第5章

CANN模型训练

第2～4章对异构计算架构CANN进行了全面介绍，它通过提供统一编程接口AscendCL，架起了底层昇腾处理器与上层AI应用之间的桥梁。第5～7章将聚焦实际AI应用的开发，以案例为驱动，介绍基于昇腾架构的模型训练、模型推理和完整的行业应用落地流程。

本章首先对主流的深度学习训练框架进行梳理，引入更能充分发挥昇腾产业技术优势的开源框架MindSpore，再以典型模型ResNet-50为例，介绍各主流框架与CANN的适配原理及在昇腾平台上的执行方式，最后介绍在昇腾平台上的模型迁移方法与一些训练过程中使用的实用工具。

5.1 深度学习训练框架

人工智能从理论研究到应用落地的过程中会涉及多个不同的步骤和工具。虽然实际应用场景千变万化，但是其中的深度学习算法具有较大的通用性，如常用于计算机视觉领域的卷积神经网络(CNN)和常用于自然语言处理领域的长短期记忆网络(LSTM)的工作过程，都可以分为模型搭建、自动微分、计算加速、推理调优等多个过程，这就使得抽象出统一的训练框架成为可能。

深度学习训练框架的出现大大降低了编写深度学习代码的成本，其中集成的大量基础算子和各种优化算法，可以有效地帮助用户摆脱烦琐的外围工作，更聚焦实际业务场景和模型设计本身。近几年来，深度学习爆炸式发展，其理论体系得到了长足的进步，基础架构也不断推陈出新，它们共同奠定了深度学习繁荣发展的基础。

如图5-1所示，近十几年来涌现出的深度学习框架可以被划分为三个阶段，第一个阶段是以theano为代表的工具库时代，这一阶段的框架奠定了基于Python、自动微分、计算图等的基本设计思路；第二阶段则以TensorFlow和PyTorch为代表，前者通过分布式训练能力在工业界得到广泛的认可，后者则通过动态图的灵活性在学术界广受青睐；第三阶段则以华为公司提出的面向全场景的端边云训练框架MindSpore为代表，从芯片、模型、算力等多个维度探索面向未来的深度学习训练架构。

本节首先对新一代深度学习框架MindSpore进行简单的介绍，再介绍TensorFlow和PyTorch的框架特点，最后再将各主流框架进行横向对比。

图 5-1　深度学习框架发展历程

5.1.1　MindSpore

在深度学习框架领域，同时满足易开发和高效执行两个目标是很困难的。为了帮助用户更简单高效地开发和使用 AI 技术，更好地发挥 AI 处理器性能，华为公司推出面向全场景 AI 计算框架 MindSpore，并在 2020 年 3 月宣布开源。

MindSpore 着力于实现三个目标：易开发、高效执行、全场景覆盖。为了达成这些目标，MindSpore 开发了一种新的策略，即基于源码转换的自动微分。一方面，MindSpore 支持流程控制的自动微分，可以非常方便地搭建模型；另一方面，MindSpore 可以对神经网络进行静态编译优化，从而获得良好的性能。

从架构上看，MindSpore 可分为四个主要组件：MindExpression(ME)、GraphEngine(GE)、MindData(MD)和 MindArmour(MA)，如图 5-2 所示。

ME 提供了 Python 接口和自动微分功能。具体来看，MindSpore 采用基于源码转换(Source Code Transformation，SCT)的自动微分机制，兼顾了可编程性和性能。一方面，MindSpore 能够提供给用户与编程语言 Python 一致的编程体验，另一方面，它可以用控制流表示复杂的组合，将函数转化为函数中间表达(Intermediate Representation，IR)，中间表达式构造出一个能够在不同设备上解析和执行的计算图，并通过解析 Python 代码，生成抽象语法树(AST)，然后将其转换为图形化的 A-Normal-Form(ANF)图。如果用户需要训练神经网络，则 ME 流水线也会自动生成反向计算节点，并添加到 ANF 图中。流水线在构造完整图之后进行许多优化(如内存复用、算子融合、常数消除等)。如果用户在分布式环境中训练模型，流水线则会通过自动并行策略进行优化。待优化完成后，GM 的虚拟机会通过会话管理计算图，调用 GE 来执行图，并控制图的生命周期。

GE 位于 ME 和底层硬件设备之间，负责硬件相关的资源管理和优化。它实际包含了 CANN 软件栈中昇腾计算编译引擎的图编译器(Graph Compiler)和昇腾计算执行引擎的图执行器(Graph Executor)。从 MindSpore 的角度来看，GE 接收来自 ME 的数据流图，并将该图中的算子调度到目标设备上执行。GE 将数据流图分解为优化后的子图，将它们调度到不同的设备上。GE 将每个设备抽象为一个执行引擎(Execution Engine)，并提供执行引擎插件机制，用来支持各种不同的设备，这样的机制使得端、边、云协同训练成为可能，进而实现了框架的全场景覆盖。

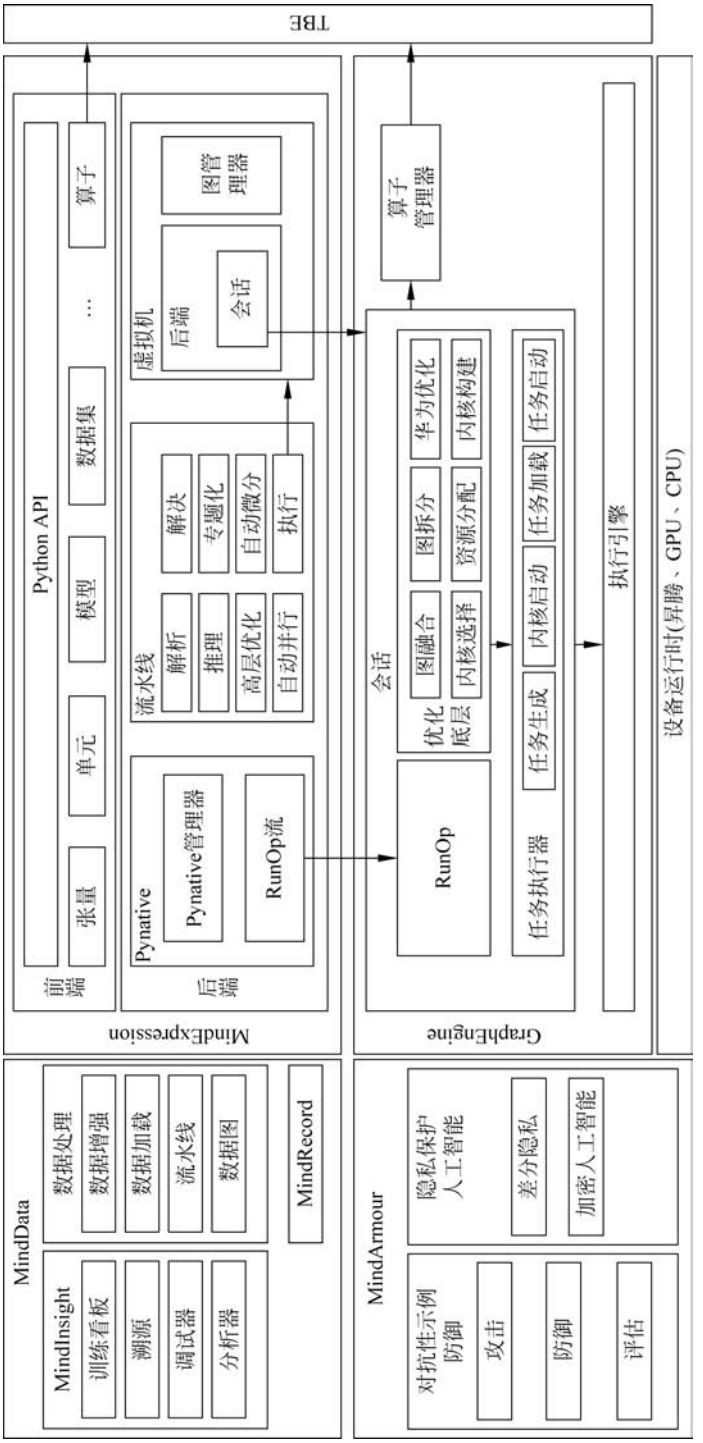

图 5-2　MindSpore 架构

MD 负责数据处理，并提供工具来帮助用户调试和优化模型。该组件通过自动数据加速技术实现了高性能的流水线进而完成数据处理。伴随着各种自动增强策略的出现，用户不必再额外寻找合适的数据增强策略。除此之外，如图 5-3 所示，训练看板将多种数据集成在一个页面，方便用户查看训练过程。分析器可以打开执行黑匣子，收集执行时间和内存使用的相关数据，进而实现有针对性的性能优化。

图 5-3　MindInsight 训练看板

MA 负责提供工具，帮助用户防御对抗性攻击，实现隐私保护的机器学习。在形式方面，MA 提供了以下功能：生成对抗代码、评估模型在特定对抗环境中的性能、开发更健壮的模型。MA 还支持丰富的隐私保护能力，如差分隐私、机密人工智能计算、可信协同学习等。

从特性上看，MindSpore 具有以下 5 个显著特点。

1. 基于源码转换的自动微分

MindSpore 采用基于源码转换的自动微分机制，在训练或推理阶段，可以将一段 Python 代码转换为数据流图。因此，用户可以方便地使用 Python 原生控制逻辑来构建复杂的神经网络模型。自动微分的实现原理可以理解为对程序本身进行符号微分。MindSpore IR 是函数式的中间表达，它与基本代数中的复合函数有直观的对应关系，只要已知基础函数的求导公式，就能推导出由任意基础函数组成的复合函数的求导公式。MindSpore IR 中每个原语操作可以对应为基础代数中的基础函数，这些基础函数可以构建更复杂的流程控制，这样的原理机制也就形成了如图 5-4 所示的基于源码转化的自动微分方式。

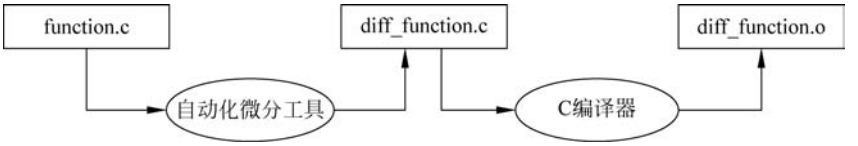

图 5-4　基于源码转换的自动微分

2. 自动并行

由于大规模模型和数据集的不断增加，分布式训练已经成为一种常见做法，但随着计算需求的不断扩张，深度学习框架不仅需要支持数据并行和模型并行，还需要支持混合并行。MindSpore 在并行化策略搜索中引入了张量重排布（Tensor Redistribution，TR），当前一个算子的输出张量模型和后一个算子的输入张量模型不一致时，就需要引入计算、通信操作的方式实现张量排布间的变化。如图 5-5 所示，张量重排布算法使输出张量的设备布局在输入到后续算子之前能够被转换，配合反向算子、半自动并行等功能，最终实现了透明且高效的并行化训练任务。“透明”是指用户只需更改一行配置，提交一个版本的 Python 代码，就可以在多个设备上运行这一版本的 Python 代码进行训练；“高效”是指该算法以最小的代价选择并行策略，降低了计算和通信开销。

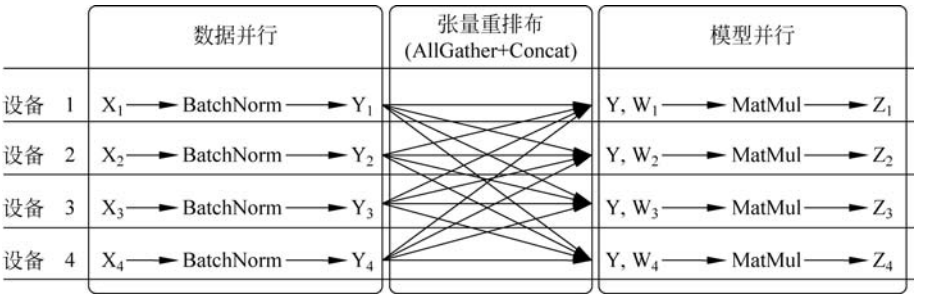

图 5-5　数据并行性向模型并行性转换

3. 动静态图结合

MindSpore 使用统一自动微分引擎兼容动静态图，如图 5-6 所示，无须引入额外的自

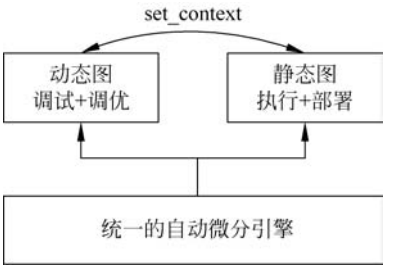

图 5-6　统一自动微分引擎兼容动静态图

动微分机制(如算子重载微分机制)就可以快速地完成转换,大大提高了动态图和静态图的兼容性。而且从用户的角度来看,仅需一行代码就可以完成动静态图模式的灵活转换,转换的代码如程序清单 5-1 所示。

程序清单 5-1　MindSpore 动静图模式的灵活转换

```
#切换为动态图模式
context.set_context(mode = contex.PYNATIVE_MODE)
#切换为静态图模式
context.set_context(mode = contex.GRAPH_MODE)

#调试通过的代码,使用静态图模式执行
@ms_function
def sub_net(self, x):
    x = self.conv(x)
    return x
#待调试的代码,使用动态图模式执行
def construct(self, x):
    x = self.sub_net(x)
    x = self.relu(x)
    return x
```

4. 二阶优化

当前主流模型都需要多次循环训练才能达成目标,以 ResNet-50 为例,使用常见的一阶方法(如梯度下降法)需要 90 个周期才能收敛至精度 0.759。而 MindSpore 实现了二阶优化算法 THOR,它引入了二阶信息矩阵来指导参数的更新,通过矩阵求逆优化,实现了训练效率的有效提升,其算法的具体流程如图 5-7 所示。这样的二阶优化方法在实际的训练过程中也取得了如图 5-8 的优秀表现,在 ResNet-50 上进行测试,仅需 42 个周期就能收敛至同等水平,训练性能大幅提升。

5. 全栈协同加速

MindSpore 作为昇腾计算体系中重要的组成部分,是最“亲和”昇腾处理器的执行引擎,配合 CANN 实现了软硬件的协同优化。具体来看,MindSpore 能将整网下沉到昇腾硬件上执行,减少了 Host CPU 与昇腾处理器之间的交互开销;它还通过格式转换消除、类型转换消除、图算子融合等方式实现了计算图深度优化,发挥了昇腾硬件的极致性能。

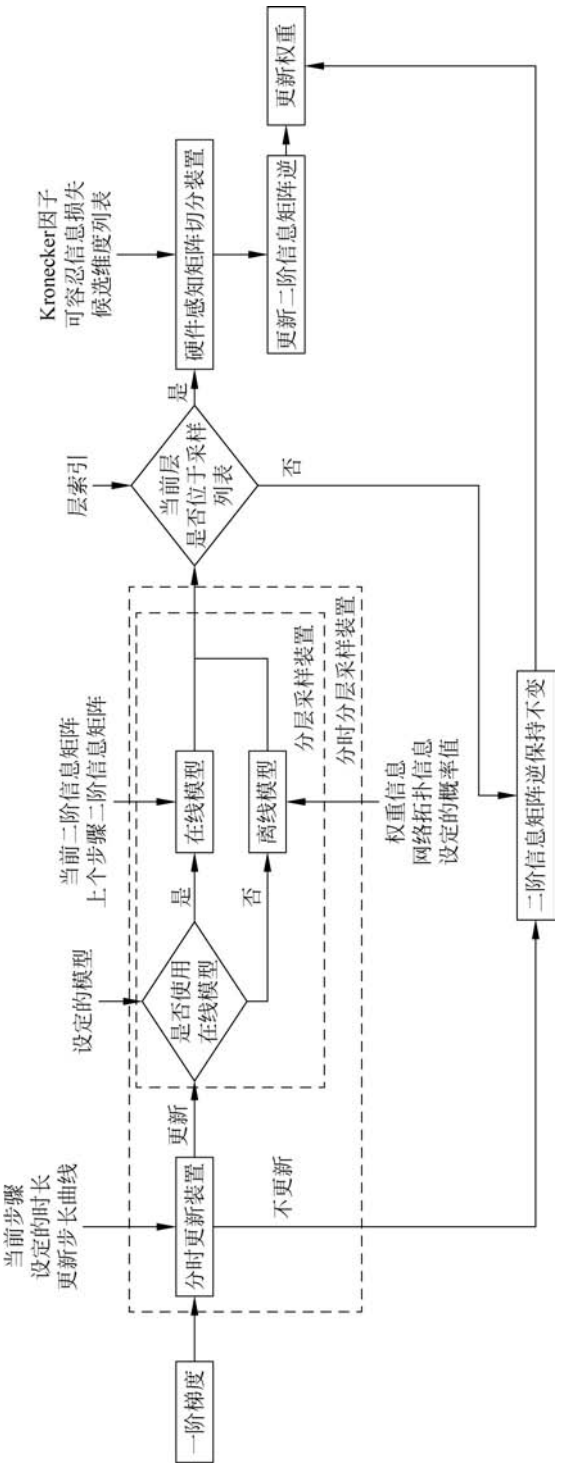

图 5-7　二阶优化算法 THOR 流程

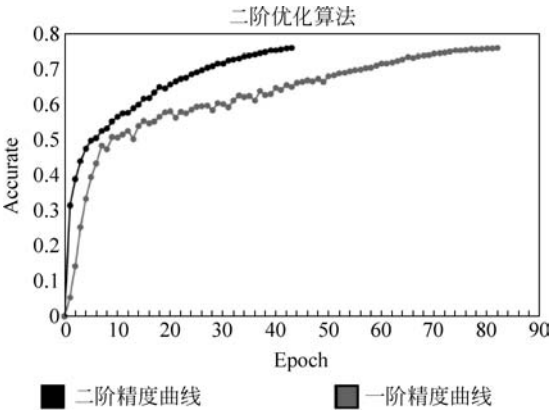

图 5-8　二阶优化算法 THOR 精度表现

5.1.2　TensorFlow

Google 公司在 2015 年 11 月正式开源发布 TensorFlow。TensorFlow 在很大程度上可以看作早期深度学习框架 Theano 的后继者，不仅因为它们有很大一批共同的用户，更因为它们有相近的设计理念——基于静态计算图实现的自动微分系统。

TensorFlow 采用静态图的运行模式，在编译执行前就构建一个静态计算图，定义所有的网络结构，然后再执行相应操作。从理论上讲，静态计算允许编译器做更大程度上的优化，但这也意味着程序与编译器之间存在着更多的代沟，这带来了计算图在运行时无法修改且代码中的错误难以发现等问题。尽管如此，TensorFlow 还是凭借着支持分布式训练、部署能力强、社区活跃度高等特点得到了工业界广泛的应用。因此，CANN 对 TensorFlow 框架进行了适配和支持，很好地发挥了昇腾产业技术优势。

作为当前主流的深度学习框架，TensorFlow 获得了巨大的成功，但图方法这种非 Python 原生的编程方式却始终备受争议；TensorFlow 创造了图、会话、命名空间等诸多抽象概念，需要普通用户花费较多的时间进行学习。且在代码层面，面对同一个功能，TensorFlow 提供了多种“良莠不齐”的实现，使用中有细微的差异，接口还一直处于快速迭代之中，版本之间存在不兼容的问题，这也引发了用户的颇多争议。

5.1.3　PyTorch

2017 年 1 月，Facebook 人工智能研究院在 GitHub 上开源了 PyTorch，迅速占领了 GitHub 热度榜榜首。与 TensorFlow 的静态计算图不同的是，PyTorch 采用了动态图模式，在每次前向传播时都会创建一幅新的计算图，这也就代表着用户可以随时定义、更改和执行结点，这种更贴近 Python 编程习惯的机制也使得调试更加容易。PyTorch 专注于快速原型设计和研究的灵活性，很快就成为 AI 研究人员的热门选择。

但这种动态图机制也有明显的漏洞，PyTorch 需要依赖宿主语言的编译器，并且使用 Tape 模式去记录运行过程，因此会产生较大的开销，且这种动态方式也不利于模型整体的性能优化。当需要将模型部署在跨平台和嵌入式平台上时，PyTorch 也往往显得“力不从心”，通常需要将模型转换为 Caffe2 并使用 C++ 改写推理代码，或者使用 REST 来配置服务器，模型的部署和加速难度较大，难以满足性能、体积、能耗、可信等工业级诉求。

CANN 对 PyTorch 进行了适配性开发，仅需少量的代码迁移工作就能使用昇腾平台的强劲计算能力，这使得使用昇腾计算体系的用户能够很好地跟进学术前沿发展。

5.1.4　主流框架对比

总体来说，MindSpore 是一种适用于全场景的新型开源深度学习训练框架，对下利用 CANN 能最大程度发挥昇腾处理器能力，对上提供网络编程 API 供使用者高效便捷地开发 AI 应用程序。表 5-1 总结了市面上主流深度学习框架的比较，由于篇幅所限，本书不再展开介绍每一种框架的具体表现，可以参考官网了解更多详情。

表 5-1　主流深度学习框架的比较

竞争力		TensorFlow	PyTorch	PaddlePaddle	MindSpore
高阶特性	并行度	数据或模型并行	数据或模型并行	数据并行	数据与模型自动并行
	二阶优化	不支持	不支持	不支持	支持
	动静一致	静态图好 动态图不足	动态图好 静态图不足	动、静态图支持	动、静态图支持
	安全与隐私	TF-Privacy/ TF-encrypted	Opacus/AdverBox	PaddlePaddleFL/ AdvBox	MindArmour
完备特性	端边云全场景	需要转换	需要转换	需要转换	架构统一
	支持硬件	GPU CPU TPU	GPU CPU	GPU CPU TPU	GPU CPU TPU
	运行平台	Linux Mac Windows Andriod	Linux Mac Wndows	Linux Mac Windows Android	Windows Linux Android
	语言支持	Python C/C++，Java， Go，R，Julia，Swift	Python	Python，Go，R， C/C++，Java	Python C/C++
	可视化	TensorBoard	TensorBoardX Visdom	VisulDL	Mindinsight
生态	预训练模型	CV、NLP、Rec、 Speech 场景 700＋	CV、NLP 场景 30＋	CV、NLP、Rec、 Speech 场景 200＋	CV、NLP、Rec 场景 60＋

5.2 深度学习训练流程

第 1 章对深度学习的基本概念作了简要介绍。本节将从流程上对相关知识进行细化。借助深度学习框架的强大能力，用户可以很大程度上省去部署和适配环境的烦恼，也可以省去编写大量底层代码的精力。但无论使用何种训练框架，整个训练流程都可以被定义为如图 5-9 所示的五个步骤。

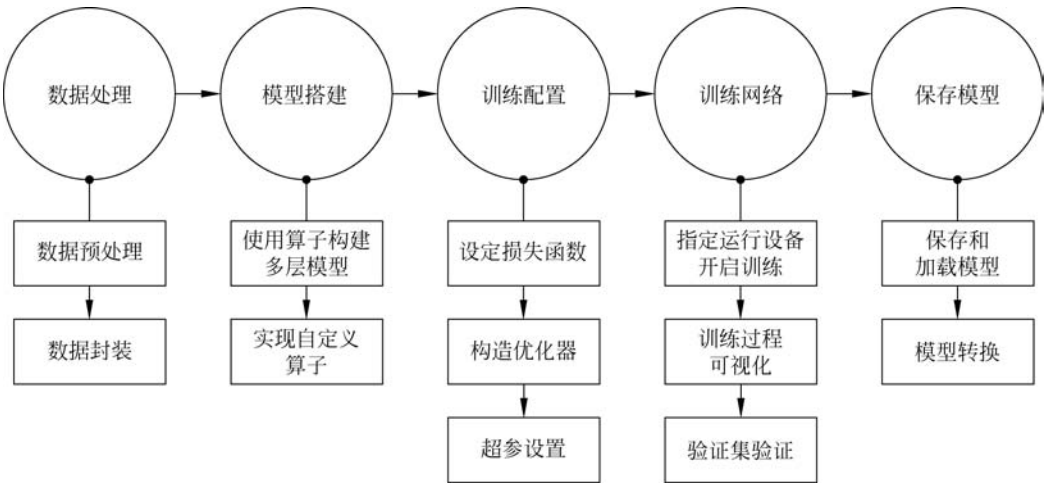

图 5-9 深度学习训练过程

5.2.1 数据处理

深度学习本质上是一种数据驱动的算法，数据质量在模型训练的过程中起到了至关重要的作用。为了训练出符合预期的模型，第一步就要对原始数据进行预处理。

从广义上讲，数据预处理可以视作正式进入深度网络计算前的一切操作。从狭义上看，除了常规的对数据进行归一化、白化、缩放、裁剪、仿射变换外，用于缓解数据不平衡的数据增广和用于提高模型鲁棒性的数据随机打乱(Shuffle)都可以视作数据预处理的一个环节。而当面对真实场景的工业问题时，往往还要面对数据质量不佳、数据规格不统一等问题。对于原始数据的清洗、缺省值处理、格式统一也是数据预处理环节的重要内容。

随着深度学习的不断发展，无论是在视觉领域还是在文本处理领域，数据都常因容量限制无法直接全部读入内存，因此需要分批次(Batch)读取数据并传递给深度学习模型。

绝大多数的深度学习训练框架都提供了实现相关功能的接口，如图 5-10 所示，MindSpore 提供的 mindspore.dataset 模块可以帮助用户构建数据集对象，使得数据在训练过程中能像经过管道中的水一样源源不断地流向训练系统。具体来看，用户可以将非标准的数据集和常用的数据集转换为 MindSpore 数据格式，即 MindRecord，从而方便地加载到 MindSpore 中进行训练。同时，MindSpore 在部分场景做了性能优化，相关的数据集构建方式将在 5.3 节中展开介绍。

图 5-10　MindSpore 构造 Dataset

5.2.2　模型搭建与训练配置

数据在传入深度学习模型之后，会进行多轮计算。每轮模型训练的过程都可以分为以下三步：第一步，进行前向传播，根据输入数据计算出模型预测的输出；第二步，根据预测值和真实值计算损失；第三步，根据损失（通常是最小化损失）进行反向传播，传递梯度并更新参数。这三个步骤也突显了建构深度学习模型时三个重要的关注点：网络结构、损失函数（loss function）和优化算法。

由多层组成的神经网络模型是训练过程中的核心，模型的不同网络结构代表着不同的表征能力，得益于强大的深度学习框架，绝大部分常见的算子如卷积、池化、激活、反卷积等均已被实现并封装成 API 供用户使用。如在 MindSpore 中，就可以基于 nn.Cell 基类，通过初始化__init__方法和 construct 构造方法构造网络模型。昇腾官方也已经预先实现好了很多场景下的典型模型并开源在 ModelZoo 中。用户可以通过昇腾官网来查看和下载丰富的深度学习模型，借助昇腾计算的强大能力进行学习和训练。

用户也可以自行构建网络，将最新结构和自己的创新思路进行实现。当需要自定义一些操作时，可以参考第 3 章介绍的 TBE 相关知识进行开发；将第三方代码直接迁移到昇腾计算平台训练的方法将在 5.5 节中介绍。

损失函数，又称目标函数，用来衡量预测值与真实值差异的程度。在深度学习中，模型训练就是通过不停地迭代来缩小损失函数值的过程。损失函数越小，一般就代表模型的学习效果越好，也正是损失函数指导了模型的学习。因此，在模型训练过程中损失函数的选择非常重要，定义一个好的损失函数，可以有效提高模型的性能。如程序清单 5-2 实现的 L1 损失函数所示，MindSpore 提供了许多通用损失函数供用户选择，但这些通用损失函数并不适用于所有场景，很多时候需要用户自定义所需的损失函数。

程序清单 5-2　L1 损失函数

```
import mindspore.nn as nn
import mindspore.ops as ops
class L1Loss(nn.Cell):
    def __init__(self):
        super(L1Loss, self).__init__()
        self.abs = ops.Abs()
        self.reduce_mean = ops.ReduceMean()
    def construct(self, base, target):
        x = self.abs(base - target)
        return self.reduce_mean(x)
```

当根据输入得到一批次数据的损失函数值之后，通常使用随机梯度下降（SGD）或自适应矩估计（Adam）等优化算法来更新参数。许多深度学习框架都需要用户手动求导并计算梯度，MindSpore 的自动微分机制采用函数式可微分编程架构，帮助用户聚焦模型算法的数学原生表达而无须手动进行求导，自动微分的样例代码如程序清单 5-3 所示。

程序清单 5-3　自动微分样例代码

```
import mindspore as ms
from mindspore import ops
grad_all = ops.composite.GradOperation()

def func(x): return x * x * x
def df_func(x):
return grad_all(func)(x)

@ms.ms_function
def df2_func(x):
return grad_all(df_func)(x)

if __name__ == "__main__":
    print(df2_func(ms.Tensor(2, ms.float32)))
```

其中，第一步定义了 func 函数，第二步利用 MindSpore 提供的反向接口进行自动微分，定义了一个一阶导数函数，第三步定义了一个二阶导数函数，最后给定输入就能获取第一步定义的函数在指定处的二阶导数，二阶导数求导结果为 12。当然，这些函数在后续执行中都会被解析为一个子图。

优化算法需要传入参数才能使用。机器学习领域一般有两类参数，一类是模型内部参数，依靠训练数据来对模型参数进行迭代优化；另一类则是模型外部的设置参数，需要人工配置，这类参数被称为“超参数”。以学习率为代表的一系列超参数也会很大程度上影响模型的训练效果，如图 5-11 所示，太大的学习率会使得模型无法收敛至最优状态，而

太小的学习率会使得训练时间过久，甚至陷入局部最小值而无法跳出，所以设置合适的超参数也是训练出预期模型的关键。而在训练前，用户往往不清楚一个特定问题设置成怎样的学习率是合理的，因此在训练时往往需要设置不同的超参数进行实验，类似的模型调优过程在深度学习训练中是十分必要的，一套优秀的超参往往是在理论和经验的共同指导下产生的，在这个过程中可以配合 MindInsight 工具，通过观察 Loss 下降的情况判断合适的学习率。

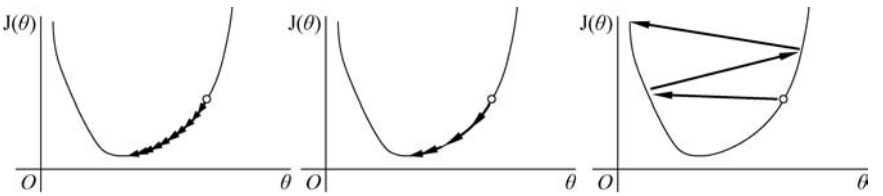

图 5-11　不同学习率对收敛效果的影响

5.2.3　训练网络与保存模型

在完成数据处理和网络模型搭建后，就可以调用 train 函数执行整个训练过程，程序清单 5-4 展示了 MindSpore 从模型搭建到执行训练的完整代码流程。

程序清单 5-4　执行训练代码示例

```
from mindspore.train.loss_scale_manager import FixedLossScaleManager

dataset = create_custom_dataset()
net = Net()
loss = nn.SoftmaxCrossEntropyWithLogits()
# 由于使用混合精度进行训练，故使用 loss scale manager 进行管理
loss_scale_manager = FixedLossScaleManager()
optim = Momentum(params=net.trainable_params(), learning_rate=0.1, momentum=0.9)
model = Model(net, loss_fn=loss, optimizer=optim, metrics=None, loss_scale_manager=
loss_scale_manager)

model.train(2, dataset)
```

在训练复杂模型时，往往需要花费大量的时间，使用可视化工具可以有效监督训练过程。Tensorboard 就是一个有效的可视化工具包，可以跟踪可视化损失及准确率等指标，也可以可视化模型图（操作和层）。由 MindStudio 提供的 MindInsight 还实现了模型溯源、数据溯源等强大的功能，甚至可以通过其提供的 MindOptimizer 进行超参搜索，根据用户配置，从训练日志中提取以往训练记录，推荐超参，最后自动地执行训练脚本。关于

MindInsight 的设计原理和具体使用方法可以参考官方规格文档[①]。

在模型训练过程中，可以添加检查点(Checkpoint)用于保存模型的参数，以便执行推理及再训练使用。具体来看，深度学习框架使用回调机制（Callback）传入 ModelCheckpoint 对象，可以保存模型参数，生成 Checkpoint 文件，以供推理或迁移学习时使用。用户也可以通过 CheckpointConfig 对象设置保存策略，选定保存格式和数量，程序清单 5-5 是保存模型的一个示例。

程序清单 5-5　MindSpore 保存模型示例

```
from mindspore.train.callback import ModelCheckpoint, CheckpointConfig
config_ck = CheckpointConfig(save_checkpoint_steps=32, keep_checkpoint_max=10)
ckpoint_cb = ModelCheckpoint(prefix='resnet50',directory=None,config=config_ck)
model.train(epoch_num, dataset, callbacks=ckpoint_cb)
```

5.3　CANN 训练实例之 MindSpore

5.1 节和 5.2 节已经对市面上主流的深度学习框架和深度学习训练的整体流程进行介绍。本节以图像分类算法 ResNet-50 为例，详细讲解如何在 CANN 统一异构计算架构上使用 MindSpore 进行模型训练。

5.3.1　环境搭建

本书第 2 章介绍了通过命令行方式安装 OS 依赖、固件、驱动和 CANN 软件包的方法。除了上述内容外，这里还需要安装 MindSpore 生产环境。目前 MindSpore 支持在 euleros_aarch64/ centos_aarch64/ centos_x86/ ubuntu_aarch64/ ubuntu_x86 上运行，安装方式也可以采用 pip 安装、source 安装和 Docker 安装三种方式，其中 pip 是一个安装、管理 Python 软件包的工具。本节将以使用 pip 安装昇腾 910 环境的 Linux 为例进行示范。

第一步，安装 pip 并确认系统环境。对于 Ubuntu 18.04/EulerOS 2.8 用户，需要保证 GCC >= 7.3.0，还需要确认安装 GNU 多重精度运算库(GNU Multiple Precision Arithmetic Library)。如果在昇腾 910 上开发代码，还需要确认安装昇腾 910 AI 处理器软件配套包，程序清单 5-6 列出了用于查看相应环境时可能使用到的一些指令。

① MindInsight 相关内容参见链接 https://www.mindspore.cn/doc/note/zh-CN/r1.1/design/mindinsight.html。

程序清单 5-6　配置 MindSpore 环境时常用指令

```
# 在 Ubuntu/Linux 64 bit 环境下安装 pip
sudo apt-get install python3-pip
# 查看 pip 版本
Python3 -m pip --version
# 查看 gcc 版本
gcc --version
# 在 Acend910 环境下安装配套软件包,参考以下命令,{version}需要根据实机版本替换
pip install /usr/local/Ascend/ascend-toolkit/latest/fwkacllib/lib64/topi-{version}-
py3-none-any.whl
pip install /usr/local/Ascend/ascend-toolkit/latest/fwkacllib/lib64/te-{version}-py3
-none-any.whl
pip install /usr/local/Ascend/ascend-toolkit/latest/fwkacllib/lib64/hccl-{version}-
py3-none-any.whl
```

第二步,获取 MindSpore 安装指令。目前,MindSpore 已经推出了支持昇腾 910、昇腾 310、CPU 和 GPU 的稳定版,用户可以在官网上[①]根据实际需求获得相对应的指令,其安装指令通式如程序清单 5-7 所示。

程序清单 5-7　MindSpore 安装指令通式

```
pip install https://ms-release.obs.cn-north-4.myhuaweicloud.com/{version}/
MindSpore/ascend/{system}/mindspore_ascend-{version}-cp37-cp37m-linux_{arch}.
whl \
--trusted-host ms-release.obs.cn-north-4.myhuaweicloud.com \
-i https://pypi.tuna.tsinghua.edu.cn/simple
```

指令中{version}表示 MindSpore 版本号,例如安装 1.1.0 版本 MindSpore 时,{version}应写为 1.1.0。{arch}表示系统架构,例如使用的 Linux 系统是 x86 架构 64 位时,{arch}应写为 x86_64。如果系统是 ARM 架构 64 位,则写为 aarch64。{system}表示系统版本,例如使用的欧拉系统 ARM 架构,{system}应写为 euleros_aarch64。在联网状态下,安装 whl 包时也会自动下载 MindSpore 安装包的依赖项,具体依赖可参考 MIndSpore 开源代码库中 requirements.txt 文件[②]。

第三步,配置环境变量。如果昇腾 910 AI 处理器配套软件包没有安装在默认路径,安装好 MindSpore 之后,需要导出 Runtime 相关环境变量,下述命令中 LOCAL_ASCEND=/usr/local/Ascend 的/usr/local/Ascend 表示配套软件包的安装路径,需注意将其改为配套软件包的实际安装路径。昇腾 910 配置 MindSpore 环境变量如程序清单 5-8 所示。

① MindSpore 安装指令获取地址为 https://www.mindspore.cn/install。

② 完整的安装依赖项的参考链接为 https://gitee.com/mindspore/mindspore/blob/r1.1/requirements.txt。

程序清单 5-8　昇腾 910 配置 MindSpore 环境变量

```
# Log 等级 0 - DEBUG, 1 - INFO, 2 - WARNING, 3 - ERROR, 默认 warning
export GLOG_v = 2
# 配置 conda 环境
LOCAL_ASCEND = /usr/local/Ascend
# 配置依赖环境
Export LD_LIBRARY_PATH = ${LOCAL_ASCEND}/add - ons/: ${LOCAL_ASCEND}/ascend - toolkit/
latest/fwkacllib/lib64: ${LOCAL_ASCEND}/driver/lib64: ${LOCAL_ASCEND}/ascend - toolkit/
latest/opp/op_impl/built - in/ai_core/tbe/op_tiling: ${LD_LIBRARY_PATH}

# 需要配置的其他变量
export TBE_IMPL_PATH = ${LOCAL_ASCEND}/ascend - toolkit/latest/opp/op_impl/ built - in/ai
_core/tbe
export ASCEND_OPP_PATH = ${LOCAL_ASCEND}/ascend - toolkit/latest/opp
export PATH = ${LOCAL_ASCEND}/ascend - toolkit/latest/fwkacllib/ccec_compiler /bin/:
${PATH}
export PYTHONPATH = ${TBE_IMPL_PATH}: ${PYTHONPATH}
```

第四步，验证安装。安装完成后就可以使用 Python 或 Python3 进入编译器，输入如程序清单 5-9 所示的代码进行简单的张量相加验证，如果其输出了一个大小为[1,2]的全为 2 的张量，则代表 MindSpore 安装成功。

程序清单 5-9　验证 MindSpore 是否安装成功

```
import numpy as np
from mindspore import Tensor
import mindspore.ops as ops
import mindspore.context as context

context.set_context(device_target = "Ascend")
x = Tensor(np.ones([1,2]).astype(np.float32))
y = Tensor(np.ones([1,2]).astype(np.float32))
print(ops.tensor_add(x, y))
```

5.3.2　ResNet-50 实现图像分类

在深度学习算法的发展中，ResNet 是一个具有里程碑式意义的网络，它的出现让训练成百上千层的网络成为可能。5.2 节已经介绍了 MindSpore 运行所需的硬件、后端等基本信息并完成了相关配置。本节将基于 CIFAR-10 数据集，使用 ResNet-50 完成图像分类模型的训练，相关的代码已在昇腾模型库中开源，用户可以通过 gitee 上的 ModelZoo

代码库[①]获取代码。

1. 数据集介绍及数据处理

CIFAR-10 是一个用于普适物体识别的计算机视觉数据集，该数据集共有 60000 张大小为 32×32 的彩色图片，共分为 10 个类，每类 6000 张图。其中测试集是由从每类中随机选取 1000 张图片组合而成的，剩下的 5000 张图片则随机排列组成了训练集。与 mnist 数据集相比，CIFAR-10 含有的是现实世界中真实的物体，不仅噪声很大，而且物体的比例、特征都不尽相同，这为识别带来很大困难。用户可以在其官网下载完整的数据集[②]，图 5-12 为 CIFAR-10 数据集示例。

图 5-12　CIFAR-10 数据集示例

在 CIFAR-10 数据集中，文件 data_batch_1. bin、data_batch_2. bin 、data_batch_5. bin 和 test_ batch. bin 中各有 10000 个样本。一个样本由 3073 字节组成，第一个字节为标签 label，剩下 3072 字节为图像数据，CIFAR-10 中各文件的说明如表 5-2 所示。

表 5-2　CIFAR-10 中各文件的说明

文　件　名	解 释 说 明
batches. meta. txt	保存 10 个类别的类别名
readme. html	数据集介绍文件
data_batch_1. bin … data_batch_5. bin	训练数据，每个文件以二进制格式保存 10000 张彩色图片和对应的标签，一共 50000 张
test_batch. bin	测试图像和测试图像标签

① Modelzoo 中 ResNet-50 代码地址：https://gitee. com/ascend/modelzoo/tree/master/built-in/MindSpore/Official/cv/image_classification/ResNet50_for_MindSpore。

② CIFAR-10 官网：https://www. cs. toronto. edu/~kriz/cifar. html。

二进制的原始文件无法直接被模型使用，用户需要首先对其进行解析。使用框架提供的数据集解析引擎可以生成供模型使用的迭代器。在 MindSpore 中，可以通过内置数据集格式 Cifar10Dataset 接口完成。在解析数据集后，可以自定义数据增强方式，并使用 map 方法在数据上执行这些预处理的算子和函数，最后通过对数据混洗（shuffle）随机打乱数据的顺序，并按 batch 读取数据进行训练。具体来看，数据解析和预处理的流程图如图 5-13 所示，其代码如程序清单 5-10 所示。

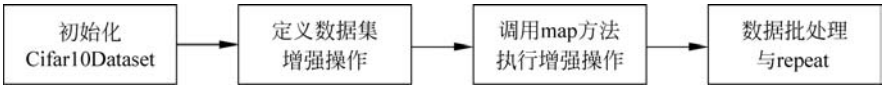

图 5-13　数据解析与预处理流程

通过程序清单 5-10 不难发现，Mindspore 还在 dataset.transforms.c_transforms 中提供了丰富的数据预处理方法。RandomCrop 函数可将图片在任意位置进行随机裁剪，RandomHorizontalFlip 可以使得图片以 prob 概率进行水平翻转，这两个操作可以在训练阶段起到很好的数据增广效果。除此之外，程序清单 5-10 中还定义了一套归一化方案，resize 函数将图片剪裁为统一大小，Rescale 和 Normalize 将各点的像素值归一化到 0～1。HWC2CHW 函数将输入图像格式从“高-宽-通道”顺序调整为“通道-高-宽”，方便后续环节进行计算加速。

程序清单 5-10　MindSpore 进行数据解析和预处理

```
import mindspore.dataset.engine as de
import mindspore.dataset.vision.c_transforms as C
import mindspore.dataset.transforms.c_transforms as C2
def create_dataset1(dataset_path, do_train, repeat_num = 1, batch_size = 32, target = "
Ascend"):
    ds = de.Cifar10Dataset(dataset_path, num_parallel_workers = 8, shuffle = True)
    # 定义数据集增强操作
    trans = []
    if do_train:
        trans += [
            C.RandomCrop((32, 32), (4, 4, 4, 4)),
            C.RandomHorizontalFlip(prob = 0.5)]
]
trans += [
        C.Resize((224, 224)),
        C.Rescale(1.0 / 255.0, 0.0),
        C.Normalize([0.4914, 0.4822, 0.4465], [0.2023, 0.1994, 0.2010]),
        C.HWC2CHW()
]
    type_cast_op = C2.TypeCast(mstype.int32)
```

```
    # 调用 map 方法执行自定义预处理方法
    ds = ds.map(operations = type_cast_op, input_columns = "label", num_parallel_workers = 8)
    ds = ds.map(operations = trans, input_columns = "image", num_parallel_workers = 8)

    # 执行 batch 和 repeat 操作
    ds = ds.batch(batch_size, drop_remainder = True)
    ds = ds.repeat(repeat_num)
    return ds
```

2. 搭建神经网络

ResNet 在传统卷积神经网络的基础上通过短路机制引入了残差单元，有效地解决了深度学习退化问题。具体来看，在定义卷积、全连接等带参网络层时，需要对其中参数进行随机初始化，随机初始化函数的代码如程序清单 5-11 所示。

程序清单 5-11　参数随机初始化函数

```
def _weight_variable(shape, factor = 0.01):
    init_value = np.random.randn( * shape).astype(np.float32) * factor
    return Tensor(init_value)
```

卷积函数已经由框架给出，在使用时只需要定义输入维度、输出维度、卷积核大小、初始化参数、边缘填充方式等内容就可以使用了。不同尺度卷积核的卷积函数如程序清单 5-12 所示。

程序清单 5-12　不同尺度卷积核的卷积函数

```
def _conv3x3(in_channel, out_channel, stride = 1, use_se = False):
    weight_shape = (out_channel, in_channel, 3, 3)
    weight = _weight_variable(weight_shape)
    return nn.Conv2d(in_channel, out_channel,kernel_size = 3, stride = stride, padding =
0, pad_mode = 'same', weight_init = weight)

def _conv1x1(in_channel, out_channel, stride = 1, use_se = False):
    weight_shape = (out_channel, in_channel, 1, 1)
    weight = _weight_variable(weight_shape)
    return nn.Conv2d(in_channel, out_channel,kernel_size = 1, stride = stride, padding =
0, pad_mode = 'same', weight_init = weight)

def _conv7x7(in_channel, out_channel, stride = 1, use_se = False):
    weight_shape = (out_channel, in_channel, 7, 7)
    weight = _weight_variable(weight_shape)
    return nn.Conv2d(in_channel, out_channel,kernel_size = 7, stride = stride, padding =
0, pad_mode = 'same', weight_init = weight)
```

同卷积类似，定义初始化参数之后，使用 nn.Dense 可以构造全连接层；ResNet 网络还使用了 BatchNorm 层，在卷积层的后面加上 BatchNorm 可以有效地提升训练过程中的数值稳定性。从代码上看，只需要调用 nn.BatchNorm2d 接口就能构建批归一化层了。

通过上述介绍的网络层，就能构造出 ResNet 中最重要的结构——残差块(Residual Block)，残差块的结构如图 5-14 所示。

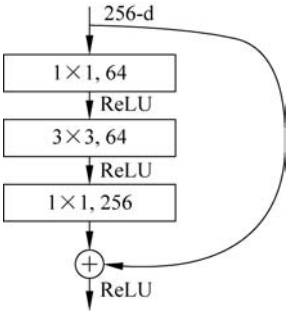

图 5-14　残差块结构

在用代码实现时，可以使用创造 Python 类的方法完成模型结构的定义，这个类需要继承 nn.Cell 父类，并且在类中定义 init 函数和 construct 函数。init 是初始化函数，construct 则是框架指定前向传播时使用的计算框架，主函数在调用模型实例时会自动执行 construct 方法。值得注意的是，在 construct 中使用的网络层都需要在 init 函数中进行声明。

在残差学习单元的结构中，残差块内共有三个卷积 1×1、3×3、1×1，它们分别完成维度压缩、卷积、恢复维度的功能，然后跟输入图片进行短接，这样的做法可以有效地降低计算复杂度。值得注意的是，如果残差块中第三次卷积输出特征图的形状与输入不一致，则对输入图片做 1×1 卷积，将其输出形状调整成一致，其具体的构造代码如程序清单 5-13 所示。

程序清单 5-13　残差块构建代码

```
class ResidualBlock(nn.Cell):
    def __init__(self,in_channel,out_channel, stride = 1):
        super(ResidualBlock, self).__init__()
        self.stride = stride
        channel = out_channel // 4
        self.conv1 = _conv1x1(in_channel, channel, stride = 1)
        self.bn1 = _bn(channel)
        self.conv2 = _conv3x3(channel, channel, stride = stride, use_se = self.use_se)
        self.bn2 = _bn(channel)
        self.conv3 = _conv1x1(channel, out_channel, stride = 1, use_se = self.use_se)
        self.bn3 = _bn_last(out_channel)
        self.relu = nn.ReLU()
        self.down_sample = False
        if stride != 1 or in_channel != out_channel:
            self.down_sample = True
        self.down_sample_layer = None
        if self.down_sample:
            self.down_sample_layer = nn.SequentialCell([_conv1x1(in_channel, out_
channel, stride), _bn(out_channel)])
        self.add = P.TensorAdd()
```

```
    def construct(self, x):
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)
        out = self.conv3(out)
        out = self.bn3(out)
        out = self.add(out, identity)
        out = self.relu(out)
        return out
```

使用残差块能够搭建 ResNet 整网。与上文介绍的类似，用户也可将模型结构代码封装为类。通过这样的方式，还可以很便捷地进行泛化操作，仅需调整传入构造函数的参数就可以实现不同层数的结构。init 函数中定义了网络各层的结构。ResNet 模型的初始化代码如程序清单 5-14 所示；在类中还定义了辅助的工具方法_make_layer，方便构建不同输入输出大小的网络层，其代码如程序清单 5-15 所示；construct 函数中包含了前向传播时的网络结构，其代码如程序清单 5-16 所示。

程序清单 5-14　ResNet 模型的初始化代码

```
def __init__(self, block, layer_nums, in_channels, out_channels, strides, num_classes):
    super(ResNet, self).__init__()

    if not len(layer_nums) == len(in_channels) == len(out_channels) == 4:
        raise ValueError("the length must be 4!")
    self.conv1 = _conv7x7(3, 64, stride=2)
    self.bn1 = _bn(64)
    self.relu = P.ReLU()
    self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, pad_mode="same")
    self.layer1 = self._make_layer(block, layer_nums[0], in_channel=in_channels[0],
out_channel=out_channels[0], stride=strides[0])
    self.layer2 = self._make_layer(block, layer_nums[1], in_channel=in_channels[1],
out_channel=out_channels[1], stride=strides[1])
    self.layer3 = self._make_layer(block, layer_nums[2], in_channel=in_channels[2],
out_channel=out_channels[2], stride=strides[2])
    self.layer4 = self._make_layer(block, layer_nums[3], in_channel=in_channels[3],
out_channel=out_channels[3], stride=strides[3])
    self.mean = P.ReduceMean(keep_dims=True)
    self.flatten = nn.Flatten()
    self.end_point = _fc(out_channels[3], num_classes)
```

程序清单 5-15　ResNet 工具方法_make_layer

```
def _make_layer(self, block, layer_num, in_channel, out_channel, stride):
    layers = []

    resnet_block = block(in_channel, out_channel, stride=stride)
    layers.append(resnet_block)

    for _ in range(1, layer_num):
        resnet_block = block(out_channel, out_channel, stride=1)
        layers.append(resnet_block)
    return nn.SequentialCell(layers)
```

程序清单 5-16　ResNet 前向传播 construct 函数

```
def construct(self, x):
    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)
    c1 = self.maxpool(x)

    c2 = self.layer1(c1)
    c3 = self.layer2(c2)
    c4 = self.layer3(c3)
    c5 = self.layer4(c4)

    out = self.mean(c5, (2, 3))
    out = self.flatten(out)
    out = self.end_point(out)
    return out
```

ResNet-50 包含多个模块，其中第 2～5 个模块分别包含 3、4、6、3 个残差块，其调用函数如程序清单 5-17 所示。

程序清单 5-17　ResNet-50 的调用函数

```
def resnet50(class_num=10):
    return ResNet(ResidualBlock,
                  [3, 4, 6, 3],
                  [64, 256, 512, 1024],
                  [256, 512, 1024, 2048],
                  [1, 2, 2, 2],
                  class_num)
```

3. 定义损失函数和优化器

定义完网络结构后只需简单的调用就能创建模型了。但正如 5.2 节介绍的，一个完整的深度学习程序还需要定义损失函数、优化器、学习率衰减等配置才能开始训练。

本案例使用带有 softmax 归一化函数的交叉熵作为损失函数，这是在分类任务中常用的损失函数。其中的 softmax 函数能使模型的原始输出 y_i 转换成 0～1 的概率，且保证所有的 softmax(x_i)相加的总和为 1，这起到了归一化的作用，其函数定义如式(5-1)所示。

$$\mathrm{softmax}(x)=\frac{\mathrm{e}^{x_i}}{\sum_j \mathrm{e}^{x_j}} \tag{5-1}$$

通过 softmax 获得模型对各类别预测的概率，直接将其与标签的真实值对比并不合理。在实际使用中，通常使用交叉熵误差作为分类的损失。用户可以从熵的角度来理解交叉熵，受限于篇幅，这里不过多从信息论的角度来介绍熵、互信息、KL 散度、交叉熵等概念，在此直接给出交叉熵的数学表达式，如式(5-2)所示。

$$L=-\left[\sum_{k=1}^{n} t_k \log y_k+(1-t_k)\log(1-y_k)\right] \tag{5-2}$$

式中，log 表示以 e 为底数的自然对数，y_k 表示模型的输出，t_k 表示真实标签，它是一个独热(one-hot)编码，也就是其中只有正确解的位置是 1，其他位置都是 0。因此，交叉熵只计算对应正确解标签输出的自然对数。正确解标签对应的输出越大，交叉熵的值越接近 0；反之，正确解标签对应的输出越小，则交叉熵的值越大。

在具体的代码实现中，可以直接使用 nn.SoftmaxCrossEntropyWithLogits 来便捷地实现交叉熵损失，这个函数可以用来衡量预测值和标签之间的差距，reduction 参数表示损失最终的聚合方式，可以选择 sum、mean、None。本实例使用 mean 作为损失聚合方式。

从代码构造的角度来看，可以将其封装为 Python 类，继承 nn.loss.loss 父类，使用 init 函数进行初始化，construct 函数表示前向传播时的逻辑。但值得注意的是，可以使用 ops.operations.OneHot 函数对输入标签进行优化，这个函数可以将负样本点的值设置为 off_value，正样本点的值设置为 on_value，这样可以实现很好的平滑效果。有关交叉熵损失函数的完整代码如程序清单 5-18 所示。

程序清单 5-18　交叉熵损失函数的完整代码

```
import mindspore.nn as nn
from mindspore import Tensor
from mindspore.common import dtype as mstype
from mindspore.nn.loss.loss import _Loss
```

```
from mindspore.ops import functional as F
from mindspore.ops import operations as P

class CrossEntropySmooth(_Loss):
    def __init__(self, sparse = True, reduction = 'mean', smooth_factor = 0., num_classes =
1000):
        super(CrossEntropySmooth, self).__init__()
        self.onehot = P.OneHot()
        self.sparse = sparse
        self.on_value = Tensor(1.0 - smooth_factor, mstype.float32)
        self.off_value = Tensor(1.0 * smooth_factor / (num_classes - 1), mstype.
float32)
        self.ce = nn.SoftmaxCrossEntropyWithLogits(reduction = reduction)

    def construct(self, logit, label):
        if self.sparse:
            label = self.onehot(label, F.shape(logit)[1], self.on_value, self.off_value)
        loss = self.ce(logit, label)
        return loss
```

正如在 5.2 节中介绍的，在训练模型的过程中会使用优化算法不断迭代模型参数以降低模型损失函数的值。本实例使用流行的动量法（Momentum）进行参数优化。动量法是传统梯度下降法的一种扩展，它不仅会使用当前梯度，还会积累之前的梯度以确定优化的走向。这样的优化算法可以加速模型的收敛，同时抑制震荡，使参数更新的方向更稳定。在具体的代码中，使用 mindspore.nn.optim.momentum 接口定义 Momentum 优化器，传入网络信息和所需的超参信息，如学习率、冲量系数等，最后调用 model.train 就可以开始训练了。

一般来说，为了获得更好的训练效果就需要在训练前期将学习率设置大一些，使得网络收敛迅速，而在训练后期将学习率设置小一些，使得网络更好地收敛到最优解。相比于固定的学习率，可以使用衰减策略动态调整学习率。常见的衰减策略有以下四种：分段常数衰减、指数衰减、多项式衰减、余弦衰减。受限于篇幅，在此不再将各种衰减方式的特点进行展开介绍。但值得介绍的是，ResNet 论文[①]提到了一种学习率预热（warmup）的方法，它在训练开始时先选择使用一个较小的学习率，训练了一些轮次后，再修改为预先设置的学习率来进行训练。这是因为刚开始训练时，模型的权重是随机初始化的，此时若选择一个较大的学习率，可能带来模型的剧烈震荡。在预热的小学习率下，模型可以慢慢趋于稳定，等模型相对稳定后再选择预先设置的学习率进行训练，模型收敛速度变得更快，模型效果更佳。带有 warmup 的余弦衰减学习率的具体实现代码如程序清单 5-19 所示。

① 论文题目为：Deep Residual Learning for Image Recognition(https://arxiv.org/pdf/1512.03385.pdf)。

程序清单 5-19　带有 warmup 的余弦衰减学习率

```
def warmup_cosine_annealing_lr(lr, steps_per_epoch, warmup_epochs, max_epoch = 120,
global_step = 0):
    """
       lr(float): 初始学习率
       steps_per_epoch(int): 每轮迭代有多少步
       warmup_epochs(int): warmup 的轮次
       max_epoch(int): 总的训练轮次
       global_step(int): 当前的训练步数
    Returns:
       np.array, 学习率数组
    """
    base_lr = lr
    warmup_init_lr = 0
    total_steps = int(max_epoch * steps_per_epoch)
    warmup_steps = int(warmup_epochs * steps_per_epoch)
    decay_steps = total_steps - warmup_steps
    lr_each_step = []
    for i in range(total_steps):
        if i < warmup_steps:
            lr = linear_warmup_lr(i + 1, warmup_steps, base_lr, warmup_init_lr)
        else:
            linear_decay = (total_steps - i) / decay_steps
            cosine_decay = 0.5 * (1 + math.cos(math.pi * 2 * 0.47 * i / decay_steps))
            decayed = linear_decay * cosine_decay + 0.00001
            lr = base_lr * decayed
        lr_each_step.append(lr)
    lr_each_step = np.array(lr_each_step).astype(np.float32)
    learning_rate = lr_each_step[global_step:]
    return learning_rate
def linear_warmup_lr(current_step, warmup_steps, base_lr, init_lr):
    lr_inc = (float(base_lr) - float(init_lr)) / float(warmup_steps)
    lr = float(init_lr) + lr_inc * current_step
    return lr
```

学习率模块返回的学习率数组可以传给优化器，有关优化器的代码如程序清单 5-20 所示。

程序清单 5-20　优化器

```
lr = warmup_cosine_annealing_lr(config.lr, step_size, config.warmup_epochs, config.
epoch_size, config.pretrain_epoch_size * step_size)
lr = Tensor(lr)
# Momentum 可传入一个固定值/迭代器/一维 Tensor 作为学习率
opt = Momentum(filter(lambda x: x.requires_grad, net.get_parameters()), lr, 0.9)
```

4. 模型训练与模型保存

完成数据预处理、网络定义、损失函数和优化器定义之后，就可以进行模型训练了。模型训练包含两层迭代，数据集的多轮迭代和一轮数据集内按分批大小进行的单步迭代。其中，单步迭代指的是按分组从数据集中抽取数据，输入网络中计算得到损失函数值，然后通过反向传播过程，借助优化器更新训练参数。

为了简化训练过程，MindSpore 封装了 Model 高阶接口。用户输入网络、损失函数和优化器完成 Model 的初始化，然后调用 train 接口进行训练，train 接口参数包括迭代次数、数据集和回调函数。

而在模型训练过程中，用户可以添加检查点用于保存模型的参数，以便进行推理及中断后再训练使用。MindSpore 的 Checkpoint 文件是一个二进制文件，存储了所有训练参数的值，采用了 Google 的 Protocol Buffers 机制，与开发语言、平台无关，具有良好的可扩展性。

在具体的实现中，通过回调函数的方式可以进行模型保存，将 ModelCheckpoint 对象传入 model.train，实现模型参数的持久化，生成 Checkpoint 文件。通过 CheckpointConfig 对象可以设置检查点的保存策略。保存的参数分为网络参数和优化器参数，可以根据具体的需求对检查点策略进行配置，程序清单 5-21 展示了配置模型保存策略的一个实例。

程序清单 5-21　使用回调机制保存检查点

```
from mindspore.train.callback import ModelCheckpoint, CheckpointConfig
config_ck = CheckpointConfig(save_checkpoint_steps=32, keep_checkpoint_max=10)
ckpoint_cb = ModelCheckpoint(prefix='resnet50', directory=None, config=config_ck)
model.train(epoch_num, dataset, callbacks=ckpoint_cb)
```

在上述代码中，首先需要初始化一个 CheckpointConfig 类对象，用来设置保存策略。save_checkpoint_steps 表示每隔多少个 step 保存一次；keep_checkpoint_max 表示最多保留 Checkpoint 文件的数量；prefix 表示生成 Checkpoint 文件的前缀名；directory 表示存放文件的目录。创建一个 ModelCheckpoint 对象把它传递给 model.train 方法，就可以在训练过程中使用检查点功能了。

5. 训练监督与模型评估

使用 model.train 后就可以开启深度学习训练了。但在面对复杂网络时，往往需要进行几十甚至几百个轮次训练。在训练之前，很难掌握在训练到第几个轮次时，模型的精度能达到满足要求的程度，所以经常会在训练的同时，在相隔固定轮次的位置对模型进行精度验证，并保存相应的模型，等训练完毕后，通过查看对应模型精度的变化就能迅速地挑选出相对最优的模型。因此在训练过程中，可以使用 Callback、metrics、MindInsight

等功能，实现对训练过程的监督和对神经网络的调试。

Callback 译为回调函数，但它其实不是一个函数而是一个类，可以使用回调函数来观察训练过程中网络内部的状态和相关信息，或在特定时期执行特定动作，例如监控损失、动态调整参数、提前终止训练任务等。

MindSpore 框架给用户提供了 ModelCheckpoint、LossMonitor、SummaryStep 等回调函数。在上文中已经使用 ModelCheckpoint 实现了模型保存，LossMonitor 可以在日志中输出损失，方便用户查看，同时它还会监控训练过程中的损失值变化情况，当损失值为 Nan 或 Inf 时终止训练。SummaryStep 可以把训练过程中的信息存储到文件中，以便后续进行查看或借助 MindInsight 可视化展示，如程序清单 5-22 就展示了传入多个回调函数时的写法。

程序清单 5-22　传入多个回调对象实现训练监督

```
from mindspore.train.callback import ModelCheckpoint, CheckpointConfig
config_ck = CheckpointConfig(save_checkpoint_steps=32, keep_checkpoint_max=10)
ckpoint_cb = ModelCheckpoint(prefix='resnet50', directory=None, config=config_ck)
model.train(epoch_num, dataset, callbacks=ckpoint_cb)
```

训练得到的模型文件可以用来预测新图像的类别，也可以使用 metrics 评估训练结果的好坏。首先通过 load_checkpoint 加载模型文件，然后调用 Model 的 eval 接口预测新图像类别。MindSpore 提供了多种 metrics 评估指标，如 accuracy、loss、precision、recall、F1，在具体实现时可以定义一个 metrics 字典对象，里面包含多种指标，传递给 model.eval 接口用来验证训练精度。模型加载与评估如程序清单 5-23 所示。

程序清单 5-23　模型加载与评估

```
metrics = {
    'accuracy': nn.Accuracy(),
    'loss': nn.Loss(),
    'precision': nn.Precision(),
    'recall': nn.Recall(),
    'f1_score': nn.F1()
}
net = ResNet()
loss = CrossEntropyLoss()
opt = Momentum()
model = Model(net, loss_fn=loss, optimizer=opt, metrics=metrics)
param_dict = load_checkpoint(args_opt.checkpoint_path)
load_param_into_net(model, param_dict)
ds_eval = create_dataset()
output = model.eval(ds_eval)
```

6. 运行并查看结果

至此，已经介绍了基于 CANN 软件栈使用 MindSpore 框架完成深度学习建模与训练的方法。在官方代码仓库中，也提供了完整的相关脚本，只需进入 scripts 目录，执行程序清单 5-24 中的脚本，即可开启训练，在屏幕上正常输出训练轮次和损失值就代表成功开始训练了。

程序清单 5-24　执行训练脚本

```
bash run_standalone_train.sh resnet50 cifar10 [DATASET_PATH]
# 执行上述脚本可以获得下述信息
...
epoch: 1 step: 195, loss is 1.9601055
epoch: 2 step: 195, loss is 1.8555021
epoch: 3 step: 195, loss is 1.6707983
epoch: 4 step: 195, loss is 1.8162166
epoch: 5 step: 195, loss is 1.393667
...
```

5.3.3　高阶技巧

1. 分布式训练

在工业实践中，很多任务都需要使用复杂的模型。复杂的模型加上海量的训练数据，经常导致模型训练耗时严重。因此，在机器资源充沛的情况下，建议采用分布式训练的方式，降低训练耗时。

常见的分布式训练有两种实现方式：模型并行与数据并行。模型并行是将一个网络拆分为多份，拆分后的模型分配到多个设备上使用相同的数据进行训练，这种方式适合于结构设计相对独立的网络模型；数据并行则是每次并行读取多份数据，读取到的数据输入给多个设备上的模型进行训练。MindSpore 同时考虑内存的计算代价和通信代价队训练时间进行建模，并设计了高效的算法来找到训练时间较短的并行策略。这种融合数据并行、模型并行及混合并行的分布式并行模式，可以自动建立代价模型，为用户选择一种新的并行模式。

在具体的实现中，需要先调用集合通信库，在 context.set_context 接口中使能分布式接口 enable_hccl，设置 device_id 参数，并通过调用 init 完成初始化操作。在程序清单 5-25 中，指定运行时使用图模式，并使用华为集合通信库（Huawei Collective Communication Library，HCCL），其中 get_rank 和 get_group_size 分别对应当前设备在集群中的 ID 和集群数量。

程序清单 5-25　分布式训练调用集合通信库

```
from mindspore import context
from mindspore.communication.management import init

if __name__ == "__main__":
    context.set_context(mode = context.GRAPH_MODE, device_target = "Ascend", enable_hccl =
True, device_id = int(os.environ["DEVICE_ID"]))
    init()
    ...
```

分布式训练时，数据是以数据并行的方式导入的。与单机不同的是，在数据集接口需要传入 num_shards 和 shard_id 参数，分别对应网卡数量和逻辑序号，建议通过 HCCL 接口获取这两个参数值，相关样例如程序清单 5-26 所示。

程序清单 5-26　分布式训练数据集改造

```
rank_id = get_rank()
rank_size = get_group_size()
data_set = ds.Cifar10Dataset(data_path, num_shards = rank_size, shard_id = rank_id)
```

context.set_auto_parallel_context 是用于设置并行参数的接口，参数 parallel_mode 可选数据并行 ParallelMode.DATA_PARALLEL 或自动并行 ParallelMode.AUTO_PARALLEL。在反向计算时，框架内部会将数据并行参数分散在多台机器的梯度进行收集，得到全局梯度值后再传入优化器中更新。mirror_mean 参数设置为 True 则对应 all reduce_mean 操作，False 对应 all reduce_sum 操作。

程序清单 5-27 的代码样例指定并行模式为自动并行，其中 dataset_sink_mode＝False 表示采用数据非下沉模式，LossMonitor 能够通过回调函数返回损失值。

程序清单 5-27　自动并行分布式训练

```
from mindspore.nn.optim.momentum import Momentum
from mindspore.train.callback import LossMonitor
from mindspore.train.model import Model, ParallelMode
from resnet import resnet50

def test_train_cifar(num_classes = 10, epoch_size = 10):
    context.set_auto_parallel_context(parallel_mode = ParallelMode.AUTO_PARALLEL,
mirror_mean = True)
    loss_cb = LossMonitor()
    dataset = create_dataset(epoch_size)
    net = resnet50(32, num_classes)
    loss = SoftmaxCrossEntropyExpand(sparse = True)
```

```
opt = Momentum(filter(lambda x: x.requires_grad, net.get_parameters()), 0.01, 0.9)
model = Model(net, loss_fn = loss, optimizer = opt)
model.train(epoch_size, dataset, callbacks = [loss_cb], dataset_sink_mode = False)
```

2. 混合精度训练

混合精度训练方法是通过混合使用 float16 和 float32 数据类型来加速深度神经网络训练的过程。使用混合精度训练主要有三个好处，一是对于中间变量的内存占用更少，节省内存的使用；二是因为内存使用减少，所以数据传出的时间也会缩短；三是 float16 的计算单元可以提供更快的计算性能。但是，混合精度训练受限于 float16 表达的精度范围，单纯将 float32 转换成 float16 会影响训练收敛情况，为了保证部分计算使用 float16 来进行加速的同时能保证训练收敛，MindSpore 还进行了额外的适配。在 MindSpore 中典型的一个混合精度计算流程如图 5-15 所示。

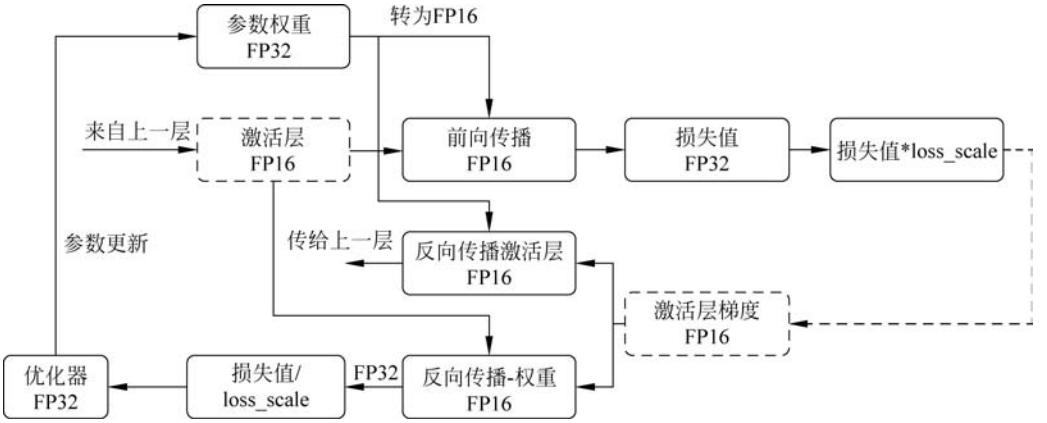

图 5-15　MindSpore 中典型混合精度计算流程

（1）MindSpore 将网络中的参数以 FP32 存储；

（2）在前向传播的过程中，遇到 FP16 算子，则把算子输入并将参数转换成 FP16 进行计算；

（3）在计算损失函数的过程中，设置为使用 FP32 进行计算；

（4）在反向传播过程中，将损失值乘以 loss_scale 值，避免反向梯度过小而产生下溢；

（5）FP16 参数参与梯度计算时，其结果将被转换回 FP32；

（6）将损失值除以 loss_scale 值，还原被放大的梯度；

（7）判断梯度是否溢出，如果溢出则跳过更新，否则对原始参数进行更新。

在代码中，可以使用 FixedLossScaleManager 来定义静态的 loss_scale 系数，实例化后传入模型构造函数，相关的代码如程序清单 5-28 所示。

程序清单 5-28　混合精度训练

```
net = resnet(class_num = config.class_num)
opt = Momentum(group_params, lr, config.momentum, loss_scale = config.loss_scale)
loss = SoftmaxCrossEntropyWithLogits(sparse = True, reduction = 'mean')
loss_scale = FixedLossScaleManager(config.loss_scale, drop_overflow_update = False)
model = Model(net, loss_fn = loss, optimizer = opt, loss_scale_manager = loss_scale,
metrics = {'acc'}, amp_level = "O2", keep_batchnorm_fp32 = False)
```

3. 高阶优化器 THOR

正如在 5.2 节中介绍的，MindSpore 推出的自研优化器 THOR 在训练速度和效果上都有着优秀的表现，在 ResNet-50＋ImageNet 上，该优化器与带 Momentum 的 SGD 相比，端到端时间可提速约 40%。在实践中，使用 THOR 训练网络非常简单，程序清单 5-29 展示了其使用方法。

程序清单 5-29　THOR 算法的使用

```
from mindspore.nn.optim import THOR #引用二阶优化器
#创建网络
net = Net()
#调用优化器
opt = THOR(net, lr, Tensor(damping), config.momentum, config.weight_decay, config.loss_
scale, config.batch_size, split_indices = split_indices)
#增加计算图提升性能
model = ConvertModelUtils().convert_to_thor_model(model = model, network = net, loss_fn = loss,
optimizer = opt, loss_scale_manager = loss_scale, metrics = {'acc'}, amp_level = "O2", keep_
batchnorm_fp32 = False, frequency = config.frequency)
#训练网络
model.train(config.epoch_size, dataset, callbacks = cb, sink_size = dataset.get_dataset_
size(), dataset_sink_mode = True)
```

导入 MindSpore 所需的二阶优化器的包，其位于 mindspore.nn.optim 中；创建所需的网络结构和 THOR 优化器，传入网络信息和 THOR 所需的超参信息后调用 convert_to_ thor_model 函数，该函数通过增加计算图使 THOR 达到更优性能。具体来看，网络运行时本身就是一张计算图，THOR 中会使用其中的二阶矩阵信息，通过额外增加一张计算图，两张计算图分别执行更新二阶矩阵和不更新二阶矩阵的操作达到更优性能。有关使用 THOR 优化器进行训练的代码可以参考官方开源代码库的实现[①]。

① ResNet_thor：https://gitee.com/mindspore/mindspore/tree/r0.7/model_zoo/official/cv/resnet_thor。

5.4 CANN 训练框架之其他框架

CANN 软件栈除了能够适配强大的 MindSpore 框架，也对市面上主流的框架进行了适配，仅需简单的修改，就能将开源代码迁移到 CANN 上运行。本节从开源的 ResNet-50 代码出发，讲解将其运行在 CANN 软件栈上的具体方法。

5.4.1 CANN 与 TensorFlow 的适配原理

CANN 软件栈中昇腾计算编译引擎的图编译器(Graph Compiler)和昇腾计算执行引擎的图执行器(Graph Executor)，合在一起常称为图引擎(Graph Engine，GE)。正如在第 2 章中介绍的，GE 能够对不同的深度学习前端框架提供统一的 IR 接口，从而支持 TensorFlow/PyTorch/MindSpore 的计算图执行。它也能优化计算图的后端执行，更充分地发挥底层昇腾处理器的计算能力。

为了实现这样的目标，华为公司开发了 TensorFlow Adapter For Ascend 组件包(简称 TF Adapter)来架起 TensorFlow 框架和 GE 之间的桥梁。图 5-16 展示了 CANN 软件栈借助 TF Adapter 实现对 TensorFlow 框架进行适配的原理图。

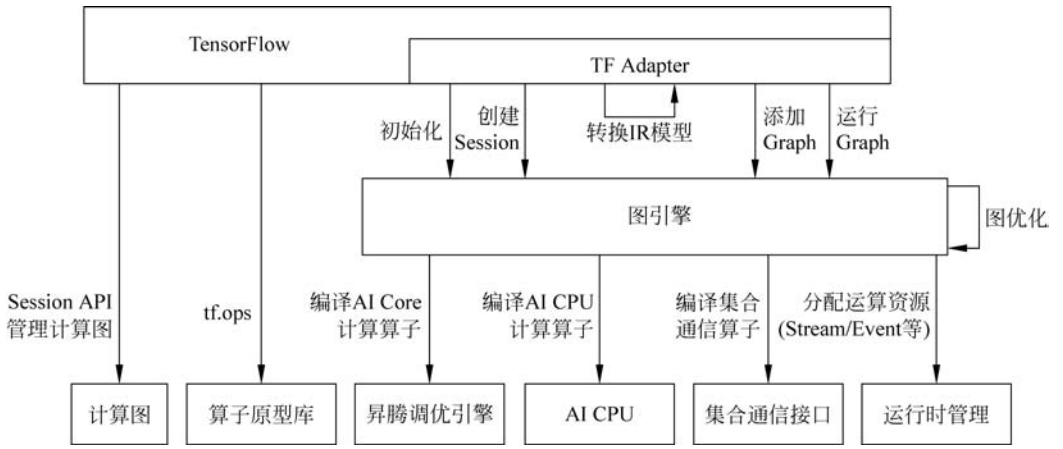

图 5-16 CANN 适配 TensorFlow 适配架构图

在 TenorFlow 中有两个十分重要的概念，op 和 kernel。可以认为 op 相当于函数声明，kernel 相当于函数实现。同一份声明在不同的设备上，最优的实现方式是不一样的，比如对于 MatMul 矩阵相乘这个操作，在 CPU 上可以用 SSE 指令优化加速，在 GPU 上可以用 GPU 实现高性能计算，在昇腾 AI 处理器上自然也有其他的执行方式。昇腾 AI

处理器也常称为 NPU(Neural Processing Unit,神经处理单元)。TF Adapter 注册了相应的 kernel 函数,在继承 tf.op 的同时能够实现自定义通信算子 HCOM 和 TBE 算子的注册,实现算子的适配。

从流程上看,当用户执行训练代码后,TensorFlow 前端会根据用户提供的训练脚本,生成训练模型,读取指定路径下的 Checkpoint 文件完成模型权重初始化或随机初始化;随后,框架前端会通过 TF Adapter 调用 GE 初始化接口,完成设备打开、计算引擎初始化、算子信息库初始化等操作。

TF Adapter 会调用 GE 接口,将前端训练模型转换为 IR 格式的模型,然后启动模型编译和执行;在图优化引擎中,它还会完成形状推导、常量折叠、算子融合等优化操作。在完成图优化后会根据算子信息库将计算图拆分为不同的子图,每个子图都可以执行在同一个设备上,如 GE 会调用图编译器接口完成 AI Core 计算算子编译,调用 AI CPU 接口完成 AI CPU 计算算子编译,调用集合通信接口(HCOM)完成集合通信算子编译。在每个具体模块中,都会进行特定的子图优化。

待计算图的编译和优化都完成后,GE 会调用 Runtime 接口分配运行资源,包含内存、Stream、Event 等,待计算资源分配完成后,就可以交由 RunTime 运行并对资源进行管理了。

上述流程是 CANN 软件栈面对 TensorFlow 代码的处理流程,借助 TF Adapter 的强大能力,只需要安装 TF Adapter 插件,并在现有 TensorFlow 脚本中添加少量配置,即可实现在昇腾 AI 处理器上加速自己的训练任务。TF-Adapter 的源码实现已经开源[①],感兴趣的读者可以通过源码深入研究。

5.4.2　使用 TensorFlow 训练 ResNet-50

5.4.1 节初步介绍了 CANN 适配 TensorFlow 的原理。本节将以具体的实例介绍如何训练出 TensorFlow 版本的 ResNet-50 模型。

1. 训练前准备

本节将以 ImageNet 数据集为例,介绍 TensorFlow 版本 ResNet-50 的训练方法。用户可以在其官方网站[②]获取数据集,原始版本的代码也可以通过 TensorFlow 官方代码库[③]中下载获取,下载后的主要文件目录结构如程序清单 5-30 所示(只列出部分涉及文件,更多文件请查看获取的 ResNet 原始网络脚本)。

其中,imagenet_main.py 文件中包含 imagenet 数据集数据预处理、模型构建定义、

① TF-Adapter 相关资料请参见 https://gitee.com/ascend/tensorflow/tree/master。

② ImageNet 数据集官网链接为 http://www.image-net.org/。

③ Modelzoo 中 TensorFlow-ResNet 的代码链接为 https://github.com/tensorflow/models/tree/r2.1_model_reference/official。

模型运行的相关函数接口。数据预处理部分包含 get_filenames、parse_record、input_fn、get_synth_input_fn，_parse_example_proto 函数，模型部分包含 ImagenetModel 类、imagenet_model_fn、run_cifar、define_cifar_flags 函数。

imagenet_preprocessing.py 文件中包含了 imagenet 图像数据预处理接口。训练过程包括使用提供的边界框对训练图像进行采样、将图像裁剪到采样边界框、随机翻转图像，然后调整到目标输出大小(不保留纵横比)。评估过程中使用图像大小调整(保留纵横比)和中央剪裁。

resnet_model.py 中包含了 ResNet 模型的实现，包括辅助构建 ResNet 模型的函数以及网络结构的定义函数。

resnet_run_loop.py 是模型运行文件，包括输入处理和运行循环两部分。输入处理包括对输入数据进行解码和格式转换，输出图像和标签，还根据是否为训练过程对数据的随机化、批次、预读取等细节做出了设定；运行循环部分包括构建 Estimator，然后进行训练和验证。总体来看，是将模型放置在具体的环境中，实现数据与误差在模型中的流动，进而利用梯度下降法更新模型参数。

程序清单 5-30　TensorFlow 版本 ResNet-50 原始网络目录结构

```
├── r1                          // 原始模型目录
│   ├── resnet                  // resnet 主目录
│       ├── __init__.py
│       ├── imagenet_main.py              // 基于 Imagenet 数据集训练网络模型
│       ├── imagenet_preprocessing.py     // Imagenet 数据集数据预处理模块
│       ├── resnet_model.py               // resnet 模型文件
│       ├── resnet_run_loop.py            // 数据输入处理与运行循环(训练、验证、测试)
│       ├── README.md                     // 项目介绍文件
│   ├── utils
│   │   ├── export.py  // 数据接收函数,定义了导出的模型将会对何种格式的参数予以响应
├── utils
│   ├── flags
│   │   ├── core.py                       // 包含了参数定义的公共接口
│   ├── logs
│   │   ├── hooks_helper.py               //自定义创建模型在测试/训练时的工具,比如
                        // 每秒钟计算步数的功能、每 N 步或捕获 CPU/GPU 分析信息的功能等
│   │   ├── logger.py                     // 日志工具
│   ├── misc
│   │   ├── distribution_utils.py         // 进行分布式运行模型的辅助函数
│   │   ├── model_helpers.py              // 定义了一些能被模型调用的函数,比如控制
                                          // 模型是否停止
```

2. 数据预处理

数据预处理流程与原始模型一致，修改 input_fn 函数内的部分代码以适配 CANN 软件栈并提升计算性能，展示的示例代码包含改动位置。

在 official/r1/resnet/imagenet_main.py 文件中增加以下头文件，如程序清单 5-31 所示，并在数据读取时获取芯片数量及芯片 id，用于支持数据并行，如程序清单 5-32 所示。

程序清单 5-31　imagenet_main 添加头文件

```
from hccl.manage.api import get_rank_size
from hccl.manage.api import get_rank_id
```

程序清单 5-32　修改 input_fn 函数

```
def input_fn(is_training, data_dir, batch_size, num_epochs = 1, dtype = tf.float32,
datasets_num_private_threads = None, parse_record_fn = parse_record, input_context = None,
drop_remainder = False, tf_data_experimental_slack = False):
     # 获取文件路径
     filenames = get_filenames(is_training, data_dir)
     # 按第一个维度切分文件
     dataset = tf.data.Dataset.from_tensor_slices(filenames)
     if input_context:
          # 获取芯片数量及芯片 id,用于支持数据并行
          dataset = dataset.shard(get_rank_size(),get_rank_id())
     if is_training:
          # 将文件顺序打乱
          dataset = dataset.shuffle(buffer_size = _NUM_TRAIN_FILES)

     dataset = dataset.interleave(
          tf.data.TFRecordDataset,
          cycle_length = 10,
          num_parallel_calls = tf.data.experimental.AUTOTUNE)
    return resnet_run_loop.process_record_dataset(
          dataset = dataset,
          is_training = is_training,
          batch_size = batch_size,
          shuffle_buffer = _SHUFFLE_BUFFER,
          parse_record_fn = parse_record_fn,
          num_epochs = num_epochs,
          dtype = dtype,
          datasets_num_private_threads = datasets_num_private_threads,
          drop_remainder = drop_remainder,
          tf_data_experimental_slack = tf_data_experimental_slack,
     )
```

3. 模型构建

模型构建的代码与原始模型代码一致，无须进行过多的适配。部分位置可以进行适配性改造以提升计算性能，例如在引入头文件之后，可以在计算精确度时使用 float32 作为标签类型以提升精度，如程序清单 5-33 所示。这个函数在 resnet_run_loop.py 的 resnet_model_fn 函数中，该类定义了由 Estimator 运行的模型。

程序清单 5-33　计算 accuracy 修改标签类型

```
from npu_bridge.hccl import hccl_ops
    # labels 使用 float32 类型来提升精度
    accuracy = tf.compat.v1.metrics.accuracy(tf.cast(labels, tf.float32), predictions['
classes'])

    # 源代码中计算 accuracy 如下:
    # accuracy = tf.compat.v1.metrics.accuracy(labels, predictions['classes'])

    accuracy_top_5 = tf.compat.v1.metrics.mean(
        tf.nn.in_top_k(predictions=logits, targets=labels, k=5, name='top_5_op'))

    # 用于分布式训练时的 accuracy 计算
    rank_size = int(os.getenv('RANK_SIZE'))
    newaccuracy = (hccl_ops.allreduce(accuracy[0], "sum") / rank_size, accuracy[1])
    newaccuracy_top_5 = (hccl_ops.allreduce(accuracy_top_5[0], "sum") / rank_size,
accuracy_top_5[1])
    metrics = {'accuracy': newaccuracy,'accuracy_top_5': newaccuracy_top_5}
    # 源代码中的 metrics 表示如下:
    # metrics = {'accuracy': accuracy,
    #            'accuracy_top_5': accuracy_top_5}
```

用户也可以使用 max_pool_with_argmax 算子替代 max_pooling2d 算子，以获得更好的计算性能。高性能算子替换如程序清单 5-34 所示。

程序清单 5-34　高性能算子替换

```
# 是否进行第一次池化
if self.first_pool_size:
    # 使用 max_pool_with_argmax 代替 max_pooling2d 能获得更好的表现
    inputs,argmax = tf.compat.v1.nn.max_pool_with_argmax(
    input=inputs, ksize=(1,self.first_pool_size,self.first_pool_size, 1),
    strides=(1,self.first_pool_stride,self.first_pool_stride,1), padding='SAME',
data_format='NCHW' if self.data_format=='channels_first' else 'NHWC')

    # 源代码使用 max_pooling2d 接口进行池化
```

```
    # inputs = tf.compat.v1.layers.max_pooling2d(
    #     inputs=inputs, pool_size=self.first_pool_size,
    #     strides=self.first_pool_stride, padding='SAME',
    #     data_format=self.data_format)

    inputs = tf.identity(inputs, 'initial_max_pool')
```

4. 训练配置

训练运行配置主要保存在 resnet_run_loop.py 文件中的 resnet_main 函数内，为了让其能够顺利迁移到昇腾平台运行，需要做三方面的修改：一是添加头文件，二是替换 Runconfig，三是替换 Estimator 接口。

参见程序清单 5-35，首先要在“/official/r1/resnet/resnet_run_loop.py”添加头文件。

程序清单 5-35　添加头文件

```
from npu_bridge.estimator.npu.npu_config import NPURunConfig
from npu_bridge.estimator.npu.npu_estimator import NPUEstimator
```

接下来，需要修改 official/r1/resnet/resnet_run_loop.py 的 resnet_main 函数，通过 NPURunconfig 替代 Runconfig 来配置运行参数，参见程序清单 5-36。

程序清单 5-36　NPURunconfig 参数配置

```
  # 使用 NPURunconfig 替换 Runconfig,适配昇腾 AI 处理器,每 115200 步保存一次 checkpoint,
每 10000 次保存一次 summary
  # 对数据进行预处理,使用混合精度模式提升训练速度
  run_config = NPURunConfig(
      model_dir=flags_obj.model_dir,
      session_config=session_config,
      save_checkpoints_steps=115200,
      enable_data_pre_proc=True,
      iterations_per_loop=100,
      # enable_auto_mix_precision=True,
      # 设置为混合精度模式
      precision_mode='allow_mix_precision',
      hcom_parallel=True
  )
  # 源代码中运行参数配置如下:
  # run_config = tf.estimator.RunConfig(
  #     train_distribute=distribution_strategy,
  #     session_config=session_config,
  #     save_checkpoints_secs=60 * 60 * 24,
  #     save_checkpoints_steps=None)
```

同样在 resnet_main 函数内，需要创建 NPUEstimator，使用 NPUEstimator 接口代替 tf.estimator.Estimator，如程序清单 5-37 所示。

程序清单 5-37　修改 NPUEstimator 接口

```
# 使用NPUEstimator接口代替 tf.estimator.Estimator
classifier = NPUEstimator(
    model_fn = model_function, model_dir = flags_obj.model_dir, config = run_config,
    params = {
        'resnet_size': int(flags_obj.resnet_size),
        'data_format': flags_obj.data_format,
        'batch_size': flags_obj.batch_size,
        'resnet_version': int(flags_obj.resnet_version),
        'loss_scale': flags_core.get_loss_scale(flags_obj,
                                                default_for_fp16 = 128),
        'dtype': flags_core.get_tf_dtype(flags_obj),
        'fine_tune': flags_obj.fine_tune,
        'num_workers': num_workers,
        'num_gpus': flags_core.get_num_gpus(flags_obj),
    })
# 源代码中创建 Estimator 如下：
# classifier = tf.estimator.Estimator(
#     model_fn = model_function, model_dir = flags_obj.model_dir, config = run_config,
#     warm_start_from = warm_start_settings, params = {
#         'resnet_size': int(flags_obj.resnet_size),
#         'data_format': flags_obj.data_format,
#         'batch_size': flags_obj.batch_size,
#         'resnet_version': int(flags_obj.resnet_version),
#         'loss_scale': flags_core.get_loss_scale(flags_obj,
#                                                 default_for_fp16 = 128),
#         'dtype': flags_core.get_tf_dtype(flags_obj),
#         'fine_tune': flags_obj.fine_tune,
#         'num_workers': num_workers,
#     })
```

5. 分布式训练

在修改完数据处理模块和模型配置的代码后，就需要修改训练函数相关接口，在代码训练模块文件中引入头文件后还需要在训练之前进行集合通信初始化，相关代码如程序清单 5-38 所示。

程序清单 5-38　引入头文件和集合通信初始化

```
from npu_bridge.estimator import npu_ops
from tensorflow.core.protobuf import rewriter_config_pb2
```

```
def main():
    # 初始化 NPU,调用 HCCL 接口
    init_sess, npu_init = resnet_run_loop.init_npu()
    init_sess.run(npu_init)

    with logger.benchmark_context(flags.FLAGS):
        run_imagenet(flags.FLAGS)
```

在完成集合通信初始化后还需要实现手动初始化集合通信的函数,参见程序清单 5-39。

程序清单 5-39　手动初始化集合通信的函数

```
# 添加如下函数
def init_npu():
    npu_init = npu_ops.initialize_system()
    config = tf.ConfigProto()
    config.graph_options.rewrite_options.remapping = rewriter_config_pb2.
RewriterConfig.OFF
    custom_op = config.graph_options.rewrite_options.custom_optimizers.add()
    custom_op.name = "NpuOptimizer"
    custom_op.parameter_map["precision_mode"].s = tf.compat.as_bytes("allow_mix_
precision")
    custom_op.parameter_map["use_off_line"].b = True
    init_sess = tf.Session(config=config)
    return init_sess, npu_init
```

在单次训练或验证结束后需要释放设备资源,在执行 classifier.train 之后添加如程序清单 5-40 所示的代码来释放 NPU 资源,在下一次进程开始之前如果还需要用到 hccl 则重新初始化。

程序清单 5-40　释放 NPU 资源

```
init_sess, npu_init = init_npu()
      npu_shutdown = npu_ops.shutdown_system()
      init_sess.run(npu_shutdown)
      init_sess.run(npu_init)
```

与单次执行训练后释放资源相似,在所有训练/验证结束后,也需要通过 npu_ops.shutdown_system 接口释放设备资源,相关代码见程序清单 5-40。由于昇腾 AI 处理器默认支持混合精度训练,loss_scale 设置过大可能导致梯度爆炸,设置过小可能会导致梯度消失。所以依据经验,修改 imagenet_main.py 的 define_imagenet_flags 函数,设置 loss_scale 为 512,确保模型正常训练。

在完成上述所有修改后，就可以执行 imagenet_main.py 进行训练了，可以通过 ModelZoo 获得完整的代码。

5.4.3 CANN 与 PyTorch 的适配原理

PyTorch 是与 TensorFlow 截然不同的框架，不同于 TensorFlow 在 Python 层构造一个完整的计算图之后去执行，PyTorch 的前向计算是由 Python 代码层驱动的，反向计算是每次迭代每个前向计算时压入栈的反向执行函数的调用串。因此，不同于 TensorFlow 整网优化的策略，CANN 采用单算子优化的方式与 PyTorch 进行适配。

使用单算子优化的对接适配方案可以最大限度上继承 PyTorch 框架动态图的特性，同时继承框架原生的体系结构，保留了比如自动微分、动态分发、Profiling、Storage 共享机制及设备侧的动态内存管理等出色的特点。与此同时，这种适配方式具有很好的扩展性。在打通流程的通路之上，对于新增的网络类型或结构，可以复用框架类算子，反向图建立和实现机制等，只需涉及相关计算类算子的开发和实现。

从用户的角度来看，可以最大限度保持与 GPU 的使用方式和风格一致。在代码迁移时，只需在 Python 侧和 device 相关操作中，指定 device 为昇腾 AI 处理器，即可完成用昇腾 AI 处理器在 PyTorch 对网络的开发、训练及调试，无须进一步关注昇腾 AI 处理器具体的底层细节，迁移成本更低。

CANN 与 PyTorch 框架对接适配的逻辑模型如图 5-17 所示。整体来看，昇腾 AI 处理器可以被当作和 GPU 同一类别的设备，具体功能包括内存管理、设备管理及算子调用实现。

为了更好地理解框架图，需要先对 PyTorch 框架的结构进行介绍。PyTorch 的代码主要由 C10、ATen、torch 三大部分组成的：C10 是 Caffe Tensor Library 的缩写，这里存放的都是最基础的 Tensor 库的代码，可以运行在服务端和移动端；Aten 是 A TENsor library for C++ 11 的缩写，这一部分声明和定义了 Tensor 运算相关的逻辑和代码；Torch 包含了其前身开源项目的代码。针对 PyTorch 这三大组成部分，CANN 都进行了适配性的开发，对上开放调用接口，对下调用 AscendCL 模块使用昇腾计算能力。

与之相配合的，基于 NPU 芯片的架构特性，昇腾计算平台还开发了 Apex 模块实现混合精度计算。Apex 是一个集优化性能、精度收敛于一身的综合优化库，在保证部分计算使用 float16 进行加速的同时能保证训练收敛。

借助 APEX 和 CANN 在框架层的适配开发，可以十分方便地完成代码迁移。5.4.4 节将通过 ResNet-50 的实例介绍代码迁移的具体流程。

5.4.4 使用 PyTorch 训练 ResNet-50

相比于对 TensorFlow 代码进行适配，修改 PyTorch 代码要简单很多。先从 PyTorch 官网中获得训练脚本[①]，获取原始代码后就可以按照如下的流程进行适配改造。

① PyTorch_resnet50 代码参见 https://github.com/pytorch/examples/tree/master/imagenet。

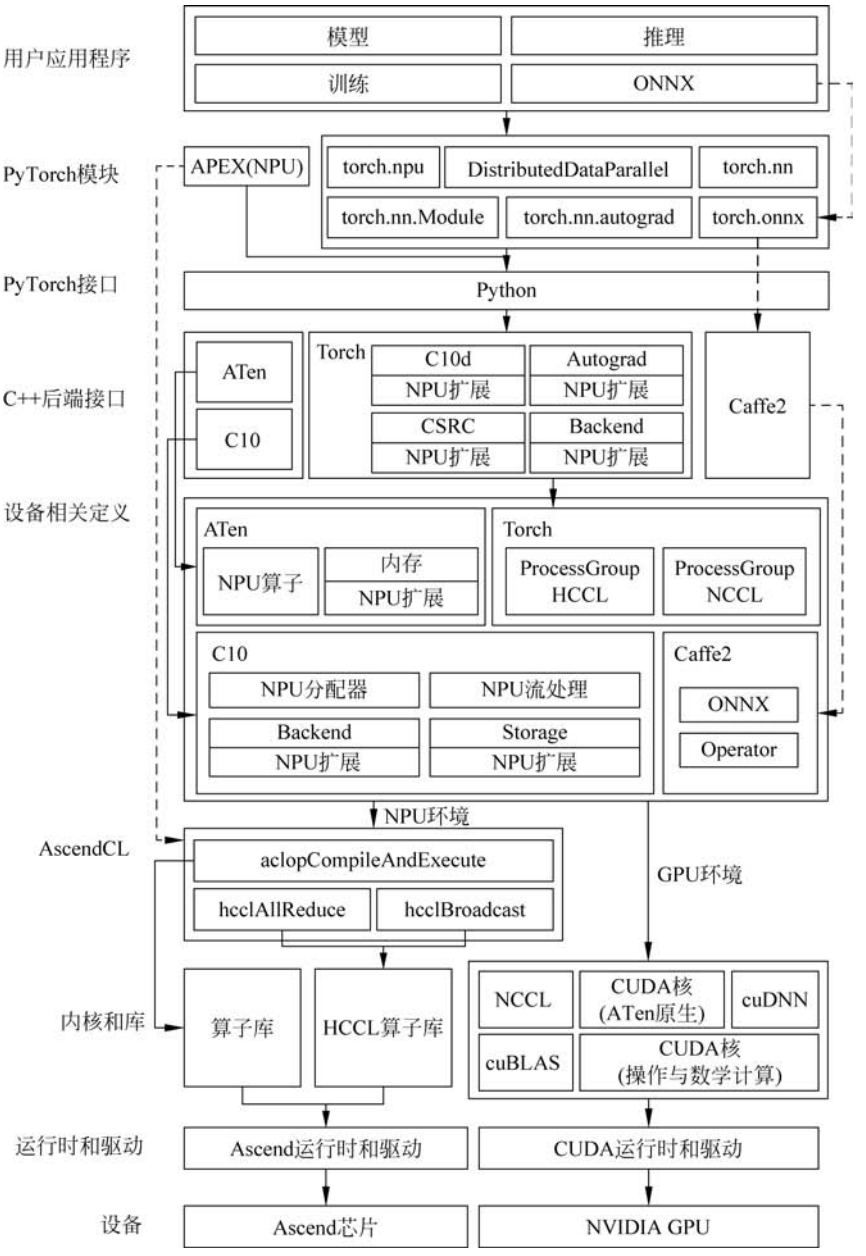

图 5-17　CANN 适配 PyTorch 逻辑模型

首先，要添加头文件以支持基于 PyTorch 框架的模型在昇腾 910 AI 处理器上训练，并在头文件后添加参数以指定使用昇腾 910 AI 处理器进行训练，相关的代码如程序清单 5-41 所示。

程序清单 5-41　PyTorch 添加头文件并指定设备

```
import torch.npu
CALCULATE_DEVICE = "npu:1"
```

接下来就要修改参数及判断选项，使其只在昇腾 910 AI 处理器上进行训练，相关的代码在 main.py 文件中的 main_worker 函数中，相关的修改代码如程序清单 5-42 所示。

程序清单 5-42　修改参数及判断选项

```
# args.gpu = gpu
args.gpu = None
# 源代码中需要判断是否在 GPU 上进行训练，源代码如下：
    # if not torch.cuda.is_available():
        # print('using CPU, this will be slow')
    # elif args.distributed:
# # # # # # # # # # # # # # npu modify begin # # # # # # # # # # # # #
    # 迁移后为直接判断是否进行分布式训练，去掉判断是否在 GPU 上进行训练
    if args.distributed:
```

在完成基本配置的修改后，要将模型及损失函数迁移到昇腾 910 AI 处理器上进行计算，相关的代码也在 main_worker 函数中，参见程序清单 5-43。

程序清单 5-43　迁移模型及损失函数

```
    elif args.gpu is not None:
        torch.cuda.set_device(args.gpu)
        model = model.cuda(args.gpu)
    else:
        # DataParallel will divide and allocate batch_size to all available GPUs
        if args.arch.startswith('alexnet') or args.arch.startswith('vgg'):
            model.features = torch.nn.DataParallel(model.features)
            model.cuda()
        else:
            # 源代码使用 torch.nn.DataParallel()类来用多个 GPU 加速训练
            # model = torch.nn.DataParallel(model).cuda()
            # 将模型迁移到 NPU 上进行训练
           model = model.to(CALCULATE_DEVICE)
    # 源代码中损失函数在 GPU 上进行计算
    # # define loss function (criterion) and optimizer
    # criterion = nn.CrossEntropyLoss().cuda(args.gpu)
    # 将损失函数迁移到 NPU 上进行计算
    criterion = nn.CrossEntropyLoss().to(CALCULATE_DEVICE)
```

为了适配 CANN 软件栈上部分算子的特性，将数据集目标结果 target 修改成 int32

类型解决算子报错问题；将数据集迁移到昇腾 910 AI 处理器上进行计算。在 train 和 validate 函数中均有数据集读取的代码，相关代码如程序清单 5-44 所示。

程序清单 5-44　修改数据集数据类型

```
for i, (images, target) in enumerate(train_loader):
    # measure data loading time
    data_time.update(time.time() - end)

    if args.gpu is not None:
        images = images.cuda(args.gpu, non_blocking=True)
    # 源代码中训练数据集在 GPU 上进行加载计算，源代码如下：
    # if torch.cuda.is_available():
        # target = target.cuda(args.gpu, non_blocking=True)
    # 将数据集迁移到 NPU 上进行计算并修改 target 数据类型
    if 'npu' in CALCULATE_DEVICE:
        target = target.to(torch.int32)
    images, target = images.to(CALCULATE_DEVICE, non_blocking=True), target.to
(CALCULATE_DEVICE, non_blocking=True)
```

完成上述修改后，如程序清单 5-45 所示，简单设置当前正在使用的 device 后，就可以直接执行脚本开启训练了。

程序清单 5-45　设置当前使用的 device

```
if __name__ == '__main__':
    if 'npu' in CALCULATE_DEVICE:
       torch.npu.set_device(CALCULATE_DEVICE)
    main()
```

通过上述的修改，开源的脚本就可以在昇腾平台的单设备下进行训练了，相关完整的代码可以通过 ModelZoo 获得。其他的开源代码也可以通过接口的替换实现网络迁移，着重考虑环境配置、高性能接口替换和数据类型改造即可。CANN 软件栈也支持分布式训练的迁移，受限于篇幅，不进行详述。感兴趣的用户可以参考 CANN 官方文档中的流程进行改造。

5.5　网络模型迁移和在线推理

5.4 节展示了如何将开源的 ResNet 代码迁移到昇腾处理器上执行。除了使用官方代码库中提供给用户的代码外，用户也可以将已有的 TensorFlow 原始代码迁移到 CANN 软件栈上。本节将具体介绍完整的迁移过程和其中可能使用到的工具。

5.5.1 模型迁移和在线推理流程

模型迁移和在线推理的主要工作就是将 TensorFlow 原始模型迁移到昇腾 AI 处理器上并执行前向传播，主要流程如图 5-18 所示。

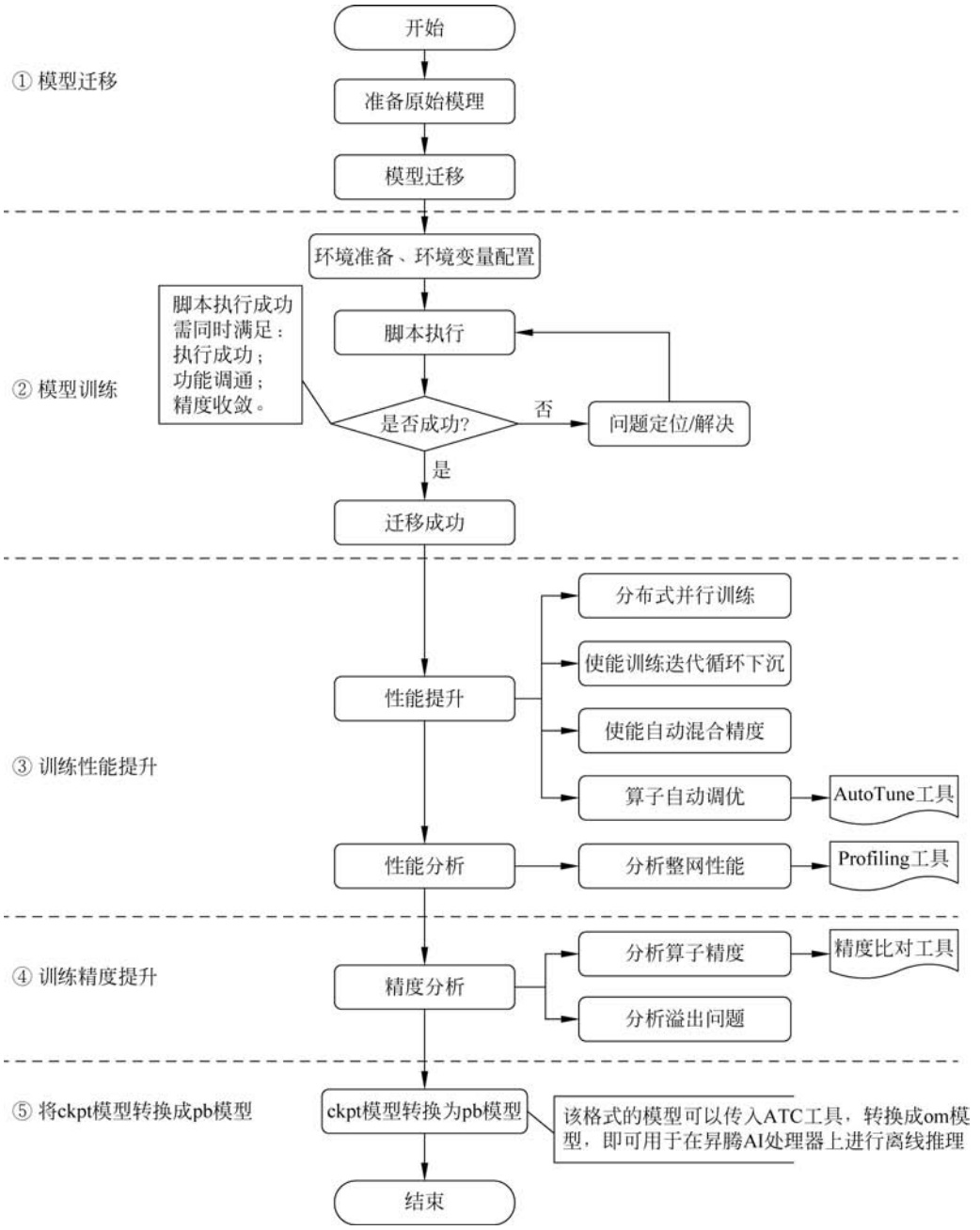

图 5-18 TensorFlow 模型迁移和在线推理主要流程

在进行代码迁移改造前，要事先准备好基于 TensorFlow 1.15 开发的训练模型及配套的数据集，用户需要在自己的设备上将其跑通，且达到预期精度和性能要求，同时记录相关精度和性能指标，用于后续在昇腾 AI 处理器上进行精度和性能对比。

在完成准备工作之后，就可以改造模型代码了。目前 CANN 支持 Estimator 和 sess.run 两种运行方式的代码迁移。接下来将对这两种方式进行具体介绍。

1. Estimator 迁移

Estimator API 属于 TensorFlow 的高阶 API，在 2018 年发布的 TensorFlow 1.10 版本中引入，它可极大简化机器学习的编程过程。Estimator 有很多优势，例如：对分布式的良好支持、简化了模型的创建工作、有利于模型用户之间的代码分享等。

使用 Estimator 进行训练脚本迁移的流程如图 5-19 所示。

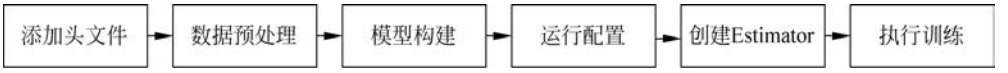

图 5-19　Estimator 脚本迁移过程

1）添加头文件

用户需要将所有涉及修改的 Python 文件都新增程序清单 5-46 的头文件，用于导入 NPU 相关库。

程序清单 5-46　添加头文件

```
from npu_bridge.npu_init import *
```

2）数据预处理

一般情况下，数据预处理的代码无须改造。当前仅支持固定 shape 下的训练，也就是在进行图编译时 shape 的值必须是已知的情况需要进行适配修改。当原始网络脚本中使用 dataset.batch(batch_size)返回动态形状时，由于数据流中剩余的样本数可能小于 batch 大小，因此在昇腾 AI 处理器上进行训练时，应将 drop_remainder 设置为 True。

这可能会丢弃文件中的最后几个样本，以确保每个批量都具有静态形状（batch_size）。但需要注意的是，推理时，当最后一次迭代的推理数据量小于 batch size 时，需要补齐空白数据到 batch size，因为有些脚本最后会添加断言，验证结果的数量要和验证数据的数量一致，此种情况会导致训练失败，相关的数据集改造代码如程序清单 5-47 所示。

程序清单 5-47　数据集模块改造

```
dataset = dataset.batch(batch_size, drop_remainder = True)
assert num_written_lines == num_actual_predict_examples
```

3）模型构建

一般情况下，此部分代码无须改造。以下情况需要进行适配修改：一是如果原始网络中使用到了 tf.device，需要删除相关代码；二是对于原始网络中的 dropout，建议替换为昇腾对应的 API 实现，以获得更优性能。相关的示例如程序清单 5-48 所示。

程序清单 5-48　dropout 的 CANN 实现

```
layers = tf.nn.dropout()                          #TensorFlow 原始代码
from npu_bridge.estimator import npu_ops
layers = npu_ops.dropout()                        #迁移后的代码
```

对于原始网络中的 gelu，建议替换为昇腾对应的 API 实现，以获得更优性能，参见程序清单 5-49。

程序清单 5-49　gelu 的 CANN 实现

```
#TensorFlow 原始代码
def gelu(x):
  cdf = 0.5 * (1.0 + tf.tanh(
     (np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3)))))
  return x * cdf
layers = gelu()
#迁移后的代码
layers = npu_unary_ops.gelu(x)
```

4）运行配置

TensorFlow 通过 Runconfig 配置运行参数，用户需要将 Runconfig 迁移为 NPURunconfig。NPURunConfig 类继承 RunConfig 类，因此在迁移的过程中按照程序清单 5-50 直接修改接口即可，大多数参数可不变。

程序清单 5-50　迁移 NPURunconfig 类

```
#TensorFlow 原始代码
config = tf.estimator.RunConfig(
  model_dir = FLAGS.model_dir,
  save_checkpoints_steps = FLAGS.save_checkpoints_steps,
  session_config = tf.ConfigProto(allow_soft_placement = True, log_device_placement =
False))
#迁移后的代码
npu_config = NPURunConfig(
  model_dir = FLAGS.model_dir,
  save_checkpoints_steps = FLAGS.save_checkpoints_steps,
  session_config = tf.ConfigProto(allow_soft_placement = True, log_device_placement =
False))
```

部分参数(包括 train_distribute/device_fn/protocol/eval_distribute/ experimental_distribute)在 NPURunConfig 中不支持,如果原始脚本使用到了,用户需要进行删除。

NPURunConfig 中新增了部分参数,从而提升训练性能,例如 iterations_per_loop、precision_mode 等,这些参数会在后面的性能提升环节详细介绍。

5) 创建 Estimator

需要将 TensorFlow 的 Estimator 迁移为 NPUEstimator,NPUEstimator 类继承了 Estimator 类,因此在迁移时按照程序清单 5-51 展示的代码直接更改接口即可,参数可保持不变。

程序清单 5-51　Estimator 代码修改示例

```
#TensorFlow 原始代码
mnist_classifier = tf.estimator.Estimator(
  model_fn = cnn_model_fn,
  config = config,
  model_dir = "/tmp/mnist_convnet_model")

#迁移后的代码
mnist_classifier = NPUEstimator(
  model_fn = cnn_model_fn,
  config = npu_config,
  model_dir = "/tmp/mnist_convnet_model"
  )
```

6) 执行训练

在 Estimator 上调用训练方法 Estimator.train,利用指定输入对模型进行固定步数的训练,无须进行特别的改造。

2. sess.run 迁移

sess.run API 属于 TensorFlow 的低阶 API,相对于 Estimator 来讲,灵活性较高,但模型的实现较为复杂。使用 sess.run API 进行训练脚本迁移开发的流程如图 5-20 所示。

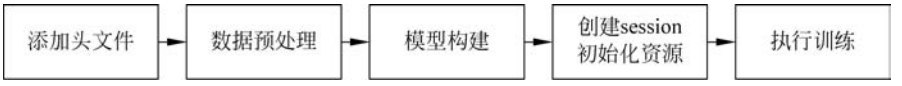

图 5-20　sess.run 代码迁移流程

与 Estimator 迁移的过程相似,都需要添加头文件、适配修改数据预处理代码及使用 NPU 实现的接口代替原生的模型接口。在创建 session 时有一些相关配置需要额外注意,其中的配置项 rewrite_options.disable_model_pruning 默认关闭,不要开启;配置项 rewrite_options.remapping 默认开启,必须显式关闭;tf.Session 原生功能在昇腾平台上全部支持,CANN 还额外支持动混合精度等功能。相关设置涉及许多相关参数,可以查

看和学习昇腾社区官方文档。sess.run 模式的原始代码见程序清单 5-52。迁移后的 sess.run 代码见程序清单 5-53。

程序清单 5-52　sess.run 模式的原始代码

```
#构造迭代器
iterator = Iterator.from_structure(train_dataset.output_types,train_dataset.output_
shapes)
#取 batch 数据
next_batch = iterator.get_next()
#迭代器初始化
training_init_op = iterator.make_initializer(train_dataset)
#变量初始化
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
#Get the number of training/validation steps per epoch
train_batches_per_epoch = int(np.floor(train_size/batch_size))
```

程序清单 5-53　迁移后的 sess.run 代码

```
#构造迭代器
iterator = Iterator.from_structure(train_dataset.output_types,train_dataset.output_
shapes)
#取 batch 数据
next_batch = iterator.get_next()
#迭代器初始化
training_init_op = iterator.make_initializer(train_dataset)
#变量初始化
init = tf.global_variables_initializer()
#创建 session
config = tf.ConfigProto()
custom_op = config.graph_options.rewrite_options.custom_optimizers.add()
custom_op.name = "NpuOptimizer"
config.graph_options.rewrite_options.remapping = RewriterConfig.OFF #必须显式关
闭 remap
sess = tf.Session(config = config)
sess.run(init)
#Get the number of training/validation steps per epoch
train_batches_per_epoch = int(np.floor(train_size/batch_size))
```

在执行训练的环节无须进行特殊的改造，但值得注意的是，如果用户训练脚本中没有使用 with 创建 session，比如将 session 对象作为自己定义的一个类成员，那么需要在迁移后的脚本中显式调用 sess.close，如程序清单 5-54 所示。

这是因为，Geop 的析构函数在 tf. session 的 close 方法中会被调用到，如果是 with 创建的 session，with 会调用 session 的__exit()__方法，里面会自动调用 close；而如果是其他情况，比如是把 session 对象作为自己定义的一个类成员，那么退出之前需要显式调用 sess. close，这样才可以保证退出的正常。

程序清单 5-54　显式调用 sess. close

```
sess = tf.Session(config = config)
sess.run(...)
sess.close()
```

在迁移完成后，就可以进行模型训练了。在模型训练的过程中，难免需要在验证集上进行模型效果的测试，在模型训练完成后，往往也需要利用在线推理的方式验证模型的有效性。推理脚本的迁移与训练代码的迁移有异曲同工之妙，只需参考上述的步骤进行简单的修改即可调用在线推理的流程，相关的代码在此不再赘述，用户可以自行体验和尝试。

5.5.2　性能分析工具——Profiling

当参考 5.5.1 节的流程进行代码迁移后，就可以将其运行在 CANN 软件栈上了。为了验证 CANN 软件栈和昇腾处理器的性能，昇腾提供了端到端 Profiling 系统，准确定位系统的软、硬件性能瓶颈，提高性能分析的效率，通过针对性的性能优化方法，以最小的代价和成本实现业务场景的极致性能。

Profiling 从功能看主要实现了两类性能数据的分析。

第一类是和训练任务相关的迭代轨迹数据，即训练任务及 CANN 软件栈采集的数据（Training Trace），通过数据增强、前后向计算、梯度聚合更新等相关数据可以实现对训练任务的迭代性能分析。具体来看，主要包括单个 Device 上 AI CPU 图计算轨迹相关的性能数据，以及 Runtime、集合通信等相关的性能数据。通过该类性能数据分析，可以得到迭代时长（$t_{(N+1)6}-t_{N6}$）、数据增强拖尾[①]（$t_{(N+1)1}-t_{N6}$）、FBBP 计算时间（$t_{N2}-t_{N1}$）及梯度聚合更新拖尾[②]（$t_{N6}-t_{N2}$）等关键性能指标项。其中各时间点取值如图 5-21 所示。

第二类则是与训练无关的任务轨迹数据，具体包括 AI Core、AI CPU、DVPP 等硬件设备的性能信息，如 CPU 占有率、内存带宽、PCIe 读写带宽等。

在训练过程中开启 Profiling 工具也是十分方便的，在 Estimator 模式下仅需进行如程序清单 5-55 的修改即可开启 Profiling 数据采集。在 sess. run 模式下，也仅需修改 session 配置项 profiling_mode、profiling_options 开启 Profiling 数据采集，如程序清单 5-56 所示。

① 数据增强拖尾：上一轮迭代结束后、本轮迭代 FP 开始前，本轮数据增强仍然在执行，这段耗时为拖尾时长。

② 梯度聚合更新拖尾：本轮迭代 BP 执行完成、迭代结束之前，仍然在执行梯度聚合/更新，这段耗时为拖尾时长。

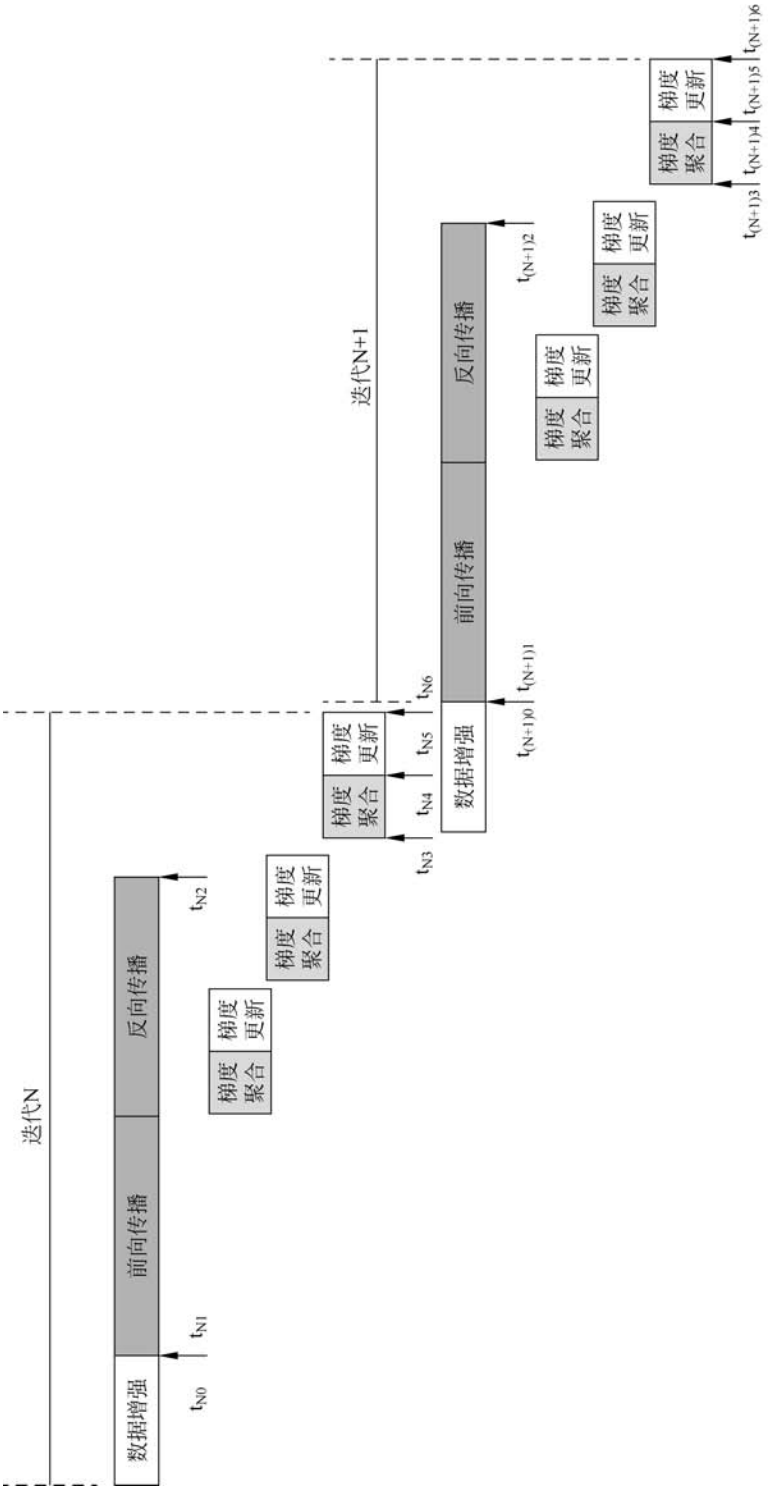

图 5-21　训练任务迭代内执行流程

程序清单 5-55　Estimator 模式下开启 Profiling 数据采集

```
from npu_bridge.npu_init import *

profiling_options = '{"output":"/tmp/profiling","training_trace":"on","fp_point":
"resnet_model/conv2d/Conv2Dresnet_model/batch_normalization/FusedBatchNormV3_Reduce",
"bp_point":"gradients/AddN_70"}'
profiling_config = ProfilingConfig(enable_profiling = True, profiling_options =
profiling_options)
session_config = tf.ConfigProto()

config = NPURunConfig(profiling_config = profiling_config, session_config = session_
config)
```

程序清单 5-56　sess.run 模式下开启 Profiling 数据采集

```
custom_op = config.graph_options.rewrite_options.custom_optimizers.add()
custom_op.name = "NpuOptimizer"
custom_op.parameter_map["use_off_line"].b = True
custom_op.parameter_map["profiling_mode"].b = True
custom_op.parameter_map["profiling_options"].s = tf.compat.as_bytes('{"output":"/tmp/
profiling","training_trace":"on","fp_point":"resnet_model/conv2d/Conv2Dresnet_model/
batch_normalization/FusedBatchNormV3_Reduce","bp_point":"gradients/AddN_70"}')
config.graph_options.rewrite_options.remapping = RewriterConfig.OFF # 关闭 remap 开关

with tf.Session(config = config) as sess:
  sess.run()
```

除了以上两种方式，用户还可以修改启动脚本中的环境变量，开启 Profiling 采集功能，相关设置如程序清单 5-57 所示。

程序清单 5-57　通过环境变量开启 Profiling 采集功能

```
export PROFILING_MODE = true
export PROFILING_OPTIONS = '{"output":"/tmp/profiling","training_trace":"on","task_
trace":"on","aicpu":"on","fp_point":"resnet_model/conv2d/Conv2Dresnet_model/batch_
normalization/FusedBatchNormV3_Reduce","bp_point":"gradients/AddN_70","aic_metrics":
"PipeUtilization"}'
```

训练结束后，切换到 output 目录下，可查看到 Profiling 数据，并且可以通过 Profiling 工具解析数据。在昇腾平台上，toolkit 工具包中的 msprof.pyc 脚本可以用于解析 System Profiling、Job Profiling 任务采集的性能原始数据。

5.5.3 算子自动调优工具——AutoTune

AI 计算芯片通常由多个计算单元、片上存储、数据传输等模块组成。运行在其上的算子，无法简单地用计算量除以算力获得耗时，更要看各个组件间的协同情况。相同的计算任务部署在相同的计算芯片上，用不同的计算流水排布效率也会差别巨大。只有精心设计好的调度逻辑才能充分发挥硬件的算力。

除芯片内的计算过程需要精心排布外，组件之间的 pipeline 也需要精心的设计才能达到最优性能。算子的理论最大性能是其瓶颈负载（计算、数据传输等）除以对应处理单元的效率。然而因为片上存储有限，一次计算任务通常会被切分成多片处理，这样就会产生计算或传输冗余，所以实际负载往往大于理论负载。相同的计算任务，用不同的计算流水排布方式其冗余度也会不同。通常会选择冗余较小的方案或将冗余转移到非瓶颈组件上。所以，为了达到最佳性能，也需要合理地设计各组件的时序。

如此复杂的调优内容，如果使用人工优化往往耗时多且结果不尽如人意。在这样的背景下，AutoTune 自动优化工具应运而生，使用它可以更充分地利用机器资源来挖掘硬件性能。在具体介绍 AutoTune 使用方法之前，不妨先对其工作原理进行简单的介绍。

网络模型生成时，AutoTune 工具调优在算子编译阶段执行，默认执行流程如图 5-22 所示。

原始框架模型首先传入图编译器进行图准备和图优化操作，具体包括算子选择、算子融合、常量折叠等操作。随后模型会进入算子编译阶段，AutoTune 工具会首先根据网络中的层信息匹配知识库。如果匹配到了知识库，则判断是否开启了 AutoTune 调优或者 REPEAT_TUNE 模式，如果未开启则直接使用知识库中的调优策略编译算子。若已开启相关配置，则会通过算法重新进行调优，进而生成自定义的知识库，若调优后的结果优于当前已存在的知识库（包括内置知识库与自定义知识库），则会将调优后的结果存入用户自定义知识库，并使用自定义知识库中的调优策略编译算子。如果没有匹配到知识库，则判断是否开启了 AutoTune 调优。如果开启了 AutoTune 调优，且调优后的结果优于默认调优策略的性能，则会将调优后的结果存入用户自定义知识库，并使用自定义知识库中的调优策略编译算子。否则，会将默认调优策略存入用户自定义知识库，并使用自定义知识库中的调优策略编译算子。

算子编译完成后，就获得了更高性能的网络模型文件，深度学习框架就能借此进行训练了。

在实际的开发过程中，如果想要使用 AutoTune 工具，则在配置好生产环境和环境变量后，仅需在训练脚本中，通过 session 配置项中的 auto_tune_mode 参数开启 AutoTune，相关的代码如程序清单 5-58 所示。

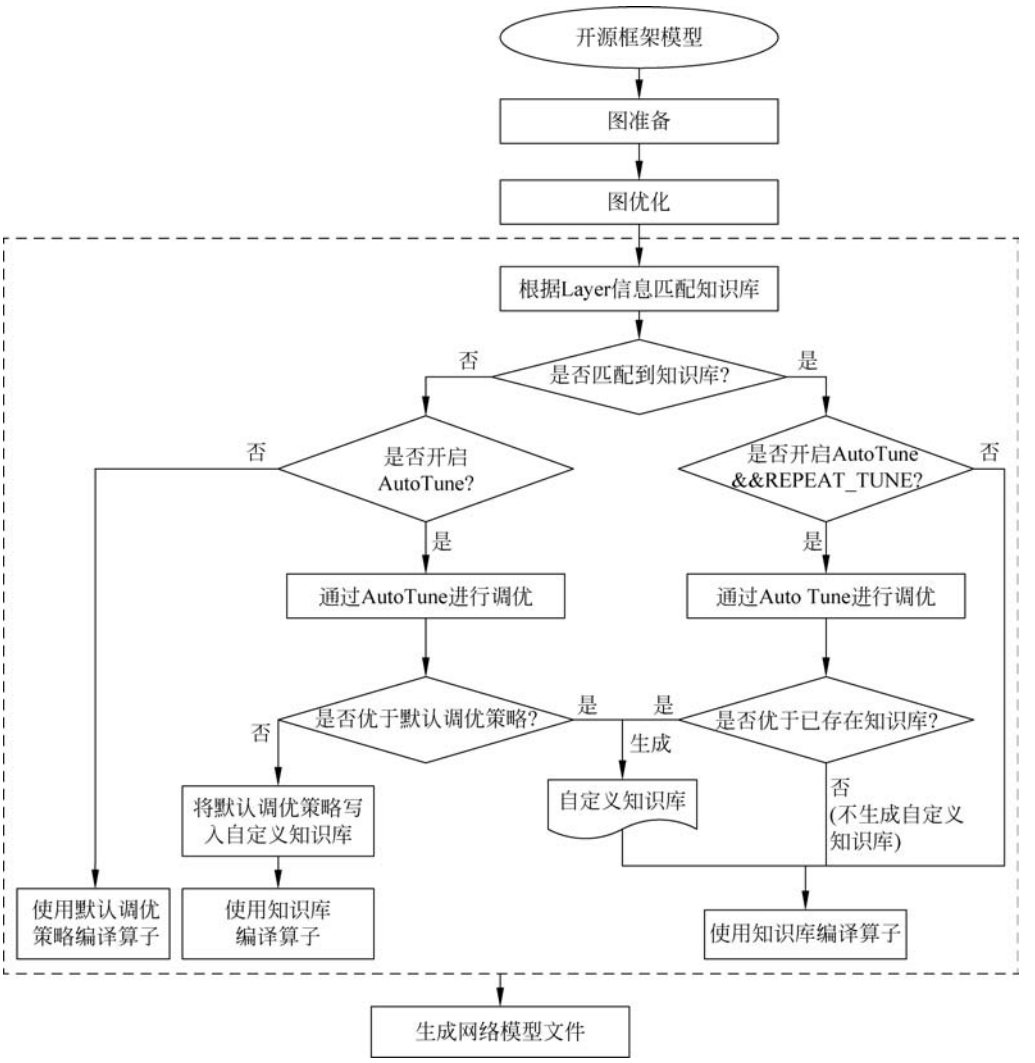

图 5-22　AutoTune 调优流程

程序清单 5-58　通过 session 配置项开启 AutoTune 工具

```
session_config = tf.config(…)
custom_op = session_config.graph_options.rewrite_options.custom_optimizers.add()
…
custom_op.parameter_map["auto_tune_mode"].s = tf.compat.as_bytes("RL,GA")
```

值得说明的是，auto_tune_mode 中的参数 RL、GA 代表两种调优模式。

RL(Reinforcement Learning)即强化学习，其主要调优原理为：将 Schedule 过程抽象为基于蒙特卡洛树搜索(Monte Carlo tree search，一种用于某些决策过程中的启发式

搜索算法)的决策链,然后使用 NN(Neural Networks)指导决策,其中 NN 基于 RL 进行训练生成。

GA(Genetic Algorithm)即遗传算法,其主要调优原理为:通过多级组合优化生成调优空间,加入人工经验进行剪枝、排序,提高调优的效率;进行多轮参数寻优,从而获得最优的 tiling 策略。

这两种算法均只能针对部分算子进行调优。在完成调优后,若满足自定义知识库生成条件(参见图 5.22 AutoTune 调优流程),则会生成自定义知识库。自定义知识库会存储到 TUNE_BANK_PATH 环境变量指定的路径中,生成的文件命名为 tune_result_pidxxx_{timestamp}.json,其中记录了调优过程和调优结果,{timestamp}为时间戳,格式为:年月日_时分秒毫秒,pidxxx 中的"xxx"为进程 ID。输出的文件内容如程序清单 5-59 所示。

程序清单 5-59　AutoTune 输出文件内容

```
"[['Operator Name']]":{
    "result_data:{
        "after_tune": 56
        "before_tune": 66,
    }
    "status_data:{
        "bank_append": true,
        "bank_hit": false,
        "bank_reserved": false,
        "bank_update": false
    }
    "ticks_best":[
        "[82, 2020-08-08 18:03:38]",
        "[104, 2020-08-08 18:03:50],
        ...
        ]
},
```

其中,各字段的含义解释如下所示。

Operator Name 为原图中算子的名称,若原图在图优化过程中进行了融合,融合后的节点对应原图中的多个节点,则会显示多个 Operator Name,例如:[['scale5a_branch1','bn5a_branch1','res5a_branch1'],['res5a'],['res5a_relu']]。

result_data 即调优结果,记录网络模型中被调优算子调优前后的执行时间。after_tune 表示 AutoTune 调优后的算子执行时间;before_tune 表示未开启 AutoTune 调优前,算子执行时间,时间单位均为 us。

status_data 表示详细调优状态信息,记录了网络模型中所有算子的调优状态信息;其中 bank_append 取值为 true,表示调优前该算子的调优策略不在知识库中,调优结束后

该调优策略追加到了知识库，其他情况取值为 false；调优前该算子的调优策略若在知识库中，bank_hit 取值为 true，若不在知识库中，取值为 false；若调优前该算子的调优策略在知识库中，并且调优结束后该调优策略没有更新，则 bank_reserved 取值为 true，其他情况取值为 false；若调优前该算子的调优策略在知识库中，并且调优结束后该调优策略进行了更新，则 bank_update 取值为 true，其他情况取值为 false。

ticks_best 记录了每轮调优的结果，包含 tiling 耗时和本轮算子的调优结束时间。

5.5.4　精度分析工具——Data Dump

无论是在训练的过程中，还是在推理过程中，都有可能出现输出结果与预期存在差异的问题，模型的精度可能有所下降。常见的精度劣化包括以下几方面的原因：算子融合、常量折叠、int8 精度不足、算子精度不达标、网络中存在“放大器结构”等。

此时需要使用精度比对工具来分析 GPU/CPU 执行结果与 NPU 执行结果之间的差距。昇腾平台提供了相应的分析工具帮助开发人员快速解决算子精度问题，目前精度分析工具可以从余弦相似度、最大绝对误差、累积相对误差、欧氏相对距离、KLD 散度、标准差几个方面进行比对。在实际使用中，可以按以下三个步骤进行分析。

1. 在 GPU 上生成参数文件

为了能让训练脚本在执行过程中输出特定格式的参数文件，需要用户对训练脚本进行一些小的改动。不论采用哪种方式(Estimator 或 session.run)，要进行精度比对的话，首先要把脚本中所有的随机全都关掉，包括但不限于对数据集的随机打乱，参数的随机初始化，以及某些算子的隐形随机初始化。

去除随机之后，就可以利用 TensorFlow 官方提供的 debug 工具 tfdbg 生成参数文件。具体来看，需要修改 tf 训练脚本，提供 debug 选项配置。在 Estimator 模式下，需要在引入包的地方添加一行，然后在生成 EstimatorSpec 对象实例的时候，也就是构造网络结构的时候，添加 tfdbg 的 hook。具体来看，相关的代码如程序清单 5-60 所示。

程序清单 5-60　Estimator 模式下添加 tfdbg

```
from tensorflow.python import debug as tf_debug
…… 
estim_specs = tf.estimator.EstimatorSpec(
mode = mode,
predictions = pred_classes,
loss = loss_op,
train_op = train_op,
training_hooks = [tf_debug.LocalCLIDebugHook()])
```

在 session.run 模式下，同样需要引入程序清单 5-61 中的库，而在 session 初始化之

后、执行计算图之前设置 tfdbg 修饰类。

程序清单 5-61　sess.run 模式下添加 tfdbg

```
from tensorflow.python import debug as tf_debug
......
sess = tf.Session()
sess.run(tf.global_variables_initializer())
sess = tf_debug.LocalCLIDebugWrapperSession(sess, ui_type = "readline")
```

修改完成后，正常启动训练，就能在控制台内进入交互界面，如图 5-23 所示，输入 run 命令，训练就会继续执行，等待执行 run 命令完成后，在命令行交互界面，可以通过 lt 查询已存储的张量，通过 pt 可以查看已存储的张量内容，保存数据为 numpy 格式文件。

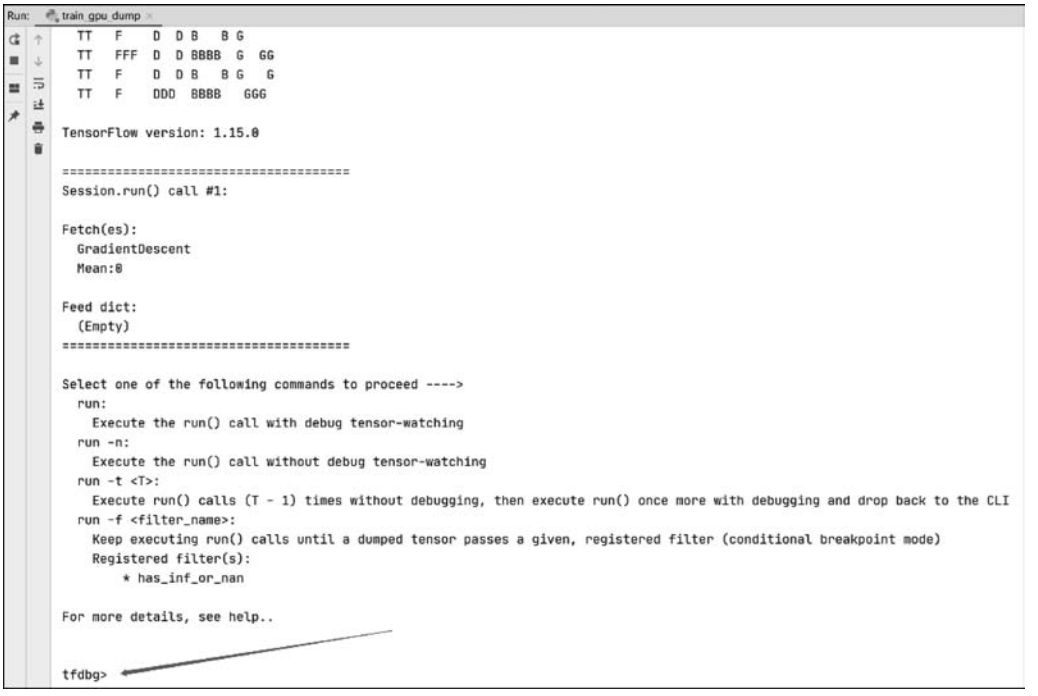

图 5-23　tfdbg 命令行交互界面

因为运行一次 tfdbg 命令只能生成一个张量，为了自动生成收集所有数据，可以按以下几个步骤操作。

(1) 执行 lt > tensor_name 将所有张量的名称暂存到文件里。

(2) 重新开启一个命令窗口，在 linux 命令行下执行下述命令，用以生成在 tfdbg 命令行执行的命令 timestamp＝$[$(date ＋%s%N)/1000]；cat tensor_name | awk '{print "pt",$4,$4}' | awk '{gsub("/","_",$3);gsub(":",".",$3);print($1,$2,"-n 0-w "$3".""'$timestamp'"".npy")}'> tensor_name_cmd.txt。

（3）在 tfdbg 命令行中，将上一步生成的 tensor_name_cmd.txt 文件内容粘贴执行，即可存储所有 npy 文件。npy 文件默认是以 numpy.save 形式存储的，上述命令会将“/”用下画线“_”替换。

执行上述操作之后，在训练脚本所在的目录中会出现很多以“.npy”为后缀的文件。至此，就完成了在 GPU 上生成参数文件。

2. 在 NPU 上生成参数文件和计算图

在昇腾平台上生成参数文件和计算图，也需要少量修改训练脚本。在具体修改之前，一定要确保代码在网络结构、算子、优化器的选择上，以及参数的初始化策略等方面跟 GPU 上训练的代码完全一致，否则比较起来是没有意义的。在对齐代码和配置后，在相应代码中，增加如下的信息。

Estimator 模式：通过 NPURunConfig 中的 dump_config 采集转储数据，在创建 NPURunConfig 之前，实例化一个 DumpConfig 类进行 dump 的配置（包括配置 dump 路径、迭代的数据、标明是算子的输入或输出数据等），如程序清单 5-62 所示。

程序清单 5-62　Estimator 模式下在 NPU 上数据

```
from npu_bridge.estimator.npu.npu_config import NPURunConfig
from npu_bridge.estimator.npu.npu_config import DumpConfig

# dump_path: dump 数据存放路径，该参数指定的目录需要在启动训练的环境上（容器或 Host
侧）提前创建且确保安装时配置的运行用户具有读写权限
# enable_dump: 是否开启 Data Dump 功能
# dump_step: 指定采集哪些迭代的 Data Dump 数据
# dump_mode: Data Dump 模式，取值：input/output/all
dump_config = DumpConfig(enable_dump = True, dump_path = "/home/HwHiAiUser/output",
dump_step = "0|5|10", dump_mode = "all")

session_config = tf.ConfigProto()

config = NPURunConfig(
  dump_config = dump_config,
  session_config = session_config
  )
```

session.run 模式：通过 session 配置项 enable_dump、dump_path、dump_step、dump_mode 来配置 dump 参数，参见程序清单 5-63。

程序清单 5-63　session.run 模式下在 NPU 上数据

```
config = tf.ConfigProto()

custom_op = config.graph_options.rewrite_options.custom_optimizers.add()
```

```
custom_op.name = "NpuOptimizer"
custom_op.parameter_map["use_off_line"].b = True

# enable_dump: 是否开启 Data Dump 功能
custom_op.parameter_map["enable_dump"].b = True
# dump_path: dump 数据存放路径,该参数指定的目录需要在启动训练的环境上(容器或 Host
侧)提前创建且确保安装时配置的运行用户具有读写权限
custom_op.parameter_map["dump_path"].s = tf.compat.as_bytes("/home/HwHiAiUser/output")
# dump_step: 指定采集哪些迭代的 Data Dump 数据
custom_op.parameter_map["dump_step"].s = tf.compat.as_bytes("0|5|10")
# dump_mode: Data Dump 模式,取值: input/output/all
custom_op.parameter_map["dump_mode"].s = tf.compat.as_bytes("all")
config.graph_options.rewrite_options.remapping = RewriterConfig.OFF

with tf.Session(config = config) as sess:
  print(sess.run(cost))
```

注意这里的 dump_path 指转储出来的参数文件保存位置,这里就保存在创建的目录中。dump_step 指想要转储出第几个训练出的结果,多个训练出之间用“|”隔开,连续多个训练出可以用诸如“5-10”来表达,这里为了举例,只转储第 0 个训练出的结果,跟 GPU 上保持一致。

接下来就可以执行训练脚本,生成转储数据文件了。转储生成的文件默认保存在{dump_path}/{time}/{deviceid}/{model_name}/{model_id}/{data_index}目录下。同时在训练脚本当前目录生成图文件,例如 ge_proto_xxxxx_Build.txt。这个文件就是后续所需的计算图,计算图名称取计算图文件下的 name 字段值。

3. 使用 MindStudio 进行精度比对

当准备好各个平台的转储数据和计算图之后,借助 MindStudio 就能将精度进行比对。

打开 MindStudio 并新建一个训练工程,菜单栏 Ascend-> Model Accuracy Analyzer 就是需要使用的模型精度比对工具。

如图 5-24 所示,在左侧的 My Output 处选择在 NPU 上 dump 出来的数据,也就是上一步的参数 dump 结果目录;在右侧的 Ground Truth 处选择第一步骤中 GPU Dump 出来的结果所在目录;在下一行的 Compare Rule Configuration 处选择生成的计算图文件。

单击下方的 Compare 按钮,就可以获得如图 5-25 所示的精度比对的结果了。

在图 5-25 的表格中出现了网络中每个算子的横向比对情况,其中每一列的含义如表 5-3 所示。

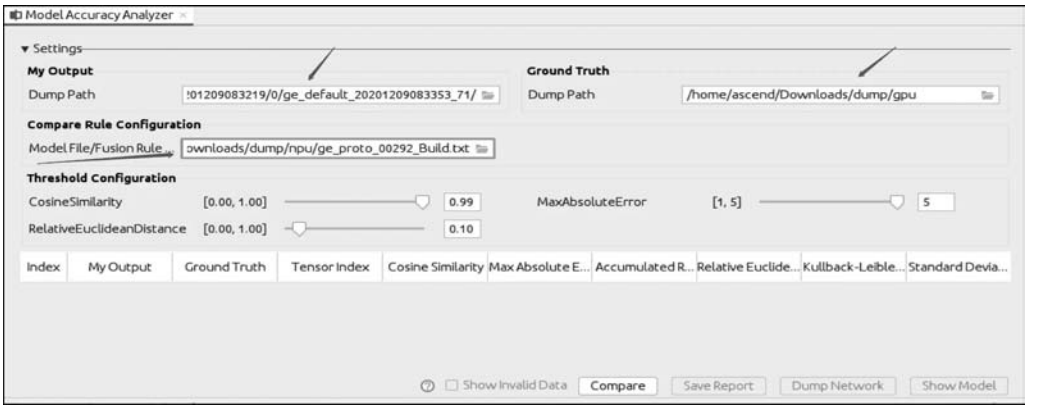

图 5-24　使用 MindStudio 进行精度比较

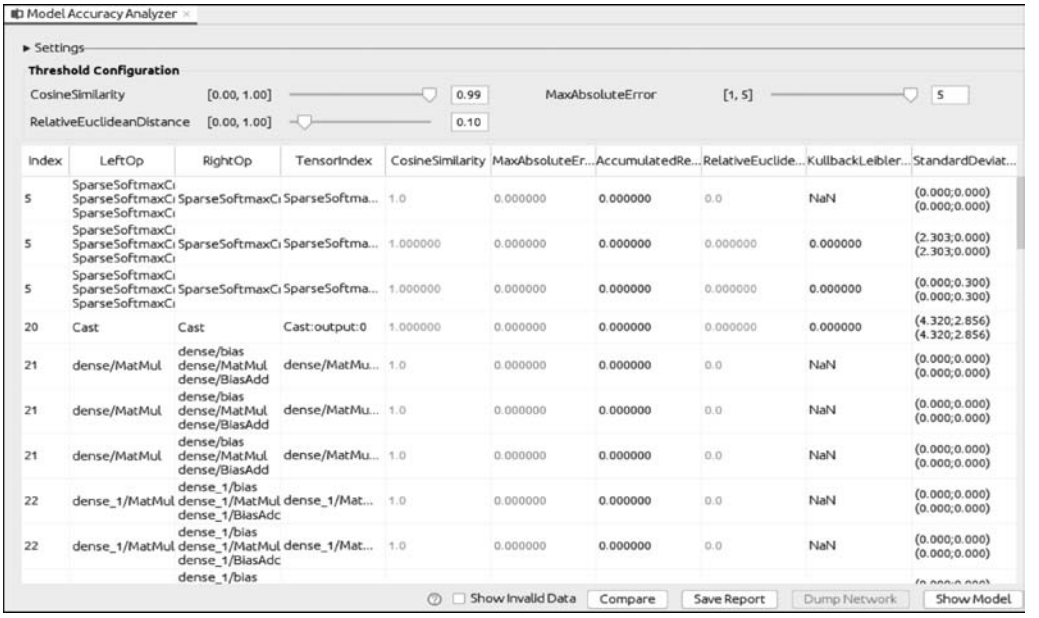

图 5-25　精度比对的结果

表 5-3　精度比对各列的含义

列　　名	含　　义
LeftOp	基于昇腾 AI 处理器运行生成的 dump 数据的算子名
RightOp	基于 GPU/CPU 运行生成的 npy 或 dump 数据的算子名
TensorIndex	基于昇腾 AI 处理器运行生成的 dump 数据的算子 input ID 和 output ID
CosineSimilarity	进行余弦相似度算法比对出来的结果，范围是[－1，1]，比对的结果越接近 1，表示两者的值越相近，越接近－1 意味着两者的值越相反

续表

列　　名	含　　义
MaxAbsoluteError	进行最大绝对误差算法比对出来的结果，值越接近于 0，表明越相近，值越大，表明差距越大
AccumulatedRelativeError	进行累积相对误差算法比对出来的结果，值越接近于 0，表明越相近，值越大，表明差距越大
RelativeEuclideanDistance	进行欧氏相对距离算法比对出来的结果，值越接近于 0，表明越相近，值越大，表明差距越大
KullbackLeiblerDivergence	进行 KLD 散度算法比对出来的结果，取值范围是 0 到无穷大。KLD 散度越小，真实分布与近似分布之间的匹配越好
StandardDeviation	进行标准差算法比对出来的结果，取值范围为 0 到无穷大。标准差越小，离散度越小，表明越接近均值。该列显示两组数据的均值和标准差，第一组展示基于昇腾 AI 处理器运行生成的 dump 数据的数值，第二组展示基于 GPU/CPU 运行生成的 dump 数据的数值

值得注意的是，如果对比表中显示“*”，则表示其新增的算子无对应的原始算子；NaN 表示无比对结果。余弦相似度和 KLD 散度比较结果为 NaN，其他算法有比较数据，则表明左侧或右侧数据为 0；KLD 散度比较结果为 Inf，表明右侧数据有一个为 0。

完成精度分析的全部流程后，如果发现某个算子上的参数差异过大，可能是整网精度较低的原因，可以分析一下这个算子的特性，针对 CANN 的特点进行适配性开发。希望通过本章的介绍，开发者能够借助强大有力的 CANN 软件栈和大量实用工具，训练出高性能的算法模型。

5.6　本章小结

本章系统介绍了基于昇腾软件栈 CANN 进行模型训练的完整流程。从当前市面上各主流深度学习框架入手，对深度学习框架的发展历程进行了简单的梳理，并将各主流框架的优劣势进行了详尽的说明。

以面向全场景 AI 计算框架——MindSpore 为例，讲解了借助深度学习框架完成深度学习任务的完整流程。无论是何种场景，都可以将深度学习训练拆解为数据处理、模型搭建、训练配置、训练网络和保存模型这五大流程。从实用性的角度看，深度学习计算框架的出现规范化了模型训练过程，也让用户能更聚焦于需要解决的任务本身。以 MindSpore 为代表的框架充分利用了底层计算能力，并将 CANN 的强大能力释放给终端用户使用。

此外，本章还以 ResNet-50 为例，具体讲解了使用 CANN 和 MindSpore 完成图像分

类的具体流程。在这个过程中，以分布式训练、混合精度训练等高阶技巧可以有效地加快模型训练速度。除通过 MindSpore 使用 CANN 和昇腾计算能力之外，CANN 也对主流的 TensorFlow 和 PyTorch 框架进行了适配，仅需很少的改动，就能将训练脚本迁移到 CANN 软件栈上执行训练。

为了追求更高的性能和模型训练效果，CANN 还开放了一些实用的工具供用户使用。用户可以使用 Profiling 进行性能分析，可以使用 AutoTune 进行算子自动调优，也可以使用 Data Dump 工具完成模型精度分析。一站式开发工具 MindStudio 在其中也起到了举足轻重的作用。用户可以参考上述模型训练的全流程，活用各种工具和技巧，训练出属于自己的深度学习模型。