

ModelArts

Model Development

Issue 01
Date 2025-01-06



Copyright © Huawei Cloud Computing Technologies Co., Ltd. 2025. All rights reserved.

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Cloud Computing Technologies Co., Ltd.

Trademarks and Permissions



HUAWEI and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.

All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Huawei Cloud Computing Technologies Co., Ltd.

Address: Huawei Cloud Data Center Jiaoxinggong Road
Qianzhong Avenue
Gui'an New District
Gui Zhou 550029
People's Republic of China

Website: <https://www.huaweicloud.com/intl/en-us/>

Contents

1 Introduction to Model Development.....	1
2 Preparing Data.....	3
3 Preparing Algorithms.....	6
3.1 Introduction to Algorithm Preparation.....	6
3.2 Using a Preset Image (Custom Script).....	7
3.2.1 Overview.....	7
3.2.2 Developing a Custom Script.....	8
3.2.3 Creating an Algorithm.....	10
3.3 Using Custom Images.....	15
3.4 Viewing Algorithm Details.....	17
3.5 Searching for an Algorithm.....	19
3.6 Deleting an Algorithm.....	19
4 Performing a Training.....	20
4.1 Creating a Training Job.....	20
4.2 Viewing Training Job Details.....	35
4.3 Viewing Training Job Events.....	37
4.4 Training Job Logs.....	39
4.4.1 Introduction to Training Job Logs.....	39
4.4.2 Common Logs.....	40
4.4.3 Viewing Training Job Logs.....	41
4.4.4 Locating Faults by Analyzing Training Logs.....	42
4.5 Cloud Shell.....	43
4.5.1 Logging In to a Training Container Using Cloud Shell.....	43
4.5.2 Keeping a Training Job Running.....	45
4.5.3 Preventing Cloud Shell Session from Disconnection.....	47
4.6 Viewing the Resource Usage of a Training Job.....	47
4.7 Evaluation Results.....	49
4.8 Viewing Training Tags.....	53
4.9 Viewing Fault Recovery Details.....	54
4.10 Viewing Environment Variables of a Training Container.....	54
4.11 Stopping, Rebuilding, or Searching for a Training Job.....	59
4.12 Releasing Training Job Resources.....	60

5 Advanced Training Operations.....	61
5.1 Automatic Recovery from a Training Fault.....	61
5.1.1 Training Fault Tolerance Check.....	61
5.1.2 Unconditional Auto Restart.....	66
5.2 Resumable Training and Incremental Training.....	67
5.3 Detecting Training Job Suspension.....	69
5.4 Priority of a Training Job.....	69
5.5 Permission to Set the Highest Job Priority.....	70
6 Distributed Training.....	72
6.1 Distributed Training Functions.....	72
6.2 Single-Node Multi-Card Training Using DataParallel.....	73
6.3 Multi-Node Multi-Card Training Using DistributedDataParallel	75
6.4 Distributed Debugging Adaptation and Code Example.....	76
6.5 Sample Code of Distributed Training.....	80
6.6 Example of Starting PyTorch DDP Training Based on a Training Job.....	86
7 Automatic Model Tuning (AutoSearch).....	90
7.1 Introduction to Hyperparameter Search.....	90
7.2 Search Algorithm.....	90
7.2.1 Bayesian Optimization (SMAC).....	90
7.2.2 TPE Algorithm.....	91
7.2.3 Simulated Annealing Algorithm.....	92
7.3 Creating a Hyperparameter Search Job.....	92

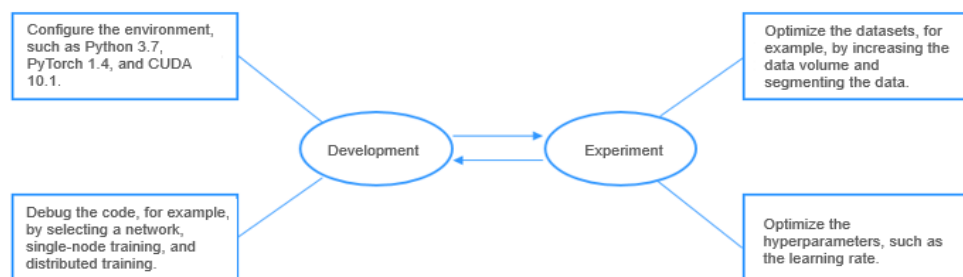
1 Introduction to Model Development

AI modeling involves two stages:

- Development: Prepare and configure the environment, and debug code for training based on deep learning. ModelArts DevEnviron is recommended for code debugging.
- Experiment: Optimize the datasets and hyperparameters, and obtain an ideal model through multiple rounds of experiments. The ModelArts training platform is recommended for training.

In the two stages, code is designed, developed and tested in repeated cycles. In the development stage, when the code becomes stable, the modeling process enters the experiment stage, during which hyperparameters are continuously optimized to iterate the model. In the experiment stage, when the training performance can be optimized, the modeling process returns to the development stage for optimizing code.

Figure 1-1 Model development process



ModelArts provides model training, which allows you to view training results and tune model parameters based on the training results. You can select resource pools with different instance flavors for model training.

The following guides you to train models on ModelArts:

- Upload the labeled data to OBS. For details, see [Preparing Data](#).

- Follow the instructions provided in [Preparing Algorithms](#) to use an algorithm for model training.
- Create a training job. You can perform this operation on the ModelArts console. For details, see [Creating a Training Job](#). For details about how to create models using custom algorithms, see [Using a Custom Algorithm to Build a Handwritten Digit Recognition Model](#).
- Follow the instructions provided in [Training Job Logs](#) to view training job logs and training resource usage.
- Follow the instructions provided in [Stopping, Rebuilding, or Searching for a Training Job](#) to stop or delete a training job.
- Follow the instructions provided in [Automatic Model Tuning \(AutoSearch\)](#) to automatically tune model hyperparameters.
- Troubleshoot if you encounter any problem during training. For details, see [Troubleshooting](#).

2 Preparing Data

ModelArts uses OBS to store data, and backs up and takes snapshots for models, achieving secure, reliable storage at low costs.

- [OBS](#)
- [Obtaining Training Data](#)

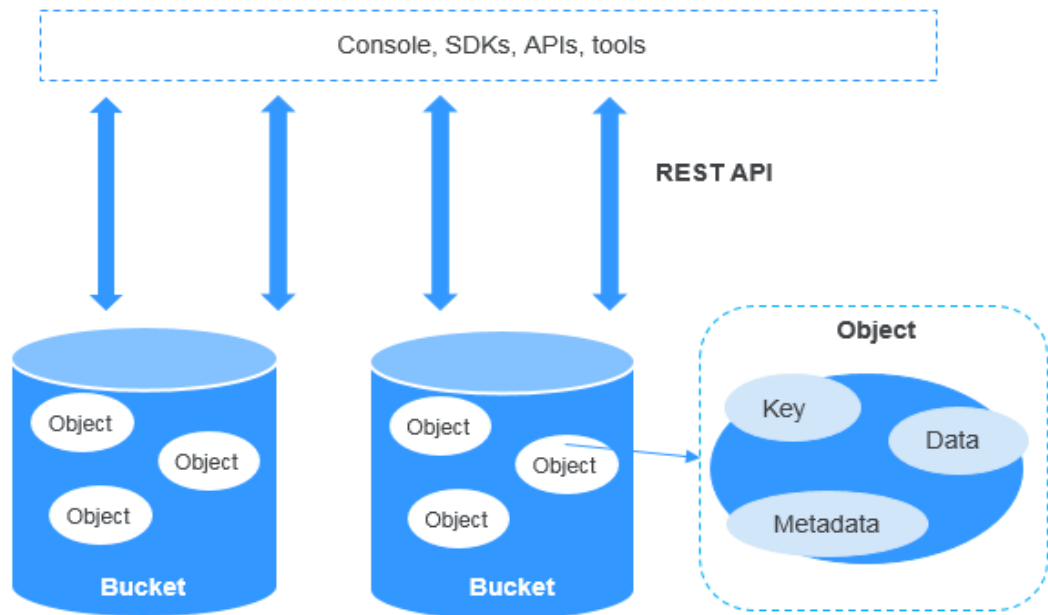
OBS

OBS provides stable, secure, and efficient cloud storage service that lets you store virtually any volume of unstructured data in any format. Bucket and objects are basic concepts in OBS. A bucket is a container for storing objects in OBS. Each bucket is specific to a region and has specific storage class and access permissions. A bucket is accessible through its domain name over the Internet. An object is the basic unit of data storage in OBS.

OBS is a data storage center for ModelArts. All the input data, output data, and cache data during AI development can be stored in OBS buckets for reading.

Before using ModelArts, [create an OBS bucket](#) and folders for storing data.

Figure 2-1 OBS



Obtaining Training Data

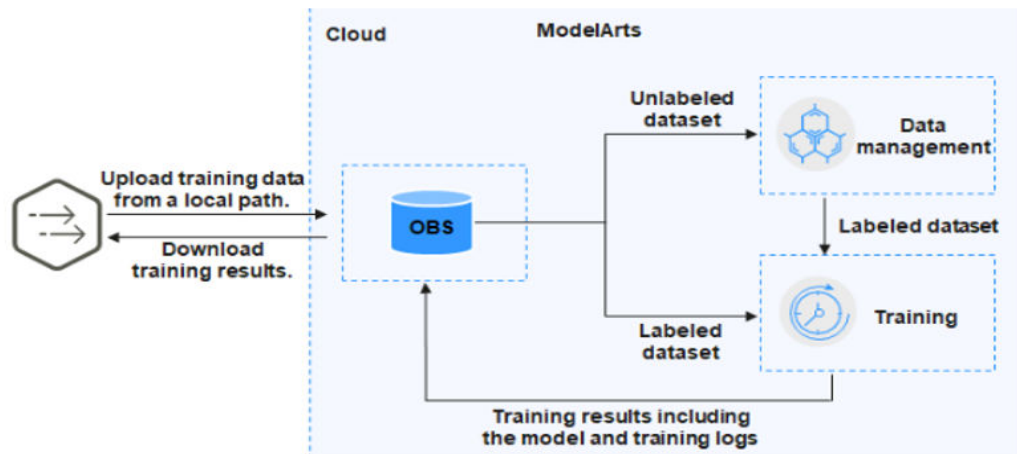
Use either of the following methods to obtain ModelArts training data:

- Datasets stored in OBS buckets
After labeling and preprocessing your dataset, upload it to an OBS bucket. When you create a training job, set **Input** to the path of the OBS bucket where the training data is stored.
- Datasets in data management
If your dataset has not labeled or requires preprocessing, import it to ModelArts data management for data preprocessing.

NOTE

ModelArts data management is being upgraded and is invisible to users who have not used data management. It is recommended that new users store their training data in OBS buckets.

Figure 2-2 Preparing data



3 Preparing Algorithms

3.1 Introduction to Algorithm Preparation

Machine learning explores general rules from limited volume of data and uses these rules to predict unknown data. To obtain more accurate prediction results, select a proper algorithm to train your model. ModelArts provides a large number of algorithm samples for different scenarios. This section describes algorithm sources and learning modes.

Algorithm Sources

You can use one of the following methods to build a ModelArts model:

- Using a preset image
To use a custom algorithm, use a framework built in ModelArts. ModelArts supports most mainstream AI engines. For details, see [Built-in Training Engines](#). These built-in engines pre-load some extra Python packages, such as NumPy. You can also use the `requirements.txt` file in the code directory to install dependency packages. For details about how to create a training job using a preset image, see [Using a Preset Image \(Custom Script\)](#).
- Using a custom image
The subscribed algorithms and built-in frameworks can be used in most training scenarios. In certain scenarios, ModelArts allows you to create custom images to train models. Custom images can be used to train models in ModelArts only after they are uploaded to Software Repository for Container (SWR). For details, see [Using a Custom Image to Train Models](#). Customizing an image requires a deep understanding of containers. Use this method only if the subscribed algorithms and custom scripts cannot meet your requirements.

Algorithm Learning Modes

ModelArts allows you to train models in different modes as required.

- Offline learning
Offline learning is the most fundamental mode for model training. In this mode, all data required for training must be provided at a time, and

optimizing the objective function stops when the training is complete. The advantage of this mode is that the trained models are stable, facilitating model verification and evaluation.

- Incremental learning

Incremental learning is a continuous learning process. Compared with offline learning, it does not need to store all training data at a time, which alleviates the problem of limited storage resources. In addition, it saves a large amount of compute power and time, and reduces economic costs in retraining.

3.2 Using a Preset Image (Custom Script)

3.2.1 Overview

If the subscribed algorithms cannot meet your requirements or you want to migrate local algorithms to ModelArts for training, use the ModelArts preset images to create algorithms. This method is also called using a preset image.

This section describes how to use a preset image to create an algorithm.

- For details about ModelArts built-in engines and models, see [Built-in Training Engines](#).
- To migrate local algorithms to ModelArts, perform code adaptation. For details, see [Developing a Custom Script](#).
- For details about how to use a preset image to create an algorithm on the ModelArts console, see [Creating an Algorithm](#).

Built-in Training Engines

The following table lists the training engines and their versions supported by ModelArts.

 **NOTE**

Supported AI engines vary depending on regions.

Table 3-1 AI engines supported by training jobs

Runtime Environment	System Architecture	System Version	AI Engine and Version	Supported CUDA or Ascend Version
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

Runtime Environment	System Architecture	System Version	AI Engine and Version	Supported CUDA or Ascend Version
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
			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 Developing a Custom Script

Before you use a preset image to create an algorithm, develop the algorithm code. This section describes how to modify local code for model training on ModelArts.

When creating an algorithm, set the code directory, boot file, input path, and output path. These settings enable the interaction between your codes and ModelArts.

- Code directory**
 Specify the code directory in the OBS bucket and upload training data such as training code, dependency installation packages, or pre-generated model to the directory. After you create the training job, ModelArts downloads the code directory and its subdirectories to the container.
 Take OBS path **obs://obs-bucket/training-test/demo-code** as an example. The content in the OBS path will be automatically downloaded to **\$_{MA_JOB_DIR}/demo-code** in the training container, and **demo-code** (customizable) is the last-level directory of the OBS path.
 Do not store training data in the code directory. When the training job starts, the data stored in the code directory will be downloaded to the backend. A large amount of training data may lead to a download failure. It is recommended that the size of the code directory does not exceed 50 MB.
- Boot file**
 The boot file in the code directory is used to start the training. Only Python boot files are supported.
- Input path**
 The training data must be uploaded to an OBS bucket or stored in the **dataset**. In the training code, **the input path** must be parsed. ModelArts automatically downloads the data in the input path to the local container directory for training. Ensure that you have the read permission to the OBS bucket. After the training job is started, ModelArts mounts a disk to the /

cache directory. You can use this directory to store temporary files. For details about the size of the **/cache** directory, see [What Are Sizes of the /cache Directories for Different Resource Specifications in the Training Environment?](#)

- Output path

You are advised to set an empty directory as the training output path. In the training code, **the output path** must be parsed. ModelArts automatically uploads the training output to the output path. Ensure that you have the write and read permissions to the OBS bucket.

The following section describes how to develop training code in ModelArts.

(Optional) Introducing Dependencies

1. If your model references other dependencies, place the required file or installation package in **Code Directory** you set during algorithm creation.
 - For details about how to install the Python dependency package, see [How Do I Create a Training Job When a Dependency Package Is Referenced by the Model to Be Trained?](#)
 - For details about how to install a C++ dependency library, see [How Do I Install a Library That C++ Depends on?](#)
 - For details about how to load parameters to a pre-trained model, see [How Do I Load Some Well Trained Parameters During Job Training?](#)

Parsing Input and Output Paths

When a ModelArts model reads data stored in OBS or outputs data to a specified OBS path, perform the following operations to configure the input and output data:

1. Parse the input and output paths in the training code. The following method is recommended:

```
import argparse
# Create a parsing task.
parser = argparse.ArgumentParser(description='train mnist')

# Add parameters.
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')

# Parse the parameters.
args = parser.parse_args()
```

After the parameters are parsed, use **data_url** and **train_url** to replace the paths to the data source and the data output, respectively.

2. When creating a training job, set the input and output paths.
Select the OBS path or dataset path as the training input, and the OBS path as the output.

Editing Training Code and Saving the Model

Training code and the code for saving the model are closely related to the AI engine you use. The following uses the TensorFlow framework as an example. Before using this case, you need to [download](#) the **mnist.npz** file and upload it to

the OBS bucket. The training input is the OBS path where the **mnist.npz** file is stored.

```
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 Creating an Algorithm

Your locally developed algorithms or algorithms developed using other tools can be uploaded to ModelArts for unified management. Note the following when creating a custom algorithm:

1. [Prerequisites](#)
2. [Accessing the Algorithm Creation Page](#)
3. [Setting Basic Parameters](#)
4. [Setting the Boot Mode](#)
5. [Configuring Pipelines](#)
6. [Defining Hyperparameters](#)
7. [Supported Policies](#)
8. [Adding Training Constraints](#)
9. [Runtime Environment Preview](#)
10. [Follow-up Operations](#)

Prerequisites

- Data is available either by creating a dataset in ModelArts or by uploading the dataset used for training to the OBS directory.
- Your training script has been uploaded to the OBS directory. For details about how to develop a training script, see [Developing a Custom Script](#).
- At least one empty folder has been created in OBS for storing the training output.

- The account is not in arrears because resources are consumed when training jobs are running.
- The OBS directory you use and ModelArts are in the same region.

Accessing the Algorithm Creation Page

1. Log in to the ModelArts management console and click **Algorithm Management** in the left navigation pane.
2. On the **My Algorithms** page, click **Create**. The **Create Algorithm** page is displayed.

Setting Basic Parameters

Enter the basic algorithm information, including **Name** and **Description**.

Setting the Boot Mode

Select a preset image to create an algorithm.

Set **Image**, **Code Directory**, and **Boot File** based on the algorithm code. Ensure that the framework of the AI image you select is the same as the one you use for editing algorithm code. For example, if TensorFlow is used for editing algorithm code, select a TensorFlow image when you create an algorithm.

Table 3-2 Parameters

Parameter	Description
Boot Mode > Preset image	Select a preset image and its version used by the algorithm. To use an old-version image, select Show Old Images .
Code Directory	<p>OBS path for storing the algorithm code. The files required for training, such as the training code, dependency installation packages, and pre-generated models, are uploaded to the code directory.</p> <p>Do not store training data in the code directory. When the training job starts, the data stored in the code directory will be downloaded to the backend. A large amount of training data may lead to a download failure.</p> <p>After you create the training job, ModelArts downloads the code directory and its subdirectories to the container.</p> <p>Take OBS path obs://obs-bucket/training-test/demo-code as an example. The content in the OBS path will be automatically downloaded to #{MA_JOB_DIR}/demo-code in the training container, and demo-code (customizable) is the last-level directory of the OBS path.</p> <p>NOTE</p> <ul style="list-style-type: none"> • Any programming language is supported. • The number of files (including files and folders) cannot exceed 1,000. • The total size of files cannot exceed 5 GB.

Parameter	Description
Boot File	The file must be stored in the code directory and end with .py. ModelArts supports boot files edited only in Python. The boot file in the code directory is used to start a training job.

Figure 3-1 Using a custom script to create an algorithm

* Boot Mode

Preset image Custom image

Code Directory ?

Boot File ?

Configuring Pipelines

A preset image-based algorithm obtains data from an OBS bucket or dataset for model training. The training output is stored in an OBS bucket. The input and output parameters in your algorithm code must be parsed to enable data exchange between ModelArts and OBS. For details about how to develop code for training on ModelArts, see [Developing a Custom Script](#).

When you use a preset image to create an algorithm, configure the input and output pipelines.

- Input configurations

Table 3-3 Input configurations

Parameter	Description
Parameter Name	Set the name based on the data input parameter in your algorithm code. The code path parameter must be the same as the training input parameter parsed in your algorithm code. Otherwise, the algorithm code cannot obtain the input data. For example, If you use argparse in the algorithm code to parse data_url into the data input, set the data input parameter to data_url when creating the algorithm.
Description	Customizable description of the input parameter,
Obtained from	Source of the input parameter. You can select Hyperparameters (default) or Environment variables .

Parameter	Description
Constraints	<p>Whether data is obtained from a storage path or ModelArts dataset.</p> <p>If you select the ModelArts dataset as the data source, the following constraints are added:</p> <ul style="list-style-type: none"> • Labeling Type: For details, see Creating a Labeling Job. • Data Format, which can be Default, CarbonData, or both. Default indicates the manifest format. • Data Segmentation: available only for image classification, object detection, text classification, and sound classification datasets. Possible values are Segmented dataset, Dataset not segmented, and Unlimited. For details, see Publishing a Data Version.
Add	Multiple data input sources are allowed.

- Output configurations

Table 3-4 Output configurations

Parameter	Description
Parameter Name	<p>Set the name based on the data output parameter in your algorithm code. The code path parameter must be the same as the training output parameter parsed in your algorithm code. Otherwise, the algorithm code cannot obtain the output path.</p> <p>For example, If you use argparse in the algorithm code to parse train_url into the data output, set the data output parameter to train_url when creating the algorithm.</p>
Description	Customizable description of the output parameter,
Obtained from	Source of the output parameter. You can select Hyperparameters (default) or Environment variables .
Add	Multiple data output paths are allowed.

Defining Hyperparameters

When you use a preset image to create an algorithm, ModelArts allows you to customize hyperparameters so you can view or modify them anytime. After the hyperparameters are defined, they are displayed in the startup command and transferred to your boot file as CLI parameters.

1. Import hyperparameters.

You can click **Add hyperparameter** to manually add hyperparameters.

2. Edit hyperparameters.
For details, see [Table 3-5](#).

Table 3-5 Hyperparameters

Parameter	Description
Name	Hyperparameter name Enter 1 to 64 characters. Only letters, digits, hyphens (-), and underscores (_) are allowed.
Type	Type of the hyperparameter, which can be String , Integer , Float , or Boolean
Default	Default value of the hyperparameter, which is used for training jobs by default
Constraints	Click Restrain . Then, set the range of the default value or enumerated value in the dialog box displayed.
Required	Select Yes or No . <ul style="list-style-type: none"> • If you select No, you can delete the hyperparameter on the training job creation page when using this algorithm to create a training job. • If you select Yes, you cannot delete the hyperparameter on the training job creation page when using this algorithm to create a training job.
Description	Description of the hyperparameter Only letters, digits, spaces, hyphens (-), underscores (_), commas (,), and periods (.) are allowed.

Supported Policies

ModelArts supports auto search. Auto search automatically finds the optimal hyperparameters without any code modification. This improves model precision and convergence speed. For details about parameter settings, see [Parameters of hyperparameter search](#).

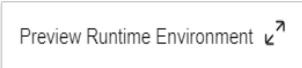
Only the `pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64` and `tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64` images are available for auto search.

Adding Training Constraints

You can add training constraints of the algorithm based on your needs.

- **Resource Type:** Select the required resource types.
- **Multicard Training:** Choose whether to support multi-card training.
- **Distributed Training:** Choose whether to support distributed training.

Runtime Environment Preview

When creating an algorithm, click the arrow on  in the lower right corner of the page to know the path of the code directory, boot file, and input and output data in the training container.

Follow-up Operations

After an algorithm is created, use it to create a training job. For details, see [Creating a Training Job](#).

3.3 Using Custom Images

The subscribed algorithms and preset images can be used in most training scenarios. In certain scenarios, ModelArts allows you to create custom images to train models.

Customizing an image requires a deep understanding of containers. Use this method only if the subscribed algorithms and preset images cannot meet your requirements. Custom images can be used to train models in ModelArts only after they are uploaded to the Software Repository for Container (SWR).

You can use custom images for training on ModelArts in either of the following ways:

- Using a preset image with customization
If you use a preset image to create a training job and you need to modify or add some software dependencies based on the preset image, you can customize the preset image. In this case, select a preset image and choose **Customize** from the framework version drop-down list box.
- Using a custom image
You can create an image based on the ModelArts image specifications, select your own image and configure the code directory (optional) and boot command to create a training job.

NOTE

When you use a custom image to create a training job, the boot command must be executed in the `/home/ma-user` directory. Otherwise, the training job may run abnormally.

Using a Preset Image with Customization

The only difference between this method and creating a training job totally based on a preset image is that you must select an image. You can create a custom image based on a preset image.

Figure 3-2 Creating an algorithm using a preset image with customization

★ Boot Mode Preset image Custom image

Customize

★ Image Select

★ Code Directory Select

★ Boot File Select

The process of this method is the same as that of creating a training job based on a preset image. For example:

- The system automatically injects environment variables.
 - `PATH=${MA_HOME}/anaconda/bin:${PATH}`
 - `LD_LIBRARY_PATH=${MA_HOME}/anaconda/lib:${LD_LIBRARY_PATH}`
 - `PYTHONPATH=${MA_JOB_DIR}:${PYTHONPATH}`
- The selected boot file will be automatically started using Python commands. Ensure that the Python environment is correct. The PATH environment variable is automatically injected. Run the following commands to check the Python version for the training job:
 - `export MA_HOME=/home/ma-user; docker run --rm {image} ${MA_HOME}/anaconda/bin/python -V`
 - `docker run --rm {image} $(which python) -V`
- The system automatically adds hyperparameters associated with the preset image.

Using a Custom Image

Figure 3-3 Creating an algorithm using a custom image

★ Boot Mode Preset image Custom image

★ Image Select

Code Directory Select

★ Boot Command ?

1

For details about how to use custom images supported by training, see [Using a Custom Image to Create a CPU- or GPU-based Training Job](#).

If all used images are customized, do as follows to use a specified Conda environment to start training:

Training jobs do not run in a shell. Therefore, you are not allowed to run the **conda activate** command to activate a specified Conda environment. In this case, use other methods to start training.

For example, Conda in your custom image is installed in the **/home/ma-user/anaconda3** directory, the Conda environment is **python-3.7.10**, and the training script is stored in **/home/ma-user/modelarts/user-job-dir/code/train.py**. Use a specified Conda environment to start training in one of the following ways:

- Method 1: Configure the correct **DEFAULT_CONDA_ENV_NAME** and **ANACONDA_DIR** environment variables for the image.

```
ANACONDA_DIR=/home/ma-user/anaconda3
DEFAULT_CONDA_ENV_NAME=python-3.7.10
```

Run the **python** command to start the training script. The following shows an example:

```
python /home/ma-user/modelarts/user-job-dir/code/train.py
```
- Method 2: Use the absolute path of Conda environment Python.

Run the **/home/ma-user/anaconda3/envs/python-3.7.10/bin/python** command to start the training script. The following shows an example:

```
/home/ma-user/anaconda3/envs/python-3.7.10/bin/python /home/ma-user/modelarts/user-job-dir/code/train.py
```
- Method 3: Configure the path environment variable.

Configure the bin directory of the specified Conda environment into the path environment variable. Run the **python** command to start the training script. The following shows an example:

```
export PATH=/home/ma-user/anaconda3/envs/python-3.7.10/bin:$PATH; python /home/ma-user/modelarts/user-job-dir/code/train.py
```
- Method 4: Run the **conda run -n** command.

Run the **/home/ma-user/anaconda3/bin/conda run -n python-3.7.10** command to execute the training. The following shows an example:

```
/home/ma-user/anaconda3/bin/conda run -n python-3.7.10 python /home/ma-user/modelarts/user-job-dir/code/train.py
```

NOTE

If there is an error indicating that the .so file is unavailable in the **\$ANACONDA_DIR/envs/\$DEFAULT_CONDA_ENV_NAME/lib** directory, add the directory to **LD_LIBRARY_PATH** and place the following command before the preceding boot command:

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

For example, the example boot command used in method 1 is as follows:

```
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 Viewing Algorithm Details

1. Log in to the ModelArts console.
2. In the navigation pane, choose **Algorithm Management**. The **My algorithm** page is displayed.

3. In the algorithm list, click the target algorithm name to go to the algorithm details page.
 - On the **Basic Information** tab, you can view the algorithm information.

Table 3-6 Basic algorithm information

Parameter	Description
Name	Algorithm name.
ID	Unique ID of an algorithm.
Description	Algorithm description. You can click the edit icon to update the description.
Preset image	Preset image and its version used by an algorithm. This parameter is available only for algorithms created using a preset image.
Custom image	Container image used by an algorithm. This parameter is available only for algorithms created using a custom engine version or custom image.
Code Directory	OBS directory for storing the algorithm code.
Boot File	OBS directory for storing the boot file. This parameter is available only for algorithms created using a preset image.
Boot Command	Boot command of an algorithm created using a custom image.
Input	Input parameters of an algorithm.
Output	Output parameters of an algorithm.
Hyperparameter	Hyperparameter information of an algorithm.
Supported Policies	Auto search policy of an algorithm. If this parameter is left blank, auto search is not supported. Otherwise, auto search parameters are displayed.
Training Constraint	Training constraints of an algorithm. If No is displayed, there is no constraint. Otherwise, the supported resource types and training scenarios are displayed.

- On the **Training** tab, you can view the information about the training jobs that use the algorithm, such as the training job name and status.
4. On the **Basic Information** tab, click **Edit** to modify algorithm information except the name and ID. After the modification, click **Save**.

3.5 Searching for an Algorithm

ModelArts allows you to quickly search for algorithms by performing the following operations.

Operation 1: Search for jobs by name, image, code directory, description, and creation time.

Operation 2: Click the refresh button in the upper right corner to refresh the algorithm list.

Operation 3: Configure the custom columns and other basic settings.

Figure 3-4 Searching for an algorithm



To sort algorithms in a column, click  in the table header of the algorithm list.

3.6 Deleting an Algorithm

Deleting Your Algorithm

Delete unused algorithms.

1. On the **Algorithm Management** > **My algorithm** page.
2. click **Delete** in the **Operation** column of the target algorithm.
3. In the displayed dialog box, click **OK** to confirm the deletion.

4 Performing a Training

4.1 Creating a Training Job

Model training continuously iterates and optimizes model weights. ModelArts training management allows you to create training jobs, view training status, and manage training versions. Through model training, you can test various combinations of model structures, data, and hyperparameters to obtain the optimal model structure and weight.

Prerequisites

- The data used for training has been uploaded to an OBS directory.
- At least one empty folder has been created in OBS for storing the training output.

NOTE

OBS buckets are not encrypted. ModelArts does not support encrypted OBS buckets. When creating an OBS bucket, do not enable bucket encryption.

- The account is not in arrears because resources are consumed when training jobs are running.
- The OBS directory you use and ModelArts are in the same region.
- Access authorization has been configured. For details, see [Configuring Access Authorization \(Global Configuration\)](#).
- (Optional) An algorithm is available in **Algorithm Management** if you want to use it to create a training job. For details, see [Introduction to Algorithm Preparation](#).
- (Optional) A custom image has been uploaded to SWR if you want to use it to create a training job. For details, see [How Can I Log In to SWR and Upload Images to It?](#)

Operation Procedure

To create a training job, follow these steps:

- Step 1** Access the page for creating a training job. For details, see [Accessing the Page for Creating a Training Job](#).

- Step 2** Configure basic information about the training job. For details, see [Configuring Basic Information About a Training Job](#).
- Step 3** Select an algorithm type for creating the training job.
- Use a preset image to create a training job by referring to [Choosing a Boot Mode \(Preset Image\)](#).
 - Use a custom image to create a training job by referring to [Choosing a Boot Mode \(Custom Image\)](#).
 - Use an existing algorithm to create a training job by referring to [Choosing an Algorithm Type \(My Algorithm\)](#).
- Step 4** Configure training parameters, including the input, output, hyperparameters, and environment variables. For details, see [Configuring Training Parameters](#).
- Step 5** Select a resource pool as required. A dedicated resource pool is recommended. For details about the differences between dedicated resource pools and public resource pools, see [Differences Between Dedicated Resource Pools and Public Resource Pools](#).
- [Configuring a Public Resource Pool](#)
 - [Configuring a Dedicated Resource Pool](#)
- Step 6** Set tags if you want to manage training jobs by group. For details, see [\(Optional\) Setting Tags](#).
- Step 7** Perform follow-up procedure. For details, see [Follow-Up Procedure](#).

----End

Accessing the Page for Creating a Training Job

1. Log in to the ModelArts console.
2. In the navigation pane, choose **Training Management > Training Jobs**. The training job list is displayed.
3. Click **Create Training Job**. The **Create Training Job** page is displayed.

Configuring Basic Information About a Training Job

On the **Create Training Job** page, set parameters.

Table 4-1 Basic information for creating a training job

Parameter	Description
Name	Name of a training job, which is mandatory. The system automatically generates a name. You can rename it based on the following naming rules: <ul style="list-style-type: none"> • The name contains 1 to 64 characters. • Letters, digits, hyphens (-), and underscores (_) are allowed.
Description	Job description, which helps you learn about the job information in the training job list.

Choosing a Boot Mode (Preset Image)

If you use a preset image to create a training job, select a boot mode by referring to [Table 4-2](#).

Table 4-2 Creating a training job using a preset image

Parameter	Description
Algorithm Type	Select Custom algorithm . This parameter is mandatory.
Boot Mode	Select Preset image and select the preset image engine and engine version to be used by the training job. If you select Customize for the engine version, select a custom image from Image .
Image	This parameter is displayed and mandatory only when the preset image version is set to Customize . You can set the container image path in either of the following ways: <ul style="list-style-type: none"> To select your image or an image shared by others, click Select on the right and select a container image for training. The required image must be uploaded to SWR beforehand. To select a public image, enter the address of the public image in SWR. Enter the image path in the format of "Organization name/Image name:Version name". Do not contain the domain name (swr.<region>.myhuaweicloud.com) in the path because the system will automatically add the domain name to the path. For example, if the SWR address of a public image is swr.<region>.myhuaweicloud.com/test-image/tensorflow2_1_1:1.1.1, enter test-images/tensorflow2_1_1:1.1.1.

Parameter	Description
Code Directory	<p>Select the OBS directory where the training code file is stored. This parameter is mandatory.</p> <ul style="list-style-type: none"> • Upload code to the OBS bucket beforehand. The total size of files in the directory cannot exceed 5 GB, the number of files cannot exceed 1000, and the folder depth cannot exceed 32. • The training code file is automatically downloaded to the <code>\${MA_JOB_DIR}/demo-code</code> directory of the training container when the training job is started. demo-code is the last-level OBS directory for storing the code. For example, if Code Directory is set to <code>/test/code</code>, the training code file is downloaded to the <code>\${MA_JOB_DIR}/code</code> directory of the training container.
Boot File	<p>Select the Python boot script of the training job in the code directory. This parameter is mandatory.</p> <p>ModelArts supports only the boot file written in Python. Therefore, the boot file must end with <code>.py</code>.</p>
Local Code Directory	<p>Specify the local directory of a training container. When a training starts, the system automatically downloads the code directory to this directory.</p> <p>The default local code directory is <code>/home/ma-user/modelarts/user-job-dir</code>. This parameter is optional.</p>
Work Directory	<p>During training, the system automatically runs the <code>cd</code> command to execute the boot file in this directory.</p>

Choosing a Boot Mode (Custom Image)

If you use a custom image to create a training job, select a boot mode by referring to [Table 4-3](#).

Table 4-3 Creating a training job using a custom image

Parameter	Description
Algorithm Type	Select Custom algorithm . This parameter is mandatory.
Boot Mode	Select Custom image . This parameter is mandatory.

Parameter	Description
Image	<p>Container image path. This parameter is mandatory. You can set the container image path in either of the following ways:</p> <ul style="list-style-type: none"> • To select your image or an image shared by others, click Select on the right and select a container image for training. The required image must be uploaded to SWR beforehand. • To select a public image, enter the address of the public image in SWR. Enter the image path in the format of "Organization name/Image name:Version name". Do not contain the domain name (swr.<region>.myhuaweicloud.com) in the path because the system will automatically add the domain name to the path. For example, if the SWR address of a public image is swr.<region>.myhuaweicloud.com/test-image/tensorflow2_1_1:1.1.1, enter test-images/tensorflow2_1_1:1.1.1.
Code Directory	<p>Select the OBS directory where the training code file is stored. If the custom image does not contain training code, you need to set this parameter. If the custom image contains training code, you do not need to set this parameter.</p> <ul style="list-style-type: none"> • Upload code to the OBS bucket beforehand. The total size of files in the directory cannot exceed 5 GB, the number of files cannot exceed 1000, and the folder depth cannot exceed 32. • The training code file is automatically downloaded to the `\${MA_JOB_DIR}/demo-code directory of the training container when the training job is started. demo-code is the last-level OBS directory for storing the code. For example, if Code Directory is set to /test/code, the training code file is downloaded to the `\${MA_JOB_DIR}/code directory of the training container.
User ID	<p>User ID for running the container. The default value 1000 is recommended.</p> <p>If the UID needs to be specified, its value must be within the specified range. The UID ranges of different resource pools are as follows:</p> <ul style="list-style-type: none"> • Public resource pool: 1000 to 65535 • Dedicated resource pool: 0 to 65535

Parameter	Description
Boot Command	<p>Command for booting an image. This parameter is mandatory.</p> <p>When a training job is running, the boot command is automatically executed after the code directory is downloaded.</p> <ul style="list-style-type: none"> If the training boot script is a .py file, train.py for example, the boot command is as follows. <pre>python \${MA_JOB_DIR}/demo-code/train.py</pre> If the training boot script is a .sh file, main.sh for example, the boot command is as follows. <pre>bash \${MA_JOB_DIR}/demo-code/main.sh</pre> <p>You can use semicolons (;) and ampersands (&&) to combine multiple commands. demo-code in the command is the last-level OBS directory where the code is stored. Replace it with the actual one.</p>
Local Code Directory	<p>Specify the local directory of a training container. When a training starts, the system automatically downloads the code directory to this directory.</p> <p>The default local code directory is /home/ma-user/modelarts/user-job-dir. This parameter is optional.</p>
Work Directory	<p>During training, the system automatically runs the cd command to execute the boot file in this directory.</p>

Choosing an Algorithm Type (My Algorithm)

Set **Algorithm Type** to **My algorithm** and select an algorithm from the algorithm list. If no algorithm meets the requirements, you can create an algorithm. For details, see [Creating an Algorithm](#).

Configuring Training Parameters

Data is obtained from an OBS bucket or dataset for model training. The training output is also stored in an OBS bucket. When creating a training job, you can configure parameters such as input, output, hyperparameters, and environment variables by referring to [Table 4-4](#).

NOTE

The input, output, and hyperparameter parameters of a training job vary depending on the algorithm type selected during training job creation. If a parameter value is dimmed, the parameter has been configured in the algorithm code and cannot be modified.

Table 4-4 Parameters for creating a training job

Parameter	Sub-Parameter	Description
Input	Parameter name	<p>The algorithm code reads the training input data based on the input parameter name.</p> <p>The recommended value is data_url. The training input parameters must match the input parameters of the selected algorithm. For details, see Table 3-3.</p>
	Dataset	<p>Click Dataset and select the target dataset and its version in the ModelArts dataset list.</p> <p>When the training job is started, ModelArts automatically downloads the data in the input path to the training container.</p> <p>NOTE ModelArts data management is being upgraded and is invisible to users who have not used data management. It is recommended that new users store their training data in OBS buckets.</p>
	Data path	<p>Click Data path and select the storage path to the training input data from an OBS bucket.</p> <p>When the training job is started, ModelArts automatically downloads the data in the input path to the training container.</p>
	Obtained from	<p>The following uses training input data_path as an example.</p> <ul style="list-style-type: none"> If you select Hyperparameters, use this code to obtain the data: <pre>import argparse parser = argparse.ArgumentParser() parser.add_argument('--data_path') args, unknown = parser.parse_known_args() data_path = args.data_path</pre> If you select Environment variables, use this code to obtain the data: <pre>import os data_path = os.getenv("data_path", "")</pre>
Output	Parameter name	<p>The algorithm code reads the training output data based on the output parameter name.</p> <p>The recommended value is train_url. The training output parameters must match the output parameters of the selected algorithm. For details, see Table 3-4.</p>

Parameter	Sub-Parameter	Description
	Data path	<p>Click Data path and select the storage path to the training output data from an OBS bucket. During training, the system automatically synchronizes files from the local code directory of the training container to the data path.</p> <p>NOTE The data path can only be an OBS path. To prevent any issues with data storage, choose an empty directory as the data path.</p>
	Obtained from	<p>The following uses the training output train_url as an example.</p> <ul style="list-style-type: none"> If you select Hyperparameters, use this code to obtain the data: <pre data-bbox="746 824 1430 954">import argparse parser = argparse.ArgumentParser() parser.add_argument('--train_url') args, unknown = parser.parse_known_args() train_url = args.train_url</pre> If you select Environment variables, use this code to obtain the data: <pre data-bbox="746 1025 1430 1084">import os train_url = os.getenv("train_url", "")</pre>
	Predownload	<p>Indicates whether to pre-download the files in the output directory to a local directory.</p> <ul style="list-style-type: none"> If you set Predownload to No, the system does not download the files in the training output data path to a local directory of the training container when the training job is started. If you set Predownload to Yes, the system automatically downloads the files in the training output data path to a local directory of the training container when the training job is started. The larger the file size, the longer the download time. To avoid excessive training time, remove any unneeded files from the local code directory of the training container as soon as possible. If you want to use resumable training and incremental training, you must select Yes.
Hyperparameter	N/A	<p>Used for training tuning. This parameter is determined by the selected algorithm. If hyperparameters have been defined in the algorithm, all hyperparameters in the algorithm are displayed.</p> <p>Hyperparameters can be modified and deleted. The status depends on the hyperparameter constraint settings in the algorithm. For details, see Defining Hyperparameters.</p>


Parameter	Sub-Parameter	Description
Environment Variable	N/A	Add environment variables based on service requirements. For details about the environment variables preset in the training container, see Viewing Environment Variables of a Training Container .
Auto Restart	N/A	<p>Once this function is enabled, you can set the number of restarts and whether to enable Unconditional auto restart.</p> <p>After you enable auto restart, ModelArts will handle any exceptions caused by environmental issues during a training job. It will either automatically handle the exception or isolate the faulty node and then restart the job, which helps to increase the success rate of the training. To avoid losing training progress and make full use of compute power, ensure that your code logic supports resumable training before enabling this function. For details, see Resumable Training and Incremental Training.</p> <p>The value ranges from 1 to 128. The default value is 3. The value cannot be changed once the training job is created. Set this parameter based on your needs.</p> <p>If Unconditional auto restart is selected, the training job will be restarted unconditionally once the system detects a training exception. To prevent invalid restarts, it supports a maximum of three consecutive unconditional restarts.</p> <p>If auto restart is triggered during training, the system records the restart information. You can check the fault recovery details on the training job details page. For details, see Viewing Fault Recovery Details.</p>

Configuring a Public Resource Pool

To configure a public resource pool, refer to [Table 4-5](#).

Table 4-5 Configuring a public resource pool for a training job

Parameter	Description
Resource Pool	Select Public resource pool .


Parameter	Description
Resource Type	<p>Select the resource type required for training. This parameter is mandatory. If a resource type has been defined in the training code, select a proper resource type based on algorithm constraints. For example, if the resource type defined in the training code is CPU and you select other types, the training fails. If some resource types are invisible or unavailable for selection, they are not supported.</p>
Specifications	<p>Select the required resource specifications based on the resource type.</p> <p>If Data path is selected for Input, you can click Check Input Size on the right to ensure the storage is larger than the input data size.</p>  <p>NOTICE The resource flavor GPU:n*tnt004 (<i>n</i> indicates a specific number) does not support multi-process training.</p>
Compute Nodes	<p>Select the number of compute nodes as required. The default value is 1.</p> <ul style="list-style-type: none"> • If only one compute node is used, a single-node training job is created. ModelArts starts one training container on this node. The training container exclusively uses the compute resources of the selected flavor. • If more than one compute nodes are used, a distributed training job is created. For more information about distributed training configurations, see Distributed Training Functions.
Persistent Log Saving	<p>If you select CPU or GPU flavors, Persistent Log Saving is available for you to set.</p> <ul style="list-style-type: none"> • This function is disabled by default. ModelArts automatically stores the logs for 30 days. You can download all logs on the job details page to a local path. • After this function is enabled, set Job Log Path. The system permanently stores training logs to the specified OBS path.
Job Log Path	<p>When enabling Persistent Log Saving, select an empty OBS directory for Job Log Path to store log files generated by the training job.</p> <p>Ensure that you have read and write permissions to the selected OBS directory.</p>

Parameter	Description
Event Notification	<p>Indicates whether to enable event notification.</p> <ul style="list-style-type: none"> • This function is disabled by default, which means SMN is disabled. • After this function is enabled, you will be notified of specific events, such as job status changes or suspected suspensions, via an SMS or email. Notifications will be billed based on SMN pricing. In this case, you must configure the topic name and events. <ul style="list-style-type: none"> – Topic: topic of event notifications. Click Create Topic to create a topic on the SMN console. – Event: events you want to subscribe to. Examples: JobStarted, JobCompleted, JobFailed, JobTerminated, and JobHanged. <p>NOTE</p> <ul style="list-style-type: none"> • After you create a topic on the SMN console, add a subscription to the topic, and confirm the subscription. Then, you will be notified of events. For details, see Adding a Subscription. • SMN charges you for the number of notification messages. For details, see Billing. • Only training jobs using GPUs support JobHanged events.
Auto Stop	<p>When using paid resources, you can determine whether to enable auto stop.</p> <ul style="list-style-type: none"> • This function is disabled by default, the training job keeps running until the training is completed. • If this function is enabled, configure the auto stop time. The value can be 1 hour, 2 hours, 4 hours, 6 hours, or Customize. The customized time must range from 1 hour to 720 hours. When you enable this function, the training stops automatically when the time limit is reached. The time limit does not count down when the training is paused.

Configuring a Dedicated Resource Pool

To configure a dedicated resource pool, refer to [Table 4-6](#).

Table 4-6 Configuring a dedicated resource pool for a training job

Parameter	Description
Resource Pool	<p>Select a dedicated resource pool.</p> <p>If you select a dedicated resource pool, you can view the status, node specifications, number of idle/fragmented nodes, number of available/total nodes, and number of cards of the resource pool. Hover over View in the Idle/Fragmented Nodes column to check fragment details and check whether the resource pool meets the training requirements.</p>
Specifications	<p>Select the required resource specifications based on the resource type.</p> <p>If Data path is selected for Input, you can click Check Input Size on the right to ensure the storage is larger than the input data size.</p>  <p>NOTICE The resource flavor GPU:n*tnt004 (<i>n</i> indicates a specific number) does not support multi-process training.</p>
Customized Specifications	<p>Indicates whether to enable customized specifications. You can customize resource specifications for training jobs based on dedicated resource pool specifications to improve resource pool utilization.</p> <ul style="list-style-type: none"> • This function is disabled by default, which means the dedicated resource pool specifications are used. • When you enable this function, jobs run with custom specifications. The custom specifications should not exceed the node specifications of the dedicated resource pool that you set. For CPU specifications, you can only customize the number of vCPUs and memory. For GPU specifications, you can customize the number of vCPUs, memory, and cards. <p>NOTE If customized specifications are enabled, the Specifications parameter is invalid.</p>

Parameter	Description
Compute Nodes	<p>Select the number of compute nodes as required. The default value is 1.</p> <ul style="list-style-type: none">• If only one compute node is used, a single-node training job is created. ModelArts starts one training container on this node. The training container exclusively uses the compute resources of the selected flavor.• If more than one compute nodes are used, a distributed training job is created. For more information about distributed training configurations, see Distributed Training Functions.
Job Priority	<p>When using a dedicated resource pool, you can set the priority of the training job. The value ranges from 1 to 3. The default priority is 1, and the highest priority is 3.</p> <ul style="list-style-type: none">• By default, the job priority can be set to 1 or 2. After the permission to set the highest job priority is configured, the priority can be set to 1 to 3.• If a training job is in the Pending state for a long time, you can change the job priority to reduce the queuing duration. For details, see Priority of a Training Job.

Parameter	Description
SFS Turbo	<p>When ModelArts and SFS Turbo are directly connected, multiple SFS Turbo file systems can be mounted to a training job to store training data. Click Add Mount Configuration and set the following parameters:</p> <ul style="list-style-type: none"> • File System: Select an SFS Turbo file system. • Mount Path: Enter the SFS Turbo mounting path in the training container. • Storage Location: Specify the SFS Turbo storage location. If you have configured the folder control permission, select a storage location. If you have not configured the folder control permission, retain the default value / or customize a location. • Mounting Mode: Permission on the mounted SFS Turbo file system. This parameter is displayed as Read/Write or Read-only based on the permission of the SFS Turbo storage location. If you have not configured the folder control permission, this parameter is unavailable. <p>NOTE</p> <ul style="list-style-type: none"> • A file system can be mounted only once and to only one path. Each mount path must be unique. A maximum of 8 disks can be mounted to a training job. • To mount an SFS Turbo file system to a training job, you need to configure network passthrough between ModelArts and the SFS Turbo file system. For details, see ModelArts Network. • The mounting path cannot be a / directory or a default mounting path, such as /cache and /home/ma-user/modelarts. • For details about how to set permissions for SFS Turbo folders, see Permissions Management.
Persistent Log Saving	<p>If you select CPU or GPU flavors, Persistent Log Saving is available for you to set.</p> <ul style="list-style-type: none"> • This function is disabled by default. ModelArts automatically stores the logs for 30 days. You can download all logs on the job details page to a local path. • After this function is enabled, set Job Log Path. The system permanently stores training logs to the specified OBS path.
Job Log Path	<p>When enabling Persistent Log Saving, select an empty OBS directory for Job Log Path to store log files generated by the training job.</p> <p>Ensure that you have read and write permissions to the selected OBS directory.</p>

Parameter	Description
Event Notification	<p>Indicates whether to enable event notification.</p> <ul style="list-style-type: none"> • This function is disabled by default, which means SMN is disabled. • After this function is enabled, you will be notified of specific events, such as job status changes or suspected suspensions, via an SMS or email. Notifications will be billed based on SMN pricing. In this case, you must configure the topic name and events. <ul style="list-style-type: none"> – Topic: topic of event notifications. Click Create Topic to create a topic on the SMN console. – Event: events you want to subscribe to. Examples: JobStarted, JobCompleted, JobFailed, JobTerminated, and JobHanged. <p>NOTE</p> <ul style="list-style-type: none"> • After you create a topic on the SMN console, add a subscription to the topic, and confirm the subscription. Then, you will be notified of events. For details, see Adding a Subscription. • SMN charges you for the number of notification messages. For details, see Billing. • Only training jobs using GPUs support JobHanged events.
Auto Stop	<p>When using paid resources, you can determine whether to enable auto stop.</p> <ul style="list-style-type: none"> • This function is disabled by default, the training job keeps running until the training is completed. • If this function is enabled, configure the auto stop time. The value can be 1 hour, 2 hours, 4 hours, 6 hours, or Customize. The customized time must range from 1 hour to 720 hours. When you enable this function, the training stops automatically when the time limit is reached. The time limit does not count down when the training is paused.

(Optional) Setting Tags

If you want to manage training jobs by group using tags, select **Configure Now** for **Advanced Configuration** to set tags for training jobs. For details about how to use tags, see [How Does ModelArts Use Tags to Manage Resources by Group?](#)

Follow-Up Procedure

After parameter setting for creating a training job, click **Submit**. On the **Confirm** dialog box, click **OK**.

A training job runs for a period of time. You can go to the training job list to view the basic information about the training job.

- In the training job list, **Status** of a newly created training job is **Pending**.
- When the status of a training job changes to **Completed**, the training job is finished, and the generated model is stored in the corresponding output path.
- If the status is **Failed** or **Abnormal**, click the job name to go to the job details page and view logs for troubleshooting.

 **NOTE**

You are billed for the resources you choose when your training job runs.

4.2 Viewing Training Job Details

1. Log in to the ModelArts management console.
2. In the navigation pane on the left, choose **Training Management > Training Jobs**.
3. In the training job list, click a job name to switch to the training job details page.
4. On the left of the training job details page, view basic job settings and algorithm parameters.
 - **Basic job settings**

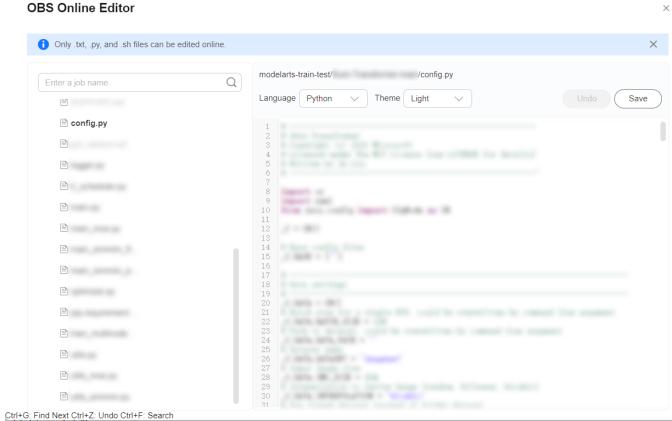
Table 4-7 Basic job settings

Parameter	Description
Job ID	Unique ID of a training job
Status	Training job status
Created	Time when the training job is created
Duration	Running duration of a training job
Retries	Number of times that a training job automatically restarts upon a fault during training. This parameter is available only when Auto Restart is enabled during training job creation.
Description	Description of a training job. You can click the edit icon to update the description of a training job.

- **Algorithm parameters**

Table 4-8 Algorithm parameters

Parameter	Description
Algorithm Name	Algorithm used in a training job You can click the algorithm name to go to the algorithm details page.
Preset images	Preset image used by a training job

Parameter	Description
Code Directory	<p>OBS path to the code directory of a training job</p> <p>You can click Edit Code on the right to edit the training script code in OBS Online Editor. OBS Online Editor is not available for a training job in the Pending, Creating, or Running status.</p>  <p>NOTE If you use the algorithm subscribed in AI Gallery to create a training job, then this parameter is not supported.</p>
Boot File	<p>Location where a boot file is stored.</p> <p>NOTE If you use the algorithm subscribed in AI Gallery to create a training job, then this parameter is not supported.</p>
User ID	ID of the user who runs the container.
Local Code Directory	Path to the training code in the training container
Work Directory	Path to the training startup file in the training container
Compute Nodes	Number of compute nodes
Dedicated resource pool	Dedicated resource pool information. This parameter is available only when a training job uses a dedicated resource pool.
Specifications	Training specifications used in a training job
Input - Input Path	OBS path where the input data is stored
Input - Parameter Name	Algorithm code parameter specified by the input path

Parameter	Description
Input - Obtained from	Method of obtaining the training job input.
Input - Local Path (Training Parameter Value)	Path for storing the input data in the ModelArts backend container. After the training is started, ModelArts downloads the data stored in OBS to the backend container.
Output - Output Path	OBS path where the output data is stored
Output - Parameter Name	Algorithm code parameter specified by the output path
Output - Obtained from	Method of obtaining the training job output.
Output - Local Path (Training Parameter Value)	Path for storing the output data in the ModelArts backend container
Hyperparameter	Hyperparameters used in a training job
Environment Variable	Environment variables for a training job

4.3 Viewing Training Job Events

Any key event of a training job will be recorded at the backend after the training job is displayed for you. You can check events on the training job details page.

This helps you better understand the running process of a training job and locate faults more accurately when a task exception occurs. The following job events are supported:

- Training job created.
- Training job failures:
- Preparations timed out. The possible cause is that the cross-region algorithm synchronization or creating shared storage timed out.
- The training job is queuing and awaiting resource allocation.
- Failed to be queued.
- The training job starts to run.
- Training job executed.

- Failed to run the training job.
- The training job is preempted.
- The system detects that your training job may be suspended. Go to the job details page to view the cause and handle the issue.
- The training job has been restarted.
- The training job has been manually stopped.
- The training job has been stopped. (Maximum running duration: 1 hour)
- The training job has been stopped. (Maximum running duration: 3 hours)
- The training job has been manually deleted.
- Billing information synchronized.
- [worker-0] The training environment is being pre-checked.
- [worker-0] [Duration: second] Pre-check completed.
- [worker-0] [Duration: second] Pre-check failed. Error: xxx
- [worker-0] [Duration: second] Pre-check failed. Error: xxx
- [worker-0] The training code is being downloaded.
- [worker-0] [Duration: second] Training code downloaded.
- [worker-0] [Duration: second] Failed to download the training code. Failure cause:
- [worker-0] The training input is being downloaded.
- [worker-0] [Duration: second] Training input (parameter: xxx) downloaded.
- [worker-0] [Duration: second] Failed to download the training input (parameter: xxx). Failure cause:
- [worker-0] Python dependency packages are being installed. Import the following files:
- [worker-0] [Duration: second] Python dependency packages installed. Import the following files:
- [worker-0] The training job starts to run.
- [worker-0] Training job executed.
- [worker-0] The training input is being uploaded.
- [worker-0] [Duration: second] Training output (parameter: xxx) uploaded.

During the training process, key events can be manually or automatically refreshed.

Procedure

1. On the ModelArts console, choose **Training Management > Training Jobs** from the navigation pane.
2. In the training job list, click the name of the target job to go to the training job details page.
3. Click **Events** to view events.

Figure 4-1 Events

Event Type	Event Message	Occurred
Normal	[Job: ma-job-454923b8-1a1e-472e-abff-d0e77e930758] WorkloadDispatcherDelete: successfully delete...	2024/01/29 09:52:20 GMT+08...
Normal	Training job completed.	2024/01/29 09:37:19 GMT+08...
Normal	[Job: ma-job-454923b8-1a1e-472e-abff-d0e77e930758] WorkloadStatusChanged: workload status was ...	2024/01/29 09:37:19 GMT+08...
Normal	[Job: ma-job-454923b8-1a1e-472e-abff-d0e77e930758] WorkloadStatusChanged: workload status was ...	2024/01/29 09:37:18 GMT+08...
Normal	[Job: ma-job-454923b8-1a1e-472e-abff-d0e77e930758] ExecuteAction: Start to execute action Complet...	2024/01/29 09:37:18 GMT+08...
Normal	[worker-0][time used: 0.133s] Upload training output(parameter name: MODEL_OUTPUT_INFER_PAT...	2024/01/29 09:37:11 GMT+08...
Normal	[worker-0] Training output(parameter name: MODEL_OUTPUT_INFER_PATH) Uploading.	2024/01/29 09:37:11 GMT+08...
Normal	[worker-0] Training finished. Exit code 0.	2024/01/29 09:37:05 GMT+08...
Normal	[worker-0] training started.	2024/01/29 09:22:41 GMT+08...
Normal	[worker-0][time used: 3.653548715ss] The training environment self-test is completed.	2024/01/29 09:22:40 GMT+08...

4.4 Training Job Logs

4.4.1 Introduction to Training Job Logs

Overview

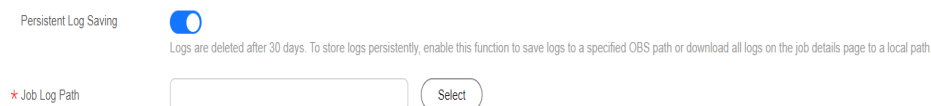
Training logs record the runtime process and exception information of training jobs and provide useful details for fault location. The standard output and standard error information in your code are displayed in training logs. If you encounter an issue during the execution of a ModelArts training job, view logs first. In most scenarios, you can locate the issue based on the error information reported in logs.

Retention Period

Logs are classified into the following types based on the retention period:

- Real-time logs: generated during training job running and can be viewed on the ModelArts training job details page.
- Historical logs: After a training job is completed, you can view its historical logs on the ModelArts training job details page. ModelArts automatically stores the logs for 30 days.
- Permanent logs: These logs are dumped to your OBS bucket. When creating a training job, you can enable persistent log saving and set a job log path for dumping.

Figure 4-2 Enabling Persistent Log Saving



Real-time logs and historical logs have no difference in content.

Related Chapters

- On the ModelArts training job details page, you can preview logs, download logs, and search for logs by keyword in the log pane. For details, see [Viewing Training Job Logs](#).
- ModelArts also enables you to quickly locate and rectify training faults. For details, see [Locating Faults by Analyzing Training Logs](#).

4.4.2 Common Logs

Common logs include the logs for **pip-requirement.txt**, training process, and ModelArts.

Log Type

Table 4-9 Log type

Type	Description
Training process log	Standard output of your training code
Installation logs for pip-requirement.txt	If pip-requirement.txt is defined in training code, PIP package installation logs are generated.
ModelArts logs	ModelArts logs are used by O&M personnel to locate service faults.

File Format

The format of a common log file is as follows. **task id** is the node ID of a training job.

Unified log format: modelarts-job-[job id]-[task id].log

Example: log/modelarts-job-95f661bd-1527-41b8-971c-eca55e513254-worker-0.log

- Single-node training jobs generate a log file, and **task id** defaults to **worker-0**.
- Distributed training generates multiple node log files, which are distinguished by **task id**, such as **worker-0** and **worker-1**.

Common logs include the logs for **pip-requirement.txt**, training process, and ModelArts.

ModelArts Logs

ModelArts logs can be filtered in the common log file **modelarts-job-[job id]-[task id].log** using the following keywords: **[ModelArts Service Log]** or **Platform=ModelArts-Service**.

- Type 1: **[ModelArts Service Log]** xxx
[ModelArts Service Log][init] download code_url: s3://dgg-test-user/snt9-test-cases/mindspore/lenet/

- Type 2: 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 Viewing Training Job Logs

On the training job details page, you can preview logs, download logs, search for logs by keyword, and filter system logs in the log pane.

- Previewing logs
You can preview training logs on the system log pane. If multiple compute nodes are used, you can choose the target node from the drop-down list on the right.

Figure 4-3 Viewing logs of different compute nodes



If a log file is oversized, the system displays only the latest logs in the log pane. To view all logs, click the link in the upper part of the log pane, which will direct you to a new page. Then you will be redirected to a new page.

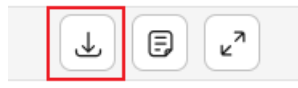
Figure 4-4 Viewing all logs



NOTE

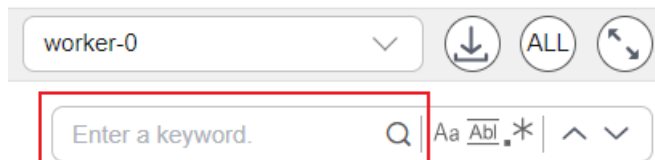
- If the total size of all logs exceeds 500 MB, the log page may be frozen. In this case, download the logs to view them locally.
- A log preview link can be accessed by anyone within one hour after it is generated. You can share the link with others.
- **Ensure that no privacy information is contained in the logs. Otherwise, information leakage may occur.**
- Downloading logs
Training logs are retained for only 30 days. To permanently store logs, click the download icon in the upper right corner of the log pane. You can download the logs of multiple compute nodes in a batch. You can also enable **Persistent Log Saving** and set a log path when you create a training job. In this way, the logs will be automatically stored in the specified OBS path.
If a training job is created on Ascend compute nodes, certain system logs cannot be downloaded in the training log pane. To obtain these logs, go to the **Job Log Path** you set when you created the training job.

Figure 4-5 Downloading logs



- Searching for logs by keyword
In the upper right corner of the log pane, enter a keyword in the search box to search for logs, as shown in [Figure 4-6](#).

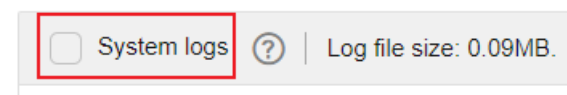
Figure 4-6 Searching for logs by keyword



The system will highlight the keyword and redirect you between search results. Only the logs loaded in the log pane can be searched for. If the logs are not fully displayed (see the message displayed on the page), obtain all the logs by downloading them or clicking the full log link and then search for the logs. On the page redirected by the full log link, press **Ctrl+F** to search for logs.

- Filtering system logs

Figure 4-7 System logs



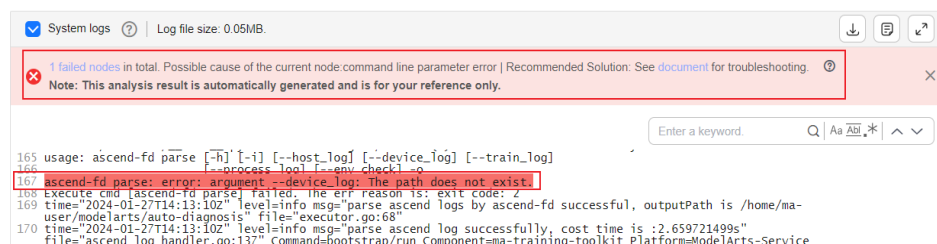
If **System logs** is selected, system logs and user logs are displayed. If **System logs** is deselected, only user logs are displayed.

4.4.4 Locating Faults by Analyzing Training Logs

If you encounter an issue during the execution of a ModelArts training job, view logs first. In most scenarios, you can locate the issue based on the error information reported in logs.

If a training job fails, ModelArts automatically identifies the failure cause and displays a message on the log page. The message consists of possible causes, recommended solutions, and error logs (marked in red).

Figure 4-8 Identifying training faults



ModelArts provides possible causes (for reference only) and solutions for some common training faults. Not all faults can be identified. For a distributed job, only the analysis result of the current node is displayed. To obtain the failure cause of a training job, check the analysis results of all nodes used by the training job.

To rectify common training faults, perform the following steps:

1. Rectify the fault based on the analysis and suggestions provided on the log page.
 - Solution 1: A troubleshooting document is provided for you to follow.
 - Solution 2: Rebuild the training job and run it again.
2. If the fault persists, analyze the error information in the logs to locate and rectify the fault.
3. If the provided solutions cannot rectify your fault, you can submit a service ticket for technical support.

4.5 Cloud Shell

4.5.1 Logging In to a Training Container Using Cloud Shell

Application Scenario

You can use Cloud Shell provided by the ModelArts console to log in to a running training container.

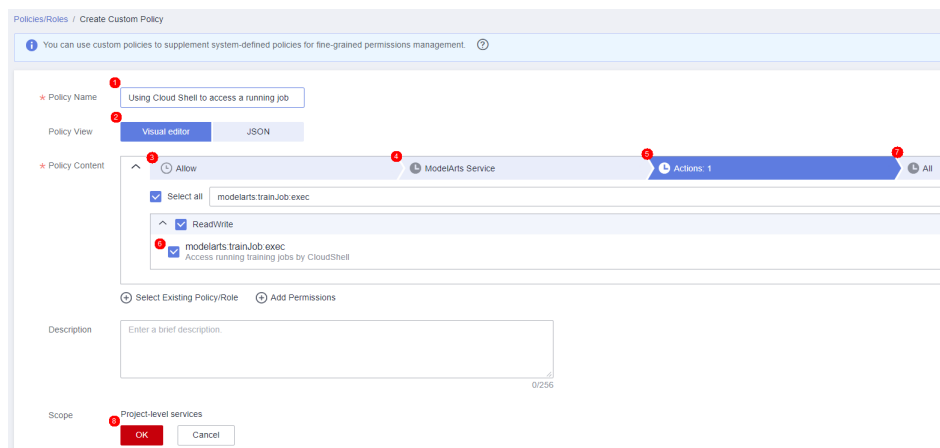
Constraints

Only dedicated resource pools support Cloud Shell. The training job must be in the **Running** state.

Preparation: Assigning the Cloud Shell Permission to an IAM User

1. Log in to the Huawei Cloud management console as a tenant user, hover the cursor over your username in the upper right corner, and choose **Identity and Access Management** from the drop-down list to switch to the IAM management console.
2. On the IAM console, choose **Permissions > Policies/Roles** from the navigation pane, click **Create Custom Policy** in the upper right corner, and configure the following parameters.
 - **Policy Name:** Enter a custom policy name, for example, **Using Cloud Shell to access a running job**.
 - **Policy View:** Select **Visual editor**.
 - **Policy Content:** Select **Allow, ModelArts Service, modelarts:trainJob:exec**, and default resources.

Figure 4-9 Creating a custom policy



3. In the navigation pane, choose **User Groups**. Then, click **Authorize** in the **Operation** column of the target user group. On the **Authorize User Group** page, select the custom policies created in 2, and click **Next**. Then, select the scope and click **OK**.

After the configuration, all users in the user group have the permission to use Cloud Shell to log in to a running training container.

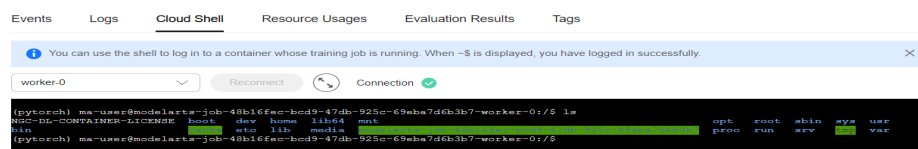
If no user group is available, create a user group, add users using the user group management function, and configure authorization. If the target user is not in a user group, you can add the user to a user group through the user group management function.

Using Cloud Shell

1. Configure parameters based on [Preparation: Assigning the Cloud Shell Permission to an IAM User](#).
2. On the ModelArts console, choose **Training Management > Training Jobs** from the navigation pane.
3. In the training job list, click the name of the target job to go to the training job details page.
4. On the training job details page, click the **Cloud Shell** tab and log in to the training container.

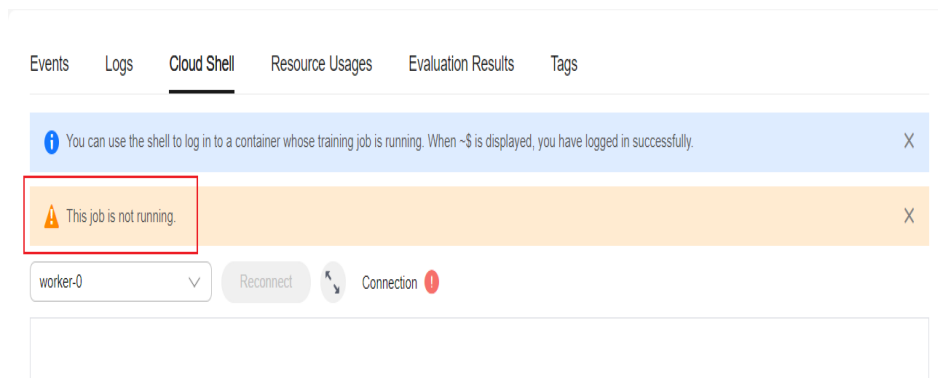
Verify that the login is successful, as shown in the following figure.

Figure 4-10 Cloud Shell page



If the job is not running or the permission is insufficient, Cloud Shell cannot be used. In this case, locate the fault as prompted.

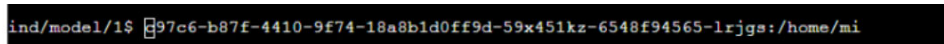
Figure 4-11 Error message



NOTE

A path display exception may occur when you log in to the Cloud Shell page. In this case, press **Enter** to rectify the fault.

Figure 4-12 Abnormal path



4.5.2 Keeping a Training Job Running

You can only log in to Cloud Shell when the training job is in **Running** state. This section describes how to log in to a running training container through Cloud Shell.

Using the sleep Command

- For training jobs using a preset image
When creating a training job, set **Algorithm Type** to **Custom algorithm** and **Boot Mode** to **Preset image**, add **sleep.py** to the code directory, and use the script as the boot file. The training job keeps running for 60 minutes. You can access the container through Cloud Shell for debugging.

Example of sleep.py

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

Figure 4-13 Using a preset image

★ Algorithm Type: Custom algorithm (selected), My algorithm, My subscription

★ Boot Mode: Preset image (selected), Custom image

★ Code Directory: /modelarts-train/ (with Select button)

★ Boot File: /modelarts-train/sleep.py (with Select button, highlighted with a red box)

Local Code Directory: /home/ma-user/ (with Select button)

Work Directory: (with Select button)

- For training jobs using a custom image
When creating a training job, set **Algorithm Type** to **Custom algorithm** and **Boot Mode** to **Custom image**, and enter **sleep 60m** in **Boot Command**. The training job keeps running for 60 minutes. You can access the container through Cloud Shell for debugging.

Figure 4-14 Using a custom image

★ Algorithm Type: Custom algorithm (selected), My algorithm, My subscription

★ Boot Mode: Preset image, Custom image (selected)

★ Image: (with Select button)

Code Directory: (with Select button)

User ID: 1000

★ Boot Command: 1 sleep 60m (highlighted with a red box)

Keeping a Failed Job Running

When creating a training job, add **|| sleep 5h** at the end of the boot command and start the training job. Run the following command:

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

If the training fails, the **sleep** command is executed. In this case, you can log in to the container image through Cloud Shell for debugging.

 NOTE

To debug a multi-node training job in Cloud Shell, you need to switch between worker-0 and worker-1 in Cloud Shell and run the boot command on each node. Otherwise, the task will wait for other nodes to join.

4.5.3 Preventing Cloud Shell Session from Disconnection

To run a job for a long time, you can use the **screen** command to prevent the job from failing due to disconnection.

1. If **screen** is not installed in the image, run **apt-get install screen** to install it.
2. Create a screen terminal.
Use **-S** to create a screen terminal named **name**.

```
screen -S name
```
3. View the created screen terminals.

```
screen -ls
```

There are screens on:

2433.pts-3.linux	(2013-10-20 16:48:59)	(Detached)
2428.pts-3.linux	(2013-10-20 16:48:05)	(Detached)
2284.pts-3.linux	(2013-10-20 16:14:55)	(Detached)
2276.pts-3.linux	(2013-10-20 16:13:18)	(Detached)

4 Sockets in /var/run/screen/S-root.
4. Connect to the screen terminal whose **screen_id** is **2276**.

```
screen -r 2276
```
5. Press **Ctrl+A+D** to exit the screen terminal. After the exit, the screen session is still active and can be reconnected at any time.

For details about how to use screens, see [Screen User's Manual](#).

4.6 Viewing the Resource Usage of a Training Job

Operations

1. On the ModelArts console, choose **Training Management > Training Jobs** from the navigation pane.
2. In the training job list, click the name of the target job to go to the training job details page.
3. On the training job details page, click the **Resource Usages** tab to view the resource usage of the compute nodes. The data of at most the last three days can be displayed. When the resource usage window is opened, the data is loading and refreshed periodically.

Operation 1: If a training job uses multiple compute nodes, choose a node from the drop-down list box to view its metrics.

Operation 2: Click **cpuUsage**, **gpuMemUsage**, **gpuUtil**, **memUsage**, **npuMemUsage**, or **npuUtil** to show or hide the usage chart of the parameter.

Operation 3: Hover the cursor on the graph to view the usage at the specific time.

Figure 4-15 Resource Usages

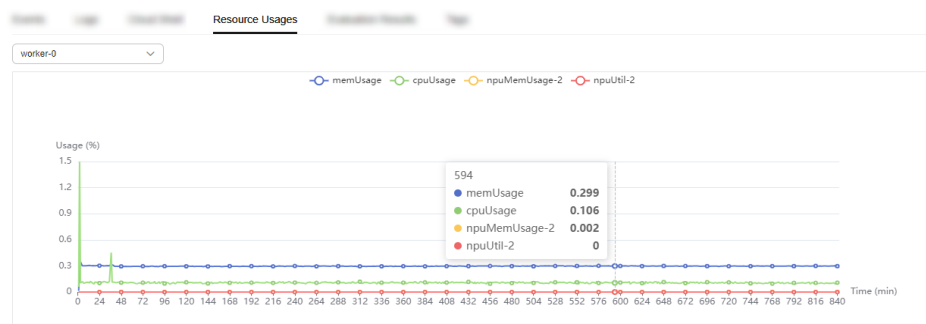


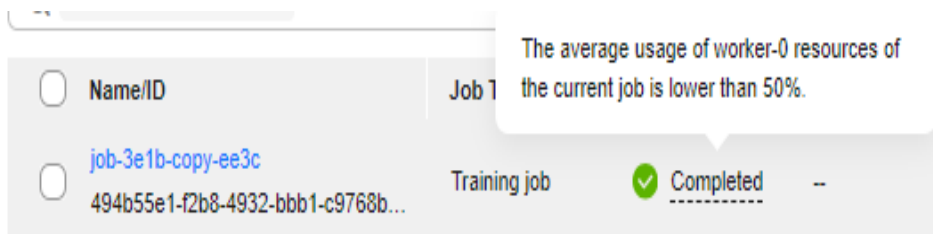
Table 4-10 Parameters

Parameter	Description
cpuUsage	CPU usage
gpuMemUsage	GPU memory usage
gpuUtil	GPU usage
memUsage	Memory usage
npuMemUsage	NPU memory usage
npuUtil	NPU usage

Alarms of Job Resource Usage

You can view the job resource usage on the training job list page. If the average GPU/NPU usage of the job's worker-0 instance is lower than 50%, an alarm is displayed in the training job list.

Figure 4-16 Job resource usage in the job list



The job resource usage here involves only GPU and NPU resources. The method of calculating the average GPU/NPU usage of a job's worker-0 instance is: Summarize the usage of each GPU/NPU accelerator card at each time point of the job's worker-0 instance and calculate the average value.

Improving Job Resource Utilization

- Increasing the value of **batch_size** increases GPU and NPU usage. You must decide the batch size that will not cause a memory overflow.
- If the time for reading data in a batch is longer than the time for GPUs or NPUs to calculate data in a batch, GPU or NPU usage may fluctuate. In this case, optimize the performance of data reading and data augmentation. For example, read data in parallel or use tools such as NVIDIA Data Loading Library (DALI) to improve the data augmentation speed.
- If a model is large and frequently saved, GPU or NPU usage is affected. In this case, do not save models frequently. Similarly, make sure that other non-GPU/NPU operations, such as log printing and training metric saving, do not affect the training process for too much time.

4.7 Evaluation Results

After a training job has been executed, ModelArts evaluates your model and provides optimization diagnosis and suggestions.

- When you use a built-in algorithm to create a training job, you can view the evaluation result without any configurations. The system automatically provides optimization suggestions based on your model metrics. Read the suggestions and guidance on the page carefully to further optimize your model.
- For a training job created by writing a training script or using a custom image, you need to add the evaluation code to the training code so that you can view the evaluation result and diagnosis suggestions after the training job is complete.

NOTE

- Only validation sets of the image type are supported.
- You can add the evaluation code only when the training scripts of the following frequently-used frameworks are used:
 - TF-1.13.1-python3.6
 - TF-2.1.0-python3.6
 - PyTorch-1.4.0-python3.6

This section describes how to use the evaluation code in a training job. To adapt and modify the training code, three steps are involved, [Adding the Output Path](#), [Copying the Dataset to the Local Host](#), and [Mapping the Dataset Path to OBS](#).

Adding the Output Path

The code for adding the output path is simple. That is, add a path for storing the evaluation result file to the code, which is called **train_url**, that is, the training output path on the console. Add **train_url** to the analysis function and use **save_path** to obtain **train_url**. The sample code is as follows:

```
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)

```

Copying the Dataset to the Local Host

Copying a dataset to the local host is to prevent the OBS connection from being interrupted due to long-time access. Therefore, copy the dataset to the local host before performing operations.

There are two methods for copying datasets. The recommended method is to use the OBS path.

- OBS path (recommended)
Call the `copy_parallel` API of MoXing to copy the corresponding OBS path.
- Dataset in ModelArts data management (manifest file format)
Call the `copy_manifest` API of MoXing to copy the file to the local host and obtain the path of the new manifest file. Then, use SDK to parse the new manifest file.

NOTE

ModelArts data management is being upgraded and is invisible to users who have not used data management. It is recommended that new users store their training data in OBS buckets.

```

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 -----')

```

Mapping the Dataset Path to OBS

The actual path of the image file, that is, the OBS path, needs to be entered in the JSON body. Therefore, after analysis and evaluation are performed on the local host, the original local dataset path needs to be mapped to the OBS path, and the new list needs to be sent to the analysis API.

If the OBS path is used as the input of `data_url`, you only need to replace the string of the local path.

```

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)

```

If the manifest file is used, the original manifest file needs to be parsed again to obtain the list and then the list is sent to the analysis API.

```
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])
```

An example code for image classification that can be used to create training jobs is as follows:

```
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 evaluation():
    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__":
    evaluation()
```

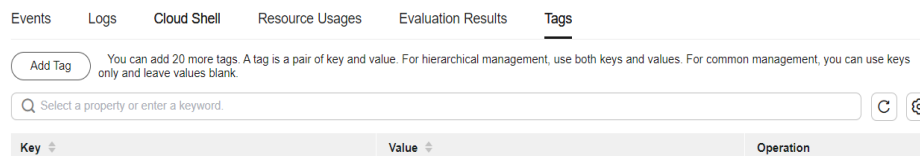
4.8 Viewing Training Tags

You can add tags to a training job for quick search.

1. On the ModelArts console, choose **Training Management > Training Jobs** from the navigation pane.
2. In the training job list, click the name of the target job to go to the training job details page.
3. Click **Tags**.

Tags can be added, modified, and deleted. For details about how to use tags, see [How Does ModelArts Use Tags to Manage Resources by Group?](#)

Figure 4-17 Viewing training tags



NOTE

You can add up to 20 tags to a training job.

4.9 Viewing Fault Recovery Details

When a training job fault occurs (such as process-level recovery, POD-level rescheduling, and job-level rescheduling), the **Fault Recovery Details** tab appears on the job details page, recording the start and stop details of the training job.

1. On the ModelArts console, choose **Training Management > Training Jobs** from the navigation pane.
2. In the training job list, click the name of the target job to go to the training job details page.
3. On the training job details page, click the **Fault Recovery Details** tab to view the fault recovery information.

Figure 4-18 Viewing fault recovery details

Scheduling	Started	Duration	Fault Source	Recovery Action Upon End	Degradation
Yes	Mar 12, 2024 12:56:04 GMT+08:00	00:07:17	worker-0	Isolated job-level rescheduling	No

4.10 Viewing Environment Variables of a Training Container

What Is an Environment Variable

This section describes environment variables preset in a training container. The environment variables include:

- Path environment variables
- Environment variables of a distributed training job
- Nvidia Collective multi-GPU Communication Library (NCCL) environment variables
- OBS environment variables
- Environment variables of the PIP source

- Environment variables of the API Gateway address
- Environment variables of job metadata

Configuring Environment Variables

When you create a training job, you can add environment variables or modify environment variables preset in the training container.

Environment Variables Preset in a Training Container

The following tables list environment variables preset in a training container, including [Table 4-11](#), [Table 4-12](#), [Table 4-13](#), [Table 4-14](#), [Table 4-15](#), [Table 4-16](#), and [Table 4-17](#).

The environment variable values are examples.

Table 4-11 Path environment variables

Variable	Description	Example
PATH	Executable file paths	PATH=/usr/local/bin:/usr/local/cuda/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
LD_LIBRARY_PATH	Dynamic load library paths	LD_LIBRARY_PATH=/usr/local/secomponent/lib:/usr/local/cuda/lib64:/usr/local/cuda/compat:/root/miniconda3/lib:/usr/local/lib:/usr/local/nvidia/lib64
LIBRARY_PATH	Static library paths	LIBRARY_PATH=/usr/local/cuda/lib64/stubs
MA_HOME	Main directory of a training job	MA_HOME=/home/ma-user
MA_JOB_DIR	Parent directory of the training algorithm folder	MA_JOB_DIR=/home/ma-user/modelarts/user-job-dir
MA_MOUNT_PATH	Path mounted to a ModelArts training container, which is used to temporarily store training algorithms, algorithm input, algorithm output, and logs	MA_MOUNT_PATH=/home/ma-user/modelarts
MA_LOG_DIR	Training log directory	MA_LOG_DIR=/home/ma-user/modelarts/log
MA_SCRIPT_INTERPRETER	Training script interpreter	MA_SCRIPT_INTERPRETER=

Variable	Description	Example
WORKSPACE	Training algorithm directory	WORKSPACE=/home/ma-user/modelarts/user-job-dir/code

Table 4-12 Environment variables of a distributed training job

Variable	Description	Example
MA_CURRENT_IP	IP address of a job container.	MA_CURRENT_IP=192.168.23.38
MA_NUM_GPUS	Number of accelerator cards in a job container.	MA_NUM_GPUS=8
MA_TASK_NAME	Name of a job container, for example: <ul style="list-style-type: none"> • worker in MindSpore and PyTorch. • learner or worker in reinforcement learning engines. • ps or worker in TensorFlow. 	MA_TASK_NAME=worker
MA_NUM_HOSTS	Number of compute nodes, which is automatically obtained from Compute Nodes .	MA_NUM_HOSTS=4
VC_TASK_INDEX	Container index, starting from 0 . This parameter is invalid for single-node training. In multi-node training jobs, you can use this parameter to determine the algorithm logic of the container.	VC_TASK_INDEX=0
VC_WORKER_NUM	Compute nodes required for a training job.	VC_WORKER_NUM=4

Variable	Description	Example
VC_WORKER_HOSTS	Domain name of each node for multi-node training. Use commas (,) to separate the domain names in sequence. You can obtain the IP address through domain name resolution.	VC_WORKER_HOSTS=modelarts-job-a0978141-1712-4f9b-8a83-000000000000-worker-0.modelarts-job-a0978141-1712-4f9b-8a83-000000000000,modelarts-job-a0978141-1712-4f9b-8a83-000000000000-worker-1.ob-a0978141-1712-4f9b-8a83-000000000000,modelarts-job-a0978141-1712-4f9b-8a83-000000000000-worker-2.modelarts-job-a0978141-1712-4f9b-8a83-00000000,ob-a0978141-1712-4f9b-8a83-000000000000-worker-3.modelarts-job-a0978141-1712-4f9b-8a83-00000000
<pre> \$ {MA_VJ_NAME} - \${MA_TASK_NAME}-N.\$ {MA_VJ_NAME} </pre>	<p>Communication domain name of a node. For example, the communication domain name of node 0 is \$ {MA_VJ_NAME}-\${MA_TASK_NAME}-0.\$ {MA_VJ_NAME}.</p> <p>N indicates the number of compute nodes.</p>	<p>For example, if there are four compute nodes, the environment variables are as follows:</p> <pre> \${MA_VJ_NAME}- \${MA_TASK_NAME}-0.\$ {MA_VJ_NAME} \${MA_VJ_NAME}- \${MA_TASK_NAME}-1.\$ {MA_VJ_NAME} \${MA_VJ_NAME}- \${MA_TASK_NAME}-2.\$ {MA_VJ_NAME} \${MA_VJ_NAME}- \${MA_TASK_NAME}-3.\$ {MA_VJ_NAME} </pre>

Table 4-13 NCCL environment variables

Variable	Description	Example
NCCL_VERSION	NCCL version	NCCL_VERSION=2.7.8
NCCL_DEBUG	NCCL log level	NCCL_DEBUG=INFO
NCCL_IB_HCA	InfiniBand NIC to use for communication	NCCL_IB_HCA=^mlx5_bond_0

Variable	Description	Example
NCCL_SOCKET_IFNAME	IP interface to use for communication	NCCL_SOCKET_IFNAME=bond0, eth0

Table 4-14 OBS environment variables

Variable	Description	Example
S3_ENDPOINT	OBS endpoint	-
S3_VERIFY_SSL	Whether to use SSL to access OBS	S3_VERIFY_SSL=0
S3_USE_HTTPS	Whether to use HTTPS to access OBS	S3_USE_HTTPS=1

Table 4-15 Environment variables of the PIP source and API Gateway address

Variable	Description	Example
MA_PIP_HOST	Domain name of the PIP source	MA_PIP_HOST=repo.myhuaweicloud.com
MA_PIP_URL	Address of the PIP source	MA_PIP_URL=http://repo.myhuaweicloud.com/repository/pypi/simple/
MA_APIGW_ENDPOINT	ModelArts API Gateway address	MA_APIGW_ENDPOINT=https://modelarts.region.cn-east-3.myhuaweicloud.com

Table 4-16 Environment variables of job metadata

Variable	Description	Example
MA_CURRENT_INSTANCE_NAME	Name of the current node for multi-node training	MA_CURRENT_INSTANCE_NAME=modelarts-job-a0978141-1712-4f9b-8a83-0000000000-worker-1

Table 4-17 Precheck environment variables

Variable	Description	Example
MA_SKIP_IMAGE_DETECT	Whether to enable ModelArts precheck. The default value is 1 , which indicates that the pre-check is enabled; the value 0 indicates that the pre-check is disabled. It is a good practice to enable precheck to detect node and driver faults before they affect services.	1

4.11 Stopping, Rebuilding, or Searching for a Training Job

Saving As an Algorithm

To modify the algorithm of a training job, click **Save As Algorithm** in the upper right corner of the training job details page.

On the **Algorithms** page, the algorithm parameters for the last training job are automatically set. You can modify the settings.

 **NOTE**

This function is not supported for algorithms subscribed in AI Gallery.

Stopping a Training Job

In the training job list, click **Stop** in the **Operation** column of a training job that is in creating, pending, or running state to stop the job.

After a training job is stopped, its billing stops on ModelArts.

A training job in completed, failed, terminated, or abnormal state cannot be stopped.

Rebuilding a Training Job

If you are not satisfied with a created training job, click **Rebuild** in the **Operation** column to rebuild it. The page for creating a training job is displayed. On this page, the parameter settings for the previous training job are automatically retained. You only need to modify certain parameter settings.

Searching for a Training Job

If you log in to ModelArts using an IAM account, all training jobs under this account are displayed in the training job list. To quickly search for a training job, use the following methods:

Method 1: Click **Only my jobs**. Then, only jobs created under the current IAM user account are displayed in the training job list.

Method 2: Search for jobs by name, ID, job type, status, creation time, algorithm, and resource pool.

Method 3: Click the refresh button in the upper right corner of the job list to refresh it.

Method 4: Configure the custom columns and other basic settings.

Figure 4-19 Searching for a training job



4.12 Releasing Training Job Resources

Release resources of a training job when not in use to avoid unnecessary charges.

- On the **Training Jobs** page, click **Delete** in the **Operation** column. In the displayed dialog box, click **OK** to delete the training job.
- Go to OBS and delete the OBS bucket and files used by the training job.

After the resources are released, check the resource usage on the **Dashboard** page.

Figure 4-20 Checking the resource usage

Usage Details ↔

Text Classification - ExeML		Image Classification - ExeML		Training Jobs-Beta New - Trai...		AI Application Management		Real-Time Services - Service ...	
Running	Projects	Running	Projects	Running	Projects	AI App...	AI Versions	Running	Services
0	1	0	1	0	23	20	20	0	1

5 Advanced Training Operations

5.1 Automatic Recovery from a Training Fault

5.1.1 Training Fault Tolerance Check

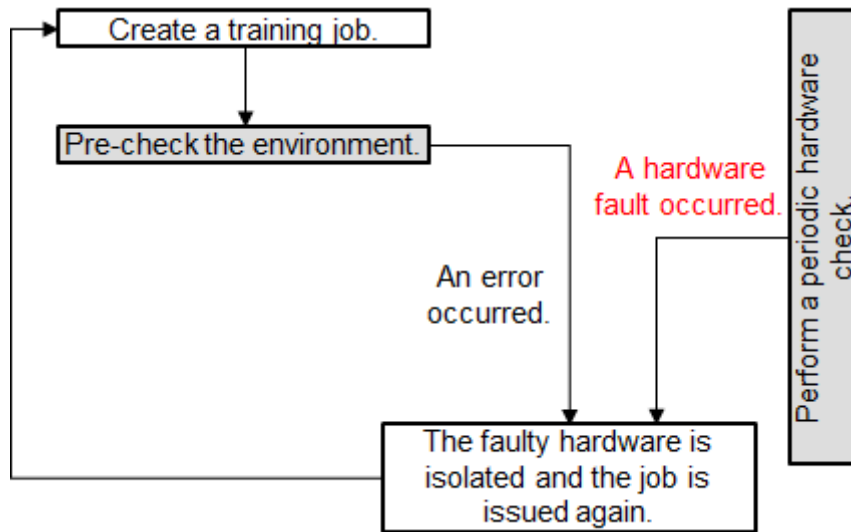
During model training, a training failure may occur due to a hardware fault. For hardware faults, ModelArts provides fault tolerance check to isolate faulty nodes to improve user experience in training.

The fault tolerance check involves environment pre-check and periodic hardware check. If any fault is detected during either of the checks, ModelArts automatically isolates the faulty hardware and issues the training job again. In distributed training, the fault tolerance check will be performed on all compute nodes used by the training job.

The following shows four failure scenarios, among which the failure in scenario 4 is not caused by a hardware fault. You can enable fault tolerance in the other three scenarios to automatically resume the training job.

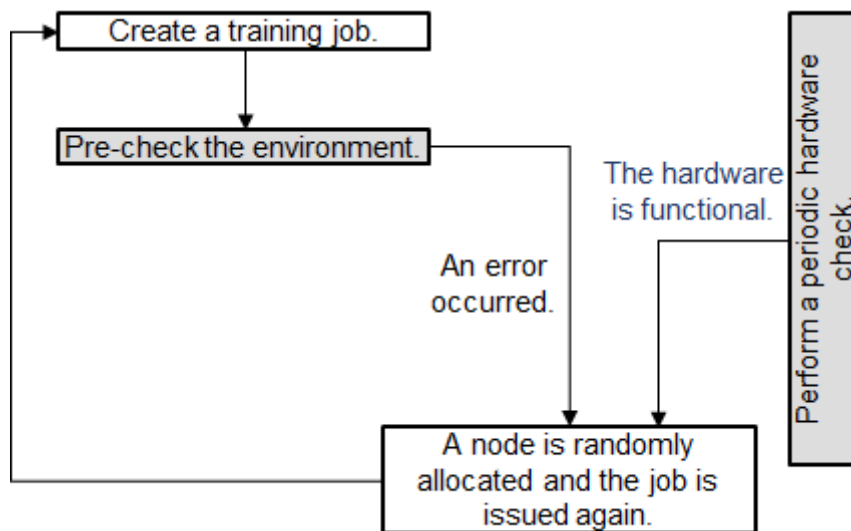
- Scenario 1: The environment pre-check fails, and the hardware is faulty. Then, ModelArts automatically isolates all faulty nodes and issues the training job again.

Figure 5-1 Pre-check failure and hardware fault



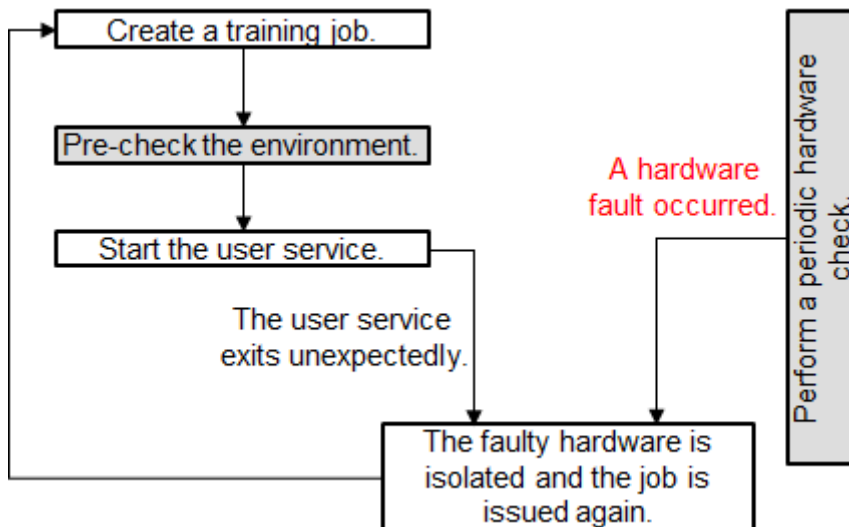
- Scenario 2: The environment pre-check fails but the hardware is functional. Then, ModelArts randomly allocates nodes and issues the training job again.

Figure 5-2 Pre-check failure but functional hardware



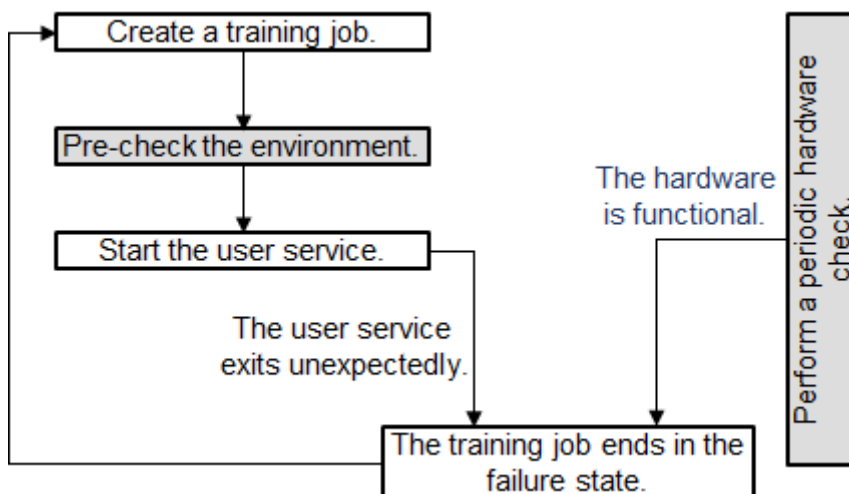
- Scenario 3: The environment pre-check is successful and the user service starts. A hardware fault occurs and the user service exits unexpectedly. Then, ModelArts automatically isolates all faulty nodes and issues the training job again.

Figure 5-3 Service failure and hardware fault



- Scenario 4: The environment pre-check is successful and the user service starts. The hardware is functional. A fault occurs in the user service, the training job ends in the failure state.

Figure 5-4 Service failure and functional hardware



After the faulty node is isolated, ModelArts creates a training job on new compute nodes. If the resources provided by the resource pool are limited, the re-issued training job will be queued with the highest priority. If the waiting time exceeds 30 minutes, the training job will automatically exit. This indicates that the resources are so limited that the training job cannot start. In this case, buy a dedicated resource pool to obtain dedicated resources.

If you use a dedicated resource pool to create a training job, the faulty nodes identified during the fault tolerance check will be removed. The system automatically adds healthy compute nodes to the dedicated resource pool. (This function is coming soon.)

More details of a fault tolerance check:

1. [Enabling Fault Tolerance Check](#)
2. [Check Items and Conditions](#)
3. [Effect of a Fault Tolerance Check](#)
4. After the environment pre-check is successful, any hardware fault will interrupt the user service. Add the reload ckpt code logic to the training so that the pre-trained model saved before the training is interrupted can be obtained. For details, see [Resumable Training and Incremental Training](#).

Enabling Fault Tolerance Check

To enable fault tolerance check, enable auto restart when creating a training job.

- Configure fault tolerance check on the ModelArts management console:
Enable **Auto Restart** on the ModelArts management console. **Auto Restart** is disabled by default, indicating that the job will not be re-issued and the environment pre-check will not be enabled. After **Auto Restart** is enabled, the number of restart retries ranges from 1 to 128.

Figure 5-5 Auto Restart



- Configure fault tolerance check using an API:
Enable auto restart upon a fault using an API. When creating a training job, configure the **fault-tolerance/job-retry-num** field in **annotations** of the **metadata** field.
If the **fault-tolerance/job-retry-num** field is added, auto restart is enabled. The value can be an integer ranging from **1** to **128**, specifying the maximum number of times that a job can be re-issued. If this hyperparameter is not specified, the default value **0** is used, indicating that the job will not be re-issued and the environment pre-check will not be enabled.

Figure 5-6 Setting the API

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

Check Items and Conditions

Check Item	Item (Log Keyword)	Execution Condition	Requirements for a Check
Domain name detection	dns	None	The domain names of the volcano containers in the .host file in /etc/volcano are successfully resolved.
Disk size - Container root directory	disk-size root	None	The directory is greater than 32 GB.
Disk size - /dev/shm	disk-size shm	None	The directory is greater than 1 GB.
Disk size - /cache	disk-size cache	None	The directory is greater than 32 GB.
ulimit check	ulimit	An IB network is used.	<ul style="list-style-type: none"> • Maximum locked memory > 16000 • Open files > 1000000 • Stack size > 8000 • Maximum user processes > 1000000
GPU check	gpu-check	GPU and the v2 training engine are used.	GPUs are detected.

Effect of a Fault Tolerance Check

- If the fault tolerance check is passed, the logs of the check items will be recorded, indicating that the check items are successful. You can search for the keyword **item** in the log file. A fault tolerance check minimizes reported runtime faults.

```
[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/
```

- If a fault tolerance check fails, check failure logs will be recorded. You can search for the keyword **item** in the log file to view the failure information.

```
[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...
```

If the number of job restarts does not reach the specified time, the job will be automatically issued again. You can search for keywords **error,exiting** to obtain the logs recording a restarted job that ends with a failure.

Using reload ckpt to Resume an Interrupted Training

With fault tolerance enabled, if a training job is restarted due to a hardware fault, you can obtain the pre-trained model in the code to restore the training to the state before the restart. To do so, add reload ckpt to the code. For details, see [Resumable Training and Incremental Training](#).

5.1.2 Unconditional Auto Restart

Context

Unexpected situations during training can lead to failures and delays in restarting the job, resulting in longer training periods. To avoid these issues, use unconditional auto restart. Unconditional auto restart means that the system will automatically restart a failed training job, regardless of the cause. This feature can improve the success rate of training and increase job stability. To prevent invalid restarts, it supports a maximum of three consecutive unconditional restarts.

To avoid losing training progress and make full use of compute power, ensure that your code logic supports resumable training before enabling this function. For details, see [Resumable Training and Incremental Training](#).

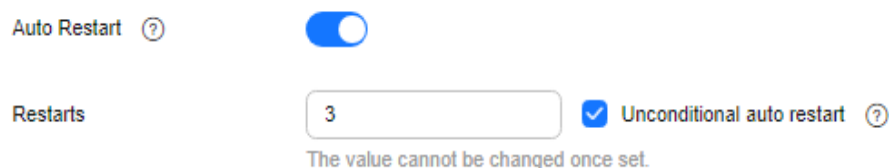
If auto restart is triggered during training, the system records the restart information. You can check the fault recovery details on the training job details page. For details, see [Viewing Fault Recovery Details](#).

Enabling Unconditional Auto Restart

You can enable unconditional auto restart either on the console or through an API.

- Using the console
On the **Create Training Job** page, enable **Auto Restart** and select **Unconditional auto restart**. If **Unconditional auto restart** is enabled, the training job will be restarted unconditionally once the system detects a training exception. If you enable auto restart but do not select **Unconditional auto restart**, the training job will only be automatically restarted if it encounters environmental issues. In case of any other problems, the status of the training job will become **Failed**.

Figure 5-7 Enabling unconditional auto restart



- Using an API

When creating a training job through an API, input the **fault-tolerance/job-retry-num** and **fault-tolerance/job-unconditional-retry** fields in **annotations** of the **metadata** field. To enable auto restart, set **fault-tolerance/job-retry-num** to a value ranging from **1** to **128**. To enable unconditional auto restart, set **fault-tolerance/job-unconditional-retry** to **true**.

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

5.2 Resumable Training and Incremental Training

Overview

Resumable training indicates that an interrupted training job can be automatically resumed from the checkpoint where the previous training was interrupted. This method is applicable to model training that takes a long time.

Incremental training is a method in which input data is continuously used to extend the existing model's knowledge to further train the model.

Checkpoints are used to resume model training or incrementally train a model.

During model training, training results (including but not limited to epochs, model weights, optimizer status, and scheduler status) are continuously saved. In this way, an interrupted training job can be automatically resumed from the checkpoint where the previous training was interrupted.

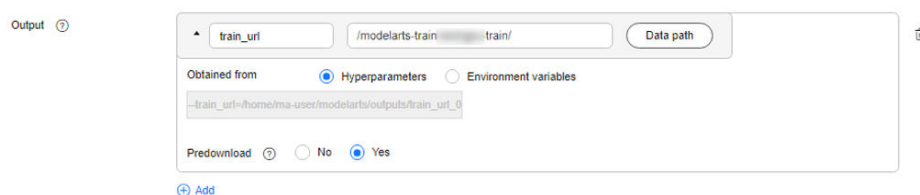
To resume a training job, load a checkpoint and use the checkpoint information to initialize the training status. To do so, add `reload ckpt` to the code.

Resumable Training and Incremental Training in ModelArts

To resume model training or incrementally train a model in ModelArts, configure **Training Output**.

When creating a training job, set the data path to the training output, save checkpoints in this data path, and set **Predownload** to **Yes**. If you set **Predownload** to **Yes**, the system automatically downloads the **checkpoint** file in the training output data path to a local directory of the training container before the training job is started.

Figure 5-8 Configuring training output



Enable fault tolerance check (auto restart) for resumable training. On the training job creation page, enable **Auto Restart**. If the environment pre-check fails, the hardware is not functional, or the training job fails, ModelArts will automatically issue the training job again.

reload ckpt for PyTorch

1. Use either of the following methods to save a PyTorch model.
 - Save model parameters only.

```
state_dict = model.state_dict()
torch.save(state_dict, path)
```
 - Save the entire model (not recommended).

```
torch.save(model, path)
```
2. Save the data generated during model training at regular intervals based on steps and time.

The data includes the network weight, optimizer weight, and epoch, which will be used to resume the interrupted training.

```
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. Check the complete code example below.

```
import os
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--train_url", type=str)
args, unparsed = parser.parse_known_args()
args = parser.parse_known_args()
# train_url is set to /home/ma-user/modelarts/outputs/train_url_0.
train_url = args.train_url

# Check whether there is a model file in the output path. If there is no file, the model will be trained
from the beginning by default. If there is a model file, the CKPT file with the maximum epoch value
will be loaded as the pre-trained model.
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)
    # Load the checkpoint.
    checkpoint = torch.load(local_ckpt_file)
    # Load the parameters that can be learned by the model.
    model.load_state_dict(checkpoint['net'])
    # Load optimizer parameters.
    optimizer.load_state_dict(checkpoint['optimizer'])
    # Obtain the saved epoch. The model will continue to be trained based on the epoch value.
    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()
...

# Save the network weight, optimizer weight, and epoch during model training.
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 Detecting Training Job Suspension

Overview

A training job may be suspended due to unknown reasons. If the suspension cannot be detected promptly, resources cannot be released, leading to a waste. To minimize resource cost and improve user experience, ModelArts provides suspension detection for training jobs. With this function, suspension can be automatically detected and displayed on the log details page. You can also enable notification so that you can be promptly notified of job suspension.

Detection Rules

Determine whether a job is suspended based on the monitored job process status and resource usage. A process is started to periodically monitor the changes of the two metrics.

- Job process status: If the process I/O of a training job changes, the next detection period starts. If the process I/O of the job remains unchanged in multiple detection periods, the resource usage detection starts.
- Resource usage: If the process I/O remains unchanged, the system collects the GPU usage within a certain period of time and determines whether the resource usage changes based on the variance and median of the GPU usage within the period. If the GPU usage is not changed, the job is suspended.

Constraints

Suspension can be detected only for training jobs that run on GPUs.

Procedure

Suspension detection is automatically performed during job running. No additional configuration is required. After detecting that a job is suspended, the system displays a message on the training job details page, indicating that the job may be suspended. If you want to be notified of suspension (by SMS or email), enable event notification on the job creation page.

5.4 Priority of a Training Job

When using a new-version dedicated resource pool for training jobs, you can set the job priority when creating a training job or adjust the priority when a job is in

the **Pending** state for a long time. By adjusting the job priority, you can reduce the job queuing duration.

Overview

Some training tasks, such as unimportant tests or experiments, are of low priority. In this case, you need to prioritize training tasks (jobs). A task with a higher priority is queued earlier than a task with a lower priority.

You can adjust the job execution sequence by configuring the priority of training jobs to ensure normal running of important services at peak hours.

Constraints

- You can set the priority of a training job only if it is created using a new-version dedicated resource pool.
- The value ranges from 1 to 3. The default priority is **1**, and the highest priority is **3**. By default, the job priority can be set to **1** or **2**. After the permission to **set the highest job priority** is configured, the priority can be set to **1** to **3**.

Configuring the Priority

Set the priority when you create a training job. The value ranges from 1 to 3. The default priority is **1**, and the highest priority is **3**.

Changing the Priority


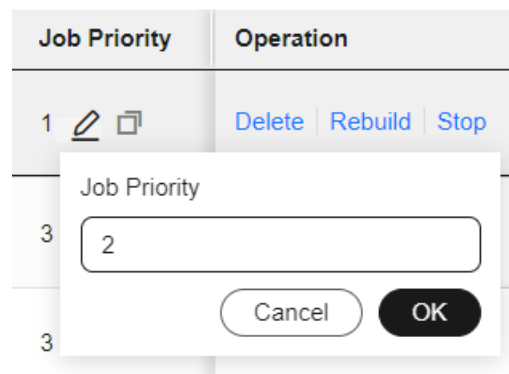
On the **Training Jobs** page, locate a training job in the **Pending** state and click  in the **Job Priority** column. In the dialog box that appears, change the priority and click **OK**.

Figure 5-9 Changing the job priority



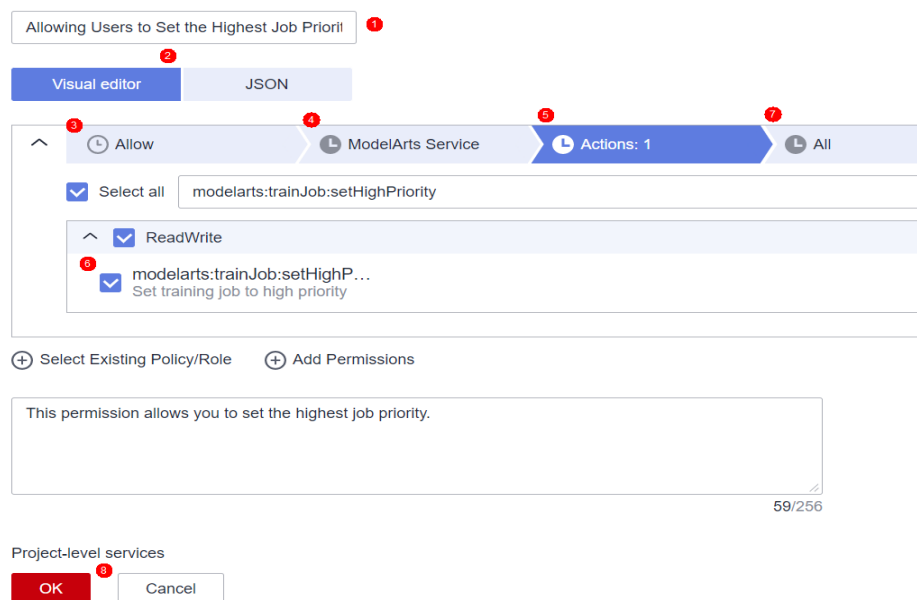
5.5 Permission to Set the Highest Job Priority

You can configure the priority when you create a training job using a new-version dedicated resource pool. You can change the priority of a pending job. The value ranges from 1 to 3. The default priority is **1**, and the highest priority is **3**. By default, the job priority can be set to **1** or **2**. After the permission to set the highest job priority is configured, the priority can be set to **1** to **3**.

Assigning the Permission to Set the Highest Job Priority to an IAM User

1. Log in to the Huawei Cloud management console as a tenant user, hover the cursor over your username in the upper right corner, and choose **Identity and Access Management** from the drop-down list to switch to the IAM management console.
2. On the IAM console, choose **Permissions > Policies/Roles** from the navigation pane, click **Create Custom Policy** in the upper right corner, and configure the following parameters.
 - **Policy Name:** Enter a custom policy name, for example, **Allowing Users to Set the Highest Job Priority**.
 - **Policy View:** Select **Visual editor**.
 - **Policy Content:** Select **Allow, ModelArts Service, modelarts:trainJob:setHighPriority**, and default resources.

Figure 5-10 Creating a custom policy



3. In the navigation pane, choose **User Groups**. Then, click **Authorize** in the **Operation** column of the target user group. On the **Authorize User Group** page, select the custom policies created in 2, and click **Next**. Then, select the scope and click **OK**.

After the configuration, all users in the user group have the permission to use Cloud Shell to log in to a running training container.

If no user group is available, create a user group, add users using the user group management function, and configure authorization. If the target user is not in a user group, you can add the user to a user group through the user group management function.

6 Distributed Training

6.1 Distributed Training Functions

ModelArts provides the following capabilities:

- Extensive built-in images, meeting your requirements
- Custom development environments set up using built-in images
- Extensive tutorials, helping you quickly understand distributed training
- Distributed training debugging in development tools such as PyCharm, VS Code, and JupyterLab

Constraints

- If the instance flavors are changed, you can only perform single-node debugging. You cannot perform distributed debugging or submit remote training jobs.
- Only the PyTorch and MindSpore AI frameworks can be used for multi-node distributed debugging. If you want to use MindSpore, each node must be equipped with eight cards.
- The OBS paths in the debugging code should be replaced with your OBS paths.
- PyTorch is used to write debugging code in this document. The process is the same for different AI frameworks. You only need to modify some parameters.

Related Chapters

- [Single-Node Multi-Card Training Using DataParallel](#): describes single-node multi-card training using DataParallel, and corresponding code modifications.
- [Multi-Node Multi-Card Training Using DistributedDataParallel](#) : describes multi-node multi-card training using DistributedDataParallel, and corresponding code modifications.
- [Distributed Debugging Adaptation and Code Example](#): describes the procedure and code example of distributed debugging adaptation.

- **Sample Code of Distributed Training:** provides a complete code sample of distributed parallel training for the classification task of ResNet18 on the CIFAR-10 dataset.
- **Debugging a Training Job:** describes how to use the SDK to debug a single-node or multi-node training job on the ModelArts development environment.

6.2 Single-Node Multi-Card Training Using DataParallel

This section describes how to perform single-node multi-card parallel training based on the PyTorch engine.

For details about the distributed training using the MindSpore engine, see [the MindSpore official website](#).

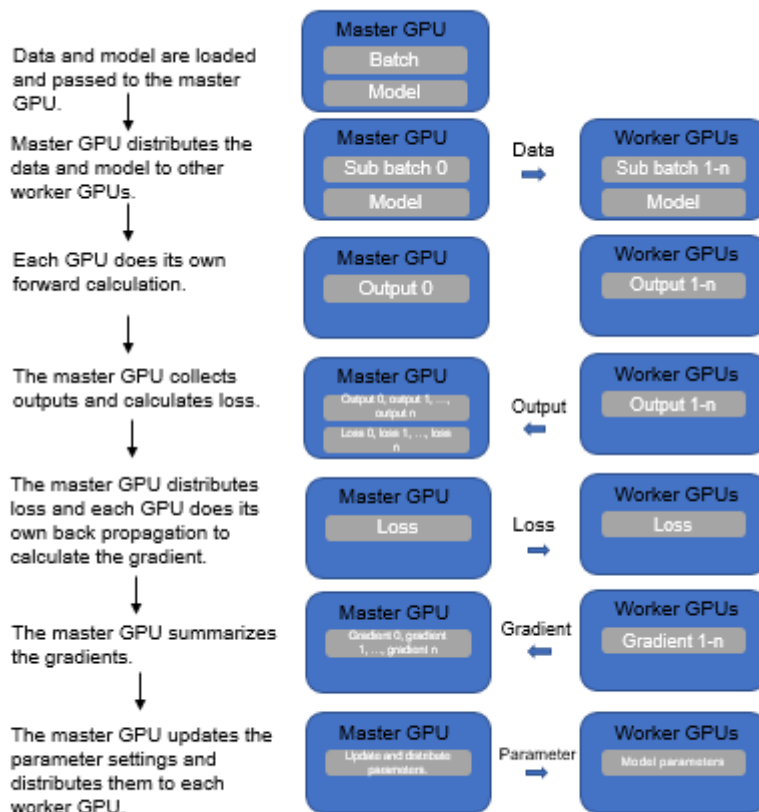
Training Process

The process of single-node multi-card parallel training is as follows:

1. A model is copied to multiple GPUs.
2. Data of each batch is distributed evenly to each worker GPU.
3. Each GPU does its own forward propagation and an output is obtained.
4. The master GPU with device ID 0 collects the output of each GPU and calculates the loss.
5. The master GPU distributes the loss to each worker GPU. Each GPU does its own backward propagation and calculates the gradient.
6. The master GPU collects gradients, updates parameter settings, and distributes the settings to each worker GPU.

The detailed flowchart is as follows.

Figure 6-1 Single-node multi-card parallel training



Advantages and Disadvantages

- Straightforward coding: Only one line of code needs to be modified.
- Bottlenecks in communication: The master GPU is used to update and distribute parameter settings, which causes high communication costs.
- Unbalanced GPU loading: The master GPU is used to summarize outputs, calculate loss, and update weights. Therefore, the GPU memory and usage are higher than those of other GPUs.

Code Modifications

Model distribution: `DataParallel(model)`

The code is slightly changed and the following is a simple example:

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

model = Net().cuda()

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

6.3 Multi-Node Multi-Card Training Using DistributedDataParallel

This section describes how to perform multi-node multi-card parallel training based on the PyTorch engine.

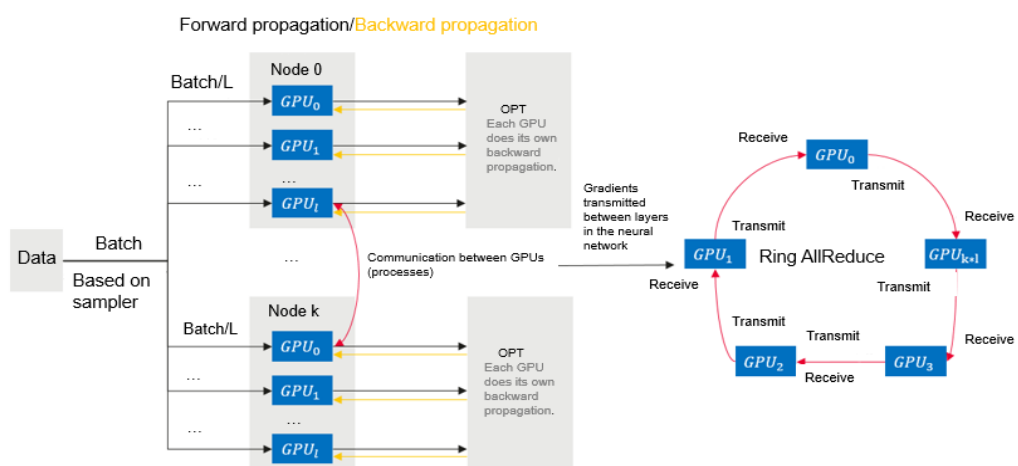
Training Process

Compared with DataParallel, DistributedDataParallel can start multiple processes for computing, greatly improving compute resource usage. Based on **torch.distributed**, DistributedDataParallel has obvious advantages over DataParallel in the distributed computing case. The process is as follows:

1. Initializes the process group.
2. Creates a distributed parallel model. Each process has the same model and parameters.
3. Creates a distributed sampler for data distribution to enable each process to load a unique subset of the original dataset in a mini batch.
4. Parameters are organized into buckets based on their shapes or sizes, which are generally determined by each layer of the network that requires parameter update in a neural network model.
5. Each process does its own forward propagation and computes its gradient.
6. After all parameter gradients at a bucket are obtained, communication is performed for gradient averaging.
7. Each GPU updates model parameters.

The detailed flowchart is as follows.

Figure 6-2 Multi-node multi-card parallel training



Advantages

- Fast communication

- Balanced load
- Fast running speed

Code Modifications

- Multi-process startup
- New variables such as rank ID and world_size are used along with the TCP protocol.
- Sampler for data distribution to avoid duplicate data between different processes
- Model distribution: DistributedDataParallel(model)
- Model saved in GPU 0

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

model = Net().cuda()

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

Related Operations

- For details about distributed debugging adaptation and code example, see [Distributed Debugging Adaptation and Code Example](#).
- This document also provides a complete code sample of distributed parallel training for the classification task of ResNet18 on the cifar10 dataset. For details, see [Sample Code of Distributed Training](#).

6.4 Distributed Debugging Adaptation and Code Example

In DistributedDataParallel, each process loads a subset of the original dataset in a batch, and finally the gradients of all processes are averaged as the final gradient. Due to a large number of samples, a calculated gradient is more reliable, and a learning rate can be increased.

This section describes the code of single-node training and distributed parallel training for the classification job of ResNet18 on the CIFAR-10 dataset. Directly execute the code to perform multi-node distributed training with CPUs or GPUs; comment out the distributed training settings in the code to perform single-node single-card training.

The training code contains three input parameters: basic training parameters, distributed parameters, and data parameters. The distributed parameters are automatically input by the platform. **custom_data** indicates whether to use custom data for training. If this parameter is set to **true**, torch-based random data is used for training and validation.

Dataset

CIFAR-10 dataset

In notebook instances, torchvision of the default version cannot be used to obtain datasets. Therefore, the sample code provides three training data loading methods.

Click **CIFAR-10 python version** on the [download page](#) to download the CIFAR-10 dataset.

- Download the CIFAR-10 dataset using torchvision.
- Download the CIFAR-10 dataset based on the URL and decompress the dataset in a specified directory. The sizes of the training set and test set are (50000, 3, 32, 32) and (10000, 3, 32, 32), respectively.
- Use Torch to obtain a random dataset similar to CIFAR-10. The sizes of the training set and test set are (5000, 3, 32, 32) and (1000, 3, 32, 32), respectively. The labels are still of 10 types. Set **custom_data** to **true**, and the training task can be directly executed without loading data.

Training Code

In the following code, those commented with `###` Settings for distributed training and `... ###` are code modifications for multi-node distributed training.

Do not modify the sample code. After the data path is changed to your path, multi-node distributed training can be executed on ModelArts.

After the distributed code modifications are commented out, the single-node single-card training can be executed. For details about the complete code, see [Sample Code of Distributed Training](#).

- **Importing dependency packages**

```
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
```

- **Defining the method and random number for loading data** (The code for loading data is not described here due to its large amount.)

```
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
```

- **Defining a network structure**

```
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

```

- **Training and validation**

```

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.')

### Settings for distributed training. Initialize DistributedDataParallel process. The init_method,
rank, and world_size parameters are automatically input by the platform. ###
dist.init_process_group(init_method=args.init_method, backend="nccl", world_size=args.world_size,
rank=args.rank)
### Settings for distributed training. Initialize DistributedDataParallel process. The init_method,
rank, and world_size parameters are automatically input by the platform. ###

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)

### Settings for distributed training. Create a sampler for data distribution to ensure that different
processes load different data. ###
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)
### Settings for distributed training. Create a sampler for data distribution to ensure that different
processes load different data. ###

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)

### Settings for distributed training. Build a DistributedDataParallel model. ###
model = nn.parallel.DistributedDataParallel(model)
### Settings for distributed training. Build a DistributedDataParallel model. ###

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

### Settings for distributed training. DistributedDataParallel sampler. Random numbers are set for
the DistributedDataParallel sampler based on the current epoch number to avoid loading duplicate
data. ###
tr_sampler.set_epoch(epoch)
### Settings for distributed training. DistributedDataParallel sampler. Random numbers are set for
the DistributedDataParallel sampler based on the current epoch number to avoid loading duplicate
data. ###

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()

```

- **Result comparison**

100-epoch **cifar-10** dataset training is completed using two resource types respectively: single-node single-card and two-node 16-card. The training duration and test set accuracy are as follows.

Table 6-1 Training result comparison

Resource Type	Single-Node Single-Card	Two-Node 16-Card
Duration	60 minutes	20 minutes
Accuracy	80+	80+

6.5 Sample Code of Distributed Training

The following provides a complete code sample of distributed parallel training for the classification task of ResNet18 on the CIFAR-10 dataset.

The content of the training boot file **main.py** is as follows (if you need to execute a single-node and single-card training job, delete the code for distributed reconstruction):

```

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.')

### Settings for distributed training. Initialize DistributedDataParallel process. The
init_method, rank, and world_size parameters are automatically input by the platform. ###
dist.init_process_group(init_method=args.init_method, backend="nccl",
                       world_size=args.world_size, rank=args.rank)
### Settings for distributed training. Initialize DistributedDataParallel process. The
init_method, rank, and world_size parameters are automatically input by the platform. ###

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)

### Settings for distributed training. Create a sampler for data distribution to ensure that
different processes load different data. ###
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)
### Settings for distributed training. Create a sampler for data distribution to ensure that
different processes load different data. ###

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)

### Settings for distributed training. Build a DistributedDataParallel model. ###
model = nn.parallel.DistributedDataParallel(model)
### Settings for distributed training. Build a DistributedDataParallel model. ###

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

### Settings for distributed training. DistributedDataParallel sampler. Random numbers are set
for the DistributedDataParallel sampler based on the current epoch number to avoid loading
duplicate data. ###
tr_sampler.set_epoch(epoch)
### Settings for distributed training. DistributedDataParallel sampler. Random numbers are set
for the DistributedDataParallel sampler based on the current epoch number to avoid loading
duplicate data. ###

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()
```

FAQs

1. How Do I Use Different Datasets in the Sample Code?

- To use the CIFAR-10 dataset in the preceding code, [download](#) and decompress the dataset and upload it to the OBS bucket. The file directory structure is as follows:

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

DDP is the code directory specified during training job creation, **main.py** is the preceding code example (the boot file specified during training job creation), and **cifar-10-batches-py** is the decompressed dataset folder that is stored in the **input_dir** folder.

- To use user-defined random data, change the value of **custom_data** in the code example to **true**.

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

Then, run **main.py**. The parameters for creating a training job are the same as those shown in the preceding figure.

2. Why Can I Leave the IP Address of the Master Node Blank for DDP?

The **init method** parameter in **parser.add_argument('--init_method', default=None, help='tcp_port')** contains the IP address and port number of the master node, which are automatically input by the platform.

6.6 Example of Starting PyTorch DDP Training Based on a Training Job

This topic describes three methods of using a training job to start PyTorch DDP training and provides their sample code.

- Use PyTorch preset images and run the **mp.spawn** command.
- Use custom images.
 - Run the **torch.distributed.launch** command.
 - Run the **torch.distributed.run** command.

Creating a Training Job

- Method 1: Use the preset PyTorch framework and run the **mp.spawn** command to start a training job.

For details about parameters for creating a training job, see [Table 6-2](#).

Table 6-2 Creating a training job (preset framework)

Parameter	Description
Algorithm Type	Select Custom algorithm .
Boot Mode	Choose Preset image and set AI Engine to PyTorch . Configure the PyTorch version based on your training requirements.
Code Directory	Select the path where the training code folder is stored in the OBS bucket, for example, obs://test-modelarts/code/ .
Boot File	Select the Python boot script of the training job in the code directory, for example, obs://test-modelarts/code/main.py .
Hyperparameters	If the resource specification is single-node multi-card, you need to specify the hyperparameters world_size and rank . If you select a resource flavor with multiple nodes (more than one compute node), you do not need to set these hyperparameters. world_size and rank are automatically injected by ModelArts.

- Method 2: Use a custom image and run the **torch.distributed.launch** command to start a training job.

For details about parameters for creating a training job, see [Table 6-3](#).

Table 6-3 Creating a training job (custom image + **torch.distributed.launch**)

Parameter	Description
Algorithm Type	Select Custom algorithm .
Boot Mode	Select Custom image .
Image	Select a PyTorch image for training.
Code Directory	Select the path where the training code folder is stored in the OBS bucket, for example, obs://test-modelarts/code/ .
Boot Command	Enter the Python startup command of the image, for example: bash \${MA_JOB_DIR}/code/torchlaunch.sh

- Method 3: Use a custom image and run the **torch.distributed.run** command to start a training job.

For details about parameters for creating a training job, see [Table 6-4](#).

Table 6-4 Creating a training job (custom image + **torch.distributed.run**)

Parameter	Description
Algorithm Type	Select Custom algorithm .
Boot Mode	Select Custom image .
Image	Select a PyTorch image for training.
Code Directory	Select the path where the training code folder is stored in the OBS bucket, for example, obs://test-modelarts/code/ .
Boot Command	Enter the Python startup command of the image, for example: bash \${MA_JOB_DIR}/code/torchrun.sh

Code Examples

Upload the following files to an OBS bucket:

```
code
├── torch_ddp.py      # Code file for PyTorch DDP training
├── main.py          # Code file for PyTorch DDP training
├── torchlaunch.sh   # Boot file for starting training using the custom image and the
torch.distributed.launch command
├── torchrun.sh      # Boot file for starting training using the custom image and the
torch.distributed.run command
```

torch_ddp.py

```
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

# Start training by running mp.spawn.
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)

# Start training by running torch.distributed.launch or torch.distributed.run.
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

```
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
    # Call the start function in the DDP sample code.
    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

```
#!/bin/bash
# Default system environment variables. Do not modify them.
```

```

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"

# Custom environment variables to specify the Python script and parameters.
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

NOTICE

In PyTorch 2.1, you must set **rdzv_backend** to **static**: **--rdzv_backend=static**.

```

#!/bin/bash
# Default system environment variables. Do not modify them.
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"

# Custom environment variables to specify the Python script and parameters.
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 Automatic Model Tuning (AutoSearch)

7.1 Introduction to Hyperparameter Search

ModelArts training supports hyperparameter search, which can automatically search for optimal hyperparameters for your models.

During model training, many hyperparameters, such as **learning_rate** and **weight_decay**, need to be adjusted based on actual requirements. This may cost an experienced algorithm engineer a lot of time and effort on manual optimization. However, the hyperparameter search supported by ModelArts can automatically optimize hyperparameters without the help of algorithm engineers and has higher optimization speed and precision than manual optimization.

ModelArts supports the following hyperparameter search algorithms:

- [Bayesian Optimization \(SMAC\)](#)
- [TPE Algorithm](#)
- [Simulated Annealing Algorithm](#)

7.2 Search Algorithm

7.2.1 Bayesian Optimization (SMAC)

In Bayesian optimization, it is assumed that there exists a functional relationship between hyperparameters and the objective function. Based on the evaluation values of the searched hyperparameters, the mean and variance of the objective function values at other search points are estimated through Gaussian process regression. The mean and variance are then used to construct the acquisition function. The next search point is the maximum value of the acquisition function. Compared with grid search, Bayesian optimization uses the previous evaluation results to reduce the number of iterations and shorten the search time. The disadvantage is that it is difficult to find the global optimal solution.

Table 7-1 Bayesian optimization parameters

Parameter	Description	Recommended Value
num_samples	Number of times to sample from the hyperparameter space	The value is an integer ranging from 10 to 20. The larger the value, the longer the search time and the better the effect.
kind	Acquisition function type	This value is string type. The default value is ucb and other value options are ei and poi . Do not change the default one.
kappa	Adjustment parameter of acquisition function type ucb , which is the upper confidence boundary	This value is a float. You are advised not to change it.
xi	Adjustment parameter of acquisition function types poi and ei	This value is a float. You are advised not to change it.

7.2.2 TPE Algorithm

The tree-structured parzen estimator (TPE) algorithm uses the Gaussian mixture model to learn the model hyperparameters. On each trial, for each parameter, TPE fits one Gaussian mixture model $l(x)$ to the set of parameter values associated with the best objective values, and another Gaussian mixture model $g(x)$ to the remaining parameter values. It chooses the hyperparameter value that maximizes the ratio $l(x)/g(x)$.

Table 7-2 TPE algorithm parameters

Parameter	Description	Recommended Value
num_samples	Number of times to sample from the hyperparameter space	The value is an integer ranging from 10 to 20. The larger the value, the longer the search time and the better the effect.
n_initial_points	Number of random evaluations of the objective function before starting to approximate it with tree parzen estimators	The value is an integer. You are advised not to change it.
gamma	Quantile of the TPE algorithm to divide $l(x)$ and $g(x)$	This value is a float ranging from 0 to 1. You are advised not to change it.

7.2.3 Simulated Annealing Algorithm

The simulated annealing algorithm is a simple but effective variant on random search that leverages smoothness in the response surface. The annealing rate is not adaptive. The annealing algorithm is to choose one of the previous trial points as a starting point, and then to sample each hyperparameter from a similar distribution to the one specified in the prior, but whose density is more concentrated around the selected trial point. The algorithm tends over time to sample from points closer and closer to the best ones. During the sampling, this algorithm may draw a runner-up trial as the best trail to avoid local optima at a certain probability.

Table 7-3 Parameters of the simulated annealing algorithm

Parameter	Description	Recommended Value
num_samples	Number of times to sample from the hyperparameter space	The value is an integer ranging from 10 to 20. The larger the value, the longer the search time and the better the effect.
avg_best_idx	Mean of geometric distribution over which trial to explore around, selecting from trials sorted by score	This value is a float. You are advised not to change it.
shrink_coef	Rate of reduction in the size of sampling neighborhood as more points have been explored	This value is a float. You are advised not to change it.

7.3 Creating a Hyperparameter Search Job

Background

If the AI engine is `pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64` or `tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64` and the hyperparameter to be optimized is of the float type, you can use hyperparameter search on ModelArts.

You can perform the hyperparameter search without any code modification. The procedure is as follows:

1. [Preparations](#)
2. [Creating an Algorithm](#)
3. [Creating a Training Job](#)
4. [Viewing Details About a Hyperparameter Search Job](#)

Preparations

- Data has been prepared. Specifically, you have created an available dataset in ModelArts, or you have uploaded the dataset used for training to the OBS directory.

- Prepare the training script and upload it to the OBS directory. For details about how to develop a training script, see [Developing a Custom Script](#).
- In the training code, print the search indicator parameters.
- At least one empty folder has been created in OBS for storing the training output.
- The account is not in arrears because resources are consumed when training jobs are running.
- The OBS directory you use and ModelArts are in the same region.

Creating an Algorithm

Log in to the ModelArts management console and create an algorithm by referring to [Creating an Algorithm](#). The image must use the **pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64** or **tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64** engine.

Hyperparameters that you want to optimize need to be defined when you configure **Hyperparameters**. You can specify the name, type, default value, and constraints. For details, see [Defining Hyperparameters](#).

Select **autoSearch(S)** to enable auto search for the algorithm. During an auto search, ModelArts uses a regular expression to obtain the search indicator parameters and performs hyperparameter optimization based on the optimization direction. Print search parameters in the code and set the following parameters.

Figure 7-1 Enabling auto search

The screenshot displays the ModelArts configuration interface for enabling auto search. It includes a table for hyperparameters, a section for supported policies, a search indicator configuration section, a section for setting automatic search parameters, and a section for search algorithm configuration.

Name	Type	Default	Required	Description	Operati...
rate	Float		Restrain	Y...	Delete

Supported Policies

- autoSearch(S)

Search Indicator

Name: [] Optimization Direction: max Counter regularization: [] [Generate Intelligently](#)

Setting Automatic Search Parameters

rate x []

rate [] ~ []

Search Algorithm Configuration

bayes_opt_search x tpe_search x anneal_search x

- bayes_opt_search
- tpe_search: avg_best_idx: 2.0
- anneal_search: shrink_coef: 0.1, num_samples: 20

- **Search Indicator**
The search indicator is the value of the objective function, which can be loss, accuracy, and so on. By optimizing and converging this value based on the optimization direction, the optimal hyperparameter can be found to improve the model precision and convergence speed.

Table 7-4 Search indicator parameters

Parameter	Description
Name	search indicator name. This parameter must be identical to the search indicator parameter in the code.
Optimization Direction	This parameter can be max or min .
Counter regularization	A regular expression needs to be entered. You can click Generate Intelligently to generate a regular expression automatically.

- **Setting Automatic Search Parameters**
Select hyperparameters that can be used for search optimization from what you configured for **Hyperparameters**. Only hyperparameters of the float type are supported. After **autoSearch(S)** is selected, set the value range.
- **Search Algorithm Configuration**
ModelArts has three built-in algorithms for hyperparameter search. You can select one or more algorithms as needed. The algorithms and their parameter description are as follows:
 - bayes_opt_search: **Bayesian Optimization (SMAC)**
 - tpe_search: **TPE Algorithm**
 - anneal_search: **Simulated Annealing Algorithm**

After the algorithm is created, use it to create a training job.

Creating a Training Job

Log in to the ModelArts console and create a training job by referring to **Creating a Training Job**. Pay attention to operations described in this section before you enable the hyperparameter search.

If you select an algorithm that supports hyperparameter search, click the button for range setting to enable hyperparameter search.

Figure 7-2 Enabling hyperparameter search

The screenshot shows the configuration interface for enabling hyperparameter search. It includes the following sections:

- Hyperparameter:** A text input field containing 'on_device_model' is followed by an equals sign and two numeric input fields with values '0' and '2'. To the right of these fields are icons for a search and a trash can.
- Add:** A button with a plus sign and the text 'Add'.
- Auto Search Metric:** A dropdown menu with 'rate' selected.
- Auto Search Algorithm:** Three buttons: 'bayes_opt_search' (highlighted in blue), 'tpe_search', and 'anneal_search'.
- Auto Search Parameters:** A list of parameter names in grey boxes, each followed by an equals sign and a text input field:
 - 'kind' = ucb
 - 'kappa' = 2.5
 - 'xi' = 0.0
 - 'num_samples' = 20
 - 'seed' = 1

After the hyperparameter search is enabled, you can configure the search indicator, search algorithm, and parameters of the selected algorithm. These parameters need to have the same values as the hyperparameters of the algorithm you created.

After a hyperparameter search job is created, it will take a period of time to run it.

Viewing Details About a Hyperparameter Search Job

After a training job is completed, you can review the results of the automated hyperparameter search to evaluate the job's performance.

If the training job is a hyperparameter search job, go to the training job details page and click the **Auto Search Results** tab to view the hyperparameter search results.

Figure 7-3 Hyperparameter search results

