ModelArts

模型训练

文档版本 01

发布日期 2024-05-23

华为云计算技术有限公司

版权所有 © 华为云计算技术有限公司 2024。 保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明

HUAWEI和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

目 录

1 模型开发简介 1
2 准备数据 3
3 准备算法 6
3.1 准备算法简介 6
3.2 使用预置框架（自定义脚本） 7
3.2.1 使用预置框架简介 7
3.2.2 开发自定义脚本 8
3.2.3 创建算法 10
3.3 使用自定义镜像 14
3.4 查看算法详情 17
3.5 查找算法 18
3.6 删除算法 18
4 完成一次训练 19
4.1 创建训练作业 19
4.2 查看训练作业详情 31
4.3 查看训练作业事件 33
4.4 查看训练作业日志 34
4.4.1 什么是训练作业日志 35
4.4.2 普通日志说明 35
4.4.3 如何查看训练作业日志 36
4.4.4 如何通过训练日志定位问题 38
4.5 Cloud Shell 38
4.5.1 使用 Cloud Shell 登录训练容器 38
4.5.2 如何训练作业保持运行中状态 40
4.5.3 如何防止 Cloud Shell 的 Session 断开 41
4.5.4 通过 py-spy 工具分析卡死进程的调用栈并结合代码分析定位卡死问题 42
4.6 查看训练作业资源利用率 42
4.7 评估训练结果 44
4.8 查看训练标签 48
4.9 查看故障恢复详情 49
4.10 查看训练容器环境变量 49
4.11 停止、重建或查找作业 53

4.12 清除训练作业资源 54

5 训练进阶 55

5.1 训练故障自动恢复 55

5.1.1 训练容错检查 55

5.1.2 无条件自动重启 59

5.2 断点续训练和增量训练 60

5.3 训练作业卡死检测 62

5.4 修改训练作业优先级 63

5.5 设置作业为高优先级权限 64

6 分布式训练 65

6.1 分布式训练功能介绍 65

6.2 单机多卡数据并行-DataParallel(DP) 66

6.3 多机多卡数据并行-DistributedDataParallel(DDP) 68

6.4 分布式调测适配及代码示例 69

6.5 分布式训练完整代码示例 73

6.6 基于训练作业启动 PyTorch DDP 训练示例 78

7 自动模型优化（AutoSearch） 83

7.1 超参搜索简介 83

7.2 搜索算法 83

7.2.1 贝叶斯优化(SMAC) 83

7.2.2 TPE 算法 84

7.2.3 模拟退火算法(Anneal) 84

7.3 创建超参搜索作业 85

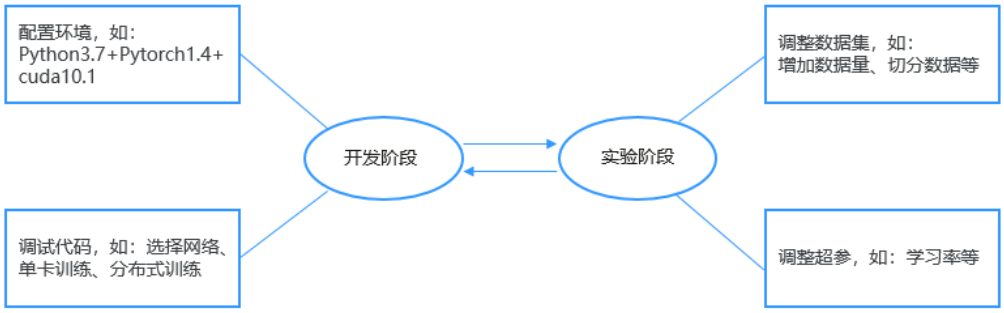
1 模型开发简介

AI模型开发的过程，称之为Modeling，一般包含两个阶段：

- 开发阶段：准备并配置环境，调试代码，使代码能够开始进行深度学习训练，推荐在ModelArts开发环境中调试。
- 实验阶段：调整数据集、调整超参等，通过多轮实验，训练出理想的模型，推荐在ModelArts训练中进行实验。

两个过程可以相互转换。如开发阶段代码稳定后，则会进入实验阶段，通过不断尝试调整超参来迭代模型；或在实验阶段，有一个可以优化训练的性能的想法，则会回到开发阶段，重新优化代码。

图 1-1 模型开发过程



ModelArts提供了模型训练的功能，方便您查看训练情况并不断调整您的模型参数。您还可以基于不同的数据，选择不同规格的资源池用于模型训练。

请参考以下指导在ModelArts上训练模型：

- 将已标注的数据上传至OBS服务使用，请参考**准备数据**。
- 训练模型的算法实现与指导请参考**准备算法**章节。
- 使用控制台创建训练作业请参考**创建训练作业**章节。使用自定义算法构建模型示例请参考**使用自定义算法在ModelArts上构建模型**。
- 关于训练作业日志、训练资源占用等详情请参考**查看训练作业日志**。
- 停止或删除模型训练作业，请参考**停止、重建或查找作业**。

- 模型超参自动调优指南，请参考**自动模型优化（AutoSearch）**。
- 如果您在训练过程中遇到问题，文档中提供了部分故障案例供参考，请参考**训练故障排查**。

2 准备数据

ModelArts使用对象存储服务（Object Storage Service，简称OBS）进行数据存储以及模型的备份和快照，实现安全、高可靠和低成本的存储需求。

- **OBS简介**
- **使用训练数据的两种方式**

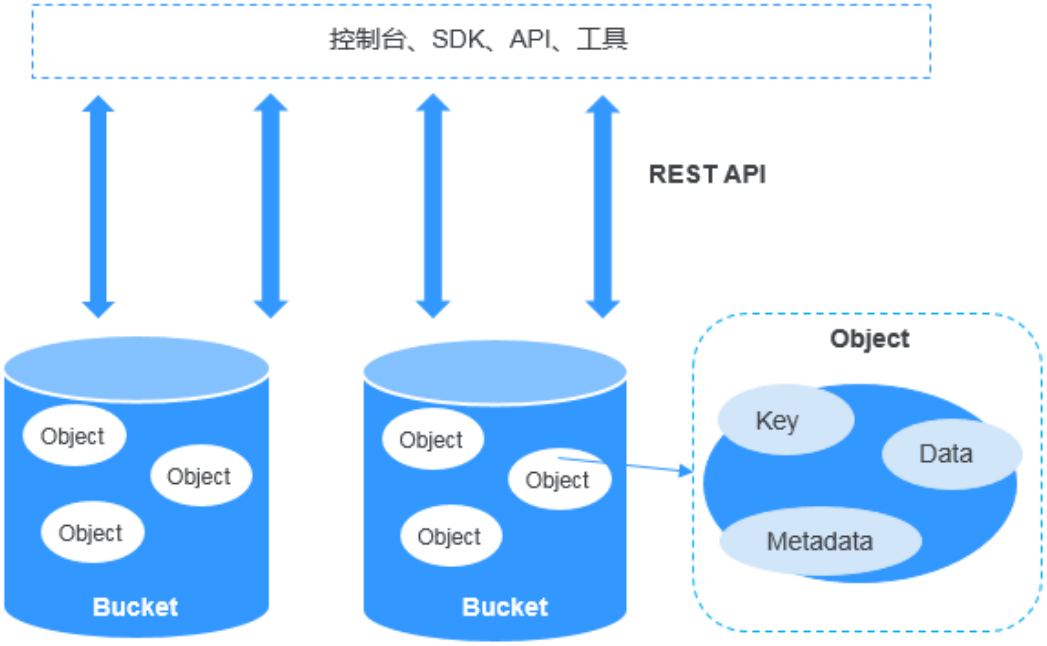
OBS 简介

对象存储服务OBS是一个基于对象的海量存储服务，为客户提供海量、安全、高可靠、低成本的数据存储能力。对象存储服务OBS的基本组成是桶和对象。桶是OBS中存储对象的容器，每个桶都有自己的存储类别、访问权限、所属区域等属性，用户在互联网上通过桶的访问域名来定位桶。对象是OBS中数据存储的基本单位。

对ModelArts来说，obs服务是一个数据存储中心。AI开发过程中的输入数据、输出数据、中间缓存数据都可以在obs桶中进行存储、读取。

因此，在使用ModelArts之前您需要**创建一个OBS桶**，然后在OBS桶中创建文件夹用于存放数据。

图 2-1 对象存储服务 OBS



使用训练数据的两种方式

ModelArts模型训练支持2种读取训练数据的方式：

- 使用OBS桶中存储的数据集

  如果您的数据集已完成数据标注和数据预处理，可以将数据上传至OBS桶。当创建训练作业时，在训练输入参数位置填写训练数据所在的OBS桶路径即可完成训练配置。

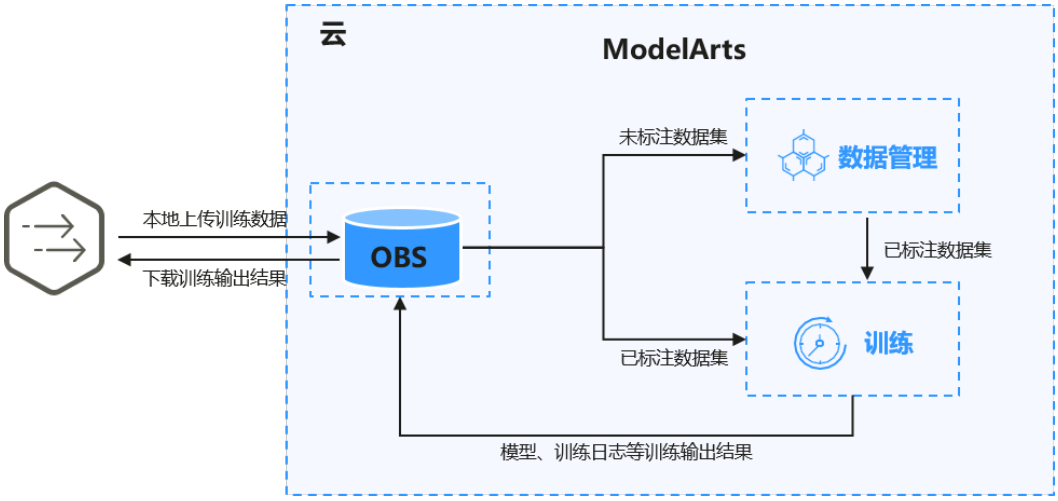
- 使用数据管理中的数据集

  如果您的数据集未标注或者需要进一步的数据预处理，可以将数据导入ModelArts数据管理模块进行数据预处理。

说明

ModelArts数据管理模块在重构升级中，对未使用过数据管理的用户不可见。建议新用户将训练数据存放至OBS桶中使用。

图 2-2 准备数据


云
ModelArts
本地上传训练数据
下载训练输出结果
OBS
未标注数据集
数据管理
已标注数据集
已标注数据集
训练
模型、训练日志等训练输出结果

3 准备算法

3.1 准备算法简介

机器学习从有限的观测数据中学习一般性的规律，并利用这些规律对未知的数据进行预测。为了获取更准确的预测结果，用户需要选择一个合适的算法来训练模型。针对不同的场景，ModelArts提供大量的算法样例。以下章节提供了关于业务场景、算法学习方式、算法实现方式的指导。

选择算法的实现方式

ModelArts提供如下方式实现模型训练。

- 使用预置框架

  如果您需要使用自己开发的算法，可以选择使用ModelArts预置框架。ModelArts支持了大多数主流的AI引擎，详细请参见**预置训练引擎**。这些预置引擎预加载了一些额外的python包，例如numpy等；也支持您通过在代码目录中使用“requirements.txt”文件安装依赖包。使用预置框架创建训练作业请参考**使用预置框架（自定义脚本）**指导。

- 使用自定义镜像

  订阅算法和预置框架涵盖了大部分的训练场景。针对特殊场景，ModelArts支持用户构建自定义镜像用于模型训练。自定义镜像需上传至容器镜像服务（SWR），才能用于ModelArts上训练，请参考**使用自定义镜像训练模型**。由于自定义镜像的制作要求用户对容器相关知识有比较深刻的了解，除非订阅算法和预置引擎无法满足需求，否则不推荐使用。

选择算法的学习方式

ModelArts支持用户根据实际需求进行不同方式的模型训练。

- 离线学习

  离线学习是训练中最基本的方式。离线学习需要一次性提供训练所需的所有数据，在训练完成后，目标函数的优化就停止了。使用离线学习的优势是模型稳定性高，便于做模型的验证与评估。

- 增量学习

增量学习是一个连续不断的学习过程。相较于离线学习，增量学习不需要一次性存储所有的训练数据，缓解了存储资源有限的问题；另一方面，增量学习节约了重新训练中需要消耗大量算力、时间以及经济成本。

3.2 使用预置框架（自定义脚本）

3.2.1 使用预置框架简介

如果订阅算法不能满足需求或者用户希望迁移本地算法至ModelArts上训练，可以考虑使用ModelArts支持的预置框架实现算法构建。这种方式在创建算法时被称为“使用预置框架”模式。

以下章节介绍了如何使用预置框架创建算法。

- 如果需要了解ModelArts模型训练支持的预置引擎和模型，请参考预置的训练引擎。
- 本地开发的算法迁移至ModelArts需要做代码适配，如何适配请参考开发自定义脚本章节。
- 通过ModelArts控制台界面使用预置框架创建算法可以参考创建算法章节。

预置的训练引擎

当前ModelArts支持的训练引擎及对应版本如下所示。

说明

不同区域支持的AI引擎有差异，请以实际环境为准。

表 3-1 训练作业支持的 AI 引擎

工作环境	系统架构	系统版本	AI引擎与版本	支持的cuda或Ascend版本
TensorFlow	x86_64	Ubuntu18.04	tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64	cuda10.1
PyTorch	x86_64	Ubuntu18.04	pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64	cuda10.2
MPI	x86_64	Ubuntu18.04	mindspore_1.3.0-cuda_10.1-py_3.7-ubuntu_1804-x86_64	cuda_10.1
Horovod	x86_64	ubuntu_18.04	horovod_0.20.0-tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64	cuda_10.1

工作环境	系统架构	系统版本	AI引擎与版本	支持的cuda或Ascend版本
			horovod_0.22.1-pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64	cuda_10.2

3.2.2 开发自定义脚本

当您使用预置框架创建算法时，您需要提前完成算法的代码开发。本章详细介绍如何改造本地代码以适配ModelArts上的训练。

创建算法时，您需要在创建页面提供代码目录路径、代码目录路径中的启动文件、训练输入路径参数和训练输出路径参数。这四种输入搭建了用户代码和ModelArts后台交互的桥梁。

- 代码目录路径

  您需要在OBS桶中指定代码目录，并将训练代码、依赖安装包或者预生成模型等训练所需文件上传至该代码目录下。训练作业创建完成后，ModelArts会将代码目录及其子目录下载至后台容器中。

  例如：OBS路径“obs://obs-bucket/training-test/demo-code”作为代码目录，OBS路径下的内容会被自动下载至训练容器的“${MA_JOB_DIR}/demo-code”目录中，demo-code为OBS存放代码路径的最后一级目录，用户可以根据实际修改。

  请注意不要将训练数据放在代码目录路径下。训练数据比较大，训练代码目录在训练作业启动后会下载至后台，可能会有下载失败的风险。建议训练代码目录大小小于或等于50MB。

- 代码目录路径中的启动文件

  代码目录路径中的启动文件作为训练启动的入口，当前只支持python格式。

- 训练输入路径参数

  训练数据需上传至OBS桶或者存储至**数据集**中。在训练代码中，用户需解析**输入路径参数**。系统后台会自动下载输入参数路径中的训练数据至训练容器的本地目录。请保证您设置的桶路径有读取权限。在训练作业启动后，ModelArts会挂载硬盘至“/cache”目录，用户可以使用此目录来存储临时文件。“/cache”目录大小请参考**训练环境中不同规格资源“/cache”目录的大小**。

- 训练输出路径参数

  建议设置一个空目录为训练输出路径。在训练代码中，您需要解析**输出路径参数**。系统后台会自动上传训练输出至指定的训练输出路径，请保证您设置的桶路径有写入权限和读取权限。

在ModelArts中，训练代码需包含以下步骤：

（可选）引入依赖

1. 当您使用自定义脚本创建算法的时候，如果您的模型引用了其他依赖，您需要在“算法管理 > 创建算法”的“代码目录”下放置相应的文件或安装包。

- 安装python依赖包请参考**模型中引用依赖包时，如何创建训练作业?**
- 安装C++的依赖库请参考**如何安装C++的依赖库?**
- 在预训练模型中加载参数请参考**如何在训练中加载部分训练好的参数?**

解析输入路径参数、输出路径参数

运行在ModelArts的模型读取存储在OBS服务的数据，或者输出至OBS服务指定路径，输入和输出数据需要配置2个地方：

1. 训练代码中需解析输入路径参数和输出路径参数。ModelArts推荐以下方式实现参数解析。

```
import argparse
# 创建解析
parser = argparse.ArgumentParser(description='train mnist')

# 添加参数
parser.add_argument('--data_url', type=str, default="./Data/mnist.npz", help='path where the dataset is saved')
parser.add_argument('--train_url', type=str, default="./Model", help='path where the model is saved')

# 解析参数
args = parser.parse_args()
```

   完成参数解析后，用户使用“data_url”、“train_url”代替算法中数据来源和数据输出所需的路径。

2. 在创建训练作业时，填写输入路径和输出路径。

   训练输入选择对应的OBS路径或者数据集路径；训练输出选择对应的OBS路径。

   图 3-1 训练输入和输出设置

训练代码正文和保存模型

训练代码正文和保存模型涉及的代码与您使用的AI引擎密切相关，以下案例以Tensorflow框架为例。案例中使用到的“mnist.npz”文件需要提前**下载**并上传至OBS桶中，训练输入为“mnist.npz”所在OBS路径。

```
import os
import argparse
import tensorflow as tf

parser = argparse.ArgumentParser(description='train mnist')
parser.add_argument('--data_url', type=str, default="./Data/mnist.npz", help='path where the dataset is saved')
parser.add_argument('--train_url', type=str, default="./Model", help='path where the model is saved')
args = parser.parse_args()

mnist = tf.keras.datasets.mnist
```

```
(x_train, y_train), (x_test, y_test) = mnist.load_data(args.data_url)
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10)
])

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)

model.save(os.path.join(args.train_url, 'model'))
```

3.2.3 创建算法

针对您在本地或使用其他工具开发的算法，支持上传至ModelArts中统一管理。在创建自定义算法过程中，您需要关注以下内容：

1. 前提条件
2. 进入创建算法页面
3. 设置算法基本信息
4. 设置算法启动方式
5. 输入输出管道设置
6. 定义超参
7. 支持的策略
8. 添加训练约束
9. 运行环境预览
10. 后续操作

前提条件

- 数据已完成准备：已在ModelArts中创建可用的数据集，或者您已将用于训练的数据集上传至OBS目录。
- 请准备好训练脚本，并上传至OBS目录。训练脚本开发指导参见开发自定义脚本。
- 已在OBS创建至少1个空的文件夹，用于存储训练输出的内容。
- 由于训练作业运行需消耗资源，确保账户未欠费。
- 确保您使用的OBS目录与ModelArts在同一区域。

进入创建算法页面

1. 登录ModelArts管理控制台，单击左侧菜单栏的“算法管理”。
2. 在“我的算法”管理页面，单击“创建”，进入“创建算法”页面。

设置算法基本信息

填写算法的基本信息，包含“名称”和“描述”。

设置算法启动方式

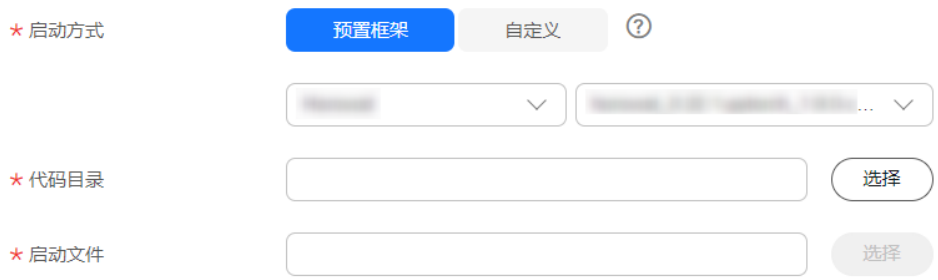
选择“预置框架”创建算法。

用户需根据实际算法代码情况设置“镜像”、“代码目录”和“启动文件”。选择的AI镜像和编写算法代码时选择的框架必须一致。例如编写算法代码使用的是TensorFlow，则在创建算法时也要选择TensorFlow镜像。

表 3-2 启动方式参数说明

参数	说明
“启动方式-预置框架”	选择算法使用的预置框架引擎和引擎版本。勾选“显示旧版镜像”，可以选择旧版预置框架。
“代码目录”	算法代码存储的OBS路径。训练代码、依赖安装包或者预生成模型等训练所需文件上传至该代码目录下。 请注意不要将训练数据放在代码目录路径下。训练数据比较大，训练代码目录在训练作业启动后会下载至后台，可能会有下载失败的风险。 训练作业创建完成后，ModelArts会将代码目录及其子目录下载至训练后台容器中。 例如：OBS路径“obs://obs-bucket/training-test/demo-code”作为代码目录，OBS路径下的内容会被自动下载至训练容器的“${MA_JOB_DIR}/demo-code”目录中，demo-code为OBS存放代码路径的最后一级目录，用户可以根据实际修改。 **说明** ● 编程语言不限。 ● 文件数（含文件、文件夹数量）小于或等于1000个。 ● 文件总大小小于或等于5GB。
“启动文件”	必须为“代码目录”下的文件，且以“.py”结尾，即ModelArts目前只支持使用Python语言编写的启动文件。 代码目录路径中的启动文件为训练启动的入口。

图 3-2 使用自定义脚本创建算法



输入输出管道设置

训练过程中，基于预置框架的算法需要从OBS桶或者数据集中获取数据进行模型训练，训练产生的输出结果也需要存储至OBS桶中。用户的算法代码中需解析输入输出

参数实现ModelArts后台与OBS的数据交互，用户可以参考开发自定义脚本完成适配ModelArts训练的代码开发。

创建基于预置框架的算法时，用户需要配置算法代码中定义的输入输出参数。

- 输入配置

表 3-3 输入配置

参数	参数说明
参数名称	根据实际代码中的输入数据参数定义此处的名称。此处设置的代码路径参数必须与算法代码中解析的训练输入数据参数保持一致，否则您的算法代码无法获取正确的输入数据。 例如，算法代码中使用argparse解析的data_url作为输入数据的参数，那么创建算法时就需要配置输入数据的参数名称为“data_url”。
描述	输入参数的说明，用户可以自定义描述。
获取方式	输入参数的获取方式，默认使用“超参”，也可以选择“环境变量”。
输入约束	开启后，用户可以根据实际情况限制数据输入来源。输入来源可以选择“数据存储位置”或者“ModelArts数据集”。 如果用户选择数据来源为ModelArts数据集，还可以约束以下三种： ● 标注类型。数据类型请参考标注数据。 ● 数据格式。可选“Default”和“CarbonData”，支持多选。其中“Default”代表Manifest格式。 ● 数据切分。仅“图像分类”、“物体检测”、“文本分类”和“声音分类”类型数据集支持进行数据切分功能。可选“仅支持切分的数据集”、“仅支持未切分数据集”和“无限制”。数据切分详细内容可参考发布数据版本。
添加	用户可以根据实际算法添加多个输入数据来源。

- 输出配置

表 3-4 输出配置

参数	参数说明
参数名称	根据实际代码中的训练输出参数定义此处的名称。此处设置的代码路径参数必须与算法代码中解析的训练输出参数保持一致，否则您的算法代码无法获取正确的输出路径。 例如，算法代码中使用argparse解析的train_url作为训练输出数据的参数，那么创建算法时就需要配置输出数据的参数名称为“train_url”。
描述	输出参数的说明，用户可以自定义描述。

参数	参数说明
获取方式	输出参数的获取方式，默认使用“超参”，也可以选择“环境变量”。
添加	用户可以根据实际算法添加多个输出数据路径。

定义超参

使用预置框架创建算法时，ModelArts支持用户自定义超参，方便用户查阅或修改。定义超参后会体现在启动命令中，以命令行参数的形式传入您的启动文件中。

1. 导入超参

   您可以单击“增加超参”手动添加超参。

   图 3-3 添加超参

2. 编辑超参

   超参的参数说明参见**表3-5**。

   表 3-5 超参编辑参数

参数	说明
名称	填入超参名称。 超参名称支持64个以内字符，仅支持大小写字母、数字、下划线和中划线。
类型	填入超参的数据类型。支持String、Integer、Float和Boolean。
默认值	填入超参的默认值。创建训练作业时，默认使用该值进行训练。
约束	单击“约束”。在弹出对话框中，支持用户设置默认值的取值范围或者枚举值范围。
必需	选择是或否。 ● 选择否，则在使用该算法创建训练作业时，支持在创建训练作业页面删除该超参。 ● 选择是，则在使用该算法创建训练作业时，不支持在创建训练作业页面删除该超参。
描述	填入超参的描述说明。 超参描述支持大小写字母、中文、数字、空格、中划线、下划线、中英文逗号和中英文句号。

支持的策略

ModelArts支持用户使用自动化搜索功能。自动化搜索功能在零代码修改的前提下，自动找到最合适的超参，有助于提高模型精度和收敛速度。详细的参数配置请参考超参搜索配置。

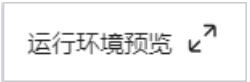
自动搜索目前仅支持pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64，tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64镜像。

添加训练约束

用户可以根据实际情况定义此算法的训练约束。

- 资源类型：选择适用的资源类型，支持多选。
- 多卡训练：选择是否支持多卡训练。
- 分布式训练：选择是否支持分布式训练。

运行环境预览

创建算法时，可以打开创建页面右下方的运行环境预览窗口 

，辅助您了解代码目录、启动文件、输入输出等数据配置在训练容器中的路径。

后续操作

创建算法完成后，可以使用算法创建训练作业，详细操作请参见创建训练作业。

3.3 使用自定义镜像

订阅算法和预置框架涵盖了大部分的训练场景。针对特殊场景，ModelArts支持用户构建自定义镜像用于模型训练。

自定义镜像的制作要求用户对容器相关知识有比较深刻的了解，除非订阅算法和预置框架无法满足需求，否则不推荐使用。自定义镜像需上传至容器镜像服务（SWR），才能用于ModelArts上训练。

ModelArts上使用自定义镜像训练支持2种方式：

- 使用预置框架 + 自定义镜像：

  如果先前基于预置框架且通过指定代码目录和启动文件的方式来创建的训练作业；但是随着业务逻辑的逐渐复杂，您期望可以基于预置框架修改或增加一些软件依赖的时候，此时您可以使用预置框架 + 自定义镜像的功能，即选择预置框架名称后，在预置框架版本下拉列表中选择“自定义”。

- 完全自定义镜像：

  用户遵循ModelArts镜像的规范要求制作镜像，选择自己的镜像，并且通过指定代码目录（可选）和启动命令的方式来创建的训练作业。

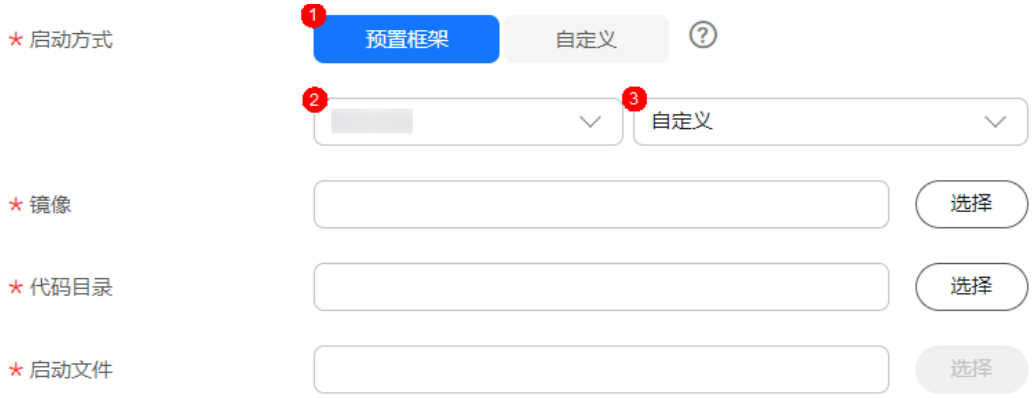
  说明

  当使用完全自定义镜像创建训练作业时，“启动命令”必须在“/home/ma-user”目录下执行，否则训练作业可能会运行异常。

使用预置框架 + 自定义镜像

此功能与直接基于预置框架创建训练作业的区别仅在于，镜像是由用户自行选择的。用户可以基于预置框架制作自定义镜像。

图 3-4 使用预置框架+自定义镜像创建算法



该功能的行为与直接基于预置框架创建的训练作业相同，例如：

- 系统将会自动注入一系列环境变量
  - PATH=${MA_HOME}/anaconda/bin:${PATH}
  - LD_LIBRARY_PATH=${MA_HOME}/anaconda/lib:${LD_LIBRARY_PATH}
  - PYTHONPATH=${MA_JOB_DIR}:${PYTHONPATH}
- 您选择的启动文件将会被系统自动以python命令直接启动，因此请确保镜像中的Python命令为您预期的Python环境。注意到系统自动注入的PATH环境变量，您可以参考下述命令确认训练作业最终使用的Python版本：
  - export MA_HOME=/home/ma-user; docker run --rm {image} ${MA_HOME}/anaconda/bin/python -V
  - docker run --rm {image} $(which python) -V
- 系统将会自动添加预置框架关联的超参

完全使用自定义镜像

图 3-5 完全使用自定义镜像创建算法

训练支持的自定义镜像使用说明请参考**使用自定义镜像创建训练作业**。

完全使用自定义镜像场景下，指定的“conda env”启动训练方法如下：

由于训练作业运行时不是shell环境，因此无法直接使用“conda activate”命令激活指定的“conda env”，需要使用其他方式以达成使用指定“conda env”来启动训练的效果。

假设您的自定义镜像中的“conda”安装于“/home/ma-user/anaconda3”目录“conda env”为“python-3.7.10”，训练脚本位于“/home/ma-user/modelarts/user-job-dir/code/train.py”。可通过以下方式使用指定的“conda env”启动训练：

- 方式一：为镜像设置正确的“DEFAULT_CONDA_ENV_NAME”环境变量与“ANACONDA_DIR”环境变量。

  您可以使用Python命令启动训练脚本。启动命令示例如下：
  ```
  python /home/ma-user/modelarts/user-job-dir/code/train.py
  ```
- 方式二：使用“conda env python”的绝对路径。

  您可以使用“/home/ma-user/anaconda3/envs/python-3.7.10/bin/python”命令启动训练脚本。启动命令示例如下：
  ```
  /home/ma-user/anaconda3/envs/python-3.7.10/bin/python /home/ma-user/modelarts/user-job-dir/code/train.py
  ```
- 方式三：设置PATH环境变量。

  您可以将指定的“conda env bin”目录配置到PATH环境变量中。您可以使用Python命令启动训练脚本。启动命令示例如下：
  ```
  export PATH=/home/ma-user/anaconda3/envs/python-3.7.10/bin:$PATH; python /home/ma-user/modelarts/user-job-dir/code/train.py
  ```
- 方式四：使用“conda run -n”命令。

  您可以使用“/home/ma-user/anaconda3/bin/conda run -n python-3.7.10”命令来执行训练命令，启动命令示例如下：
  ```
  /home/ma-user/anaconda3/bin/conda run -n python-3.7.10 python /home/ma-user/modelarts/user-job-dir/code/train.py
  ```

说明

如果在训练时发生找不到“$ANACONDA_DIR/envs/$DEFAULT_CONDA_ENV_NAME/lib ”目录下“.so”文件的相关报错，可以尝试将该目录加入到“LD_LIBRARY_PATH”，将以下命令放在上述启动方式命令前：

```
export LD_LIBRARY_PATH=$ANACONDA_DIR/envs/$DEFAULT_CONDA_ENV_NAME/
lib:$LD_LIBRARY_PATH;
```

例如，方式一的启动命令示例此时变为：

```
export LD_LIBRARY_PATH=$ANACONDA_DIR/envs/$DEFAULT_CONDA_ENV_NAME/
lib:$LD_LIBRARY_PATH; python /home/ma-user/modelarts/user-job-dir/code/train.py
```

3.4 查看算法详情

1. 登录ModelArts管理控制台。
2. 在左侧导航栏中，选择“算法管理”，进入“我的算法”页面。
3. 在“我的算法”列表中，单击算法名称进入详情页。
   - 选择“基本信息”页签可以查看算法信息。

表 3-6 算法基本信息

参数	说明
名称	算法的名称。
ID	算法的唯一标识。
描述	算法的简介。 单击编辑图标，可以更新描述。
预置框架	算法使用的预置框架引擎及版本。只有使用预置框架创建的算法才有该参数。
自定义镜像	算法使用的容器镜像。只有使用自定义引擎版本或自定义镜像创建的算法才有该参数。
代码目录	算法代码存放的OBS目录。
启动文件	启动文件的OBS存放目录。只有使用预置框架创建的算法才有该参数。
启动命令	显示启动命令。只有使用自定义镜像创建的算法才有该参数。
输入	算法的输入参数。
输出	算法的输出参数。
超参	算法的超参信息。
支持的策略	算法的自动搜索策略。如果空白即表示不支持自动搜索，否则显示自动搜索的参数信息。
训练约束	算法的训练约束信息，如果显示“否”则表示不存在约束，否则会显示支持的资源类型和训练场景的支持情况。

- 选择“训练列表”页签可以查看使用该算法的训练作业信息，例如训练作业名称、状态。

4. “基本信息”页签，单击“编辑”，支持修改除名称和ID之外的算法信息。修改完成，单击“保存”即可完成修改。

3.5 查找算法

ModelArts提供查找算法功能帮助用户快速查找算法。

操作一：按照名称、镜像、代码目录、描述、创建时间筛选的高级搜索。

操作二：单击右上角“刷新”图标，刷新算法列表。

操作三：自定义列功能设置。

图 3-6 查找算法

如果需要对算法排序，可单击表头中的 箭头进行排序。

3.6 删除算法

删除我的算法

在“算法管理 > 我的算法”页面，“删除”运行结束的训练作业。您可以单击“操作”列的“删除”，在弹出的提示框中单击“确认”，删除对应的算法。

4 完成一次训练

4.1 创建训练作业

模型训练是一个不断迭代和优化模型权重的过程。ModelArts的训练模块支持创建训练作业、查看训练情况以及管理训练版本。通过模型训练试验模型结构、数据和超参的各种组合，便于找到最佳的模型结构和权重。

前提条件

- 已经将用于训练作业的数据上传至OBS目录。
- 已经在OBS目录下创建了至少1个空的文件夹，用于存储训练输出的内容。

  说明

  ModelArts不支持加密的OBS桶，创建OBS桶时，请勿开启桶加密。
- 由于训练作业运行需消耗资源，为了避免训练失败请确保账户未欠费。
- 确保使用的OBS目录与ModelArts在同一区域。
- 检查是否配置了访问授权。如果未配置，请参见**使用委托授权**完成操作。
- （可选）如果使用已有算法创建训练作业，需要确认“算法管理”中已准备好算法，具体操作请参见**准备算法简介**。
- （可选）如果使用自定义镜像创建训练作业，需要上传镜像到SWR服务中，具体操作请参见**如何登录并上传镜像到SWR**。

操作流程介绍

创建训练作业的操作步骤如下所示。

步骤1 **进入创建训练作业页面**。

步骤2 **配置训练作业基本信息**。

步骤3 根据不同的算法来源，选择不同的训练作业创建方式。

- 使用预置镜像创建训练作业：**选择创建方式（使用预置镜像）**
- 使用自定义镜像创建训练作：**选择创建方式（使用自定义镜像）**
- 使用已有算法创建训练作业：**选择创建方式（使用我的算法）**

步骤4 配置训练参数：配置训练作业的输入、输出、超参、环境变量等参数。

步骤5 根据需要选择不同的资源池用于训练作业，推荐使用专属资源池，两者的差异说明请参见专属资源池和公共资源池的能力差异。

- 配置资源池（公共资源池）
- 配置资源池（专属资源池）

步骤6 （可选）设置标签：如果需要对训练作业进行资源分组管理，可以设置标签。

步骤7 后续操作。

----结束

进入创建训练作业页面

1. 登录ModelArts管理控制台。
2. 在左侧导航栏中，选择“训练管理 > 训练作业”进入训练作业列表。
3. 单击“创建训练作业”，进入创建训练作业页面。

配置训练作业基本信息

在创建训练作业页面填写训练作业基本信息。

表 4-1 创建训练作业的基本信息

参数名称	说明
名称	必填，训练作业的名称。 系统会自动生成一个名称，可以根据业务需求重新命名，命名规则如下： ● 支持1~64位字符。 ● 可以包含大小写字母、数字、中划线（-）或下划线（_）。
描述	训练作业的简介，便于在训练作业列表了解作业信息。

选择创建方式（使用预置镜像）

如果选择使用预置镜像创建训练作业，则参考表4-2选择训练作业的创建方式。

表 4-2 创建训练作业的创建方式（使用预置镜像）

参数名称	说明
创建方式	必选，选择“自定义算法”。
启动方式	必选，选择“预置框架”，并选择训练作业要使用的预置框架引擎和引擎版本。 如果引擎版本选择“自定义”，则需要配置“镜像”参数，选择自定义镜像用于训练作业。

参数名称	说明
镜像	仅当预置框架的引擎版本选择“自定义”时才显示该参数，且是必填参数。 容器镜像地址的填写支持如下方式。 ● 选择自有镜像或他人共享的镜像：单击右边的“选择”，从容器镜像中选择用于训练的容器镜像。所需镜像需要提前上传到SWR服务中。 ● 选择公开镜像：直接输入SWR服务中公开镜像的地址。地址直接填写“组织名称/镜像名称:版本名称”，不需要带域名信息（swr.<region>.xxx.com），系统会自动拼接域名地址。例如：某公开镜像的SWR地址为“swr.<region>.xxx.com/test-image/tensorflow2_1_1:1.1.1”，则此处填写“test-images/tensorflow2_1_1:1.1.1”。
代码目录	必填，选择训练代码文件所在的OBS目录。 ● 需要提前将代码上传至OBS桶中，目录内文件总大小要小于或等于5GB，文件数要小于或等于1000个，文件深度要小于或等于32。 ● 训练代码文件会在训练作业启动的时候被系统自动下载到训练容器的“${MA_JOB_DIR}/demo-code”目录中，“demo-code”为存放代码目录的最后一级OBS目录。例如，“代码目录”选择的是“/test/code”，则训练代码文件会被下载到训练容器的“${MA_JOB_DIR}/code”目录中。
启动文件	必填，选择代码目录中训练作业的Python启动脚本。 ModelArts只支持使用Python语言编写的启动文件，因此启动文件必须以“.py”结尾。
本地代码目录	指定训练容器的本地目录，启动训练时系统会将代码目录下载至此目录。 此参数可选，默认本地代码目录为“/home/ma-user/modelarts/user-job-dir”。
工作目录	训练时，系统会自动cd到此目录下执行启动文件。

选择创建方式（使用自定义镜像）

如果选择使用自定义镜像创建训练作业，则参考表4-3选择训练作业的创建方式。

表 4-3 创建训练作业的创建方式（使用自定义镜像）

参数名称	说明
创建方式	必选，选择“自定义算法”。
启动方式	必选，选择“自定义”。

参数名称	说明
镜像	必填，填写容器镜像的地址。 容器镜像地址的填写支持如下方式。 ● 选择自有镜像或他人共享的镜像：单击右边的“选择”，从容器镜像中选择用于训练的容器镜像。所需镜像需要提前上传到SWR服务中。 ● 选择公开镜像：直接输入SWR服务中公开镜像的地址。地址直接填写“组织名称/镜像名称:版本名称”，不需要带域名信息（swr.<region>.xxx.com），系统会自动拼接域名地址。例如：某公开镜像的SWR地址为“swr.<region>.xxx.com/test-image/tensorflow2_1_1:1.1.1”，则此处填写“test-images/tensorflow2_1_1:1.1.1”。
代码目录	选择训练代码文件所在的OBS目录。如果自定义镜像中不含训练代码则需要配置该参数，如果自定义镜像中已包含训练代码则不需要配置。 ● 需要提前将代码上传至OBS桶中，目录内文件总大小要小于或等于5GB，文件数要小于或等于1000个，文件深度要小于或等于32。 ● 训练代码文件会在训练作业启动的时候被系统自动下载到训练容器的“${MA_JOB_DIR}/demo-code”目录中，“demo-code”为存放代码目录的最后一级OBS目录。例如，“代码目录”选择的是“/test/code”，则训练代码文件会被下载到训练容器的“${MA_JOB_DIR}/code”目录中。
运行用户ID	容器运行时的用户ID，该参数为选填参数，建议使用默认值1000。 如果需要指定uid，则uid数值需要在规定范围内，不同资源池的uid范围如下： ● 公共资源池：1000-65535 ● 专属资源池：0-65535

启动命令	必填，镜像的启动命令。
	运行训练作业时，当“代码目录”下载完成后，“启动命令”会被自动执行。
	● 如果训练启动脚本用的是py文件，例如“train.py”，则启动命令如下所示。
	`python ${MA_JOB_DIR}/demo-code/train.py`
	● 如果训练启动脚本用的是sh文件，例如“main.sh”，则启动命令如下所示。
	`bash ${MA_JOB_DIR}/demo-code/main.sh`
	启动命令支持使用“;”和“&&”拼接多条命令，命令中的“demo-code”为存放代码目录的最后一级OBS目录，以实际情况为准。

参数名称	说明
本地代码目录	指定训练容器的本地目录，启动训练时系统会将代码目录下载至此目录。 此参数可选，默认本地代码目录为“/home/ma-user/modelarts/user-job-dir”。
工作目录	训练时，系统会自动cd到此目录下执行启动文件。

选择创建方式（使用我的算法）

如果选择使用已有算法创建训练作业，则“创建方式”选择“我的算法”，在算法列表中选择算法。如果没有满足条件的算法，也可以新建算法，具体操作请参见创建算法。

配置训练参数

训练过程中需要从OBS桶或者数据集中获取输入数据进行模型训练，训练输出的结果也要存储至OBS桶中。创建训练作业时可以参考表4-4配置输入、输出、超参、环境变量等参数。

说明

创建训练作业时选择的创建方式不同，训练作业的输入、输出和超参显示不同。如果参数值置灰，即表示该参数已经在算法代码中配置了且不支持修改。

表 4-4 创建训练作业的训练参数

参数名称	子参数	说明
输入	参数名称	算法代码需要通过“输入”的“参数名称”去读取训练的输入数据。 建议设置为“data_url”。训练输入参数要与所选算法的“输入”参数匹配，请参见创建算法时的表3-3。
	数据集	单击“数据集”，在ModelArts数据集列表中勾选目标数据集并选择对应的版本。 训练启动时，系统将自动下载输入路径中的数据到训练运行容器。 **说明** ModelArts数据管理模块在重构升级中，对未使用过数据管理的用户不可见。建议新用户将训练数据存放至OBS桶中使用。
	数据存储位置	单击“数据存储位置”，从OBS桶中选择训练输入数据的存储位置。 训练启动时，系统将自动下载输入路径中的数据到训练运行容器。

参数名称	子参数	说明
	获取方式	以参数名称为“data_path”的训练输入为例，说明获取方式的作用。 ● 当参数的“获取方式”为“超参”时，可以参考如下代码来读取数据。 import argparse parser = argparse.ArgumentParser() parser.add_argument('--data_path') args, unknown = parser.parse_known_args() data_path = args.data_path ● 当参数的“获取方式”为“环境变量”时，可以参考如下代码来读取数据。 import os data_path = os.getenv("data_path", "")
输出	参数名称	算法代码需要通过“输出”的“参数名称”去读取训练的输出目录。 建议设置为“train_url”。训练输出参数要与所选算法的“输出”参数匹配，请参见创建算法时的表3-4。
	数据存储位置	单击“数据存储位置”，从OBS桶中选择训练输出数据的存储位置。训练过程中，系统将自动从训练容器的本地代码目录下同步文件到数据存储位置。 **说明** 数据存储位置仅支持OBS路径。为避免数据存储冲突，建议选择一个空目录用作“数据存储位置”。
	获取方式	以参数名称为“train_url”的训练输出为例，说明获取方式的作用。 ● 当参数的“获取方式”为“超参”时，可以参考如下代码来读取数据。 import argparse parser = argparse.ArgumentParser() parser.add_argument('--train_url') args, unknown = parser.parse_known_args() train_url = args.train_url ● 当参数的“获取方式”为“环境变量”时，可以参考如下代码来读取数据。 import os train_url = os.getenv("train_url", "")
	预下载至本地目录	选择是否将输出目录下的文件预下载至本地目录。 ● 不下载：表示启动训练作业时不会将输出数据的存储位置中的文件下载到训练容器的本地代码目录中。 ● 下载：表示系统会在启动训练作业时自动将输出数据的存储位置中的所有文件下载到训练容器的本地代码目录中。下载时间会随着文件变大而变长，为了防止训练时间过长，请及时清理训练容器的本地代码目录中的无用文件。如果要使用断点续训练和增量训练，则必须选择“下载”。

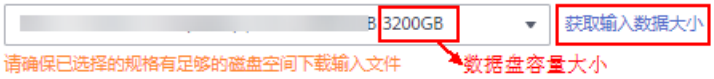
参数名称	子参数	说明
超参	-	超参用于训练调优。此参数由选择的算法决定，如果在算法中已经定义了超参，则此处会显示算法中所有的超参。 超参支持修改和删除，状态取决于算法中的超参“约束”设置，详情请参见定义超参。
环境变量	-	根据业务需求增加环境变量。训练容器中预置的环境变量请参见查看训练容器环境变量。
自动重启	-	打开开关后，可以设置重启次数和是否启用无条件自动重启。 打开自动重启开关后，当由于环境问题导致训练作业异常时，系统将自动修复异常或隔离节点，并重启训练作业，提高训练成功率。为了避免丢失训练进度、浪费算力，开启此功能前请确认代码已适配断点续训，操作指导请参见断点续训练和增量训练。 重启次数的取值范围是1~128，缺省值为3。创建训练后不支持修改重启次数，请合理设置次数。 勾选无条件自动重启后，只要系统检测到训练异常，就无条件重启训练作业。为了避免无效重启浪费算力资源，系统最多只支持连续无条件重启3次。 当训练过程中触发了自动重启，则系统会记录重启信息，在训练作业详情页可以查看故障恢复详情，具体请参见查看故障恢复详情。

配置资源池（公共资源池）

如果使用公共资源池创建训练作业，则参考表4-5配置公共资源池。

表 4-5 创建训练作业的公共资源池

参数名称	说明
资源池	必选，选择“公共资源池”。
资源类型	必选，选择训练需要的资源类型。当训练代码中已定义资源类型时，则根据算法的约束条件选择合适的资源类型。例如，训练代码中定义的资源类型为CPU，这里选择其他类型时会导致训练失败。如果部分资源类型不可见或不可选，表示不支持。

参数名称	说明
规格	必选，根据不同的资源类型，选择所需的资源规格。 当“输入”参数选择“数据存储位置”时，在选择资源池规格时可以单击右侧的“获取输入数据大小”，检查输入数据的大小是否超出数据盘的容量限制，避免训练过程中出现内存不足的情况。  **须知** 资源规格为“GPU:n*tnt004”（n表示具体数字）的资源不支持多进程的训练任务。
计算节点个数	必填，根据需要选择计算节点的个数。默认值为“1”。 ● 当“计算节点个数 = 1”时，创建的是单机训练作业，ModelArts只会在一个节点上启动一个训练容器，该训练容器独享所选规格的计算资源。 ● 当“计算节点个数 > 1”时，创建的是分布式训练作业，更多分布式训练配置请参见**分布式训练功能介绍**。
永久保存日志	选择CPU或者GPU资源时，支持选择是否打开“永久保存日志”开关。 ● 开关关闭（默认关闭）：表示不永久保存日志，则训练日志会在30天后会被清理。可以在作业详情页下载全部日志至本地。 ● 开关打开：表示永久保存日志，此时必须配置“作业日志路径”，系统会将训练日志永久保存至指定的OBS路径。
作业日志路径	打开“永久保存日志”开关时，必须配置“作业日志路径”，用于存放训练作业产生的日志文件。 建议选择一个空的OBS文件目录存放运行中产生的日志文件，同时需要OBS文件目录的读写权限。

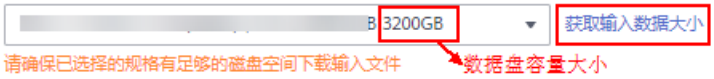
参数名称	说明
事件通知	选择是否打开“事件通知”开关。 ● 开关关闭（默认关闭）：表示不启用消息通知服务。 ● 开关打开：表示订阅消息通知服务，当训练作业发生特定事件（如作业状态变化或疑似卡死）时会发送通知。此时必须配置“主题名”和“事件”。 – “主题名”：事件通知的主题名称。单击“创建主题”，前往消息通知服务中创建主题。 – “事件”：选择要订阅的事件类型。例如“作业开始”、“作业结束”、“作业失败”、“作业终止”、“作业疑似卡死”等。 **说明** ● 需要为消息通知服务中创建的主题添加订阅，当订阅状态为“已确认”后，方可收到事件通知。订阅主题的详细操作请参见**添加订阅**。 ● 使用消息通知服务会产生相关服务费用，详细信息请参见**计费说明**。 ● 只有资源类型为“GPU”的训练作业才支持通知“作业疑似卡死”的事件。
自动停止	当使用付费资源时，可以选择是否打开“自动停止”开关。 ● 开关关闭（默认关闭）：表示训练作业将一直运行直至训练完成。 ● 开关打开：表示启用自动停止功能，此时必须配置自动停止时间，支持设置为“1小时”、“2小时”、“4小时”、6小时或“自定义”，自定义时间取值范围为1~720小时。启用该参数并设置时间后，运行时长到期后将会自动终止训练，准备排队等状态不扣除运行时长。

配置资源池（专属资源池）

如果使用专属资源池创建训练作业，则参考**表4-6**配置公共资源池。

表 4-6 创建训练作业的专属资源池

参数名称	说明
资源池	必选，选择“专属资源池”并选择要使用的资源池。 选择专属资源池时，支持查看当前资源池的状态、节点规格、空闲/碎片节点数、可用节点/总节点数以及卡数信息。单击“空闲/碎片节点数”列的“查看”可以查看碎片详情，确认资源池是否满足训练需求。

参数名称	说明
规格	必选，根据不同的资源类型，选择所需的资源规格。 当“输入”参数选择“数据存储位置”时，在选择资源池规格时可以单击右侧的“获取输入数据大小”，检查输入数据的大小是否超出数据盘的容量限制，避免训练过程中出现内存不足的情况。  **须知** 资源规格为“GPU:n*tnt004”（n表示具体数字）的资源不支持多进程的训练任务。
自定义规格	选择是否打开“自定义规格”开关。训练作业支持基于专属资源池的规格自定义资源规格，进而提升资源池的利用率。 ● 开关关闭（默认关闭）：表示直接使用专属资源池的规格。 ● 开关打开：表示自定义专属资源池的规格，自定义规格只能小于或等于专属资源池的“节点规格”。CPU规格只支持自定义核数和内存，GPU规格支持自定义核数、内存和卡数。 **说明** 如果启用“自定义规格”，则“规格”参数将配置无效。
计算节点个数	必填，根据需要选择计算节点的个数。默认值为“1”。 ● 当“计算节点个数 = 1”时，创建的是单机训练作业，ModelArts只会在一个节点上启动一个训练容器，该训练容器独享所选规格的计算资源。 ● 当“计算节点个数 > 1”时，创建的是分布式训练作业，更多分布式训练配置请参见**分布式训练功能介绍**。
作业优先级	使用专属资源池创建训练作业时，支持设置训练作业的优先级。取值为1~3，默认优先级为1，最高优先级为3。 ● 默认用户权限可选择优先级1和2，配置了“**设置作业为高优先级权限**”的用户可选择优先级1~3。 ● 如果训练作业长时间处于“等待中”的状态，则可以通过修改作业优先级来减少排队时长，请参见**修改训练作业优先级**。

参数名称	说明
SFS Turbo	当ModelArts和SFS Turbo间网络直通时，训练作业支持挂载多个SFS Turbo存放训练数据。单击“增加挂载配置”，填写如下参数。 ● “文件系统”：选择一个SFS Turbo。 ● “云上挂载路径”：输入SFS Turbo对应在训练容器内的云上挂载路径。 ● “存储位置”：选择SFS Turbo的存储位置。如果用户配置了文件夹控制权限，请选择存储位置；如果用户未配置文件夹控制权限，可以保持默认值“/”或者自定义位置。 ● “挂载方式”：显示挂载SFS Turbo的权限。根据SFS Turbo存储位置的权限显示“读写”或“只读”，如果用户未配置文件夹控制权限，则该参数不可见。 **说明** ● 相同的文件系统只能挂载一次，且只能对应一个挂载路径，挂载路径均不可重复。最多可以挂载8个盘。 ● 如果要使用训练作业挂载SFS Turbo功能，需要配置ModelArts和SFS Turbo间网络直通，具体操作请参见**ModelArts网络**。 ● 云上挂载路径有如下限制：不能为 / 目录，不能为 /cache、 /home/ma-user/modelarts等系统已经默认挂载的路径。 ● 如果需要设置SFS Turbo的文件夹权限，请参考**权限管理**文档配置。
永久保存日志	选择CPU或者GPU资源时，支持选择是否打开“永久保存日志”开关。 ● 开关关闭（默认关闭）：表示不永久保存日志，则训练日志会在30天后会被清理。可以在作业详情页下载全部日志至本地。 ● 开关打开：表示永久保存日志，此时必须配置“作业日志路径”，系统会将训练日志永久保存至指定的OBS路径。
作业日志路径	打开“永久保存日志”开关时，必须配置“作业日志路径”，用于存放训练作业产生的日志文件。 建议选择一个空的OBS文件目录存放运行中产生的日志文件，同时需要OBS文件目录的读写权限。

参数名称	说明
事件通知	选择是否打开“事件通知”开关。 ● 开关关闭（默认关闭）：表示不启用消息通知服务。 ● 开关打开：表示订阅消息通知服务，当训练作业发生特定事件（如作业状态变化或疑似卡死）时会发送通知。此时必须配置“主题名”和“事件”。 – “主题名”：事件通知的主题名称。单击“创建主题”，前往消息通知服务中创建主题。 – “事件”：选择要订阅的事件类型。例如“作业开始”、“作业结束”、“作业失败”、“作业终止”、“作业疑似卡死”等。 **说明** ● 需要为消息通知服务中创建的主题添加订阅，当订阅状态为“已确认”后，方可收到事件通知。订阅主题的详细操作请参见**添加订阅**。 ● 使用消息通知服务会产生相关服务费用，详细信息请参见**计费说明**。 ● 只有资源类型为“GPU”的训练作业才支持通知“作业疑似卡死”的事件。
自动停止	当使用付费资源时，可以选择是否打开“自动停止”开关。 ● 开关关闭（默认关闭）：表示训练作业将一直运行直至训练完成。 ● 开关打开：表示启用自动停止功能，此时必须配置自动停止时间，支持设置为“1小时”、“2小时”、“4小时”、6小时或“自定义”，自定义时间取值范围为1~720小时。启用该参数并设置时间后，运行时长到期后将会自动终止训练，准备排队等状态不扣除运行时长。

（可选）设置标签

如果需要通过标签实现资源分组管理，可以在“高级选项”处勾选“现在配置”，可以设置训练作业的“标签”。标签详细用法请参见**ModelArts如何通过标签实现资源分组管理**。

后续操作

当创建训练作业的参数配置完成后，单击“提交”，在信息确认页面单击“确定”，提交创建训练作业任务。

训练作业一般需要运行一段时间，前往训练作业列表，可以查看训练作业的基本情况。

- 在训练作业列表中，刚创建的训练作业状态为“等待中”。
- 当训练作业的状态变为“已完成”时，表示训练作业运行结束，其生成的模型将存储至对应的“输出”目录中。

- 当训练作业的状态变为“运行失败”或“异常”时，可以单击训练作业的名称进入详情页面，通过查看日志等手段处理问题。

说明

训练作业运行过程中将按照选择的资源进行计费。

4.2 查看训练作业详情

1. 登录ModelArts管理控制台。
2. 在左侧导航栏中，选择“训练管理 > 训练作业”，进入“训练作业”列表。
3. 在“训练作业”列表中，单击作业名称，进入训练作业详情页。
4. 在训练作业详情页的左侧，可以查看此次训练作业的基本信息和算法配置的相关信息。
   - **训练作业基本信息**

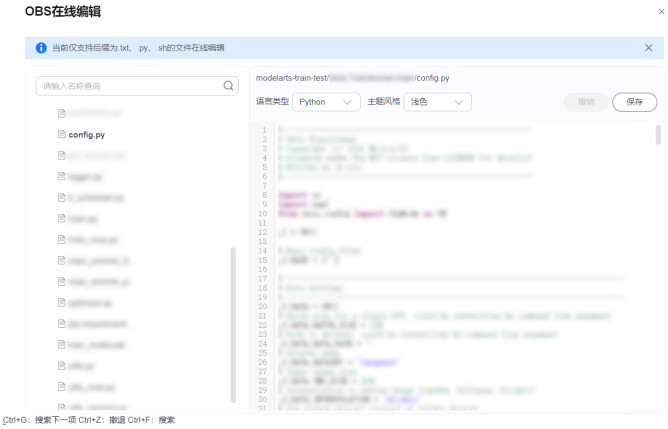
     表 4-7 训练作业基本信息

参数	说明
“作业ID”	训练作业唯一标识。
“作业状态”	训练作业状态。
“创建时间”	记录训练作业创建时间。
“运行时长”	记录训练作业运行时长。
“重启次数”	记录训练过程中时如果出现故障，作业自动重启的次数。仅当创建训练作业时开启“自动重启”功能时可见。
“描述”	训练作业的描述。 您可以单击编辑图标，更新训练作业的描述。

   - **训练作业参数**

     表 4-8 训练作业参数

参数	说明
“算法名称”	本次训练作业使用的算法。单击算法名称，可以跳转至算法详情页面。
“预置镜像”	本次训练作业使用的预置镜像框架。

参数	说明
“代码目录”	训练作业代码目录所在的OBS路径。 您可以单击代码目录后的“编辑代码”，在“OBS在线编辑”对话框中实时编辑训练脚本代码。当训练作业状态为“等待中”、“创建中”和“运行中”时，不支持“OBS在线编辑”功能。  **说明** 当您使用AI Gallery中订阅的算法创建训练作业时，不支持该参数。
“启动文件”	训练作业启动文件位置。 **说明** 当您使用AI Gallery中订阅的算法创建训练作业时，不支持该参数。
“运行用户ID”	容器运行时的用户ID。
“本地代码目录”	训练代码在训练容器中的存放路径。
“工作目录”	训练启动文件在训练容器中的路径。
“计算节点个数”	本次训练作业设置的计算节点个数。
“专属资源池”	专属资源池信息，仅当训练作业使用专属资源池时可见。
“规格”	本次训练作业使用的训练规格。
“输入-输入路径”	本次训练中，输入数据的OBS路径。
“输入-参数名称”	算法代码中，输入路径指代的参数。
“输入-获取方式”	本次训练作业的输入采用的获取方式。

参数	说明
“输入-本地路径（训练参数值）”	训练启动后，ModelArts将OBS路径中的数据下载至后台容器，本地路径指ModelArts后台容器中存储输入数据的路径。
“输出-输出路径”	本次训练中，输出数据的OBS路径。
“输出-参数名称”	算法代码中，输出路径指代的参数。
“输出-获取方式”	本次训练作业的输出采用的获取方式。
“输出-本地路径（训练参数值）”	ModelArts后台容器中存储训练输出的路径。
“超参”	本次训练作业使用的超参。
“环境变量”	本次训练作业设置的环境变量。

4.3 查看训练作业事件

训练作业的（从用户可看见训练任务开始）整个生命周期中，每一个关键事件点在系统后台均有记录，用户可随时在对应训练作业的详情页面进行查看。

方便用户更清楚的了解训练作业运行过程，遇到任务异常时，更加准确的排查定位问题。当前支持的作业事件如下所示：

- 训练作业创建成功
- 训练作业创建失败报错：
- 准备阶段超时。可能原因是跨区域算法同步或者创建共享存储超时
- 训练作业已排队，正在等待资源分配
- 训练作业排队失败
- 训练作业开始运行
- 训练作业运行成功
- 训练作业运行失败
- 训练作业被抢占
- 系统检测到您的作业疑似卡死，请及时前往作业详情界面查看并处理
- 训练作业已重启
- 训练作业已被手动终止
- 训练作业已被终止（最大运行时长：1h）
- 训练作业已被终止（最大运行时长：3h）
- 训练作业已被手动删除
- 计费信息同步结束

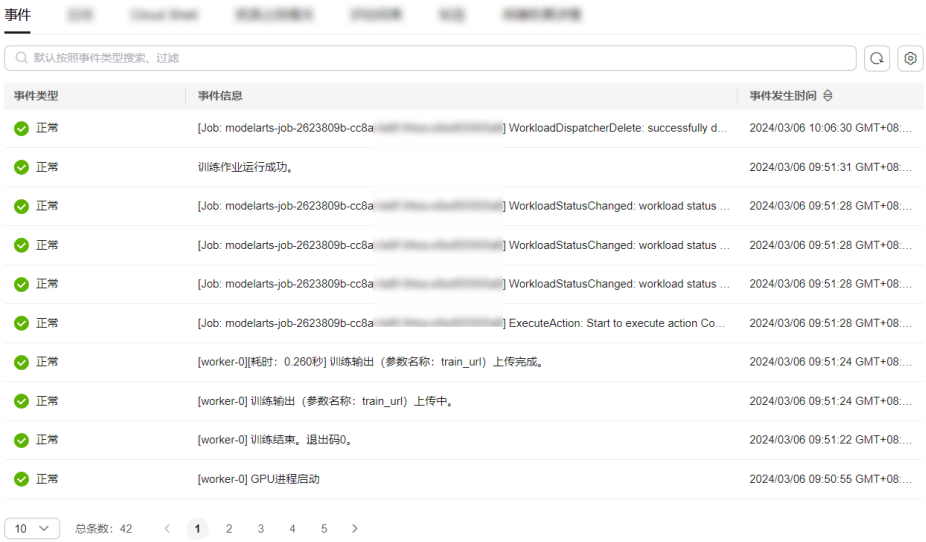
- [worker-0] 训练环境预检中
- [worker-0] [耗时： 秒] 预检完成
- [worker-0] [耗时： 秒] 检查失败。发现异常：
- [worker-0] [耗时： 秒] 检查失败。发现错误：
- [worker-0] 训练代码下载中
- [worker-0] [耗时： 秒] 训练代码下载完成
- [worker-0] [耗时： 秒] 训练代码下载失败，失败原因：
- [worker-0] 训练输入下载中
- [worker-0] [耗时： 秒] 训练输入（参数名称： ）下载完成
- [worker-0] [耗时： 秒] 训练输入（参数名称： ）下载失败，失败原因：
- [worker-0] 正在安装Python依赖包，导入文件：
- [worker-0] [耗时： 秒] Python依赖包安装完成，导入文件：
- [worker-0] 训练任务开始运行
- [worker-0] 训练任务运行结束，退出码
- [worker-0] 训练输入上传中
- [worker-0] [耗时： 秒] 训练输出（参数名称： ）上传完成

训练运行到结束的过程中，关键事件支持手动/自动刷新。

查看操作

1. 在ModelArts管理控制台的左侧导航栏中选择“训练管理 > 训练作业”。
2. 在训练作业列表中，单击作业名称进入训练作业详情页面。
3. 在训练作业详情页面，单击“事件”页签查看事件信息。

图 4-1 查看事件信息



4.4 查看训练作业日志

4.4.1 什么是训练作业日志

训练日志定义

训练日志用于记录训练作业运行过程和异常信息，为快速定位作业运行中出现的问题提供详细信息。用户代码中的标准输出、标准错误信息会在训练日志中呈现。在ModelArts中训练作业遇到问题时，可首先查看日志，多数场景下的问题可以通过日志报错信息直接定位。

训练日志的时效性

从日志产生的时效性上可以分为以下3种情况：

- 实时日志：训练作业实时运行时产生，在ModelArts训练作业详情页面上可以查看。
- 历史日志：训练作业结束后，可以在ModelArts训练作业详情页面上查看历史日志，ModelArts系统自动保存30天。
- 永久日志：转存到OBS桶中的训练日志，在创建训练作业时，打开永久保存日志开关设置作业日志路径即可将日志转存至OBS路径。

图 4-2 开启永久保存日志开关

实时日志和历史日志都是标准日志输出，内容上没有区别。

相关章节

- 在ModelArts训练作业详情页，训练日志窗口提供了日志预览、日志下载、日志中搜索关键字能力，具体请参见**如何查看训练作业日志**。
- ModelArts还提供了训练作业失败定位与分析功能，方便用户通过日志快速定位并解决训练作业问题，具体请参见**如何通过训练日志定位问题**。

4.4.2 普通日志说明

普通日志中包含训练进程日志、pip-requirement.txt安装日志和ModelArts平台日志。

普通日志类型

表 4-9 普通日志类型

日志类型	说明
训练进程日志	用户训练代码的标准输出。
pip-requirement.txt安装日志	如果用户有定义pip-requirement.txt文件，会产生pip包安装日志。

日志类型	说明
ModelArts平台日志	ModelArts平台产生的系统日志，主要用于运维人员定位平台问题。

普通日志的文件格式

普通日志的文件格式如下，其中task id为训练作业中的节点id。

```
统一日志格式：modelarts-job-[job id]-[task id].log
样例：log/modelarts-job-95f661bd-1527-41b8-971c-eca55e513254-worker-0.log
```

- 单机训练作业只会生成一个日志文件，单机作业的task id默认为worker-0。
- 分布式场景下有多个节点日志文件并存，通过task id区分不同节点，例如：worker-0，worker-1等。

训练进程日志、pip-requirement.txt安装日志和ModelArts平台日志都包含在普通日志文件modelarts-job-[job id]-[task id].log中

ModelArts 平台日志

ModelArts平台日志可以通过关键字在训练的普通日志文件modelarts-job-[job id]-[task id].log中筛查，筛查关键字有：[ModelArts Service Log]或Platform=ModelArts-Service。

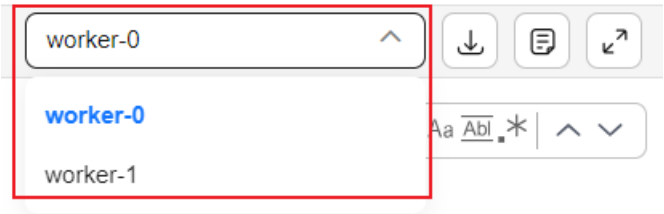
- 类型一：[ModelArts Service Log] xxx
  ```
  [ModelArts Service Log][init] download code_url: s3://dgg-test-user/snt9-test-cases/mindspore/lenet/
  ```
- 类型二：time=“xxx” level=“xxx” msg=“xxx” file=“xxx” Command=xxx Component=xxx Platform=xxx
  ```
  time="2021-07-26T19:24:11+08:00" level=info msg="start the periodic upload task, upload period = 5 seconds " file="upload.go:46" Command=obs/upload Component=ma-training-toolkit Platform=ModelArts-Service
  ```

4.4.3 如何查看训练作业日志

在训练作业详情页，训练日志窗口提供日志预览、日志下载、日志中搜索关键字、系统日志过滤能力。

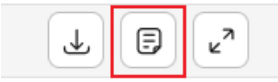
- 预览
  系统日志窗口提供训练日志预览功能，如果训练作业有多个节点，则支持查看不同计算节点的日志，通过右侧下拉框可以选择目标节点预览。

  图 4-3 查看不同计算节日志

  当日志文件过大时，系统日志窗口仅加载最新的部分日志，并在日志窗口上方提供全量日志访问链接。打开该链接可在新页面查看全部日志。

图 4-4 查看全量日志



说明

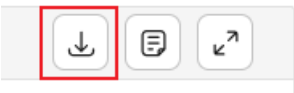
- 如果全部日志超过500M，可能会引起浏览页面卡顿，建议您直接下载日志查看。
- 预览链接在生成后的一小时内，支持任何人打开并查看。您可以分享链接至他人。
- **请注意日志中不能包含隐私内容，否则会造成信息泄露。**

- 下载

  训练日志仅保留30天，超过30天会被清理。如果用户需要永久保存日志，请单击系统日志窗口右上角下载按钮下载日志至本地保存，支持批量下载多节点日志。用户也可以在创建训练作业时打开永久保存日志按钮，保存训练日志至指定OBS路径。

  针对使用Ascend规格创建的训练作业，部分系统日志暂不支持直接在训练日志窗口下载，请在创建训练作业时指定OBS路径用于保存训练日志。

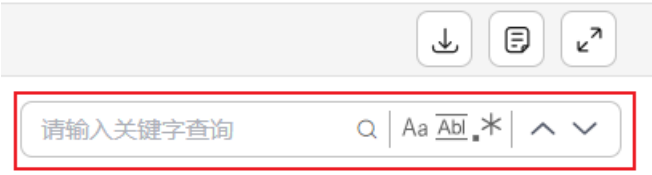
  图 4-5 下载日志

- 搜索关键字

  用户可以在系统日志右上角的搜索框搜索关键字，如图4-6所示。

  图 4-6 搜索关键字

  系统支持高亮关键字并实现搜索结果间的跳转。搜索功能仅支持搜索当前页面加载的日志，如果日志加载不全（请关注页面提示）则需要下载或者通过打开全量日志访问链接进行搜索。全量日志访问链接打开的新页面可以通过Ctrl+F进行搜索。

- 系统日志过滤

  图 4-7 系统日志复选框

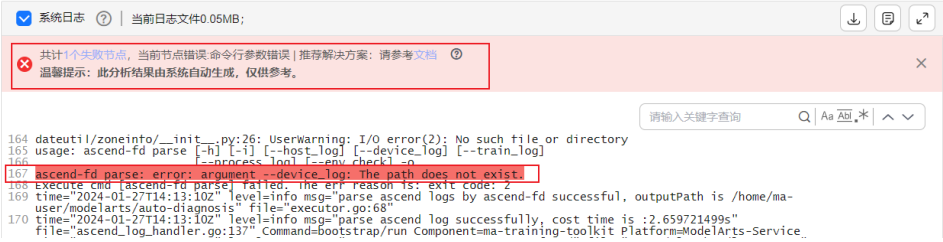
  如果勾选了系统日志复选框，则日志中呈现系统日志和用户日志。如果去勾选，则只显示用户日志。

4.4.4 如何通过训练日志定位问题

在ModelArts中训练作业遇到问题时，可首先查看日志，多数场景下的问题可以通过日志报错信息直接定位。

ModelArts提供了训练作业失败定位与分析功能，如果训练作业运行失败，ModelArts会自动识别导致作业失败的原因，在训练日志界面上给出提示。提示包括三部分：失败的可能原因、推荐的解决方案以及对应的日志（底色标红部分）。

图 4-8 训练故障识别



ModelArts会对部分常见训练错误给出分析建议，目前还不能识别所有错误，提供的失败可能原因仅供参考。针对分布式作业，只会显示当前节点的一个分析结果，作业的失败需要综合各个节点的失败原因做一个综合判断。

常见训练问题定位思路如下：

1. 根据日志界面提示中提供的分析建议解决。
   - 参考案例解决：会提供当前故障对应的指导文档链接，请参照文档中的解决方案修复问题。
   - 重建作业：建议重建作业进行重试，大概率能修复问题。
2. 上一步不能解决问题时，可以尝试分析日志中提示的错误信息，定位并解决问题。
3. 最后，如果以上均不能解决问题，可以提工单进行人工咨询。

4.5 Cloud Shell

4.5.1 使用 Cloud Shell 登录训练容器

使用场景

允许用户使用ModelArts控制台提供的Cloud Shell登录运行中的训练容器。

约束限制

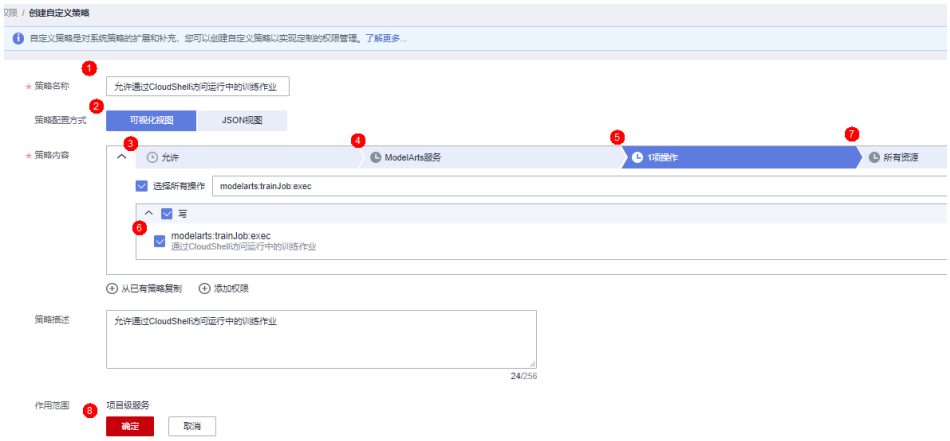
仅专属资源池均支持使用Cloud Shell，且训练作业必须处于“运行中”状态。

前提条件：给子账号配置允许使用 Cloud Shell 的权限

1. 使用主用户账号登录华为云的管理控制台，单击右上角用户名，在下拉框中选择“统一身份认证”，进入统一身份认证（IAM）服务。
2. 在统一身份认证服务页面的左侧导航选择“权限管理 > 权限”，单击右上角的“创建自定义策略”按如下要求设置完成后单击“确定”。

- “策略名称”：设置自定义策略名称，例如：允许通过Cloud Shell访问运行中的训练作业。
- “策略配置方式”：选择可视化视图。
- “策略内容”：允许，云服务中搜索ModelArts服务并选中，操作列中搜索关键词modelarts:trainJob:exec并选中，所有资源选择默认值。

图 4-9 创建自定义策略



3. 在统一身份认证服务页面的左侧导航选择“用户组”，在用户组页面查找待授权的用户组名称，在右侧的操作列单击“授权”，勾选步骤2创建的自定义策略，单击“下一步”，选择授权范围方案，单击“确定”。

   此时，该用户组下的所有用户均有权限通过Cloud Shell登录运行中的训练作业容器。

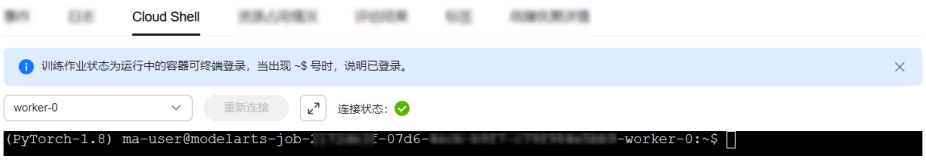
   如果没有用户组，也可以创建一个新的用户组，并通过“用户组管理”功能添加用户，并配置授权。如果指定的子用户没有在用户组中，也可以通过“用户组管理”功能增加用户。

如何使用 Cloud Shell

1. 参考**前提条件：给子账号配置允许使用Cloud Shell的权限**，完成配置。
2. 在ModelArts管理控制台的左侧导航栏中选择“训练管理 > 训练作业”。
3. 在训练作业列表中，单击作业名称进入训练作业详情页面。
4. 在训练作业详情页面，单击“Cloud Shell”页签，登录训练容器。

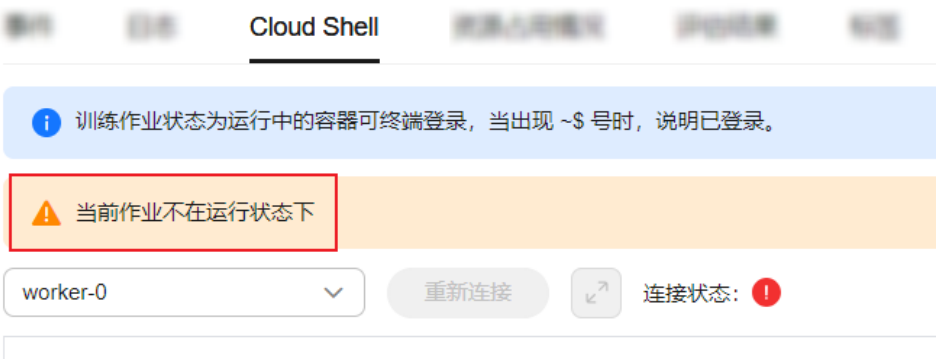
   连接成功后，Cloud Shell界面提示如下。

图 4-10 Cloud Shell 界面



当作业处于非运行状态或权限不足时会导致无法使用Cloud Shell，请根据提示定位原因即可。

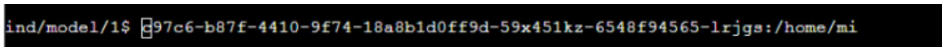
图 4-11 报错提示



说明

部分用户登录Cloud Shell界面时，可能会出现路径显示异常情况，此时在Cloud Shell中单击回车键即可恢复正常。

图 4-12 路径异常



4.5.2 如何训练作业保持运行中状态

由于需要训练作业处于“运行中”状态才能登录Cloud Shell，因此本文介绍如何使训练作业保持运行中状态，方便您快速通过Cloud Shell登录运行中的训练容器。

通过 Sleep 命令使训练作业保持运行

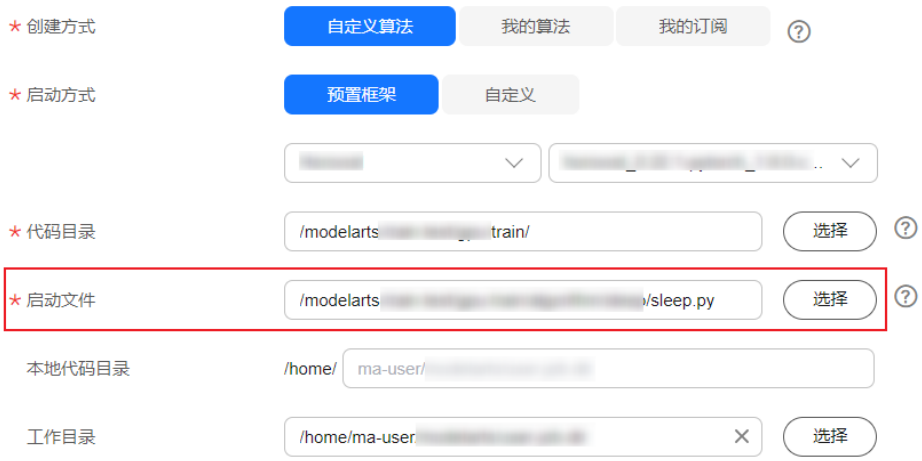
- 如果训练作业使用的是预置框架：

  在创建训练作业时，“创建方式”选择“自定义算法”，“启动方式”选择“预置框架”，代码目录中新增sleep.py并将此脚本作为“启动文件”。这样启动的作业将会持续运行60分钟。您可通过Cloud Shell进入容器进行调试。

  sleep.py示例：

```
import os
os.system('sleep 60m')
```

图 4-13 预置框架启动方式



- 如果训练作业使用的是自定义镜像

  在创建训练作业时，“创建方式”选择“自定义算法”，“启动方式”选择“自定义”，“启动命令”输入“sleep 60m”。这样启动的作业将会持续运行60分钟。您可通过Cloud Shell进入容器进行调试。

  图 4-14 自定义启动方式

出错的任务如何卡在运行中状态

创建训练作业时，启动命令末尾新增"|| sleep 5h"，并启动训练任务，例如下方的cmd为您的启动命令：

```
cmd || sleep 5h
```

如果训练失败，则会执行sleep命令，此时可通过Cloud Shell登录容器镜像中调试。

说明

在Cloud Shell中调试多节点训练作业时，需要在Cloud Shell中切换work0、work1来实现对不同节点下发启动命令，否则任务会处于等待其他节点的状态。

4.5.3 如何防止 Cloud Shell 的 Session 断开

如果需要长时间运行某一个任务，为避免在期间连接断开导致任务失败，可通过使用screen命令使得任务在远程终端窗口运行。

1. 如果镜像中未安装screen，则执行“apt-get install screen”安装。
2. 创建screen终端。

   ```
   # 使用 -S 创建一个叫name的screen终端
   screen -S name
   ```

3. 显示已创建的screen终端。

   ```
   screen -ls
   There are screens on:
   2433.pts-3.linux    (2013年10月20日 16时48分59秒)    (Detached)
   2428.pts-3.linux    (2013年10月20日 16时48分05秒)    (Detached)
   2284.pts-3.linux    (2013年10月20日 16时14分55秒)    (Detached)
   2276.pts-3.linux    (2013年10月20日 16时13分18秒)    (Detached)
   4 Sockets in /var/run/screen/S-root.
   ```

4. 连接“screen_id”为“2276”的screen终端。
```
screen -r 2276
```
5. 按下“Ctrl”+“a”+“d”键离开screen终端。离开后，screen会话仍将是活跃的，之后可以随时重新连接。

更多Screen使用说明可参考**Screen User’s Manual**。

4.5.4 通过 py-spy 工具分析卡死进程的调用栈并结合代码分析定位卡死问题

使用场景

本文指导用户通过py-spy工具分析卡死进程的调用栈并结合代码分析定位卡死问题。

操作步骤

步骤1 在ModelArts控制台，选择“训练管理>训练作业”。

步骤2 在训练作业详情页面，选择Cloud Shell页签，登录训练容器（训练作业需处于运行中）。

步骤3 安装py-spy工具。
```
# 通过utils.sh脚本自动配置python环境
source /home/ma-user/modelarts/run/utils.sh

# 安装py-spy
pip install py-spy

# 如果超时提示connection broken by 'ProxyError('Cannot connect to proxy.')则表示用户设置了proxy，需要先关掉
export no_proxy=$no_proxy,repo.myhuaweicloud.com（此处需要替换成对应局点的pip源地址）
pip install py-spy
```

步骤4 查看堆栈。py-spy工具的具体使用方法可参考**py-spy官方文档**。
```
# 找到训练进程的PID
ps -ef

# 查看进程12345的进程堆栈
# 如果是8卡的训练作业，一般用此命令依次去查看主进程起的对应的8个进程的堆栈情况
py-spy dump --pid 12345
```

----结束

4.6 查看训练作业资源利用率

如何查看训练作业资源使用详情

1. 在ModelArts管理控制台的左侧导航栏中选择“训练管理 > 训练作业”。
2. 在训练作业列表中，单击作业名称进入训练作业详情页面。
3. 在训练作业详情页面，单击“资源占用情况”页签查看计算节点的资源使用情况，最多可显示最近三天的数据。在“资源占用情况”窗口打开时，会定期向后台获取最新的资源使用率数据并刷新。

   操作一：如果训练作业使用多个计算节点，可以通过实例名称的下拉框切换节点。

操作二：单击图例“cpuUsage”、“gpuMemUsage”、“gpuUtil”、“memUsage”“npuMemUsage”、“npuUtil”、可以添加或取消对应参数的使用情况图。

操作三：鼠标悬浮在图片上的时间节点，可查看对应时间节点的占用率情况。

图 4-15 资源占用情况

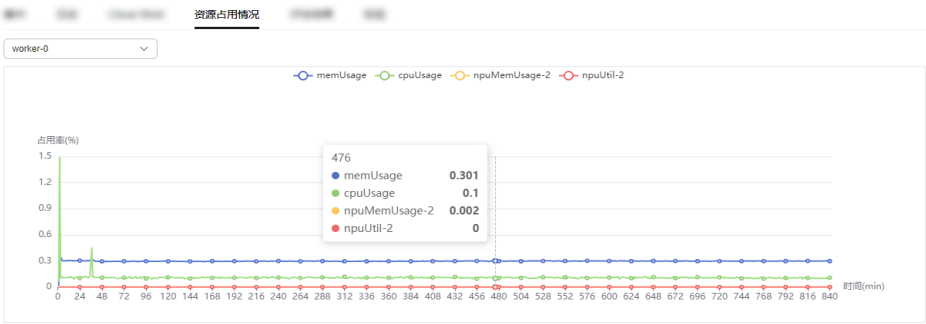


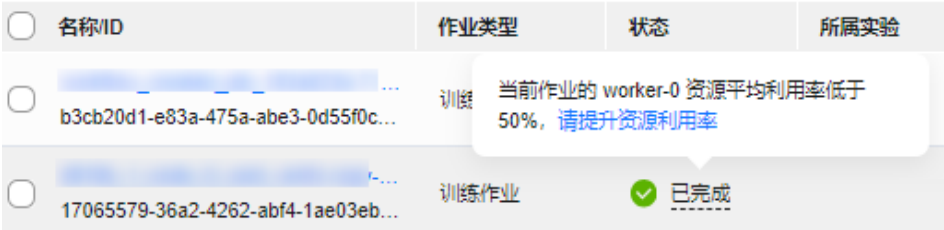
表 4-10 参数说明

参数	说明
cpuUsage	cpu使用率。
gpuMemUsage	gpu内存使用率。
gpuUtil	gpu使用情况。
memUsage	内存使用率。
npuMemUsage	npu内存使用率。
npuUtil	npu使用情况。

如何判断训练作业资源利用率高低

在训练管理的训练作业列表页可以查看作业资源利用率情况。当作业worker-0实例的GPU/NPU的平均利用率低于50%时，在训练作业列表中会进行告警提示。

图 4-16 作业列表显示作业资源利用率情况



此处的作业资源利用率只涉及GPU和NPU资源。作业worker-0实例的GPU/NPU平均利用率计算方法：将作业worker-0实例的各个GPU/NPU加速卡每个时间点的利用率汇总取平均值。

如何提高训练作业资源利用率

- 适当增大batch_size：较大的batch_size可以让GPU/NPU计算单元获得更高的利用率，但是也要根据实际情况来选择batch_size，防止batch_YLLsize过大导致内存溢出。
- 提升数据读取的效率：如果读取一个batch数据的时间要长于GPU/NPU计算一个batch的时间，就有可能出现GPU/NPU利用率上下浮动的情况。建议优化数据读取和数据增强的性能，例如将数据读取并行化，或者使用NVIDIA Data Loading Library（DALI）等工具提高数据增强的速度。
- 模型保存不要太频繁：模型保存操作一般会阻塞训练，如果模型较大，并且较频繁地进行保存，就会影响GPU/NPU利用率。同理，其他非GPU/NPU操作尽量不要阻塞训练主进程太多的时间，如日志打印，保存训练指标信息等。

4.7 评估训练结果

训练作业运行结束后，ModelArts可为您的模型进行评估，并且给出调优诊断和建议。

- 针对使用预置算法创建训练作业，无需任何配置，即可查看此评估结果（由于每个模型情况不同，系统将自动根据您的模型指标情况，给出一些调优建议，请仔细阅读界面中的建议和指导，对您的模型进行进一步的调优）。
- 针对用户自己编写训练脚本或自定义镜像方式创建的训练作业，则需要在您的训练代码中添加评估代码，才可以在训练作业结束后查看相应的评估诊断建议。

说明

- 只支持验证集的数据格式为图片
- 目前，仅如下常用框架的训练脚本支持添加评估代码。
  - TF-1.13.1-python3.6
  - TF-2.1.0-python3.6
  - PyTorch-1.4.0-python3.6

下文将介绍如何在训练中使用评估代码。对训练代码做一定的适配和修正，分为三个方面：**添加输出目录**、**复制数据集到本地**、**映射数据集路径到OBS**。

添加输出目录

添加输出目录的代码比较简单，即在代码中添加一个输出评估结果文件的目录，被称为train_url，也就是页面上的训练输出位置。并把train_url添加到使用的函数analysis中，使用save_path来获取train_url。示例代码如下所示：

```
FLAGS = tf.app.flags.FLAGS
tf.app.flags.DEFINE_string('model_url', '', 'path to saved model')
tf.app.flags.DEFINE_string('data_url', '', 'path to output files')
tf.app.flags.DEFINE_string('train_url', '', 'path to output files')
tf.app.flags.DEFINE_string('adv_param_json',
                           '{"attack_method":"FGSM","eps":40}',
                           'params for adversarial attacks')
FLAGS(sys.argv, known_only=True)

...

# analyse
res = analyse(
    task_type=task_type,
    pred_list=pred_list,
    label_list=label_list,
```

```
    name_list=file_name_list,
    label_map_dict=label_dict,
    save_path=FLAGS.train_url)
```

复制数据集到本地

复制数据集到本地主要是为了防止长时间访问OBS容易导致OBS连接中断使得作业卡住，所以一般先将数据复制到本地再进行操作。

数据集复制有两种方式，推荐使用OBS路径复制。

- OBS路径（推荐）

  直接使用moxing的copy_parallel接口，复制对应的OBS路径。

- ModelArts数据管理中的数据集（即manifest文件格式）

  使用moxing的copy_manifest接口将文件复制到本地并获取新的manifest文件路径，然后使用SDK解析新的manifest文件。

说明

ModelArts数据管理模块在重构升级中，对未使用过数据管理的用户不可见。建议新用户将训练数据存放至OBS桶中使用。

```
if data_path.startswith('obs://'):
    if '.manifest' in data_path:
        new_manifest_path, _ = mox.file.copy_manifest(data_path, '/cache/data/')
        data_path = new_manifest_path
    else:
        mox.file.copy_parallel(data_path, '/cache/data/')
        data_path = '/cache/data/'
    print('------------- download dataset success ------------')
```

映射数据集路径到 OBS

由于最终JSON体中需要填写的是图片文件的真实路径，也就是OBS对应的路径，所以在复制到本地做完分析和评估操作后，需要将原来的本地数据集路径映射到OBS路径，然后将新的list送入analysis接口。

如果使用的是OBS路径作为输入的data_url，则只需要替换本地路径的字符串即可。

```
if FLAGS.data_url.startswith('obs://'):
    for idx, item in enumerate(file_name_list):
        file_name_list[idx] = item.replace(data_path, FLAGS.data_url)
```

如果使用manifest文件，需要再解析一遍原版的manifest文件获取list，然后再送入analysis接口。

```
if or FLAGS.data_url.startswith('obs://'):
    if 'manifest' in FLAGS.data_url:
            file_name_list = []
            manifest, _ = get_sample_list(
                manifest_path=FLAGS.data_url, task_type='image_classification')
            for item in manifest:
                if len(item[1]) != 0:
                    file_name_list.append(item[0])
```

完整的适配了训练作业创建的图像分类样例代码如下：

```
import json
import logging
import os
import sys
import tempfile
```

```
import h5py
import numpy as np
from PIL import Image

import moxing as mox
import tensorflow as tf
from deep_moxing.framework.manifest_api.manifest_api import get_sample_list
from deep_moxing.model_analysis.api import analyse, tmp_save
from deep_moxing.model_analysis.common.constant import TMP_FILE_NAME

logging.basicConfig(level=logging.DEBUG)

FLAGS = tf.app.flags.FLAGS
tf.app.flags.DEFINE_string('model_url', '', 'path to saved model')
tf.app.flags.DEFINE_string('data_url', '', 'path to output files')
tf.app.flags.DEFINE_string('train_url', '', 'path to output files')
tf.app.flags.DEFINE_string('adv_param_json',
                           '{"attack_method":"FGSM","eps":40}',
                           'params for adversarial attacks')
FLAGS(sys.argv, known_only=True)


def _preprocess(data_path):
    img = Image.open(data_path)
    img = img.convert('RGB')
    img = np.asarray(img, dtype=np.float32)
    img = img[np.newaxis, :, :, :]
    return img


def softmax(x):
    x = np.array(x)
    orig_shape = x.shape
    if len(x.shape) > 1:
        # Matrix
        x = np.apply_along_axis(lambda x: np.exp(x - np.max(x)), 1, x)
        denominator = np.apply_along_axis(lambda x: 1.0 / np.sum(x), 1, x)
        if len(denominator.shape) == 1:
            denominator = denominator.reshape((denominator.shape[0], 1))
        x = x * denominator
    else:
        # Vector
        x_max = np.max(x)
        x = x - x_max
        numerator = np.exp(x)
        denominator = 1.0 / np.sum(numerator)
        x = numerator.dot(denominator)
    assert x.shape == orig_shape
    return x


def get_dataset(data_path, label_map_dict):
    label_list = []
    img_name_list = []
    if 'manifest' in data_path:
        manifest, _ = get_sample_list(
            manifest_path=data_path, task_type='image_classification')
        for item in manifest:
            if len(item[1]) != 0:
                label_list.append(label_map_dict.get(item[1][0]))
                img_name_list.append(item[0])
            else:
                continue
    else:
        label_name_list = os.listdir(data_path)
        label_dict = {}
        for idx, item in enumerate(label_name_list):
            label_dict[str(idx)] = item
            sub_img_list = os.listdir(os.path.join(data_path, item))
```

```
        img_name_list += [
            os.path.join(data_path, item, img_name) for img_name in sub_img_list
        ]
        label_list += [label_map_dict.get(item)] * len(sub_img_list)
    return img_name_list, label_list


def deal_ckpt_and_data_with_obs():
    pb_dir = FLAGS.model_url
    data_path = FLAGS.data_url

    if pb_dir.startswith('obs://'):
        mox.file.copy_parallel(pb_dir, '/cache/ckpt/')
        pb_dir = '/cache/ckpt'
        print('------------- download success ------------')
    if data_path.startswith('obs://'):
        if '.manifest' in data_path:
            new_manifest_path, _ = mox.file.copy_manifest(data_path, '/cache/data/')
            data_path = new_manifest_path
        else:
            mox.file.copy_parallel(data_path, '/cache/data/')
            data_path = '/cache/data/'
        print('------------- download dataset success ------------')
    assert os.path.isdir(pb_dir), 'Error, pb_dir must be a directory'
    return pb_dir, data_path


def evalution():
    pb_dir, data_path = deal_ckpt_and_data_with_obs()
    index_file = os.path.join(pb_dir, 'index')
    try:
        label_file = h5py.File(index_file, 'r')
        label_array = label_file['labels_list'][:].tolist()
        label_array = [item.decode('utf-8') for item in label_array]
    except Exception as e:
        logging.warning(e)
        logging.warning('index file is not a h5 file, try json.')
        with open(index_file, 'r') as load_f:
            label_file = json.load(load_f)
        label_array = label_file['labels_list'][:]
    label_map_dict = {}
    label_dict = {}
    for idx, item in enumerate(label_array):
        label_map_dict[item] = idx
        label_dict[idx] = item
    print(label_map_dict)
    print(label_dict)

    data_file_list, label_list = get_dataset(data_path, label_map_dict)

    assert len(label_list) > 0, 'missing valid data'
    assert None not in label_list, 'dataset and model not match'

    pred_list = []
    file_name_list = []
    img_list = []

    for img_path in data_file_list:
        img = _preprocess(img_path)
        img_list.append(img)
        file_name_list.append(img_path)

    config = tf.ConfigProto()
    config.gpu_options.allow_growth = True
    config.gpu_options.visible_device_list = '0'
    with tf.Session(graph=tf.Graph(), config=config) as sess:
        meta_graph_def = tf.saved_model.loader.load(
            sess, [tf.saved_model.tag_constants.SERVING], pb_dir)
        signature = meta_graph_def.signature_def
```

```
        signature_key = 'predict_object'
        input_key = 'images'
        output_key = 'logits'
        x_tensor_name = signature[signature_key].inputs[input_key].name
        y_tensor_name = signature[signature_key].outputs[output_key].name
        x = sess.graph.get_tensor_by_name(x_tensor_name)
        y = sess.graph.get_tensor_by_name(y_tensor_name)
        for img in img_list:
            pred_output = sess.run([y], {x: img})
            pred_output = softmax(pred_output[0])
            pred_list.append(pred_output[0].tolist())

    label_dict = json.dumps(label_dict)
    task_type = 'image_classification'

    if FLAGS.data_url.startswith('obs://'):
        if 'manifest' in FLAGS.data_url:
            file_name_list = []
            manifest, _ = get_sample_list(
                manifest_path=FLAGS.data_url, task_type='image_classification')
            for item in manifest:
                if len(item[1]) != 0:
                    file_name_list.append(item[0])
        for idx, item in enumerate(file_name_list):
            file_name_list[idx] = item.replace(data_path, FLAGS.data_url)
    # analyse
    res = analyse(
        task_type=task_type,
        pred_list=pred_list,
        label_list=label_list,
        name_list=file_name_list,
        label_map_dict=label_dict,
        save_path=FLAGS.train_url)

if __name__ == "__main__":
    evalution()
```

4.8 查看训练标签

通过给训练作业添加标签，可以标识云资源，便于快速搜索训练作业。

1. 在ModelArts管理控制台的左侧导航栏中选择“训练管理 > 训练作业”。
2. 在训练作业列表中，单击作业名称进入训练作业详情页面。
3. 在训练作业详情页面，单击“标签”页签查看标签信息。

   支持添加、修改、删除标签。标签详细用法请参见**ModelArts如何通过标签实现资源分组管理**。

图 4-17 查看训练标签

说明

最多支持添加20个标签。

4.9 查看故障恢复详情

当训练作业发生故障恢复时（例如进程级恢复、POD级重调度、JOB级重调度等），作业详情页面中会出现“故障恢复详情”页签，里面记录了训练作业的启停情况。

1. 在ModelArts管理控制台的左侧导航栏中选择“训练管理 > 训练作业”。
2. 在训练作业列表中，单击作业名称进入训练作业详情页面。
3. 在训练作业详情页面，单击“故障恢复详情”页签查看故障恢复信息。

图 4-18 查看故障恢复详情

4.10 查看训练容器环境变量

什么是环境变量

本章节展示了训练容器环境中预置的环境变量，方便用户查看，主要包括以下类型。

- 路径相关环境变量
- 分布式训练任务环境变量
- NCCL（Nvidia Collective multi-GPU Communication Library）环境变量
- OBS环境变量
- PIP源环境变量
- API网关地址环境变量
- 作业元信息环境变量

如何修改环境变量

用户可以在创建训练作业页面增加新的环境变量，也可以设置新的取值覆盖当前训练容器中预置的环境变量值。

训练容器中预置的环境变量

训练容器中预置的环境变量如下面表格所示，包括**表4-11**、**表4-12**、**表4-13**、**表4-14**、**表4-15**、**表4-16**、**表4-17**。

此处的环境变量取值仅为示例，涉及不同规格、引擎、Region可能取值不一样，此处仅供参考。

表 4-11 路径相关环境变量

变量名	说明	示例
PATH	可执行文件路径，已包含常用的可执行文件路径。	“PATH=/usr/local/nvidia/bin:/usr/local/cuda/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin”
LD_LIBRARY_PATH	动态链接库路径，已包含常用的动态库路径。	“LD_LIBRARY_PATH=/usr/local/seccomponent/lib:/usr/local/cuda/lib64:/usr/local/cuda/compat:/root/miniconda3/lib:/usr/local/nvidia/lib:/usr/local/nvidia/lib64”
LIBRARY_PATH	静态库路径，已包含常用的静态库路径。	“LIBRARY_PATH=/usr/local/cuda/lib64/stubs”
MA_HOME	训练作业的主目录。	“MA_HOME=/home/ma-user”
MA_JOB_DIR	训练算法文件夹所在的父目录。	“MA_JOB_DIR=/home/ma-user/modelarts/user-job-dir”
MA_MOUNT_PATH	ModelArts挂载至训练容器内的路径，用于临时存放训练算法、算法输入、算法输出、日志等文件。	“MA_MOUNT_PATH=/home/ma-user/modelarts”
MA_LOG_DIR	训练日志目录。	“MA_LOG_DIR=/home/ma-user/modelarts/log”
MA_SCRIPT_INTERPRETER	训练脚本解释器。	“MA_SCRIPT_INTERPRETER=”
WORKSPACE	训练算法目录。	“WORKSPACE=/home/ma-user/modelarts/user-job-dir/code”

表 4-12 分布式训练任务环境变量

变量名	说明	示例
MA_CURRENT_IP	作业容器IP。	“MA_CURRENT_IP=192.168.23.38”
MA_NUM_GPUS	作业容器的加速卡数量。	“MA_NUM_GPUS=8”

变量名	说明	示例
MA_TASK_NAME	作业容器的角色名，例如： ● MindSpore、PyTorch为worker ● 强化学习引擎为learner，worker ● TensorFlow为ps，worker	“MA_TASK_NAME=worker”
MA_NUM_HOSTS	训练作业使用的节点数量。	“MA_NUM_HOSTS=4”
VC_TASK_INDEX	多节点训练时，当前作业容器的序列号，第一个容器的值为0。	“VC_TASK_INDEX=0”
VC_WORKER_NUM	训练作业使用的节点数量。	“VC_WORKER_NUM=4”
VC_WORKER_HOSTS	多节点训练时，每个节点的域名地址，按顺序以英文逗号分隔，可以通过域名解析获取IP地址。	“VC_WORKER_HOSTS=modelarts-job-a0978141-1712-4f9b-8a83-00000000000000-worker-0.modelarts-job-a0978141-1712-4f9b-8a83-00000000000000,modelarts-job-a0978141-1712-4f9b-8a83-00000000000000-worker-1.ob-a0978141-1712-4f9b-8a83-00000000000000,modelarts-job-a0978141-1712-4f9b-8a83-00000000000000-worker-2.modelarts-job-a0978141-1712-4f9b-8a83-00000000000000,ob-a0978141-1712-4f9b-8a83-00000000000000-worker-3.modelarts-job-a0978141-1712-4f9b-8a83-00000000000000”

表 4-13 NCCL 环境变量

变量名	说明	示例
NCCL_VERSION	NCCL版本。	“NCCL_VERSION=2.7.8”
NCCL_DEBUG	NCCL日志等级。	“NCCL_DEBUG=INFO”
NCCL_IB_HCA	指定NCCL使用的IB网卡。	“NCCL_IB_HCA=^mlx5_bond_0”

变量名	说明	示例
NCCL_SOCKET_IFNAME	指定NCCL使用的SOCKET网卡。	“NCCL_SOCKET_IFNAME=bond0,eth0”

表 4-14 OBS 环境变量

变量名	说明	示例
S3_ENDPOINT	OBS地址。	“-”
S3_VERIFY_SSL	访问OBS是否使用SSL。	“S3_VERIFY_SSL=0”
S3_USE_HTTPS	访问OBS是否使用HTTPS。	“S3_USE_HTTPS=1”

表 4-15 PIP 源和 API 网关地址环境变量

变量名	说明	示例
MA_PIP_HOST	PIP源域名。	“MA_PIP_HOST=repo.myhuaweicloud.com”
MA_PIP_URL	PIP源地址。	“MA_PIP_URL=http://repo.myhuaweicloud.com/repository/pypi/simple/”
MA_APIGW_ENDPOINT	ModelArts API网关地址。	“MA_APIGW_ENDPOINT=https://modelarts.region.cn-east-3.myhuaweicloud.com”

表 4-16 作业元信息环境变量

变量名	说明	示例
MA_CURRENT_INSTANCE_NAME	多节点训练时，当前节点的名称。	“MA_CURRENT_INSTANCE_NAME=modelarts-job-a0978141-1712-4f9b-8a83-000000000000-worker-1”

表 4-17 预检相关环境变量

变量名	说明	示例
MA_SKIP_IMAGE_DETECT	ModelArts预检是否开启。默认为1，1表示开启预检，0表示关闭预检。 推荐开启预检，预检可提前发现节点故障、驱动故障。	“1”

4.11 停止、重建或查找作业

另存为算法

当您需要修改训练作业的算法时，可以在训练作业详情页面右上角，单击“另存为算法”。

在“创建算法”页面中，会自动填充上一次训练作业的算法参数配置，您可以根据业务需求在原来算法配置基础上进行修改。

说明

在AI Gallery中订阅的算法不支持另存为算法。

停止训练作业

在训练作业列表中，针对“创建中”、“等待中”、“运行中”的训练作业，您可以单击“操作”列的“终止”，停止正在运行中的训练作业。

训练作业停止后，ModelArts将停止计费。

运行结束的训练作业，如“已完成”、“运行失败”、“已终止”、“异常”的作业，不涉及“终止”操作。

重建训练作业

当对创建的训练作业不满意时，您可以单击操作列的重建，重新创建训练作业。在重创训练作业页面，会自动填入上一次训练作业设置的参数，您仅需在原来的基础上进行修改即可重新创建训练作业。

查找训练作业

当用户使用IAM账号登录时，训练作业列表会显示IAM账号下所有训练作业。ModelArts提供查找训练作业功能帮助用户快速查找训练作业。

操作一：单击“只显示自己”按钮，训练作业列表仅显示当前子账号下创建的训练作业。

操作二：按照名称、ID、作业类型、状态、创建时间、算法、资源池等条件筛选的高级搜索。

操作三：单击作业列表右上角“刷新”图标，刷新作业列表。

操作四：自定义列功能设置。

图 4-19 查找训练作业

4.12 清除训练作业资源

如果不再需要使用此训练任务，建议清除相关资源，避免产生不必要的费用。

- 在“训练作业”页面，“删除”运行结束的训练作业。您可以单击“操作”列的“删除”，在弹出的提示框中单击“确认”，删除对应的训练作业。
- 进入OBS，删除本示例使用的OBS桶及文件。

完成资源清除后，您可以在总览页面的使用情况确认资源删除情况。

图 4-20 查看使用情况

5 训练进阶

5.1 训练故障自动恢复

5.1.1 训练容错检查

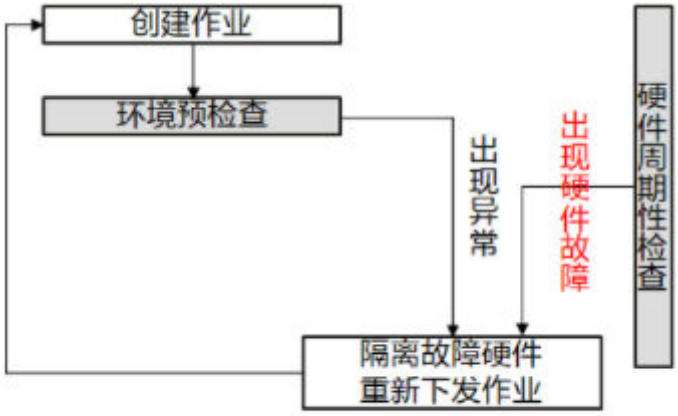
用户在训练模型过程中，存在因硬件故障而产生的训练失败场景。针对硬件故障场景，ModelArts提供容错检查功能，帮助用户隔离故障节点，优化用户训练体验。

容错检查包括两个检查项：环境预检测与硬件周期性检查。当环境预检查或者硬件周期性检查任一检查项出现故障时，隔离故障硬件并重新下发训练作业。针对于分布式场景，容错检查会检查本次训练作业的全部计算节点。

下图中有四个场景，其中场景四为正常训练作业失败场景，其他三个场景下可开启容错功能进行训练作业自动恢复。

- 场景一：环境预检测失败、硬件检测出现故障，系统隔离所有故障节点并重新下发训练作业。

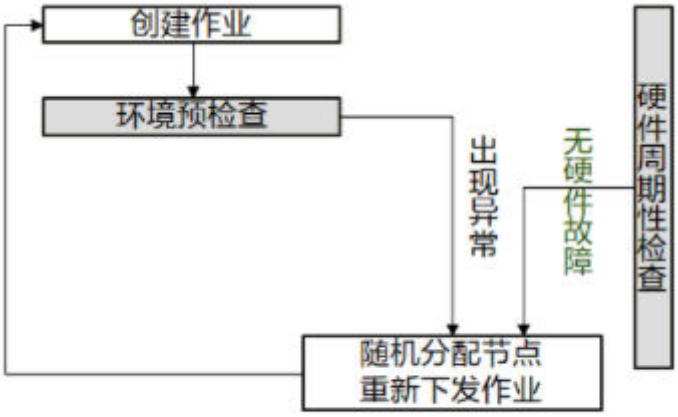
图 5-1 预检失败&硬件故障



1. 预检失败&硬件故障

- 场景二：环境预检测失败、硬件无故障，系统随机再分配节点并重新下发训练作业。

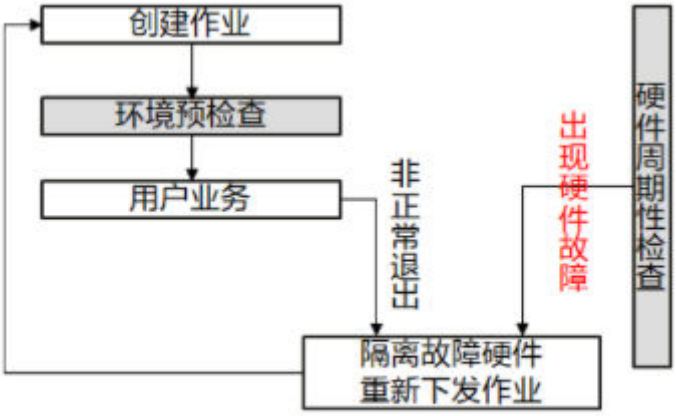
图 5-2 预检失败&硬件正常



2. 预检失败&硬件正常

- 场景三：环境预检测成功并进入用户业务阶段，硬件检测出现故障并且用户业务非正常退出，系统隔离所有故障节点并重新下发训练作业。

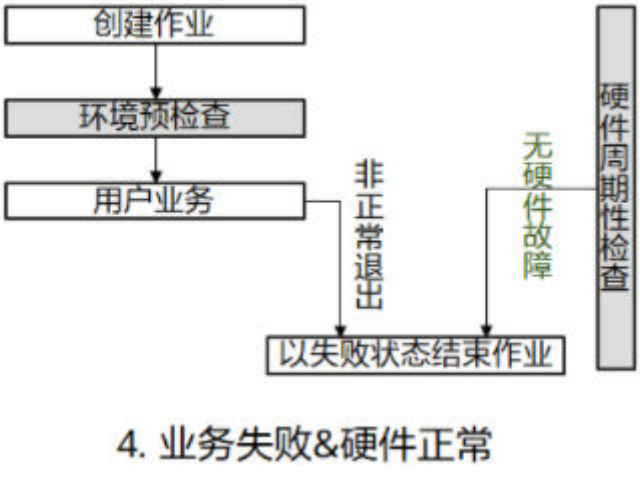
图 5-3 业务失败&硬件故障



3. 业务失败&硬件故障

- 场景四：环境预检测成功并进入用户业务阶段，硬件无故障，当用户业务异常时系统以失败状态结束作业。

图 5-4 业务失败&硬件正常



隔离故障节点后，系统会在新的计算节点上重新创建训练作业。如果资源池规格紧张，重新下发的训练作业会以第一优先级进行排队。如果排队时间超过30分钟，训练任务会自动退出。该现象表明资源池规格任务紧张，训练作业无法正常启动，推荐您购买专属资源池补充计算节点。

如果您使用专属资源池创建训练作业，容错检查识别的故障节点会被剔除。系统自动补充健康的计算节点至专属资源池。（该功能即将上线）

容错检查详细介绍请参考：

1. 开启容错检查
2. 检测项目与执行条件
3. 触发容错环境检测达到的效果
4. 环境预检查通过后，如果发生硬件故障会导致用户业务中断。您可以在训练中补充reload ckpt的代码逻辑，使能读取训练中断前保存的预训练模型。指导请参考断点续训练和增量训练。

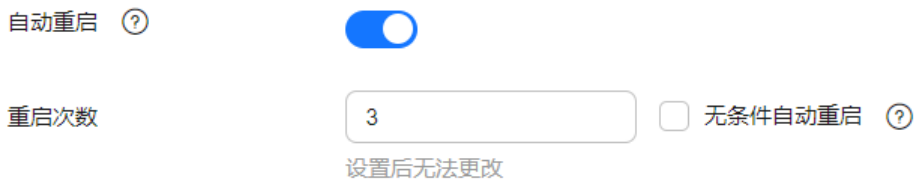
开启容错检查

用户可以在创建训练作业时通过设置自动重启的方式开启容错检查。

- 使用ModelArts控制台的创建训练作业页面设置自动重启：

  用户可以在控制台页面通过开关的方式开启自动重启。“自动重启”开关默认不开启，表示不做重新下发作业，也不会启用环境检测。打开开关后，允许设置重启次数为1~128次。

  图 5-5 自动重启设置



- 使用API接口设置容错检查：

  用户可以通过API接口的方式开启自动重启。创建训练作业时，在“metadata”字段的“annotations”中传入“fault-tolerance/job-retry-num”字段。

  添加“fault-tolerance/job-retry-num”字段，视为开启自动重启，value的范围可以设置为1~128的整数。value值表示最大允许重新下发作业的次数。如果不传入则默认为0，表示不做重新下发作业，也不会启用环境检测。

  图 5-6 设置 API

```
{
    "kind": "job",
    "metadata": {
        "annotations": {
            "fault-tolerance/job-retry-num": "3"
        },
```

检测项目与执行条件

检测项目	item（日志关键字）	执行条件	检测成功要求
域名检测	dns	无	volcano容器的域名都解析成功（/etc/volcano下的“.host”文件中的域名解析成功）
磁盘空间-容器根目录	disk-size root	无	大于32GB
磁盘空间-/dev/shm目录	disk-size shm	无	大于1GB
磁盘空间-/cache目录	disk-size cache	无	大于32GB
ulimit检查	ulimit	使用IB网络时	• max locked memory > 16000 • open files > 1000000 • stack size > 8000 • max user processes > 1000000
gpu检查	gpu-check	使用gpu，且使用v2训练引擎时	检测到gpu

触发容错环境检测达到的效果

- 容错检查正常通过时，会打印检测项目的日志，表示具体涉及的检查项目成功。您可以通过在日志中搜索“item”关键字查看。当容错检查正常通过时，可以减少运行故障上报问题。

```
[Modelarts Service Log][task] Detect
[Modelarts Service Log][INFO][detect] code: 0, message: ok, item: dns
[Modelarts Service Log][INFO][detect] code: 0, message: ok, item: disk-size root
[Modelarts Service Log][INFO][detect] code: 0, message: ok, item: disk-size shm
[Modelarts Service Log][INFO][detect] code: 0, message: ok, item: disk-size cache
[Modelarts Service Log][init] download code_url: s3://test-qianjiajun/tolerance_test/
```

- 容错检查失败时，会打印检查失败的日志。您可以通过在日志中搜索“item”关键字查看失败信息。

```
[Modelarts Service Log][init] running
[Modelarts Service Log][init] ip of the pod: 172.16.0.160
[Modelarts Service Log][INFO]detect item: dns; json:{"code": 0, "message": "ok"}
[Modelarts Service Log][INFO][task][detect] code: 0, message: ok, item: dns
[Modelarts Service Log][INFO]detect item: disk-size root; json:{"code": 13, "message": "the disk space of the path
\"/\" is 4892852224, which is less than 34359738368"}
[Modelarts Service Log][ERROR][task][detect] code: 13, message: the disk space of the path "/" is 4892852224, which is
less than 34359738368, item: disk-size root
[Modelarts Service Log][init] exiting...
[Modelarts Service Log][init] wait python processes exit...
```

如果作业重启次数没有达到设定的次数，则会自动做重新下发作业。您可以通过搜索“error,exiting”关键字查找作业重启失败结束的日志。

使用 reload ckpt 恢复中断的训练

在容错机制下，如果因为硬件问题导致训练作业重启，用户可以在代码中读取预训练模型，恢复至重启前的训练状态。用户需要在代码里加上reload ckpt的代码，使能读取训练中断前保存的预训练模型。具体请参见断点续训练和增量训练。

5.1.2 无条件自动重启

背景信息

训练过程中可能会碰到预期外的情况导致训练失败，且无法及时重启训练作业，导致训练周期长，而无条件自动重启可以避免这类问题。无条件自动重启是指当训练作业失败时，不管什么原因系统都会自动重启训练作业，提高训练成功率和提升作业的稳定性。为了避免无效重启浪费算力资源，系统最多只支持连续无条件重启3次。

为了避免丢失训练进度、浪费算力，开启此功能前请确认代码已适配断点续训，操作指导请参见断点续训练和增量训练。

当训练过程中触发了自动重启，则系统会记录重启信息，在训练作业详情页可以查看故障恢复详情，具体请参见查看故障恢复详情。

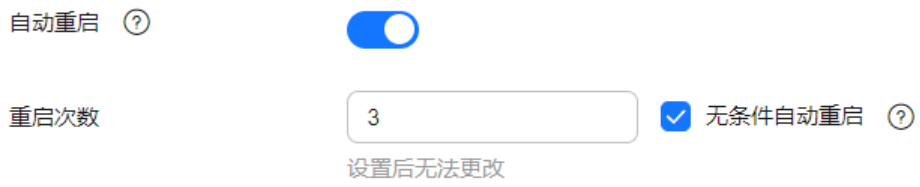
开启无条件自动重启

开启无条件自动重启有2种方式：控制台设置或API接口设置。

- 控制台设置

  在创建训练作业页面，开启“自动重启”开关，并勾选“无条件自动重启”，开启无条件自动重启。开启无条件自动重启后，只要系统检测到训练异常，就无条件重启训练作业。如果未勾选“无条件自动重启”只是打开了“自动重启”开关，则表示仅环境问题导致训练作业异常时才会自动重启，其他问题导致训练作业异常时会直接返回“运行失败”。

图 5-7 开启无条件重启



- API接口设置

  通过API接口创建训练作业时，在“metadata”字段的“annotations”中传入“fault-tolerance/job-retry-num”和“fault-tolerance/job-unconditional-retry”字段。“fault-tolerance/job-retry-num”赋值为1~128表示开启自动重启，“fault-tolerance/job-unconditional-retry”赋值为“true”表示启用了无条件自动重启。

```
{
    "kind": "job",
    "metadata": {
        "annotations": {
            "fault-tolerance/job-retry-num": "8",
            "fault-tolerance/job-unconditional-retry": "true"
        }
    }
}
```

5.2 断点续训练和增量训练

什么是断点续训练和增量训练

断点续训练是指因为某些原因（例如容错重启、资源抢占、作业卡死等）导致训练作业还未完成就被中断，下一次训练可以在上一次的训练基础上继续进行。这种方式对于需要长时间训练的模型而言比较友好。

增量训练是指增加新的训练数据到当前训练流程中，扩展当前模型的知识和能力。

断点续训练和增量训练均是通过checkpoint机制实现。

checkpoint的机制是：在模型训练的过程中，不断地保存训练结果（包括但不限于EPOCH、模型权重、优化器状态、调度器状态）。即便模型训练中断，也可以基于checkpoint接续训练。

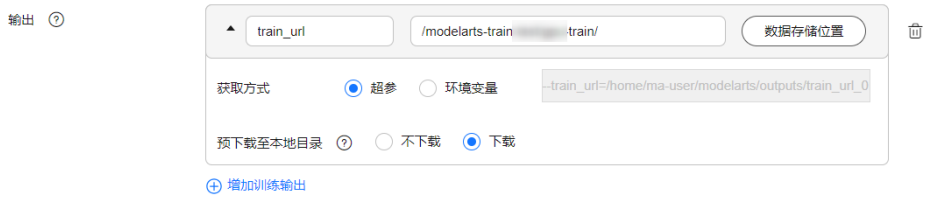
当需要从训练中断的位置接续训练，只需要加载checkpoint，并用checkpoint信息初始化训练状态即可。用户需要在代码里加上reload ckpt的代码，使能读取前一次训练保存的预训练模型。

ModelArts 中如何实现断点续训练和增量训练

在ModelArts训练中实现断点续训练或增量训练，建议使用“训练输出”功能。

在创建训练作业时，设置训练“输出”参数为“train_url”，在指定的训练输出的数据存储位置中保存**checkpoint**，且**“预下载至本地目录”**选择“下载”。选择预下载至本地目录时，系统在训练作业启动前，自动将数据存储位置中的**checkpoint**文件下载到训练容器的本地目录。

图 5-8 训练输出设置



断点续训练建议和训练容错检查（即自动重启）功能同时使用。在创建训练作业页面，开启“自动重启”开关。训练环境预检测失败、或者训练容器硬件检测故障、或者训练作业失败时会自动重新下发并运行训练作业。

PyTorch 版 reload ckpt

1. PyTorch模型保存有两种方式。
   - 仅保存模型参数
     ```
     state_dict = model.state_dict()
     torch.save(state_dict, path)
     ```
   - 保存整个Model（不推荐）
     ```
     torch.save(model, path)
     ```
2. 可根据step步数、时间等周期性保存模型的训练过程的产物。

   将模型训练过程中的网络权重、优化器权重、以及epoch进行保存，便于中断后继续训练恢复。
   ```
   checkpoint = {
           "net": model.state_dict(),
           "optimizer": optimizer.state_dict(),
           "epoch": epoch
   }
   if not os.path.isdir('model_save_dir'):
       os.makedirs('model_save_dir')
   torch.save(checkpoint,'model_save_dir/ckpt_{}.pth'.format(str(epoch)))
   ```
3. 完整代码示例。
   ```
   import os
   import argparse
   parser.add_argument("--train_url", type=str)
   args = parser.parse_known_args()
   # train_url 将被赋值为"/home/ma-user/modelarts/outputs/train_url_0"
   train_url = args.train_url

   # 判断输出路径中是否有模型文件。如果无文件则默认从头训练，如果有模型文件，则加载epoch值最大的ckpt文件当做预训练模型。
   if os.listdir(train_url):
       print('> load last ckpt and continue training!!')
       last_ckpt = sorted([file for file in os.listdir(train_url) if file.endswith(".pth")])[-1]
       local_ckpt_file = os.path.join(train_url, last_ckpt)
       print('last_ckpt:', last_ckpt)
       # 加载断点
       checkpoint = torch.load(local_ckpt_file)
       # 加载模型可学习参数
       model.load_state_dict(checkpoint['net'])
       # 加载优化器参数
       optimizer.load_state_dict(checkpoint['optimizer'])
       # 获取保存的epoch，模型会在此epoch的基础上继续训练
       start_epoch = checkpoint['epoch']
   start = datetime.now()
   total_step = len(train_loader)
   for epoch in range(start_epoch + 1, args.epochs):
       for i, (images, labels) in enumerate(train_loader):
           images = images.cuda(non_blocking=True)
           labels = labels.cuda(non_blocking=True)
   ```

```
    # Forward pass
    outputs = model(images)
    loss = criterion(outputs, labels)
    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    ...

# 保存模型训练过程中的网络权重、优化器权重、以及epoch
checkpoint = {
      "net": model.state_dict(),
      "optimizer": optimizer.state_dict(),
      "epoch": epoch
    }
if not os.path.isdir(train_url):
    os.makedirs(train_url)
    torch.save(checkpoint, os.path.join(train_url, 'ckpt_best_{}.pth'.format(epoch)))
```

5.3 训练作业卡死检测

什么是训练作业卡死检测

训练作业在运行中可能会因为某些未知原因导致作业卡死，如果不能及时发现，就会导致无法及时释放资源，从而造成极大的资源浪费。为了节省训练资源成本，提高使用体验，ModelArts提供了卡死检测功能，能自动识别作业是否卡死，并在日志详情界面上展示，同时能配置通知及时提醒用户作业卡死。

检测规则

卡死检测主要是通过监控作业进程的状态和资源利用率来判定作业是否卡死。会启动一个进程来周期性地监控上述两个指标的变化情况。

- 进程状态：只要训练作业中存在进程IO有变化，进入下一个检测周期。如果在多个检测周期内，作业所有进程IO都没有变化，则进入资源利用率检测阶段。
- 资源利用率：在作业进程IO没有变化的情况下，采集一定时间段内的GPU利用率，并根据这段时间内的GPU利用率的方差和中位数来判断资源使用率是否有变化。如果没有变化，则判定作业卡死。

约束限制

目前卡死检测仅支持资源类型为GPU的训练作业。

操作步骤

卡死检测无需额外配置，作业运行中会自动执行检测。检测到作业卡死后会在训练作业详情页提示作业疑似卡死。如需检测到卡死后发送通知（短信、邮件等）请在作业创建页面配置事件通知。

常见案例

训练作业卡死常见现象的案例和解决方案如下：

复制数据卡死

训练前卡死

训练中途卡死

训练最后一个epoch卡死

5.4 修改训练作业优先级

使用专属资源池（New）进行训练作业时，支持在创建训练作业时设置任务优先级，也支持作业在长时间处于“等待中”的状态时调整优先级。如通过调整作业优先级可以减少作业的排队时长。

什么是训练作业优先级

在用户运行训练作业过程中，有需要对训练任务（也叫训练作业）做优先级划分。比如有一些任务是低优先级，可能是跑一些测试、也可能是跑一些简单的不重要的实验。在这类场景下，当有高优先级任务的时候，需要能比低优先级任务更快进入排队队列。

在资源使用高峰期，用户可以通过提供或降低训练作业的优先级，来动态调节作业的执行顺序，保障关键业务的及时运行。

约束限制

- 仅使用新版专属资源池训练时才支持设置训练作业优先级。公共资源池和旧版专属资源池均不支持设置训练作业优先级。
- 作业优先级取值为1~3，默认优先级为1，最高优先级为3。默认用户权限可选择优先级1和2，配置了“设置作业为高优先级权限”的用户可选择优先级1~3。

如何设置训练作业优先级

在创建训练作业页面可以设置训练的“作业优先级”。取值为1~3，默认优先级为1，最高优先级为3。

如何修改训练作业优先级

在训练作业列表页面，选择“状态”为“等待中”的训练作业，单击“作业优先级”列的，在弹窗中修改优先级后单击“确定”。

图 5-9 修改作业优先级

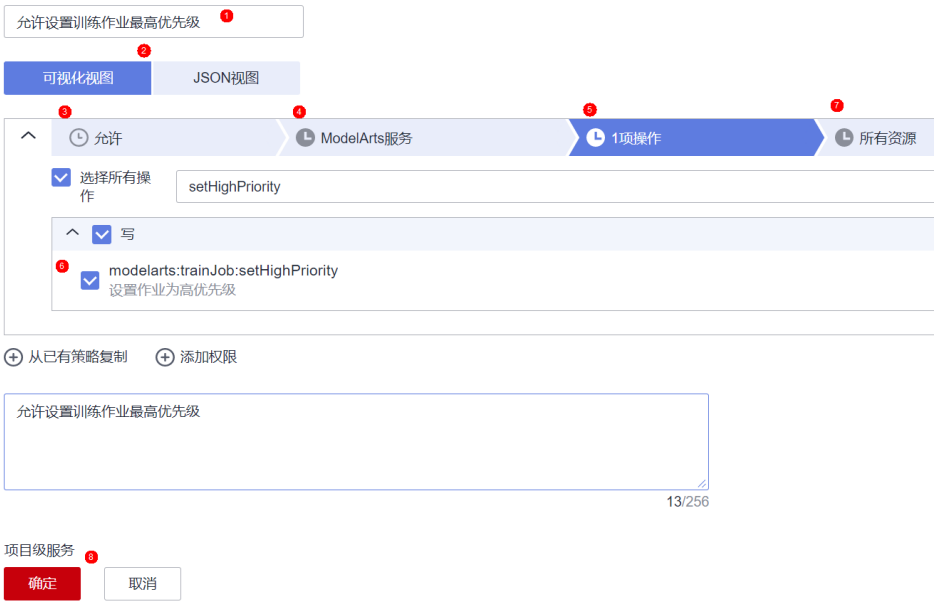
5.5 设置作业为高优先级权限

使用专属资源池（New）进行训练的作业，支持在创建训练作业时设置任务优先级，支持在作业排队过程中调整优先级。优先级取值为1~3，默认优先级为1，最高优先级为3。默认用户权限可选择优先级1和2，配置了"设置作业为高优先级"权限的用户可选择优先级1~3。

给子账号配置"设置作业为高优先级"权限

1. 使用主用户账号登录华为云的管理控制台，单击右上角用户名，在下拉框中选择“统一身份认证”，进入统一身份认证（IAM）服务。
2. 在统一身份认证服务页面的左侧导航选择“权限管理 > 权限”，单击右上角的“创建自定义策略”按如下要求设置完成后单击“确定”。
   - “策略名称”：设置自定义策略名称，例如：允许用户设置训练作业最高优先级。
   - “策略配置方式”：选择可视化视图。
   - “策略内容”：允许，云服务中搜索ModelArts服务并选中，操作列中搜索关键词“modelarts:trainJob:setHighPriority”并选中，所有资源选择默认值。

   图 5-10 创建自定义策略

3. 在统一身份认证服务页面的左侧导航选择“用户组”，在用户组页面查找待授权的用户组名称，在右侧的操作列单击“授权”，勾选步骤2创建的自定义策略，单击“下一步”，选择授权范围方案，单击“确定”。

   此时，该用户组下的所有用户均有权限通过Cloud Shell登录运行中的训练作业容器。

   如果没有用户组，也可以创建一个新的用户组，并通过“用户组管理”功能添加用户，并配置授权。如果指定的子用户没有在用户组中，也可以通过“用户组管理”功能增加用户。

6 分布式训练

6.1 分布式训练功能介绍

ModelArts提供了如下能力：

- 丰富的官方预置镜像，满足用户的需求。
- 支持基于预置镜像自定义制作专属开发环境，并保存使用。
- 丰富的教程，帮助用户快速适配分布式训练，使用分布式训练极大减少训练时间。
- 分布式训练调测的能力，可在PyCharm/VSCode/JupyterLab等开发工具中调试分布式训练。

约束限制

- 如果切换了Notebook的规格，那么只能在Notebook进行单机调测，不能进行分布式调测，也不能提交远程训练任务。
- 当前仅支持PyTorch和MindSpore AI框架，如果MindSpore要进行多机分布式训练调试，则每台机器上都必须有8张卡。
- 本文档提供的调测代码中涉及到的OBS路径，请用户替换为自己的实际OBS路径。
- 本文档提供的调测代码是以PyTorch为例编写的，不同的AI框架之间，整体流程是完全相同的，只需要修改个别的参数即可。

相关章节

- **单机多卡数据并行-DataParallel(DP)**：介绍单机多卡数据并行分布式训练原理和代码改造点。
- **多机多卡数据并行-DistributedDataParallel(DDP)**：介绍多机多卡数据并行分布式训练原理和代码改造点。
- **分布式调测适配及代码示例**：提供了分布式训练调测具体的代码适配操作过程和代码示例。
- **分布式训练完整代码示例**：针对Resnet18在cifar10数据集上的分类任务，给出了分布式训练改造(DDP)的完整代码示例，供用户学习参考。

- **基于开发环境使用SDK调测训练作业**：介绍如何在ModelArts的开发环境中，使用SDK调测单机和多机分布式训练作业。

6.2 单机多卡数据并行-DataParallel(DP)

本章节介绍基于PyTorch引擎的单机多卡数据并行训练。

MindSpore引擎的分布式训练参见**MindSpore官网**。

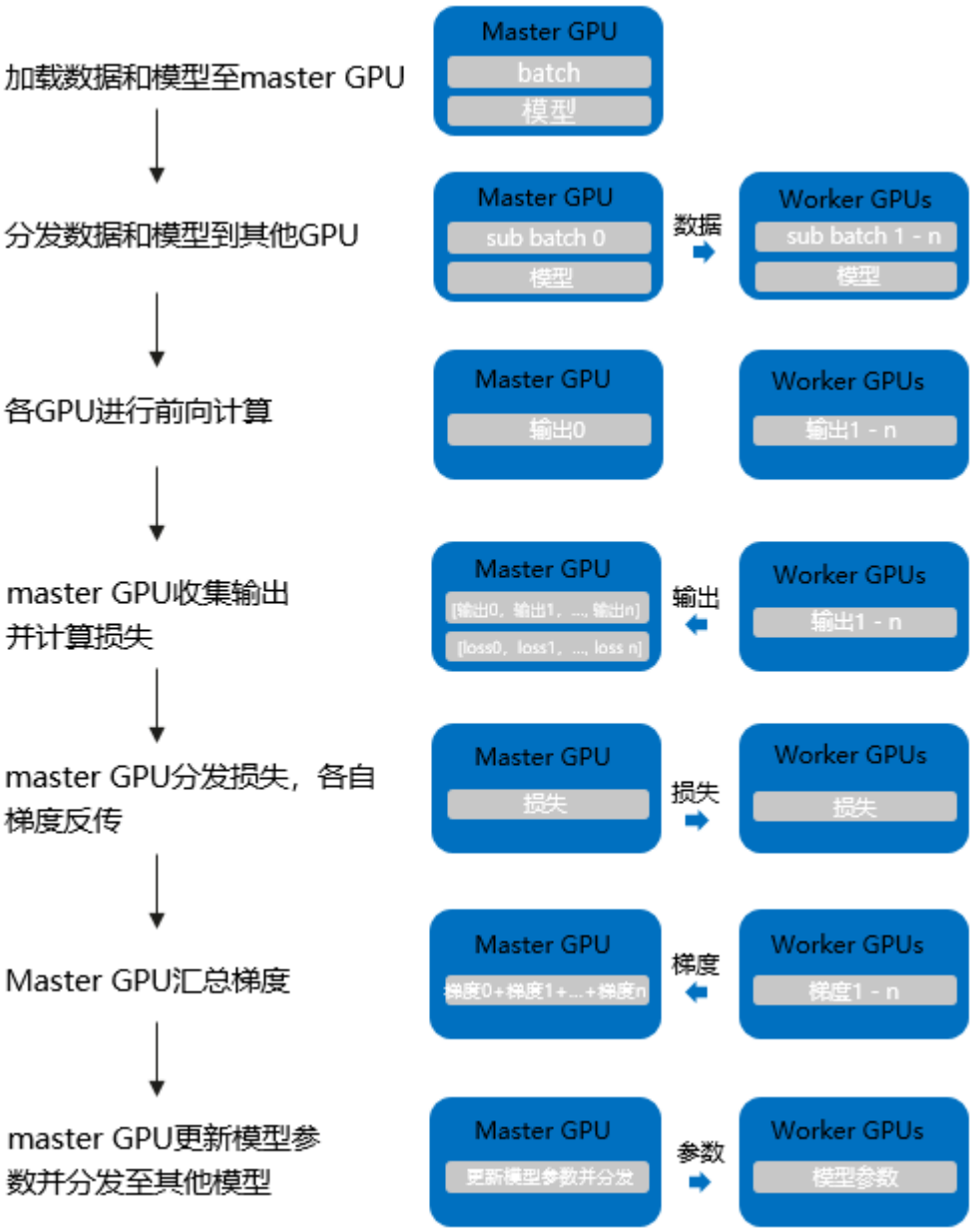
训练流程简述

单机多卡数据并行训练流程介绍如下：

1. 将模型复制到多个GPU上
2. 将一个Batch的数据均分到每一个GPU上
3. 各GPU上的模型进行前向传播，得到输出
4. 主GPU（逻辑序号为0）收集各GPU的输出，汇总后计算损失
5. 分发损失，各GPU各自反向传播梯度
6. 主GPU收集梯度并更新参数，将更新后的模型参数分发到各GPU

具体流程图如下：

图 6-1 单机多卡数据并行训练



DataParallel 进行单机多卡训练的优缺点

- 代码简单：仅需修改一行代码。
- 通信瓶颈 ： 负责reducer的GPU更新模型参数后分发到不同的GPU，因此有较大的通信开销。
- GPU负载不均衡：负责reducer的GPU需要负责汇总输出、计算损失和更新权重，因此显存和使用率相比其他GPU都会更高。

代码改造点

模型分发：DataParallel(model)

完整代码由于代码变动较少，此处进行简略介绍。

```
import torch
class Net(torch.nn.Module):
    pass

model = Net().cuda()

### DataParallel Begin ###
model = torch.nn.DataParallel(Net().cuda())
### DataParallel End ###
```

6.3 多机多卡数据并行-DistributedDataParallel(DDP)

本章节介绍基于PyTorch引擎的多机多卡数据并行训练。

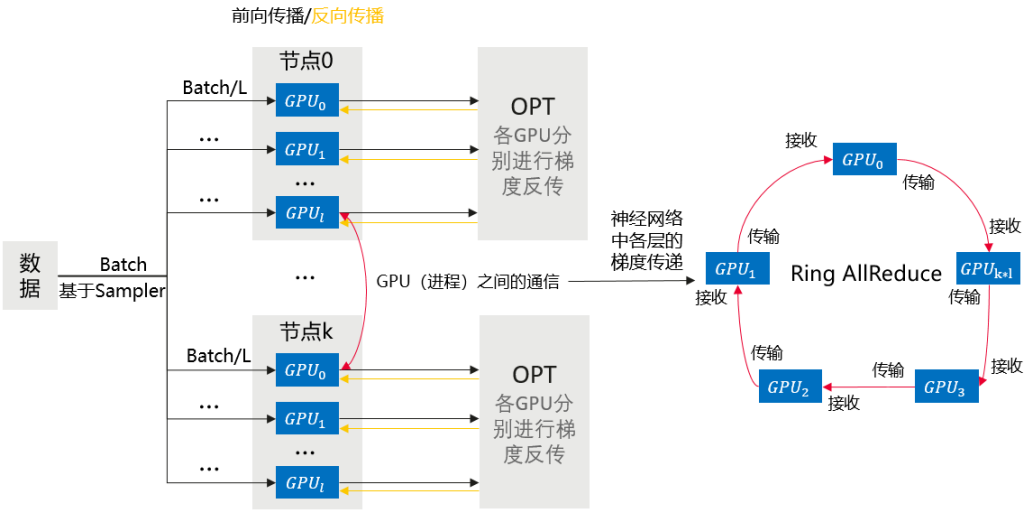
训练流程简述

相比于DP，DDP能够启动多进程进行运算，从而大幅度提升计算资源的利用率。可以基于torch.distributed实现真正的分布式计算，具体的原理此处不再赘述。大致的流程如下：

1. 初始化进程组。
2. 创建分布式并行模型，每个进程都会有相同的模型和参数。
3. 创建数据分发Sampler，使每个进程加载一个mini batch中不同部分的数据。
4. 网络中相邻参数分桶，一般为神经网络模型中需要进行参数更新的每一层网络。
5. 每个进程前向传播并各自计算梯度。
6. 模型某一层的参数得到梯度后会马上进行通讯并进行梯度平均。
7. 各GPU更新模型参数。

具体流程图如下：

图 6-2 多机多卡数据并行训练



DistributedDataParallel 进行多机多卡训练的优缺点

- **通信更快**：相比于DP，通信速度更快
- **负载相对均衡**：相比于DP，GPU负载相对更均衡

- **运行速度快**：因为通信时间更短，效率更高，能更快速的完成训练任务

代码改造点

- 引入多进程启动机制：初始化进程
- 引入几个变量：tcp协议，rank进程序号，worldsize开启的进程数量
- 分发数据：DataLoader中多了一个Sampler参数，避免不同进程数据重复
- 模型分发：DistributedDataParallel(model)
- 模型保存：在序号为0的进程下保存模型

```
import torch
class Net(torch.nn.Module):
    pass

model = Net().cuda()

### DistributedDataParallel Begin ###
model = torch.nn.parallel.DistributedDataParallel(Net().cuda())
### DistributedDataParallel End ###
```

相关操作

- 分布式训练调测具体的代码适配操作过程和代码示例请参见**分布式调测适配及代码示例**章节。
- 文档还针对Resnet18在cifar10数据集上的分类任务，给出了分布式训练改造(DDP)的完整代码示例，供用户学习参考，具体请参见**分布式训练完整代码示例**。

6.4 分布式调测适配及代码示例

在DistributedDataParallel中，不同进程分别从原始数据中加载batch的数据，最终将各个进程的梯度进行平均作为最终梯度，由于样本量更大，因此计算出的梯度更加可靠，可以适当增大学习率。

以下对resnet18在cifar10数据集上的分类任务，给出了单机训练和分布式训练改造(DDP)的代码。直接执行代码为多节点分布式训练且支持CPU分布式和GPU分布式，将代码中的分布式改造点注释掉后即可进行单节点单卡训练。

训练代码中包涵三部分入参，分别为训练基础参数、分布式参数和数据相关参数。其中分布式参数由平台自动入参，无需自行定义。数据相关参数中的custom_data表示是否使用自定义数据进行训练，该参数为“true”时使用基于torch自定义的随机数据进行训练和验证。

数据集

cifar10数据集

在Notebook中，无法直接使用默认版本的torchvision获取数据集，因此示例代码中提供了三种训练数据加载方式。

cifar-10数据集**下载链接**，单击“CIFAR-10 python version”。

- 尝试基于torchvision获取cifar10数据集。
- 基于数据链接下载数据并解压，放置在指定目录下，训练集和测试集的大小分别为(50000，3，32，32)和(10000，3，32，32)。

- 考虑到下载cifar10数据集较慢，基于torch生成类似cifar10的随机数据集，训练集和测试集的大小分别为(5000，3，32，32)和(1000，3，32，32)，标签仍为10类，指定custom_data = 'true'后可直接进行训练任务，无需加载数据。

训练代码

以下代码中以“### 分布式改造，... ###”注释的代码即为多节点分布式训练需要适配的代码改造点。

不对示例代码进行任何修改，适配数据路径后即可在ModelArts上完成多节点分布式训练。

注释掉分布式代码改造点，即可完成单节点单卡训练。完整代码见**分布式训练完整代码示例**。

- **导入依赖包**

```
import datetime
import inspect
import os
import pickle
import random

import argparse
import numpy as np
import torch
import torch.distributed as dist
from torch import nn, optim
from torch.utils.data import TensorDataset, DataLoader
from torch.utils.data.distributed import DistributedSampler
from sklearn.metrics import accuracy_score
```

- **定义加载数据的方法和随机数**，由于加载数据部分代码较多，此处省略

```
def setup_seed(seed):
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)
    random.seed(seed)
    torch.backends.cudnn.deterministic = True

def get_data(path):
    pass
```

- **定义网络结构**

```
class Block(nn.Module):

    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        self.residual_function = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels)
        )

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        out = self.residual_function(x) + self.shortcut(x)
        return nn.ReLU(inplace=True)(out)
```

```
class ResNet(nn.Module):

    def __init__(self, block, num_classes=10):
        super().__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True))
        self.conv2 = self.make_layer(block, 64, 64, 2, 1)
        self.conv3 = self.make_layer(block, 64, 128, 2, 2)
        self.conv4 = self.make_layer(block, 128, 256, 2, 2)
        self.conv5 = self.make_layer(block, 256, 512, 2, 2)
        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.dense_layer = nn.Linear(512, num_classes)

    def make_layer(self, block, in_channels, out_channels, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(in_channels, out_channels, stride))
            in_channels = out_channels
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv1(x)
        out = self.conv2(out)
        out = self.conv3(out)
        out = self.conv4(out)
        out = self.conv5(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.dense_layer(out)
        return out
```

- **进行训练和验证**

```
def main():
    file_dir = os.path.dirname(inspect.getframeinfo(inspect.currentframe()).filename)

    seed = datetime.datetime.now().year
    setup_seed(seed)

    parser = argparse.ArgumentParser(description='Pytorch distribute training',
                                     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    parser.add_argument('--enable_gpu', default='true')
    parser.add_argument('--lr', default='0.01', help='learning rate')
    parser.add_argument('--epochs', default='100', help='training iteration')

    parser.add_argument('--init_method', default=None, help='tcp_port')
    parser.add_argument('--rank', type=int, default=0, help='index of current task')
    parser.add_argument('--world_size', type=int, default=1, help='total number of tasks')

    parser.add_argument('--custom_data', default='false')
    parser.add_argument('--data_url', type=str, default=os.path.join(file_dir, 'input_dir'))
    parser.add_argument('--output_dir', type=str, default=os.path.join(file_dir, 'output_dir'))
    args, unknown = parser.parse_known_args()

    args.enable_gpu = args.enable_gpu == 'true'
    args.custom_data = args.custom_data == 'true'
    args.lr = float(args.lr)
    args.epochs = int(args.epochs)

    if args.custom_data:
        print('[warning] you are training on custom random dataset, '
              'validation accuracy may range from 0.4 to 0.6.')

    ### 分布式改造，DDP初始化进程，其中init_method, rank和world_size参数均由平台自动入参 ###
    dist.init_process_group(init_method=args.init_method, backend="nccl", world_size=args.world_size,
rank=args.rank)
    ### 分布式改造，DDP初始化进程，其中init_method, rank和world_size参数均由平台自动入参 ###
```

```
    tr_set, val_set = get_data(args.data_url, custom_data=args.custom_data)

    batch_per_gpu = 128
    gpus_per_node = torch.cuda.device_count() if args.enable_gpu else 1
    batch = batch_per_gpu * gpus_per_node

    tr_loader = DataLoader(tr_set, batch_size=batch, shuffle=False)

    ### 分布式改造，构建DDP分布式数据sampler，确保不同进程加载到不同的数据 ###
    tr_sampler = DistributedSampler(tr_set, num_replicas=args.world_size, rank=args.rank)
    tr_loader = DataLoader(tr_set, batch_size=batch, sampler=tr_sampler, shuffle=False, drop_last=True)
    ### 分布式改造，构建DDP分布式数据sampler，确保不同进程加载到不同的数据 ###

    val_loader = DataLoader(val_set, batch_size=batch, shuffle=False)

    lr = args.lr * gpus_per_node
    max_epoch = args.epochs
    model = ResNet(Block).cuda() if args.enable_gpu else ResNet(Block)

    ### 分布式改造，构建DDP分布式模型 ###
    model = nn.parallel.DistributedDataParallel(model)
    ### 分布式改造，构建DDP分布式模型 ###

    optimizer = optim.Adam(model.parameters(), lr=lr)
    loss_func = torch.nn.CrossEntropyLoss()

    os.makedirs(args.output_dir, exist_ok=True)

    for epoch in range(1, max_epoch + 1):
        model.train()
        train_loss = 0

        ### 分布式改造，DDP sampler, 基于当前的epoch为其设置随机数，避免加载到重复数据 ###
        tr_sampler.set_epoch(epoch)
        ### 分布式改造，DDP sampler, 基于当前的epoch为其设置随机数，避免加载到重复数据 ###

        for step, (tr_x, tr_y) in enumerate(tr_loader):
            if args.enable_gpu:
                tr_x, tr_y = tr_x.cuda(), tr_y.cuda()
            out = model(tr_x)
            loss = loss_func(out, tr_y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            train_loss += loss.item()
        print('train | epoch: %d | loss: %.4f' % (epoch, train_loss / len(tr_loader)))

        val_loss = 0
        pred_record = []
        real_record = []
        model.eval()
        with torch.no_grad():
            for step, (val_x, val_y) in enumerate(val_loader):
                if args.enable_gpu:
                    val_x, val_y = val_x.cuda(), val_y.cuda()
                out = model(val_x)
                pred_record += list(np.argmax(out.cpu().numpy(), axis=1))
                real_record += list(val_y.cpu().numpy())
                val_loss += loss_func(out, val_y).item()
        val_accu = accuracy_score(real_record, pred_record)
        print('val | epoch: %d | loss: %.4f | accuracy: %.4f' % (epoch, val_loss / len(val_loader), val_accu),
'\n')

        if args.rank == 0:
            # save ckpt every epoch
            torch.save(model.state_dict(), os.path.join(args.output_dir, f'epoch_{epoch}.pth'))
```

```
if __name__ == '__main__':
    main()
```

- **结果对比**

  分别以单机单卡和两节点16卡两种资源类型完成100epoch的cifar-10数据集训练，训练时长和测试集准确率如下。

  表 6-1 训练结果对比

资源类型	单机单卡	两节点16卡
耗时	60分钟	20分钟
准确率	80+	80+

6.5 分布式训练完整代码示例

以下对resnet18在cifar10数据集上的分类任务，给出了分布式训练改造(DDP)的完整代码示例。

训练启动文件main.py内容如下（如果需要执行单机单卡训练任务，则将分布式改造的代码删除）：

```
import datetime
import inspect
import os
import pickle
import random
import logging

import argparse
import numpy as np
from sklearn.metrics import accuracy_score
import torch
from torch import nn, optim
import torch.distributed as dist
from torch.utils.data import TensorDataset, DataLoader
from torch.utils.data.distributed import DistributedSampler

file_dir = os.path.dirname(inspect.getframeinfo(inspect.currentframe()).filename)


def load_pickle_data(path):
    with open(path, 'rb') as file:
        data = pickle.load(file, encoding='bytes')
    return data


def _load_data(file_path):
    raw_data = load_pickle_data(file_path)
    labels = raw_data[b'labels']
    data = raw_data[b'data']
    filenames = raw_data[b'filenames']

    data = data.reshape(10000, 3, 32, 32) / 255
    return data, labels, filenames
```

```
def load_cifar_data(root_path):
    train_root_path = os.path.join(root_path, 'cifar-10-batches-py/data_batch_')
    train_data_record = []
    train_labels = []
    train_filenames = []
    for i in range(1, 6):
        train_file_path = train_root_path + str(i)
        data, labels, filenames = _load_data(train_file_path)
        train_data_record.append(data)
        train_labels += labels
        train_filenames += filenames
    train_data = np.concatenate(train_data_record, axis=0)
    train_labels = np.array(train_labels)

    val_file_path = os.path.join(root_path, 'cifar-10-batches-py/test_batch')
    val_data, val_labels, val_filenames = _load_data(val_file_path)
    val_labels = np.array(val_labels)

    tr_data = torch.from_numpy(train_data).float()
    tr_labels = torch.from_numpy(train_labels).long()
    val_data = torch.from_numpy(val_data).float()
    val_labels = torch.from_numpy(val_labels).long()
    return tr_data, tr_labels, val_data, val_labels


def get_data(root_path, custom_data=False):
    if custom_data:
        train_samples, test_samples, img_size = 5000, 1000, 32
        tr_label = [1] * int(train_samples / 2) + [0] * int(train_samples / 2)
        val_label = [1] * int(test_samples / 2) + [0] * int(test_samples / 2)
        random.seed(2021)
        random.shuffle(tr_label)
        random.shuffle(val_label)
        tr_data, tr_labels = torch.randn((train_samples, 3, img_size, img_size)).float(),
torch.tensor(tr_label).long()
        val_data, val_labels = torch.randn((test_samples, 3, img_size, img_size)).float(),
torch.tensor(
            val_label).long()
        tr_set = TensorDataset(tr_data, tr_labels)
        val_set = TensorDataset(val_data, val_labels)
        return tr_set, val_set
    elif os.path.exists(os.path.join(root_path, 'cifar-10-batches-py')):
        tr_data, tr_labels, val_data, val_labels = load_cifar_data(root_path)
        tr_set = TensorDataset(tr_data, tr_labels)
        val_set = TensorDataset(val_data, val_labels)
        return tr_set, val_set
    else:
        try:
            import torchvision
            from torchvision import transforms
            tr_set = torchvision.datasets.CIFAR10(root='./data', train=True,
                                                  download=True, transform=transforms)
            val_set = torchvision.datasets.CIFAR10(root='./data', train=False,
                                                   download=True, transform=transforms)
            return tr_set, val_set
        except Exception as e:
            raise Exception(
                f"{e}, you can download and unzip cifar-10 dataset manually, "
                "the data url is http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz")


class Block(nn.Module):
```

```
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        self.residual_function = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1,
bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels)
        )

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        out = self.residual_function(x) + self.shortcut(x)
        return nn.ReLU(inplace=True)(out)


class ResNet(nn.Module):

    def __init__(self, block, num_classes=10):
        super().__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True))
        self.conv2 = self.make_layer(block, 64, 64, 2, 1)
        self.conv3 = self.make_layer(block, 64, 128, 2, 2)
        self.conv4 = self.make_layer(block, 128, 256, 2, 2)
        self.conv5 = self.make_layer(block, 256, 512, 2, 2)
        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.dense_layer = nn.Linear(512, num_classes)

    def make_layer(self, block, in_channels, out_channels, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(in_channels, out_channels, stride))
            in_channels = out_channels
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv1(x)
        out = self.conv2(out)
        out = self.conv3(out)
        out = self.conv4(out)
        out = self.conv5(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.dense_layer(out)
        return out


def setup_seed(seed):
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
```

```
    np.random.seed(seed)
    random.seed(seed)
    torch.backends.cudnn.deterministic = True


def obs_transfer(src_path, dst_path):
    import moxing as mox
    mox.file.copy_parallel(src_path, dst_path)
    logging.info(f"end copy data from {src_path} to {dst_path}")


def main():
    seed = datetime.datetime.now().year
    setup_seed(seed)

    parser = argparse.ArgumentParser(description='Pytorch distribute training',
                                     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    parser.add_argument('--enable_gpu', default='true')
    parser.add_argument('--lr', default='0.01', help='learning rate')
    parser.add_argument('--epochs', default='100', help='training iteration')

    parser.add_argument('--init_method', default=None, help='tcp_port')
    parser.add_argument('--rank', type=int, default=0, help='index of current task')
    parser.add_argument('--world_size', type=int, default=1, help='total number of tasks')

    parser.add_argument('--custom_data', default='false')
    parser.add_argument('--data_url', type=str, default=os.path.join(file_dir, 'input_dir'))
    parser.add_argument('--output_dir', type=str, default=os.path.join(file_dir, 'output_dir'))
    args, unknown = parser.parse_known_args()

    args.enable_gpu = args.enable_gpu == 'true'
    args.custom_data = args.custom_data == 'true'
    args.lr = float(args.lr)
    args.epochs = int(args.epochs)

    if args.custom_data:
        logging.warning('you are training on custom random dataset, '
            'validation accuracy may range from 0.4 to 0.6.')

    ### 分布式改造，DDP初始化进程，其中init_method, rank和world_size参数均由平台自动入参 ###
    dist.init_process_group(init_method=args.init_method, backend="nccl", world_size=args.world_size, rank=args.rank)
    ### 分布式改造，DDP初始化进程，其中init_method, rank和world_size参数均由平台自动入参 ###

    tr_set, val_set = get_data(args.data_url, custom_data=args.custom_data)

    batch_per_gpu = 128
    gpus_per_node = torch.cuda.device_count() if args.enable_gpu else 1
    batch = batch_per_gpu * gpus_per_node

    tr_loader = DataLoader(tr_set, batch_size=batch, shuffle=False)

    ### 分布式改造，构建DDP分布式数据sampler，确保不同进程加载到不同的数据 ###
    tr_sampler = DistributedSampler(tr_set, num_replicas=args.world_size, rank=args.rank)
    tr_loader = DataLoader(tr_set, batch_size=batch, sampler=tr_sampler, shuffle=False, drop_last=True)
    ### 分布式改造，构建DDP分布式数据sampler，确保不同进程加载到不同的数据 ###

    val_loader = DataLoader(val_set, batch_size=batch, shuffle=False)
```

```
    lr = args.lr * gpus_per_node * args.world_size
    max_epoch = args.epochs
    model = ResNet(Block).cuda() if args.enable_gpu else ResNet(Block)

    ### 分布式改造，构建DDP分布式模型 ###
    model = nn.parallel.DistributedDataParallel(model)
    ### 分布式改造，构建DDP分布式模型 ###

    optimizer = optim.Adam(model.parameters(), lr=lr)
    loss_func = torch.nn.CrossEntropyLoss()

    os.makedirs(args.output_dir, exist_ok=True)

    for epoch in range(1, max_epoch + 1):
        model.train()
        train_loss = 0

        ### 分布式改造，DDP sampler, 基于当前的epoch为其设置随机数，避免加载到重复数据
###
        tr_sampler.set_epoch(epoch)
        ### 分布式改造，DDP sampler, 基于当前的epoch为其设置随机数，避免加载到重复数据
###

        for step, (tr_x, tr_y) in enumerate(tr_loader):
            if args.enable_gpu:
                tr_x, tr_y = tr_x.cuda(), tr_y.cuda()
            out = model(tr_x)
            loss = loss_func(out, tr_y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            train_loss += loss.item()
        print('train | epoch: %d | loss: %.4f' % (epoch, train_loss / len(tr_loader)))

        val_loss = 0
        pred_record = []
        real_record = []
        model.eval()
        with torch.no_grad():
            for step, (val_x, val_y) in enumerate(val_loader):
                if args.enable_gpu:
                    val_x, val_y = val_x.cuda(), val_y.cuda()
                out = model(val_x)
                pred_record += list(np.argmax(out.cpu().numpy(), axis=1))
                real_record += list(val_y.cpu().numpy())
                val_loss += loss_func(out, val_y).item()
        val_accu = accuracy_score(real_record, pred_record)
        print('val | epoch: %d | loss: %.4f | accuracy: %.4f' % (epoch, val_loss / len(val_loader),
val_accu), '\n')

        if args.rank == 0:
            # save ckpt every epoch
            torch.save(model.state_dict(), os.path.join(args.output_dir, f'epoch_{epoch}.pth'))


if __name__ == '__main__':
    main()
```

常见问题

1、示例代码中如何使用不同的数据集？

- 上述代码如果使用cifar10数据集，则将数据集下载并解压后，上传至OBS桶中，文件目录结构如下：

```
DDP
|--- main.py
|--- input_dir
|------ cifar-10-batches-py
|-------- data_batch_1
|-------- data_batch_2
|-------- ...
```

  其中“DDP”为创建训练作业时的“代码目录”，“main.py”为上文代码示例（即创建训练作业时的“启动文件”），“cifar-10-batches-py”为解压后的数据集文件夹（放在input_dir文件夹下）。

- 如果使用自定义的随机数据，则将代码示例中的参数“custom_data”改为“true”，修改后内容如下：

```
parser.add_argument('--custom_data', default='true')
```

  然后直接运行代码示例“main.py”即可，创建训练作业的参数与上图相同。

2、为什么DDP可以不输入主节点ip?

“parser.add_argument('--init_method', default=None, help='tcp_port')”中的init method参数值会包含主节点的ip和端口，由平台自动入参，不需要用户输入主节点的ip和端口。

6.6 基于训练作业启动 PyTorch DDP 训练示例

本文介绍三种使用训练作业来启动PyTorch DDP训练的方法及对应代码示例：

- 使用PyTorch预置框架功能，通过mp.spawn命令启动
- 使用自定义镜像功能
  - 通过torch.distributed.launch命令启动
  - 通过torch.distributed.run命令启动

创建训练作业

- 方式一：使用PyTorch预置框架功能，通过mp.spawn命令启动训练作业。

  创建训练作业的关键参数如表6-2所示。

表 6-2 创建训练作业（预置框架）

参数名称	说明
创建方式	选择“自定义算法”。
启动方式	选择“预置框架”，引擎选择“PyTorch”，PyTorch版本根据训练要求选择。
代码目录	选择OBS桶中训练code文件夹所在路径，例如“obs://test-modelarts/code/”。
启动文件	选择代码目录中训练作业的Python启动脚本。例如“obs://test-modelarts/code/main.py”。

参数名称	说明
超参	当资源规格为单机多卡时，需要指定超参world_size和rank。 当资源规格为多机时（即计算节点个数大于 1），无需设置超参world_size和rank，超参会由平台自动注入。

- 方式二：使用自定义镜像功能，通过torch.distributed.launch命令启动训练作业。创建训练作业的关键参数如表6-3所示。

表 6-3 创建训练作业（自定义镜像+torch.distributed.launch 命令）

参数名称	说明
创建方式	选择“自定义算法”。
启动方式	选择“自定义”。
镜像	选择用于训练的PyTorch镜像。
代码目录	选择OBS桶中训练code文件夹所在路径，例如“obs://test-modelarts/code/”。
启动命令	输入镜像的Python启动命令，例如： `bash ${MA_JOB_DIR}/code/torchlaunch.sh`

- 方式三：使用自定义镜像功能，通过torch.distributed.run命令启动训练作业。创建训练作业的关键参数如表6-4所示。

表 6-4 创建训练作业（自定义镜像+torch.distributed.run 命令）

参数名称	说明
创建方式	选择“自定义算法”。
启动方式	选择“自定义”。
镜像	选择用于训练的PyTorch镜像。
代码目录	选择OBS桶中训练code文件夹所在路径，例如“obs://test-modelarts/code/”。
启动命令	输入镜像的Python启动命令，例如： `bash ${MA_JOB_DIR}/code/torchrun.sh`

代码示例

文件目录结构如下所示，将以下文件上传至OBS桶中：

```
code                    # 代码根目录
└─torch_ddp.py            # PyTorch DDP训练代码文件
└─main.py               # 使用PyTorch预置框架功能，通过mp.spawn命令启动训练的启动文件
└─torchlaunch.sh          # 使用自定义镜像功能，通过torch.distributed.launch命令启动训练的启动文件
└─torchrun.sh           # 使用自定义镜像功能，通过torch.distributed.run命令启动训练的启动文件
```

torch_ddp.py内容如下：

```
import os
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.optim as optim
from torch.nn.parallel import DistributedDataParallel as DDP

# 用于通过 mp.spawn 启动
def init_from_arg(local_rank, base_rank, world_size, init_method):
    rank = base_rank + local_rank
    dist.init_process_group("nccl", rank=rank, init_method=init_method, world_size=world_size)
    ddp_train(local_rank)

# 用于通过 torch.distributed.launch 或 torch.distributed.run 启动
def init_from_env():
    dist.init_process_group(backend='nccl', init_method='env://')
    local_rank=int(os.environ["LOCAL_RANK"])
    ddp_train(local_rank)

def cleanup():
    dist.destroy_process_group()

class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = nn.Linear(10, 10)
        self.relu = nn.ReLU()
        self.net2 = nn.Linear(10, 5)
    def forward(self, x):
        return self.net2(self.relu(self.net1(x)))

def ddp_train(device_id):
    # create model and move it to GPU with id rank
    model = ToyModel().to(device_id)
    ddp_model = DDP(model, device_ids=[device_id])
    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)
    optimizer.zero_grad()
    outputs = ddp_model(torch.randn(20, 10))
    labels = torch.randn(20, 5).to(device_id)
    loss_fn(outputs, labels).backward()
    optimizer.step()
    cleanup()

if __name__ == "__main__":
    init_from_env()
```

main.py内容如下：

```
import argparse
import torch
import torch.multiprocessing as mp

parser = argparse.ArgumentParser(description='ddp demo args')
parser.add_argument('--world_size', type=int, required=True)
parser.add_argument('--rank', type=int, required=True)
parser.add_argument('--init_method', type=str, required=True)
args, unknown = parser.parse_known_args()

if __name__ == "__main__":
    n_gpus = torch.cuda.device_count()
    world_size = n_gpus * args.world_size
    base_rank = n_gpus * args.rank
    # 调用 DDP 示例代码中的启动函数
    from torch_ddp import init_from_arg
    mp.spawn(init_from_arg,
        args=(base_rank, world_size, args.init_method),
```

```
        nprocs=n_gpus,
        join=True)
```

torchlaunch.sh内容如下：

```
#!/bin/bash
# 系统默认环境变量，不建议修改
MASTER_HOST="$VC_WORKER_HOSTS"
MASTER_ADDR="${VC_WORKER_HOSTS%%,*}"
MASTER_PORT="6060"
JOB_ID="1234"
NNODES="$MA_NUM_HOSTS"
NODE_RANK="$VC_TASK_INDEX"
NGPUS_PER_NODE="$MA_NUM_GPUS"

# 自定义环境变量，指定python脚本和参数
PYTHON_SCRIPT=${MA_JOB_DIR}/code/torch_ddp.py
PYTHON_ARGS=""

CMD="python -m torch.distributed.launch \
    --nnodes=$NNODES \
    --node_rank=$NODE_RANK \
    --nproc_per_node=$NGPUS_PER_NODE \
    --master_addr $MASTER_ADDR \
    --master_port=$MASTER_PORT \
    --use_env \
    $PYTHON_SCRIPT \
    $PYTHON_ARGS
"
echo $CMD
$CMD
```

torchrun.sh内容如下：

须知

PyTorch 2.1版本需要将“rdzv_backend”参数设置为“static：--rdzv_backend=static”。

```
#!/bin/bash
# 系统默认环境变量，不建议修改
MASTER_HOST="$VC_WORKER_HOSTS"
MASTER_ADDR="${VC_WORKER_HOSTS%%,*}"
MASTER_PORT="6060"
JOB_ID="1234"
NNODES="$MA_NUM_HOSTS"
NODE_RANK="$VC_TASK_INDEX"
NGPUS_PER_NODE="$MA_NUM_GPUS"

# 自定义环境变量，指定python脚本和参数
PYTHON_SCRIPT=${MA_JOB_DIR}/code/torch_ddp.py
PYTHON_ARGS=""

if [[ $NODE_RANK == 0 ]]; then
    EXT_ARGS="--rdzv_conf=is_host=1"
else
    EXT_ARGS=""
fi

CMD="python -m torch.distributed.run \
    --nnodes=$NNODES \
    --node_rank=$NODE_RANK \
    $EXT_ARGS \
    --nproc_per_node=$NGPUS_PER_NODE \
    --rdzv_id=$JOB_ID \
    --rdzv_backend=c10d \
    --rdzv_endpoint=$MASTER_ADDR:$MASTER_PORT \
```

```
    $PYTHON_SCRIPT \
    $PYTHON_ARGS
    "
echo $CMD
$CMD
```

7 自动模型优化（AutoSearch）

7.1 超参搜索简介

ModelArts训练支持超参搜索功能，自动实现模型超参搜索，为您的模型匹配最合适的超参。

在模型训练过程中，有很多超参需要根据任务进行调整，比如learning_rate、weight_decay等，这一工作往往需要一个有经验的算法工程师花费一定精力和大量时间进行手动调优。ModelArts支持的超参搜索功能，在无需算法工程师介入的情况下，即可自动进行超参的调优，在速度和精度上超过人工调优。

ModelArts支持以下三种超参搜索算法：

- **贝叶斯优化(SMAC)**
- **TPE算法**
- **模拟退火算法(Anneal)**

7.2 搜索算法

7.2.1 贝叶斯优化(SMAC)

贝叶斯优化假设超参和目标函数存在一个函数关系。基于已搜索超参的评估值，通过高斯过程回归来估计其他搜索点处目标函数值的均值和方差。根据均值和方差构造采集函数（Acquisition Function），下一个搜索点为采集函数的极大值点。相比网格搜索，贝叶斯优化会利用之前的评估结果，从而降低迭代次数、缩短搜索时间；缺点是不容易找到全局最优解。

表 7-1 贝叶斯优化的参数说明

参数	说明	取值参考
num_samples	搜索尝试的超参组数	int，一般在10-20之间，值越大，搜索时间越长，效果越好

参数	说明	取值参考
kind	采集函数类型	string，默认为'ucb'，可能取值还有'ei'、'poi'，一般不建议用户修改
kappa	采集函数ucb的调节参数，可理解为上置信边界	float，一般不建议用户修改
xi	采集函数poi和ei的调节参数	float，一般不建议用户修改

7.2.2 TPE 算法

TPE算法全称Tree-structured Parzen Estimator，是一种利用高斯混合模型来学习超参模型的算法。在每次试验中，对于每个超参，TPE为与最佳目标值相关的超参维护一个高斯混合模型l(x)，为剩余的超参维护另一个高斯混合模型g(x)，选择l(x)/g(x)最大化时对应的超参作为下一组搜索值。

表 7-2 TPE 算法的参数说明

参数	说明	取值参考
num_samples	搜索尝试的超参组数	int，一般在10-20之间，值越大，搜索时间越长，效果越好
n_initial_points	采用TPE接近目标函数之前，对目标函数的随机评估数	int，一般不建议用户修改
gamma	TPE算法的一定分位数，用于划分l(x)和g(x)	float，范围(0,1)，一般不建议用户修改

7.2.3 模拟退火算法(Anneal)

模拟退火算法即Anneal算法，是随机搜索中一个简单但有效的变体，它利用了响应曲面中的平滑度。退火速率不自适应。Anneal算法从先前采样的一个试验点作为起点，然后从与先验分布相似的分布中采样每组超参数，但其密度更集中在选择的试验点周围。随着时间推移，算法会倾向于从越来越接近最佳点处采样。在采样过程中，算法可能绘制一个次佳试验作为最佳试验，以一定概率跳出局部最优解。

表 7-3 模拟退火算法的参数说明

参数	说明	取值参考
num_samples	搜索尝试的超参组数	int，一般在10-20之间，值越大，搜索时间越长，效果越好
avg_best_idx	要探索试验的几何分布平均，从按照分数排序的试验中选择	float，一般不建议用户修改

参数	说明	取值参考
shrink_coef	随着更多的点被探索，邻域采样大小的减少率	float，一般不建议用户修改

7.3 创建超参搜索作业

背景信息

如果用户使用的AI引擎为pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64和tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64，并且优化的超参类型为float类型，ModelArts支持用户使用超参搜索功能。

在0代码修改的基础下，实现算法模型的超参搜索。需要完成以下步骤：

1. 准备工作
2. 创建算法
3. 创建训练作业
4. 查看超参搜索作业详情

准备工作

- 数据已完成准备：已在ModelArts中创建可用的数据集，或者您已将用于训练的数据集上传至OBS目录。
- 请准备好训练脚本，并上传至OBS目录。训练脚本开发指导参见开发自定义脚本。
- 在训练代码中，用户需打印搜索指标参数。
- 已在OBS创建至少1个空的文件夹，用于存储训练输出的内容。
- 由于训练作业运行需消耗资源，确保账户未欠费。
- 确保您使用的OBS目录与ModelArts在同一区域。

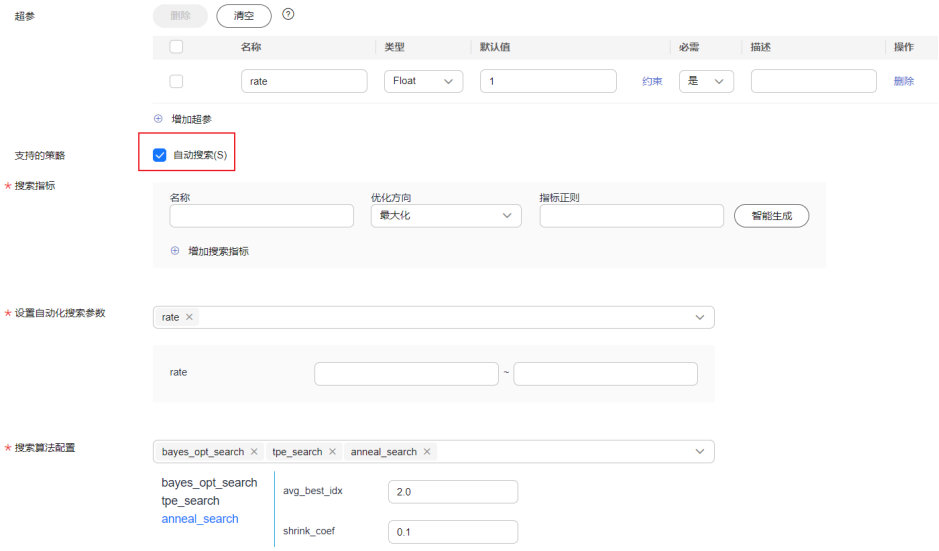
创建算法

进入ModelArts控制台，参考创建算法操作指导，创建自定义算法。镜像应该满足pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64或tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64引擎。

对于用户希望优化的超参，需在“超参”设置中定义，可以给定名称、类型、默认值、约束等，具体设置方法可以参考定义超参。

单击勾选“自动搜索”，为算法设置算法搜索功能。自动搜索作业运行过程中，ModelArts后台通过指标正则表达式获取搜索指标参数，朝指定的优化方向进行超参优化。用户需要在代码中打印搜索参数并在控制台配置以下参数。

图 7-1 设置算法搜索功能



- 搜索指标

  搜索指标为目标函数的值，通常可以设置为loss、accuracy等。通过优化搜索指标的目标值超优化方向收敛，找到最契合的超参，提高模型精度和收敛速度。

  表 7-4 搜索指标参数

参数	说明
名称	搜索指标的名称。需要与您在代码中打印的搜索指标参数保持一致。
优化方向	可选“最大化”或者“最小化”。
指标正则	填入正则表达式。您可以单击智能生成功能自动获取正则表达式。

- 设置自动化搜索参数

  从已设置的“超参”中选择可用于搜索优化的超参。优化的超参仅支持float类型，选中自动化搜索参数后，需设置取值范围。

- 搜索算法配置

  ModelArts内置三种超参搜索算法，用户可以根据实际情况选择对应的算法，支持多选。对应的算法和参数解析请参考以下：

  – bayes_opt_search:**贝叶斯优化(SMAC)**
  – tpe_search:**TPE算法**
  – anneal_search:**模拟退火算法(Anneal)**

提交创建算法完成后即可执行下一步，创建训练作业。

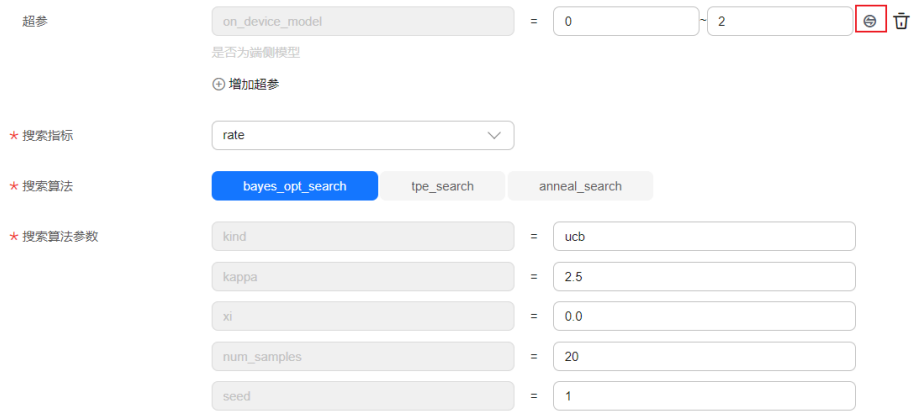
创建训练作业

登录ModelArts控制台，参考**创建训练作业**操作指导，创建训练作业。用户需关注以下操作才能开启超参搜索。

版权所有 © 华为云计算技术有限公司

当您选择支持超参搜索的算法，需单击超参的范围设置按钮才能开启超参搜索功能。

图 7-2 开启超参搜索功能



开启超参搜索功能后，用户可以设置搜索指标、搜索算法和搜索算法参数。三个参数显示的支持值与**算法管理模块的超参设置**对应。

完成超参搜索作业的创建后，训练作业需要运行一段时间。

查看超参搜索作业详情

训练作业运行结束后，可以查看自动超参搜索结果判断此训练作业是否满意。

如果训练作业是超参搜索作业，进入训练作业详情页，选择“自动超参搜索结果”页签查看超参搜索结果。

图 7-3 超参搜索结果