle	MindSpore
ResNet50	94.28	94.36	94.65
MobileNetV2	91.29	91.67	90.85
DenseNet121	94.11	93.61	93.76
ViT	53.13	54.28	56.36
ShuffleNet V2	89.17	88.66	88.72
HarDNet	87.51	86.75	87.51
EfficientNet	90.26	89.54	89.66
Gather-excite	94.49	94.51	94.53
U-Net	71.59	72.63	74.01
PSPNet_ResNet50	62.17	63.28	64.38

6 总结与展望

本文提出了一种基于领域知识图谱和抽象语法树的不同深度学习框架间 AI 源码迁移方法. 通过建立领域知识图谱来存储不同深度学习框架间算子及其参数的差异信息, 并采用抽象语法树作为代码迁移中间件提高了转换的可靠性与效率. 实验结果表明, 我们的方法在不同深度学习框架间的 AI 源码自动迁移上具有较好的转换效果, 模型功能性测试证明了代码迁移的正确性, 模型精度测试进一步证明了迁移后的模型的可靠性.

在接下来的工作中, 我们考虑完善迁移代码的完整性, 包括模型的训练代码、测试代码以及数据集读取代码等. 上述代码存在于模型构建中除模型设计外的其余步骤中, 如数据处理、训练设置、应用部署和模型评估, 其中包含若干多对多映射关系的算子. 我们计划采用基于自定义算子的组合替代实现来进行多对多算子的迁移, 并将该自定义算子覆盖到源抽象语法树中来达到改写抽象语法树的目的. 最后, 我们计划进一步完善深度学习框架的数量, 增加其他框架下模型的迁移方案, 实现更多框架之间的互联互通, 以凝聚深度学习框架间的群智力量.

References

[1] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin ZM, Gimelshein N, Antiga L, Desmaison A, Köpf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai JJ, Chintala S. PyTorch: An imperative style, high-performance deep learning library. In: Proc. of the 33rd Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2019. 8026–8037.

[2] Ma YJ, Yu DH, Wu T, Wang HF. PaddlePaddle: An open-source deep learning platform from industrial practice. Frontiers of Data and Computing, 2019, 1(1): 105–115 (in Chinese with English abstract). [doi: 10.11871/jfdc.issn.2096.742X.2019.01.011]

[3] Shridhar A, Tomson P, Innes M. Interoperating deep learning models with ONNX. Proc. of the JuliaCon Conf., 2020, 1(1): 59. [doi: 10.21105/jcon.00059]

[4] Liu Y, Chen C, Zhang R, Qin TT, Ji X, Lin HX, Yang M. Enhancing the interoperability between deep learning frameworks by model conversion. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 1320–1330. [doi: 10.1145/3368089.3417051]

[5] Jia YQ, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, Guadarrama S, Darrell T. Caffe: Convolutional architecture for fast feature embedding. In: Proc. of the 22nd ACM Int'l Conf. on Multimedia. Orlando: ACM, 2014. 675–678. [doi: 10.1145/2647868.2654889]

[6] Abadi M, Barham P, Chen JM, Chen ZF, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray DG, Steiner B, Tucker P, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng XQ. TensorFlow: A system for large-scale machine learning. In: Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation. Savannah: USENIX Association, 2016. 265–283.

[7] Seide F, Agarwal A. CNTK: Microsoft's open-source deep-learning toolkit. In: Proc. of the 22nd ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. San Francisco: ACM, 2016. 2135. [doi: 10.1145/2939672.2945397]

[8] Ren S, Guo D, Lu S. CodeBLEU: A method for automatic evaluation of code synthesis. arXiv:2009.10297, 2020.

[9] Roziere B, Lachaux MA, Chanussot L, Lample G. Unsupervised translation of programming languages. In: Proc. of the 34th Int'l Conf.

on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 20601–20611.

[10] He KM, Zhang XY, Ren SQ, Sun J. Deep residual learning for image recognition. In: Proc. of the 2016 IEEE Conf. on Computer Vision and Pattern Recognition (CVPR). Las Vegas: IEEE, 2016. 770–778. [doi: 10.1109/CVPR.2016.90]

[11] Sandler M, Howard A, Zhu ML, Zhmoginov A, Chen LC. MobileNetV2: Inverted residuals and linear bottlenecks. In: Proc. of the 2018 IEEE/CVF Conf. on Computer Vision and Pattern Recognition. Salt Lake City: IEEE, 2018. 4510–4520. [doi: 10.1109/CVPR.2018.00474]

[12] Huang G, Liu Z, van der Maaten L, Weinberger KQ. Densely connected convolutional networks. In: Proc. of the 2017 IEEE Conf. on Computer Vision and Pattern Recognition (CVPR). Honolulu: IEEE, 2017. 2261–2269. [doi: 10.1109/CVPR.2017.243]

[13] Dosovitskiy A, Beyer L, Kolesnikov A, Weissenborn D, Zhai XH, Unterthiner T, Dehghani M, Minderer M, Heigold G, Gelly S, Uszkoreit J, Houlsby N. An image is worth 16x16 words: Transformers for image recognition at scale. arXiv:2010.11929, 2021.

[14] Ma NN, Zhang XY, Zheng HT, Sun J. ShuffleNet V2: Practical guidelines for efficient CNN architecture design. In: Proc. of the 15th European Conf. on Computer Vision. Munich: Springer-Verlag, 2018. 122–138. [doi: 10.1007/978-3-030-01264-9_8]

[15] Chao P, Kao CY, Ruan YS, Huang CH, Lin YL. HarDNet: A low memory traffic network. In: Proc. of the 2019 IEEE/CVF Int'l Conf. on Computer Vision (ICCV). Seoul: IEEE, 2019. 3551–3560. [doi: 10.1109/ICCV.2019.00365]

[16] Tan MX, Le Q. EfficientNet: Rethinking model scaling for convolutional neural networks. In: Proc. of the 36th Int'l Conf. on Machine Learning. Long Beach: ICML, 2019. 6105–6114.

[17] Hu J, Shen L, Albanie S, Sun G, Vedaldi A. Gather-excite: Exploiting feature context in convolutional neural networks. In: Proc. of the 32nd Int'l Conf. on Neural Information Processing Systems. Montréal: Curran Associates Inc., 2018. 9423–9433.

[18] Ronneberger O, Fischer P, Brox T. U-Net: Convolutional networks for biomedical image segmentation. In: Proc. of the 18th Int'l Conf. on Medical Image Computing and Computer-assisted Intervention. Munich: Springer, 2015. 234–241. [doi: 10.1007/978-3-319-24574-4_28]

[19] Zhao HS, Shi JP, Qi XJ, Wang XG, Jia JY. Pyramid scene parsing network. In: Proc. of the 2017 IEEE Conf. on Computer Vision and Pattern Recognition (CVPR). Honolulu: IEEE, 2017. 6230–6239. [doi: 10.1109/CVPR.2017.660]

附中文参考文献

[2] 马艳军, 于佃海, 吴甜, 王海峰. 飞桨: 源于产业实践的开源深度学习平台. 数据与计算发展前沿, 2019, 1(1): 105–115. [doi: 10.11871/jfdc.issn.2096.742X.2019.01.011]

作者简介

丁嵘, 副教授, 博士生导师, 主要研究领域为智能软件工程, 计算机视觉, 无人系统智能算法.

刘屹洲, 硕士, 主要研究领域为代码迁移, 知识图谱.

王雨倩, 硕士, 主要研究领域为预训练语言模型, 知识图谱嵌入.

李一錡, 博士生, 主要研究领域为智能软件.