ModelArts

Model Training

Issue	01
Date	2024-06-12

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are trademarks 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 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 Technologies Co., Ltd.

Address: Huawei Industrial Base
Bantian, Longgang
Shenzhen 518129
People's Republic of China

Website: https://www.huawei.com

Email: support@huawei.com

Security Declaration

Vulnerability

Huawei's regulations on product vulnerability management are subject to the *Vul. Response Process.* For details about this process, visit the following web page:
https://www.huawei.com/en/psirt/vul-response-process
For vulnerability information, enterprise customers can visit the following web page:
https://securitybulletin.huawei.com/enterprise/en/security-advisory

Contents

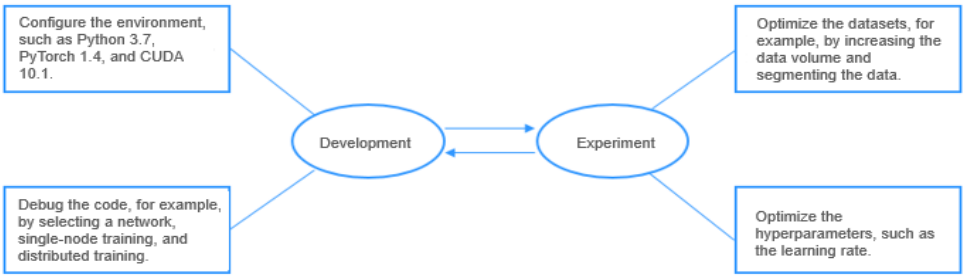
1 Introduction to Model Development

AI modeling involves two stages:

- Development: To train using deep learning, you must set up and configure the environment and debug the code. For code debugging, it is recommended to use ModelArts development environments.
- Experiment: To obtain an ideal model, you must optimize the datasets and hyperparameters through multiple rounds of experiments. ModelArts training is recommended.

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.

The following is part of the process for AI modeling.



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

To train a model on ModelArts, follow these steps:

- Upload the labeled data to OBS. For details, see **Preparing Data**.
- Create an algorithm for model training. For details, see **Preparing Algorithms**.

- Create a training job on the ModelArts console. For details, see **Creating a Training Job**.
- Review the training job logs and training resource usage. For details, see **Training Job Logs**.
- Stop or delete a training job. For details, see **Stopping, Rebuilding, or Searching for a Training Job**.
- Troubleshoot if you encounter any problem during training. For details, see "Training Jobs" in *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**

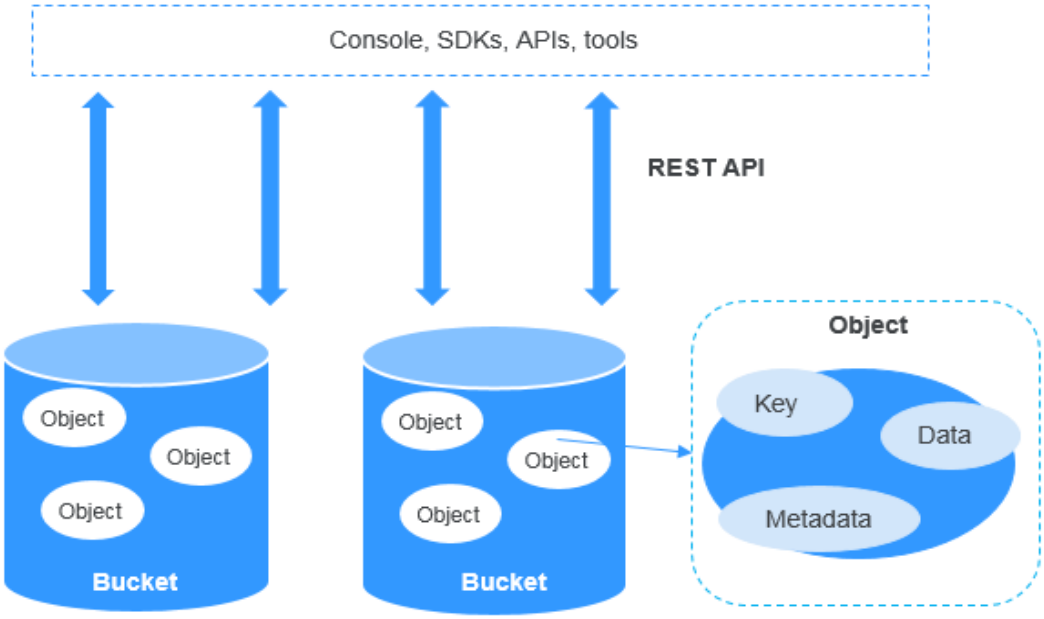
OBS

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.

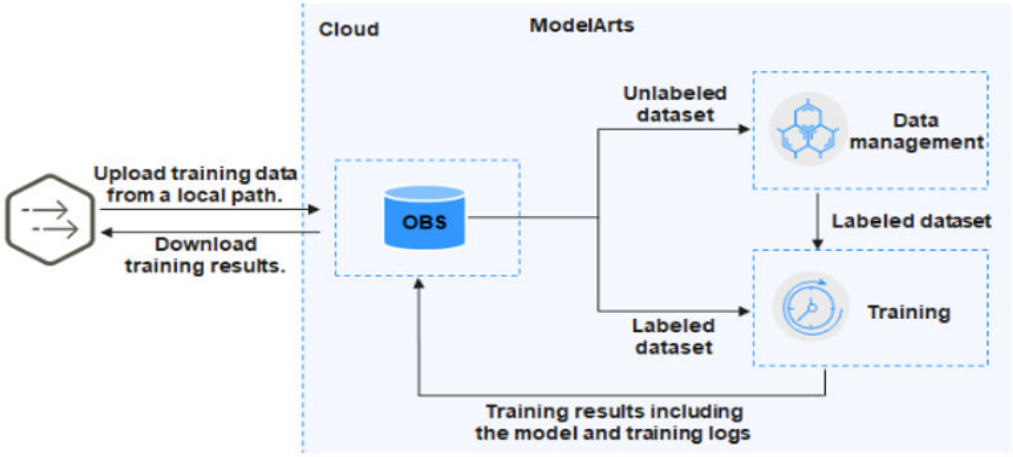
- Datasets in data management

  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 (For new-version training, see **Using a Custom Image to Train Models**.)

  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 the Software Repository for Container (SWR). 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. However, it is time-consuming and low in storage space utilization.

- 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 of the new version

Runtime Environment	Supported Chip	System Architecture	System Version	AI Engine and Version	Supported CUDA or Ascend Version
TensorFlow	CPU/GPU	x86_64	Ubuntu18.04	tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64	cuda10.1

Runtime Environment	Supported Chip	System Architecture	System Version	AI Engine and Version	Supported CUDA or Ascend Version
PyTorch	CPU/GPU	x86_64	Ubuntu18.04	pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64	cuda10.2
MPI	CPU/GPU	x86_64	Ubuntu18.04	mindspore_1.3.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64	cuda_10.1
Horovod	GPU	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 code 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 a 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 on 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?" in *FAQs*

- 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 on 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?" in *FAQs*.
   - For details about how to install a C++ dependency library, see "How Do I Install a Library That C++ Depends on?" in *FAQs*.
   - For details about how to load parameters to a pre-trained model, see "How Do I Load Some Well Trained Parameters During Job Training?" in *FAQs*.

Parsing Input and Output Paths

To enable a ModelArts model reads data stored in OBS or outputs data to a specified OBS path, follow these steps to configure the input and output data:

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

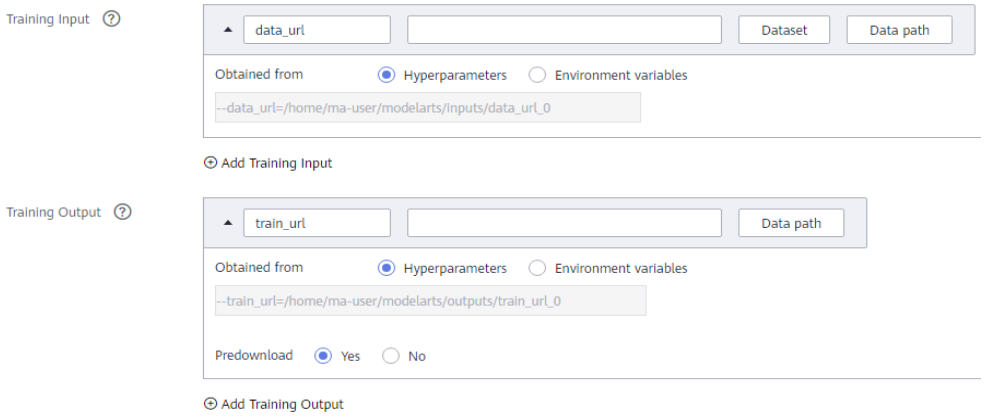
```
import argparse
# Create a parsing task.
parser = argparse.ArgumentParser(description='train mnist')

# 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, configure the input and output paths.

   Select an OBS path or dataset path as the training input, and an OBS path for the output.

   Figure 3-1 Setting training input and 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'))
```

Differences in Training Code Adaptation

In the old version, you are required to configure data input and output as follows:


```
# Parse CLI parameters.
import argparse
parser = argparse.ArgumentParser(description='MindSpore Lenet Example')
parser.add_argument('--data_url', type=str, default="./Data",
                    help='path where the dataset is saved')
parser.add_argument('--train_url', type=str, default="./Model", help='if is test, must provide\
                    path where the trained ckpt file')
args = parser.parse_args()
...
# Download data to your local container. In the code, local_data_path specifies the training input path.
mox.file.copy_parallel(args.data_url, local_data_path)
...
# Upload the local container data to the OBS path.
mox.file.copy_parallel(local_output_path, args.train_url)
```

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 Information**
4. **Setting the Boot Mode**
5. **Configuring Pipelines**
6. **Configuring Hyperparameters**
7. **Supported Policies**
8. **Adding Training Constraints**
9. **Previewing the Runtime Environment**
10. **Follow-Up Operations**

Prerequisites

- Training data is available. You can create a dataset in ModelArts or upload an existing dataset used for training to the OBS directory.
- Your training script has been uploaded to an 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.

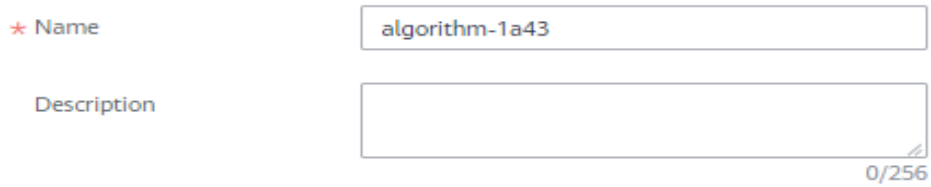
Accessing the Algorithm Creation Page

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

Setting Basic Information

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

Figure 3-2 Setting basic information



Setting the Boot Mode

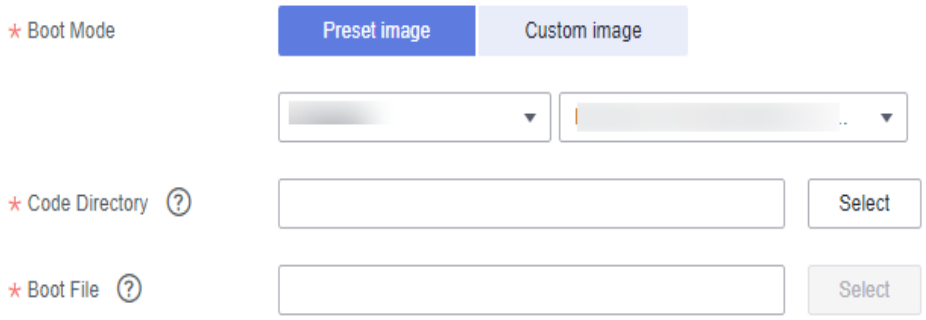
Select a preset image to create an algorithm.

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	AI images supported by the new-version training are displayed by default. For details, see **Overview**.
Code Directory	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. 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. 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. **NOTE** • Any programming language is supported. • The total number of both files and folders cannot exceed 1,000. • The total size of files cannot exceed 5 GB.
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 3-3 Using a custom script to create an algorithm



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**.

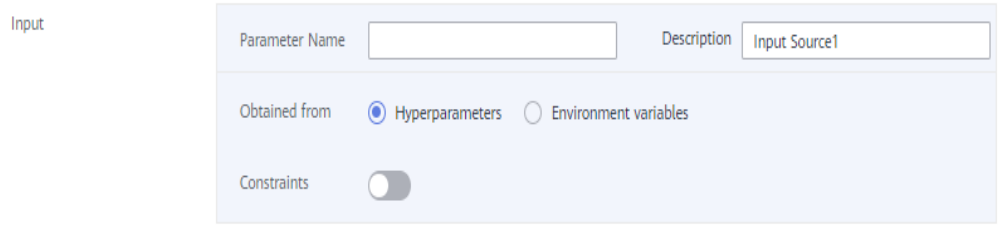
When you use a preset image to create an algorithm, configure the input and output parameters defined in the algorithm code.

- Input configurations

  Table 3-3 Input configurations

Parameter	Description
Parameter Name	Set this parameter 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	Customize the description of the input parameter.
Obtained from	Select a source of the input parameter, **Hyperparameters** (default) or **Environment variables**.
Constraints	Enable this parameter to specify the input source. The default source is a storage path. This parameter is optional.
Add	Add multiple input data sources based on your algorithm.

Figure 3-4 Input configurations

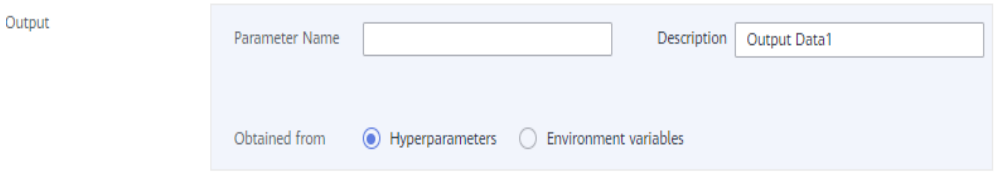


- Output configurations

Table 3-4 Output configurations

Parameter	Description
Parameter Name	Set this parameter 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. 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.
Description	Customize the description of the output parameter.
Obtained from	Select a source of the output parameter, **Hyperparameters** (default) or **Environment variables**.
Add	Add multiple output data paths based on your algorithm.

Figure 3-5 Output configurations



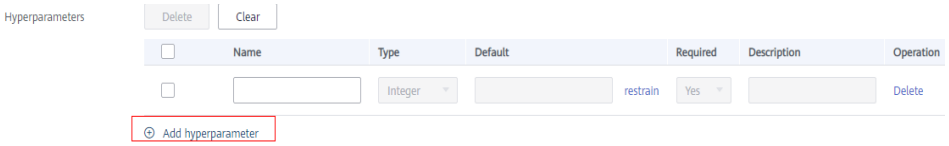
Configuring Hyperparameters

When you use a preset image to create an algorithm on ModelArts, you can customize hyperparameters so you can review or modify them anytime. Defined hyperparameters are displayed in the boot command and passed to your boot file as CLI parameters.

1. Import hyperparameters.

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

Figure 3-6 Adding hyperparameters



2. Edit hyperparameters. For details, see **Table 3-5**.

Table 3-5 Hyperparameter parameters

Parameter	Description
Name	Enter the hyperparameter name. Enter 1 to 64 characters. Only letters, digits, hyphens (-), and underscores (_) are allowed.
Type	Select a data type of the hyperparameter. The value can be **String**, **Integer**, **Float**, or **Boolean**
Default	Set the default value of the hyperparameter. This value will be used for training jobs by default.
Restrain	Click **Restrain** and set the range of the default value or enumerated value in the dialog box displayed.
Required	Select **Yes** or **No**. • 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	Enter the description of the hyperparameter. Only letters, digits, spaces, hyphens (-), underscores (_), commas (,), and periods (.) are allowed.

Supported Policies

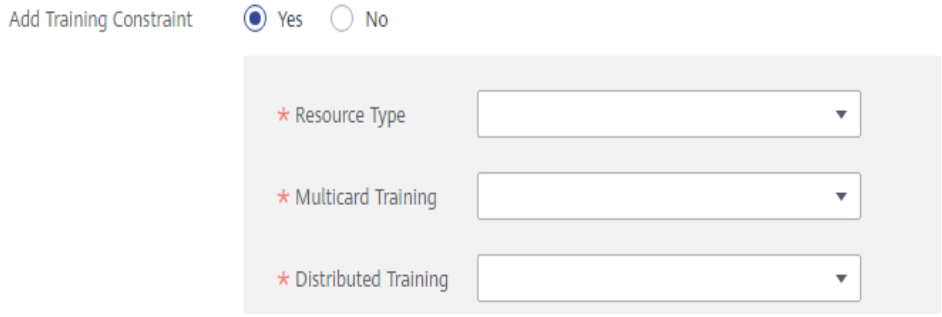
Only the pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64 and tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64 images are available for auto search.

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.

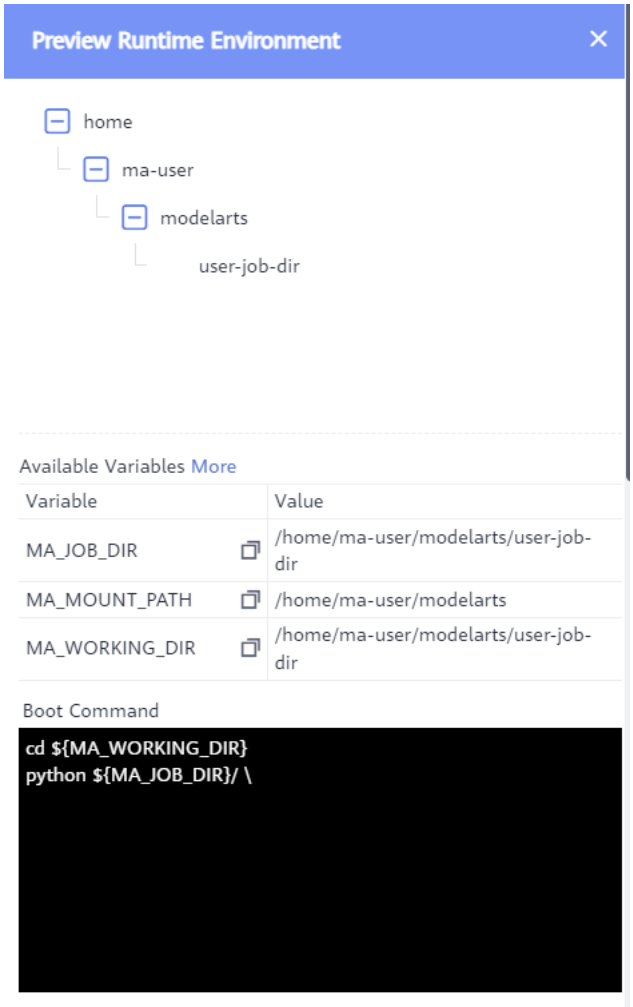
Figure 3-7 Training constraints



Previewing the Runtime Environment

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

Figure 3-8 Preview Runtime Environment



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 a Custom Image

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

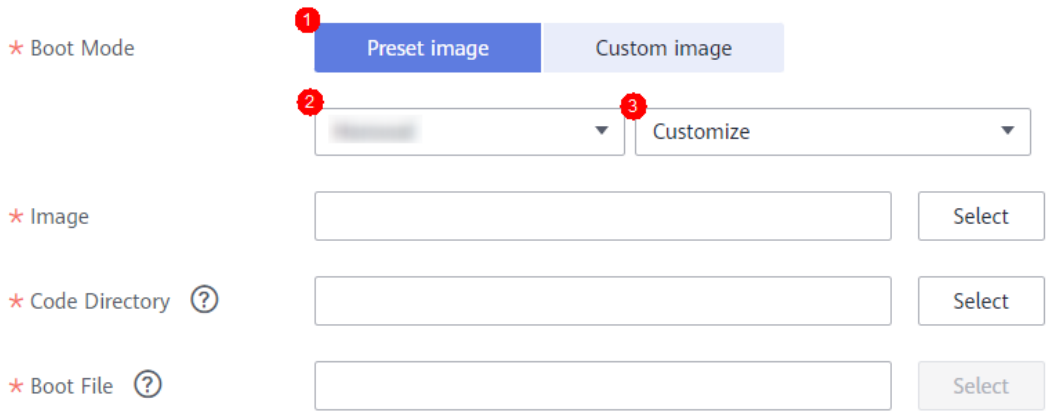
  If you use a preset image to create a training job and you need to modify or add some software dependencies based on the preset image, you can customize the preset image. In this case, select a preset image and choose **Customize** from the framework version drop-down list box.
- Using a custom image

  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.

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. For details about how to create a custom image based on a preset image, see **Using a Base Image to Create a Training Image**.

Figure 3-9 Creating an algorithm using a preset image with customization

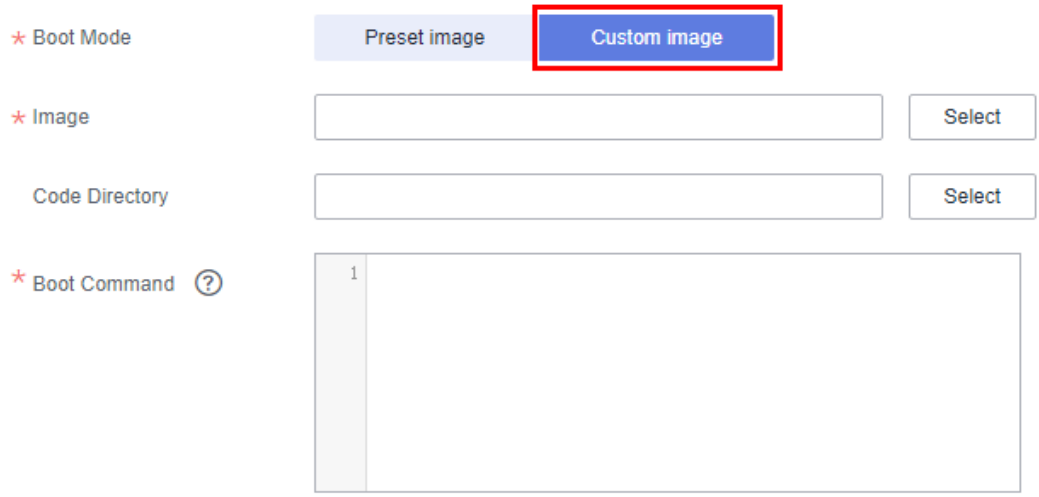


The process of this method is the same as that of creating a training job based on a preset image. For example:

- 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 3-10 Creating an algorithm using a custom image



For details about how to use custom images supported by the new-version 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.

  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 Searching for an Algorithm

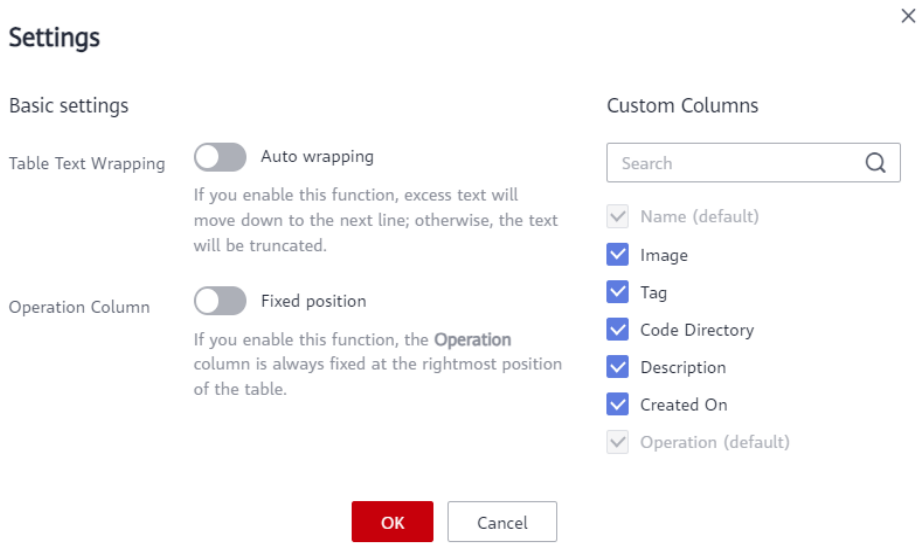
ModelArts allows you to quickly search for algorithms by performing the following operations.

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.

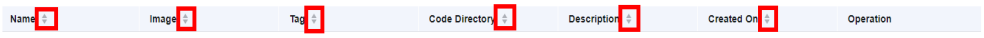
Operation 3: Configure the custom columns and other basic settings.

Figure 3-11 Configuring the custom columns and other basic settings



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

Figure 3-12 Sorting



3.5 Deleting an Algorithm

Deleting Your Algorithm

Choose **Algorithm Management** > **My algorithm** and click **Delete** in the **Operation** column of the target algorithm. In the displayed dialog box, confirm the deletion.

4 Performing a Training

4.1 Creating a Training Job

ModelArts training management enables you to create training jobs, review training statuses, and manage job versions. Model training is an iterative optimization process. Through unified training management, you can flexibly select algorithms, data, and hyperparameters to obtain the optimal input configuration and model. After comparing metrics between training versions, you can determine the most satisfactory training job.

Prerequisites

- Data is available either by creating a dataset in ModelArts or by uploading the data used for training to an OBS directory.
- An algorithm has been created either by using a preset image (**Using a Preset Image (Custom Script)**) or using a custom image (**Using a Custom Image**).
- At least one empty folder has been created in OBS for storing the training output. OBS buckets are not encrypted. ModelArts does not support encrypted OBS buckets. When creating an OBS bucket, do not enable bucket encryption.
- Access authorization has been configured. For details, see **Configuring Access Authorization (Global Configuration)**.

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**. Then, configure parameters.

Table 4-1 Basic information

Parameter	Description
Name	Name of a training job. The system automatically generates a name. You can rename it based on the following naming rules: • The name contains 1 to 64 characters. • Letters, digits, hyphens (-), and underscores (_) are allowed.
Description	Description of a training job.
Experiment	The options are **Create new**, **Use existing**, and **Not required**. If you set **Experiment** to **Create new**, enter an experiment name and description.

Table 4-2 Algorithm parameters (algorithm type)

Parameter	Option	Description
Algorithm Type > Custom algorithm > Boot Mode	Preset image	If **Boot Mode** is set to **Preset image**, select a preset engine and configure the code directory and boot file. • **Code Directory**: Select the code directory required for this training job. Upload code to an OBS bucket beforehand. The total size of files in the directory cannot exceed 5 GB, the number of files cannot exceed 1,000, and the folder depth cannot exceed 32. • **Boot File**: Select the Python boot script in the code directory. The boot file must be a .py file because ModelArts supports only boot files written in Python.

Parameter	Option	Description
Algorithm Type > Custom algorithm > Boot Mode	Preset image > Customize	If **Boot Mode** is set to **Preset image** and the engine version to **Customize**, configure the image, code directory, and boot file. • **Image**: Select a container image path. – Private images or shared images: Click **Select** on the right to select an SWR image. Ensure that the required image has been uploaded to SWR. – Public images: Enter the SWR image path in the format of *Organization name/Image name:Version name*. Do not contain the domain name (swr.<region>.xxx.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>.xxx.com/test-image/tensorflow2_1_1:1.1.1**, set this parameter to **test-images/tensorflow2_1_1:1.1.1**. • **Code Directory**: Select the code directory required for this training job. Upload code to an OBS bucket beforehand. The total size of files in the directory cannot exceed 5 GB, the number of files cannot exceed 1,000, and the folder depth cannot exceed 32. • **Boot File**: Select the Python boot script in the code directory. The boot file must be a .py file because ModelArts supports only boot files written in Python.

Parameter	Option	Description
Algorithm Type > Custom algorithm > Boot Mode	Custom image	If **Boot Mode** is set to **Custom image**, set **Image**, **Code Directory**, **User ID**, and **Boot Command**. For details, see **Using a Custom Image to Create an Algorithm**. • **Image**: Select a container image path. – Private images or shared images: Click **Select** on the right to select an SWR image. Ensure that the required image has been uploaded to SWR. – Public images: Enter the SWR image path in the format of *Organization name/Image name:Version name*. Do not contain the domain name (swr.<region>.xxx.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>.xxx.com/test-image/tensorflow2_1_1:1.1.1**, set this parameter to **test-images/tensorflow2_1_1:1.1.1**. • **Code Directory**: Select the code directory required for this training job. This parameter is optional. 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. • **User ID**: Enter the user ID for running the container. The default value **1000** is recommended. This parameter is optional. 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: – Public resource pool: **1000** to **65535** – Dedicated resource pool: **0** to **65535** • **Boot Command**: Enter the image boot command. This parameter is mandatory. The boot command will be automatically executed after the code directory is downloaded. – If the training boot script is a .py file, **train.py** for example, the boot command can be **python ${MA_JOB_DIR}/demo-code/train.py**. – If the training boot script is a .sh file, **main.sh** for example, the boot command can be **bash ${MA_JOB_DIR}/demo-code/main.sh**. Semicolons (;) and ampersands (&&) can be used to combine multiple boot commands. **demo-code**

Parameter	Option	Description
		(customizable) in the boot command is the last-level directory of the OBS path.
Algorithm Type > Custom algorithm	Local Code Directory	You can specify the local directory of a training container. When a training job starts, the system automatically downloads the code directory to this directory. The default local code directory is **/home/ma-user/modelarts/user-job-dir**. This parameter is optional.
Algorithm Type > Custom algorithm	Work Directory	Directory where the boot file in the training container is located. When a training job starts, the system automatically runs the **cd** command to change the work directory to the specified directory.
Algorithm Type	My algorithm	Select an algorithm or create an algorithm by referring to **Creating an Algorithm**.

Table 4-3 Algorithm parameters (input and output)

Parameter	Option	Description
Input	Name	The recommended value is **data_url**. The training input must match the input configuration set in your selected algorithm. For details, see **Table 3-3**. You can select a dataset or data path for data input. When the training job is started, ModelArts automatically downloads the data in the input path to the container directory for training.
	Data path	Select the training data from your OBS bucket. Click **Data path** and select the OBS bucket and folder in the dialog box displayed. **NOTE** If **Data path** is unavailable, the training data of the selected algorithm cannot be from an OBS path.

Parameter	Option	Description

	Obtained from	The following uses training input **data_path** as an example.
		If you select **Hyperparameters**, use this code to obtain the data:
		`import argparse`
		`parser = argparse.ArgumentParser()`
		`parser.add_argument('--data_path')`
		`args, unknown = parser.parse_known_args()`
		`data_path = args.data_path`
		If you select **Environment variables**, use this code to obtain the data:
		`import os`
		`data_path = os.getenv("data_path", "")`

Output	Name	The algorithm code reads the local path to the training output based on this parameter. The recommended value is **train_url**. The training output must match the output configuration set in your selected algorithm. For details, see **Table 3-4**. You can select an OBS path for data output. During training, ModelArts automatically uploads the training output to the OBS path.
	Data path	This data path stores the training output. During and after the training, the system automatically synchronizes files from the local directory to the data path. You can only select an OBS path as the data path. Select an OBS path for storing the training result. To minimize errors, select an empty directory.

	Obtained from	The following uses the training output **train_url** as an example.
		If you select **Hyperparameters**, use this code to obtain the data:
		`import argparse`
		`parser = argparse.ArgumentParser()`
		`parser.add_argument('--train_url')`
		`args, unknown = parser.parse_known_args()`
		`train_url = args.train_url`
		If you select **Environment variables**, use this code to obtain the data:
		`import os`
		`train_url = os.getenv("train_url", "")`

	Predownload	If you set **Predownload** to **Yes**, the system automatically downloads the files in the training output data path to the local directory of the training container before the training job is started. Select **Yes** for **resumable training and incremental training**.

Parameter	Option	Description
Hyperparameter	N/A	The value of this parameter varies according to the selected algorithm. If you have defined hyperparameters when creating an algorithm, all hyperparameters of the algorithm are displayed. Whether hyperparameters can be modified or deleted depends on how you configure the constraints when creating the algorithm. For details, see **Configuring Hyperparameters**.
Environment Variable	N/A	Environment variables, which you can add as required. For details about the environment variables preset in the training container, see **Viewing Environment Variables of a Training Container**.
Auto Restart	N/A	Number of retries for a failed training job. If this parameter is enabled, a failed training job will be automatically re-delivered and run. On the training job details page, you can review the number of retries for a failed training job. • This function is disabled by default. • If you enable this function, set the number of retries. The value ranges from **1** to **3** and cannot be changed.

NOTE

The training input, training output, and hyperparameters vary according to the selected algorithm.

If the system displays a message for **Input**, indicating there is no input channel for the selected algorithm, you do not need to set data input on this page.

If the system displays a message for **Output**, indicating there is no output channel for the selected algorithm, you do not need to set data output on this page.

If the system displays a message for **Hyperparameters**, indicating the selected algorithm does not support custom hyperparameters, you do not need to set hyperparameters on this page.

4. Select the instance specifications. The value range of the training parameters must comply with the constraints of the selected algorithm.

Table 4-4 Resource parameters

Parameter	Description
Resource Pool	Select a resource pool for the job. Public and dedicated resource pools are available for you to select. If you select a dedicated resource pool, you can review details about the pool. If the number of available cards of this pool is insufficient, jobs may need to be queued. In this case, use another resource pool or reduce the number of cards required. **NOTE** Dedicated resource pools can be accessed to your VPCs and subnets. For details, see **Interconnecting a VPC**. If you want to change the VPC accessible to your dedicated resource pool, see **Interconnecting a VPC**.
Resource Type	Select CPU or GPU as needed. Set this parameter based on the resource type specified in your training code.
Specifications	Select a resource flavor based on the resource type. If the type of resources to be used has been specified in your training code, only the options that comply with the constraints of the selected algorithm are available for you to choose. For example, if **GPU** is selected in the training code but you select **CPU** here, the training may fail. During training, ModelArts will mount NVME SSDs to the **/cache** directory. You can use this directory to store temporary files. The data disk size varies depending on the resource type. To prevent insufficient memory during training, click **Check Input Size** to check whether the disk size of selected instance specifications is sufficient for the input size. **NOTICE** The resource flavor **GPU:*n**nvidia-t4** (*n* indicates a specific number) does not support multi-process training.
Compute Nodes	Set the number of compute nodes. The default value is **1**.
Job Priority	When using a dedicated resource pool for training, 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**. 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**. You can change the priority of a pending job.

Parameter	Description
Persistent Log Saving	If you select CPU or GPU resources, **Persistent Log Saving** is available for you to set. This function is disabled by default. ModelArts automatically stores training logs for 30 days. You can download all logs on the job details page. After enabling this function, you can store training logs in a specified OBS directory. You are advised to select an empty OBS directory to store the log files generated during training.
Job Log Path	If you select Ascend resources, select an empty OBS directory for storing training logs. Ensure that you have read and write permissions to the selected OBS directory.
Event Notification	You can enable this function so you will be notified of specific events, such as job status changes or suspected suspensions, via an SMS or email. If you enable this function, configure the following parameters: ● **Topic**: Specify the topic of event notifications. You can create a topic on the SMN console. ● **Event**: Select events you want to subscribe to. The options include **JobStarted**, **JobCompleted**, **JobFailed**, **JobTerminated**, and **JobHanged**. **NOTE** ● 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. ● Currently, only training jobs using GPUs support **JobHanged** events.
Auto Stop	● After this parameter is enabled and the auto stop time is set, a training job automatically stops at the specified time. ● If this function is disabled, a training job will continue to run. ● The options are **1 hour**, **2 hours**, **4 hours**, **6 hours**, and **Customize** (1 hour to 720 hours).

5. Click **Submit** to create the training job.

   A training job generally runs for a period of time. To check the real-time status and basic information of a training job, switch to the training job list.

   – In the training job list, **Status** of the newly created training job is **Pending**.

   – When the status of a training job changes to **Completed**, the training job is complete, and the generated model is stored in the specified training output path.

- If the status is **Failed** or **Abnormal**, click the job name to go to the job details page and check logs for troubleshooting. For details, see **Reviewing Training Job Details**.

4.2 Reviewing Training Job Details

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

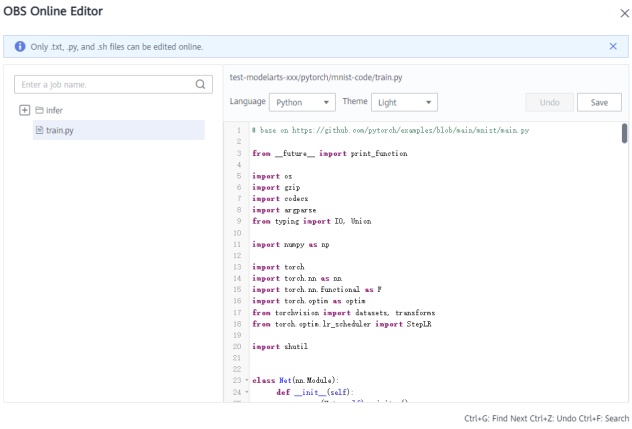
     Table 4-5 Basic job settings

Parameter	Description
Job ID	Unique ID of the training job.
Status	Status of the training job.
Created	Time when the training job is created.
Duration	Running duration of the training job.
Description	Description of the training job. You can click the edit icon to update the description of a training job.

   - Algorithm parameters

     Table 4-6 Algorithm parameters

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

Parameter	Description
Code Directory	OBS path to the code directory of the training job. 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.  **NOTE** This parameter is not supported when you use an algorithm subscribed from AI Hub to create a training job.
Boot File	Location where the training boot file is stored. **NOTE** This parameter is not supported when you use an algorithm subscribed from AI Hub to create a training job.
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 set for the training job.
Specifications	Training specifications used by the training job.
Input Path	OBS path where the input data is stored.
Parameter Name	Input path parameter specified in the algorithm code.
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 Path	OBS path where the output data is stored.
Parameter Name	Output path parameter specified in the algorithm code.

Parameter	Description
Local Path (Training Parameter Value)	Path for storing the output data in the ModelArts backend container.
Hyperparameter	Hyperparameters used in the training job.
Environment Variable	Environment variables for the training job.

4.3 Training Job Logs

4.3.1 Introduction to Training Job Logs

Overview

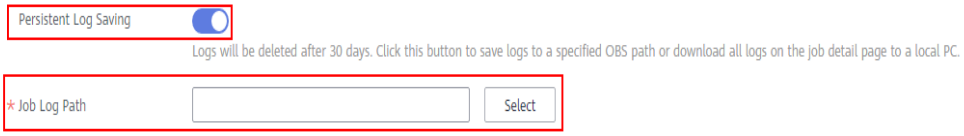
Training logs record the runtime process and exception information of training jobs and provide useful details for fault location. The standard output and standard error information in your code are displayed in training logs. If you encounter an issue during the execution of a ModelArts training job, 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 complete, you can view its historical logs on the ModelArts training job details page. ModelArts automatically stores the logs for 30 days.
- Permanent logs: dumped to your OBS bucket. When creating a training job, you can set an OBS dump path. You need to manually enable **Persistent Log Saving** for CPU- or GPU-based training jobs.

Figure 4-1 Enabling Persistent Log Saving



Real-time logs and historical logs have no difference in content. Real-time logs, historical logs, and permanent logs of CPU- or GPU-based training jobs are the same.

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**.

4.3.2 Common Logs

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

Log Type

Table 4-7 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
```

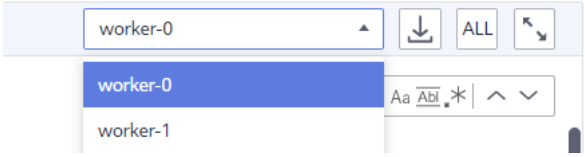
4.3.3 Viewing Training Job Logs

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

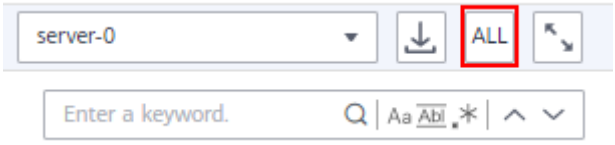
  You can preview logs of each compute node, if multiple compute nodes are used, in the training log pane by choosing the target node from the drop-down list on the right.

  Figure 4-2 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.

  Figure 4-3 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.

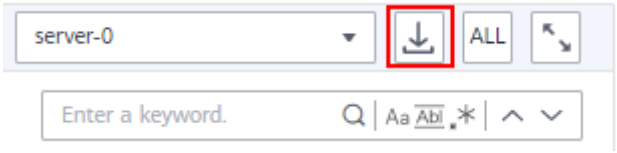
    Ensure that no privacy information is contained in the logs. Otherwise, information leakage may occur.

- 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 an Ascend compute node, 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.

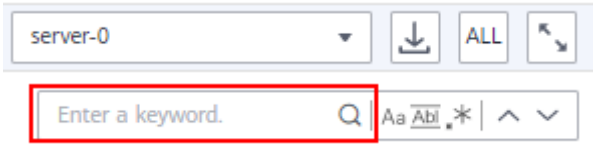
Figure 4-4 Downloading logs



- 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 4-5**.

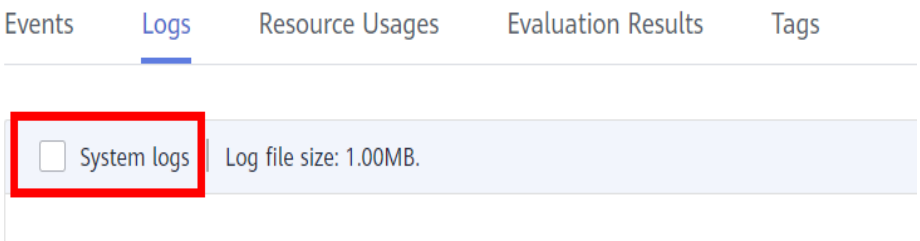
  Figure 4-5 Searching for logs by keyword

  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 4-6 System logs

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

4.3.4 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 4-7 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.

4.4 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.

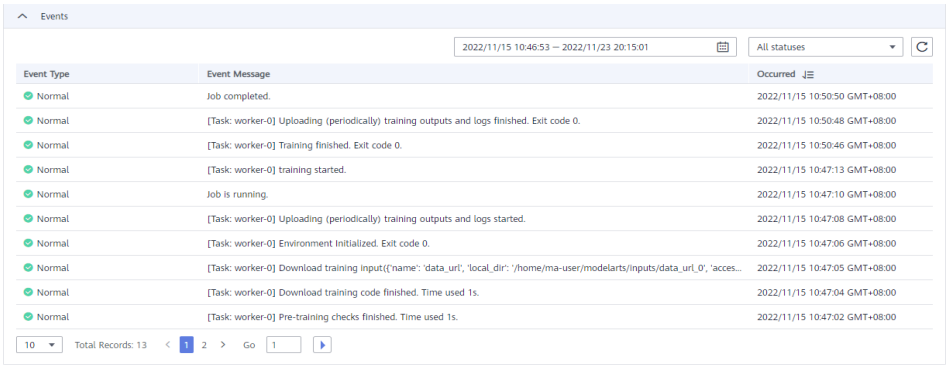
- Failed to be queued.
- The training job starts to run.
- Training job executed.
- Failed to run the training job.
- The training job is preempted.
- The system detects that your training job may be suspended. Go to the job details page to view the cause and handle the issue.
- The training job has 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. In the navigation pane of the ModelArts management console, choose **Training Management** > **Training Jobs**. In the training job list, click a job name.
2. Click **Events** to view events.

Figure 4-8 Events



4.5 Viewing the Resource Usage of a Training Job

Operations

You can view the resource usage of a compute node in the **Resource Usages** window. 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 4-9 Resource Usages

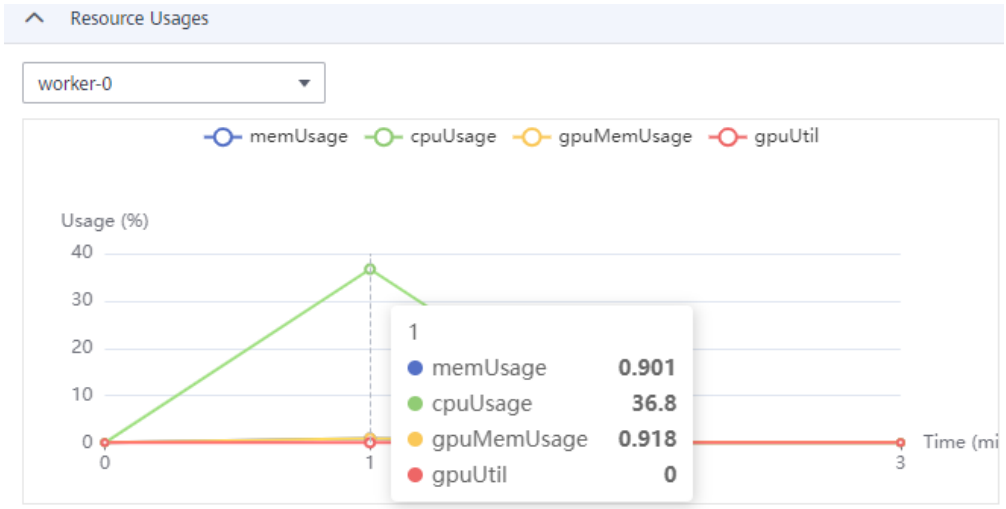


Table 4-8 Parameters

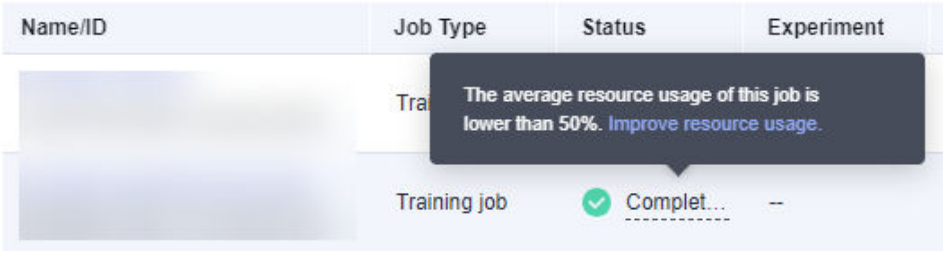
Parameter	Description
cpuUsage	CPU usage

Parameter	Description
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 a job is lower than 50%, an alarm is displayed in the training job list.

Figure 4-10 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 is: Summarize the usage of each GPU/NPU accelerator card at each time point of the job and calculate the average value. If a job uses multiple compute nodes, summarize the usage of all compute nodes and then obtain the average usage of a single job.

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.

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

    data_file_list, label_list = get_dataset(data_path, label_map_dict)

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

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

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

    config = tf.ConfigProto()
    config.gpu_options.allow_growth = True
    config.gpu_options.visible_device_list = '0'
```



```
    with tf.Session(graph=tf.Graph(), config=config) as sess:
        meta_graph_def = tf.saved_model.loader.load(
            sess, [tf.saved_model.tag_constants.SERVING], pb_dir)
        signature = meta_graph_def.signature_def
        signature_key = 'predict_object'
        input_key = 'images'
        output_key = 'logits'
        x_tensor_name = signature[signature_key].inputs[input_key].name
        y_tensor_name = signature[signature_key].outputs[output_key].name
        x = sess.graph.get_tensor_by_name(x_tensor_name)
        y = sess.graph.get_tensor_by_name(y_tensor_name)
        for img in img_list:
            pred_output = sess.run([y], {x: img})
            pred_output = softmax(pred_output[0])
            pred_list.append(pred_output[0].tolist())

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

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

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

4.7 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.

Figure 4-11 Setting environment variables

Environment Variables Preset in a Training Container

The following tables list environment variables preset in a training container, including **Table 4-9**, **Table 4-10**, **Table 4-11**, **Table 4-12**, **Table 4-13**, **Table 4-14**, and **Table 4-15**.

The environment variable values are examples.

Table 4-9 Path environment variables

Variable	Description	Example
PATH	Executable file paths	**PATH=/usr/local/nvidia/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/seccomponent/lib:/usr/local/cuda/lib64:/usr/local/cuda/compat:/root/miniconda3/lib:/usr/local/nvidia/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**

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

Table 4-10 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: • **worker** in MindSpore and PyTorch. • **learner** or **worker** in reinforcement learning engines. • **ps** or **worker** in **TensorFlow**.	**MA_TASK_NAME=worker**
MA_NUM_HOSTS	Compute nodes required for a training job.	**MA_NUM_HOSTS=4**
VC_TASK_INDEX	Sequence number of a job container for multi-node training. The value of the first container is **0**.	**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-8a83-000000000000,ob-a0978141-1712-4f9b-8a83-000000000000-worker-3.modelarts-job-a0978141-1712-4f9b-8a83-000000000000**

Table 4-11 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**
NCCL_SOCKET_IFNAME	IP interface to use for communication	**NCCL_SOCKET_IFNAME=bond0,eth0**

Table 4-12 OBS environment variables

Variable	Description	Example
S3_ENDPOINT	OBS endpoint	**S3_ENDPOINT=https://obs.region.xxx.com**
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 4-13 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.xxx.com**
MA_PIP_URL	Address of the PIP source	**MA_PIP_URL=http://repo.xxx.com/repository/pypi/simple/**
MA_APIGW_ENDPOINT	ModelArts API Gateway address	**MA_APIGW_ENDPOINT=https://modelarts.region.xxx.xxx.com**

Table 4-14 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-8a83-000000000000-worker-1**

Table 4-15 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**

4.8 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 algorithm creation 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 Hub.

Stopping a Training Job

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

A training job in the 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 target 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: Enable **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.

4.9 CloudShell

4.9.1 Logging In to a Training Container Using Cloud Shell

Application Scenarios

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

Constraints

The training container must be running in a dedicated resource pool.

Preparation: Assigning the Cloud Shell Permission to an IAM User

1. Log in to the console as a tenant user, click your username in the upper right corner, and choose **Identity and Access Management** from the drop-down list to switch to the IAM console.
2. On the IAM console, choose **Permissions** > **Policies/Roles** from the navigation pane and then click **Create Custom Policy** in the upper right corner. On the page that appears, configure parameters and click **OK**.
   - **Policy Name**: Enter a 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.
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.

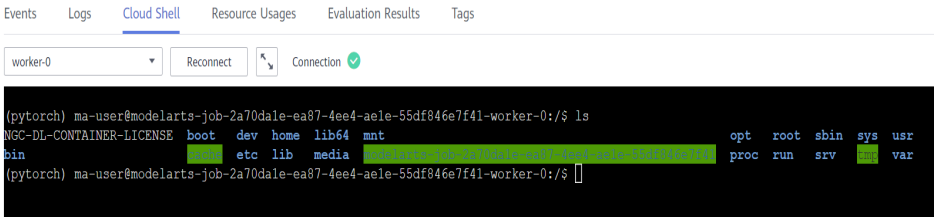
   If no user group is available, create one, add users to it through user group management, and configure authorization for the user group. If the target user is not in a user group, add the user to a user group through user group management.

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**. Go to the details page of the target training job and log in to the training container on the Cloud Shell tab.

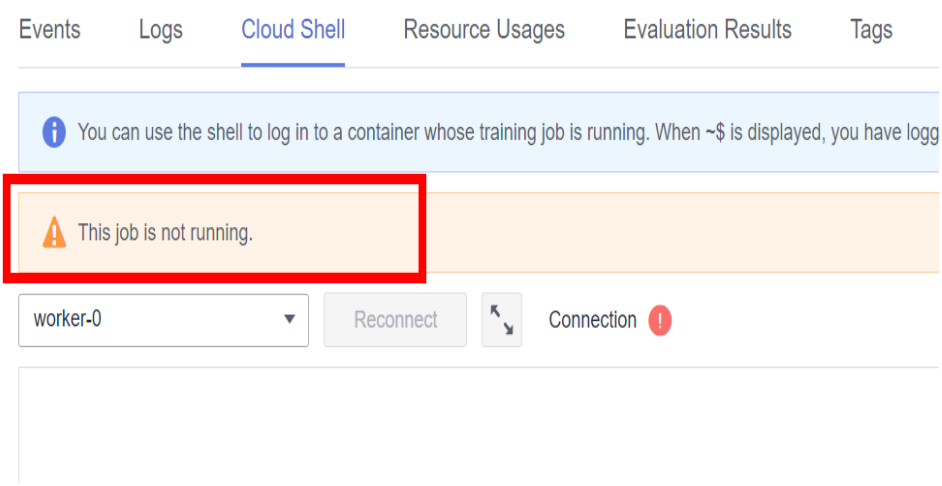
   Verify that the login is successful, as shown in the following figure.

   Figure 4-12 Cloud Shell



   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 4-13 Error message



4.10 Releasing Training Job Resources

Release resources of a training job when not in use.

- 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.

5 Training Experiment

5.1 Introduction to Experiment

An experiment is a job management capability provided by ModelArts. You can add training jobs to experiments for management.

Manage training jobs in an experiment by referring to the following instructions:

- For details about how to add a training job to an experiment, see **Adding a Training Job to an Experiment**.
- For details about how to view experiment information, see **Viewing an Experiment**.
- For details about how to delete an experiment, see **Deleting an Experiment**.

5.2 Adding a Training Job to an Experiment

To add a training job to an experiment, configure **Experiment** when creating a training job. The options are as follows:

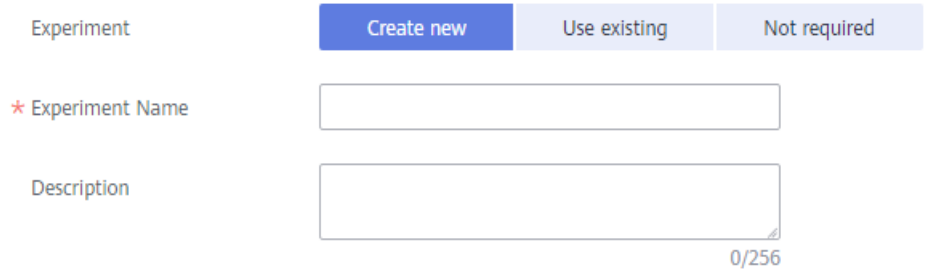
- **Create new**: An experiment can only be created when you create a training job. If you select this option, enter a new experiment name. After the job is submitted, the experiment is created and the job is added to the new experiment. The experiment name will be checked. If the experiment name already exists, the job cannot be submitted.
- **Use existing**: Select an existing experiment from the drop-down list box to add the job to the existing experiment.
- **Not required**: Select this option if you do not want to manage your job through an experiment. The experiment tab page of Training Management does not display a job that has not been added to an experiment.

Creating a Job to Be Added to an Experiment

Log in to the ModelArts management console, choose **Training Management** > **Training Jobs (New)**, and click **Create Training Job** in the upper right corner. The **Training** page is displayed.

On this page, you can configure **Experiment**, which defaults to **Create new**. In this case, enter a name for the new experiment. Then, an experiment is created after you create the training job.

Figure 5-1 Creating a training job

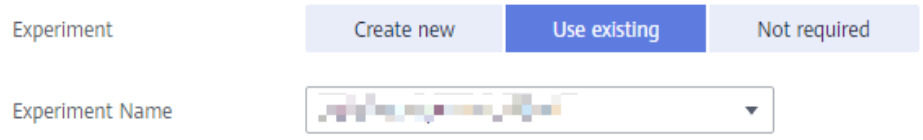


Adding a Created Job to an Experiment

Log in to the ModelArts console, choose **Training Management** > **Training Jobs (New)**, and click **Rebuild** in the **Operation** column of a target job. Alternatively, click the job name or ID in the job list. On the job details page, click **Rebuild** in the upper right corner.

- For a job that has not been added to an experiment, select **Create new** by default and enter a name for the new experiment. Then, an experiment is created after you create the training job.
- For a job that has been added to an experiment, select **Use existing** by default and select the experiment where the source job is.

Figure 5-2 Rebuilding a training job



5.3 Viewing an Experiment

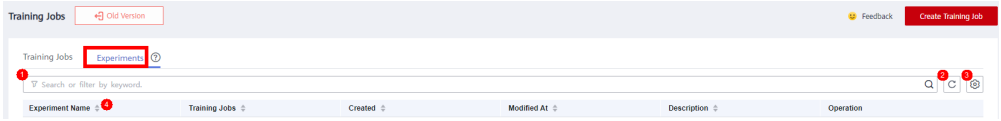
Viewing the Experiment List

1. Log in to the ModelArts console. In the left navigation pane, choose **Training Management** > **Training Jobs**. The **Training Jobs** page is displayed.
2. Click **Experiments** to go to the **Experiments** tab page. The experiment list displays some basic experiment information.

Table 5-1 Basic experiment information

Parameter	Description
Experiment Name	Experiment name, which can be changed on the experiment details page
Training Jobs	Number of training jobs in an experiment
Created	Time when an experiment is created
Modified At	Time when any of the following occurs: • Changing the experiment name • Modifying the description of the experiment • Adding a training job to or deleting a training job from the experiment
Description	Experiment description, which can be modified
Operation	You can delete the experiment.

Figure 5-3 Basic experiment information

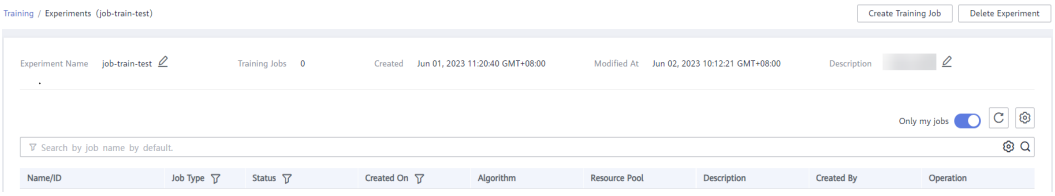


- You can search for experiments by experiment name, number of training jobs, creation time, modification time, and description.
- You can click the refresh button in the upper right corner of the job list to refresh the job list.
- You can click the setting button in the upper right corner of the experiment list to select items you want to display in the experiment list.
- You can click the arrow in the table header to sort experiments.

Viewing Experiment Details

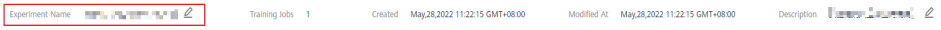
In the experiment list, click an experiment name to go to the experiment details page. Basic experiment information is displayed in the upper part of the experiment details page, and the job list of the experiment is displayed in the lower part of the experiment details page.

Figure 5-4 Viewing experiment details



- You can edit the name and description of an experiment.

  Figure 5-5 Editing the name and description of an experiment

- You can click **Only my jobs** to view the jobs that you have created and included in the experiment.

  NOTE

  By default, if an account has multiple IAM users, only the jobs of the current IAM user is displayed.

- You can search for jobs by name, ID, algorithm, status, creation time, job type, or resource pool.
- You can click the refresh button in the upper right corner of the job list to refresh the job list.
- You can click the setting button in the upper right corner of the job list to select items you want to display in the job list.

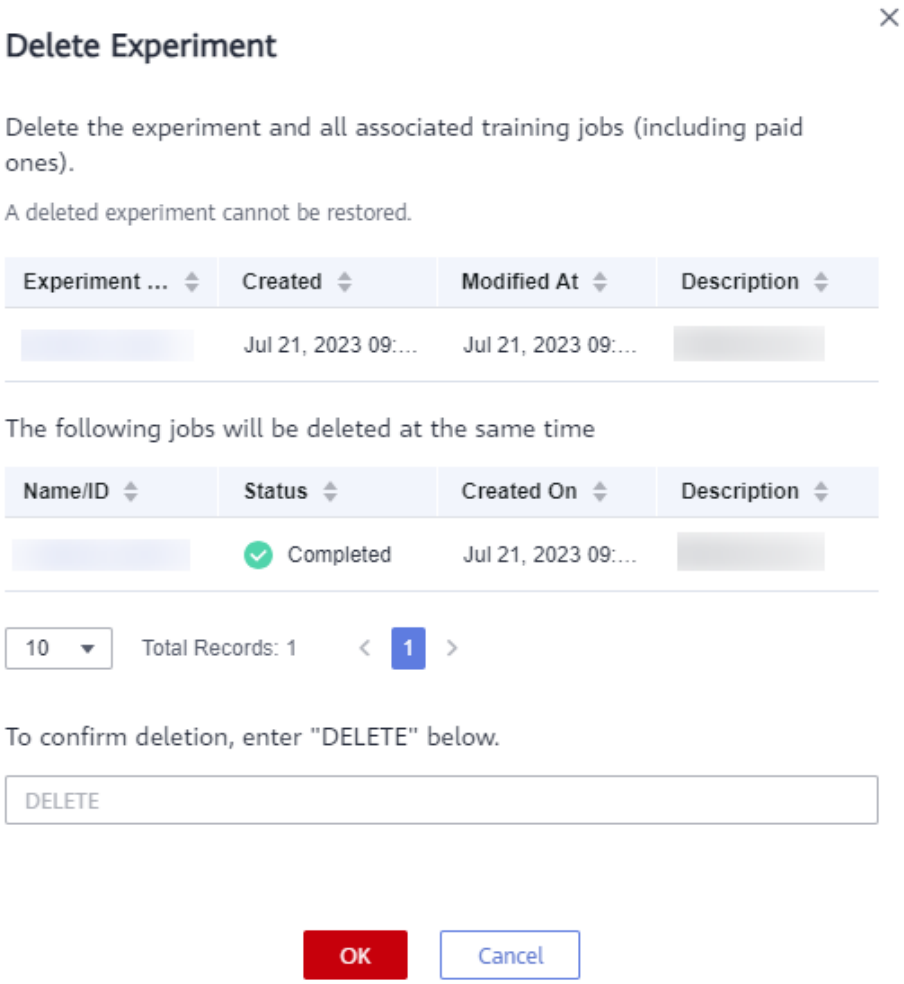
5.4 Deleting an Experiment

You can click **Delete** on the experiment list page or click **Delete Experiment** in the upper right corner of the experiment details page to delete an experiment. All jobs of the experiment are displayed on the **Delete Experiment** page. Enter **DELETE** and click **OK** to confirm the deletion.

CAUTION

After an experiment is deleted, all jobs in the experiment will be deleted accordingly and cannot be restored. Therefore, exercise cautions when performing this operation.

Figure 5-6 Deleting an experiment



6 Advanced Training Operations

6.1 Selecting a Training Mode

ModelArts provides different training modes for MindSpore engines and enables you to obtain different diagnosis information based on actual scenarios.

On the training job creation page, you can select **General**, **High performance**, or **Fault diagnosis** for training mode. The default value is **General**. For details about debugging information in **General** mode, see **Training Log Details**.

Use **High performance** and **Fault diagnosis** in the following scenarios:

- High performance: In high performance mode, certain O&M functions will be adjusted or even disabled to maximally accelerate the running speed, but this will deteriorate fault locating. This mode is suitable for stable networks requiring high performance.
- Fault diagnosis: In fault diagnosis mode, certain O&M functions will be enabled or adjusted to collect more information for locating faults. This mode provides fault diagnosis. You can select a diagnosis type as required.

Figure 6-1 Mode selection

The following table details debugging information obtained in each mode.

Table 6-1 Debugging information obtained in each mode

Debugging Information	General	High performance	Fault diagnosis	Description
MindSpore log levels	Info level	Error level	Info level	MindSpore framework runtime log

Debugging Information	General	High performance	Fault diagnosis	Description
Running Data Recorder (RDR)	Disabled	Disabled	Enabled	If a running exception occurs, the recorded MindSpore data is automatically exported to help locate the exception cause. Different data is exported for different exceptions. For details about RDR, see **MindSpore Documentation**.
analyze_fail.dat	Enabled by default and uploaded to the training job log path			Graph build failure information is automatically exported for inference process analysis.
Dump data	Enabled by default and uploaded to the training job log path			Dump data is exported when an exception occurs during backend running.

In the fault diagnosis mode, after the fault diagnosis function is enabled, you can view the following fault diagnosis data: The following data is stored in the OBS directory in the training log path.

Description of the training output log file in the fault diagnosis mode:

```
{obs-log-path}/
    modelarts-job-{job-id}-worker-{index}.log # Displayed log summary
    modelarts-job-{job-id}-proc-rank-{rank-id}-device-{device-id}.txt # Displayed logs of each device
    modelarts-job-{job-id}/
        ascend/
            npu_collect/rank_{id}/ # Output path for TFAdapter DUMP GRAPH and GE DUMP GRAPH, generated only for the TensorFlow framework
            process_log/rank_{id}/ # Plog log path
            msnpureport/{task-index}/ # msnpureport tool execution logs, which you do not need to pay attention to
        mindspore/
            log/ # MindSpore framework logs and MindSpore fault diagnosis data
```

Table 6-2 Fault diagnosis data of MindSpore

Category	Description
CANN framework logs and fault diagnosis data	Host logs of the INFO or higher levels, including CANN software stack logs and driver logs.
MindSpore framework logs and fault diagnosis data	MindSpore framework logs of the INFO or higher levels.

Category	Description
	RDR file. If a running exception occurs, the recorded MindSpore data is automatically exported to help locate the exception cause. Different data is exported for different exceptions.
	analyze_fail.dat. Graph build failure information is automatically exported for inference process analysis.
	Dump data, which is exported when an exception occurs during backend running

On the training job creation page, select the MindSpore algorithm and set **Resource Type** to **Ascend**, and then you can enable fault diagnosis.

Figure 6-2 Resource Type

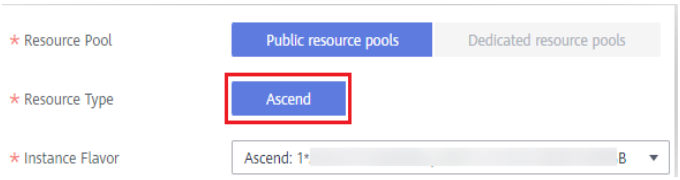


Figure 6-3 Enabling fault diagnosis

6.2 Automatic Recovery from a Training Fault

6.2.1 Training Fault Tolerance Check

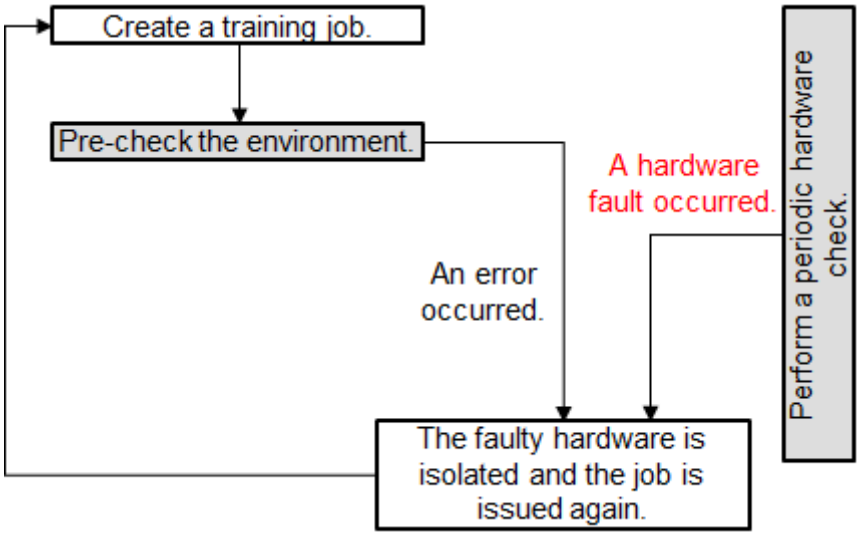
During model training, a training failure may occur due to a hardware fault. For hardware faults, ModelArts provides fault tolerance check to isolate faulty nodes to improve user experience in training.

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.

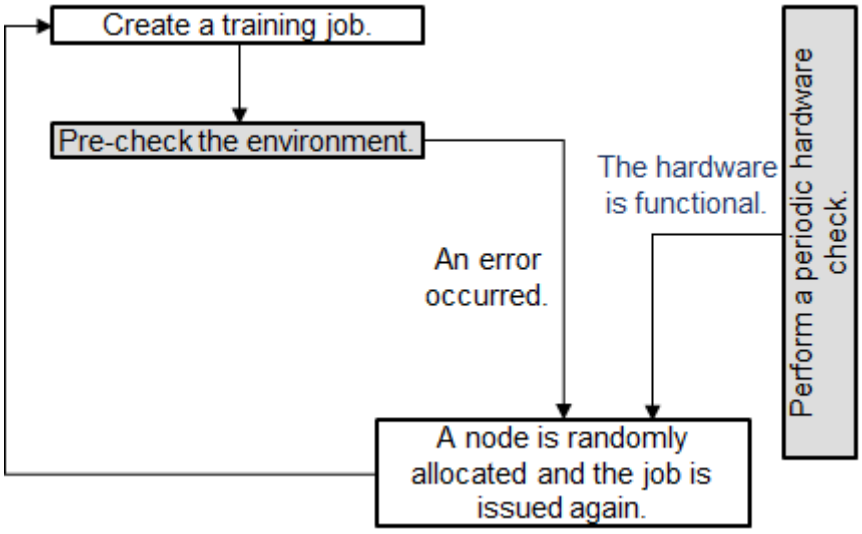
- Scenario 1: The environment pre-check fails, and the hardware is faulty. Then, ModelArts automatically isolates all faulty nodes and issues the training job again.

Figure 6-4 Pre-check failure and hardware fault



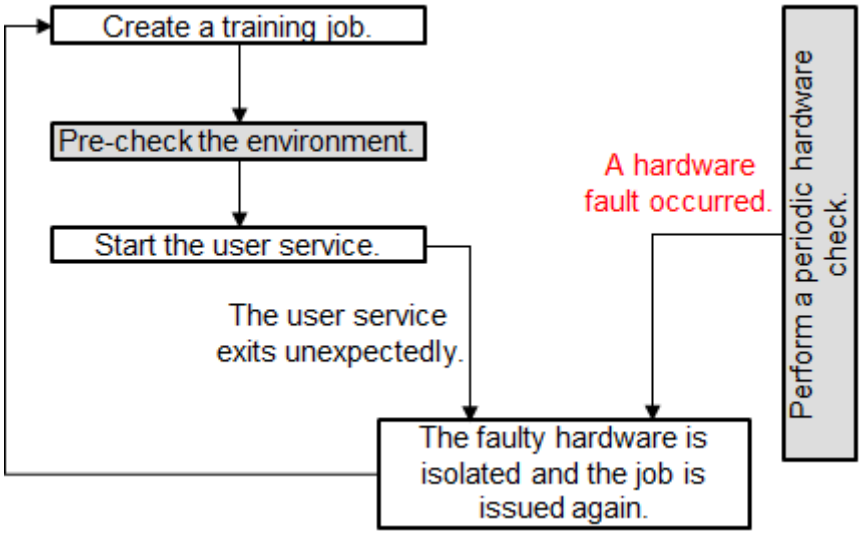
- Scenario 2: The environment pre-check fails but the hardware is functional. Then, ModelArts automatically isolates all faulty nodes and issues the training job again.

Figure 6-5 Pre-check failure but functional hardware



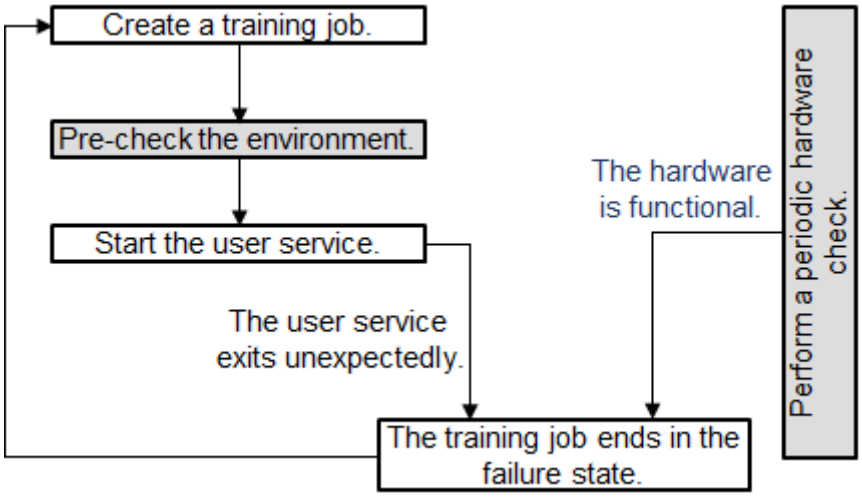
- Scenario 3: The environment pre-check is successful and the user service starts. A hardware fault occurs and the user service exits unexpectedly. Then, ModelArts automatically isolates all faulty nodes and issues the training job again.

Figure 6-6 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 6-7 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**.

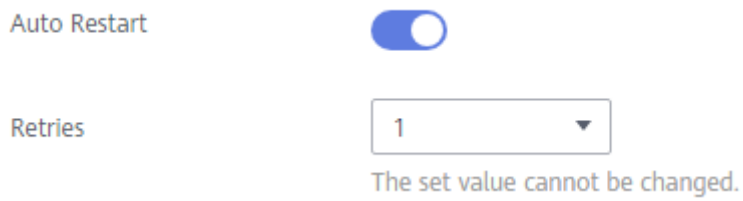
Enabling Fault Tolerance Check

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 3.

  Figure 6-8 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 **3**. 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.

```
{
    "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.	• 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**.

6.3 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.

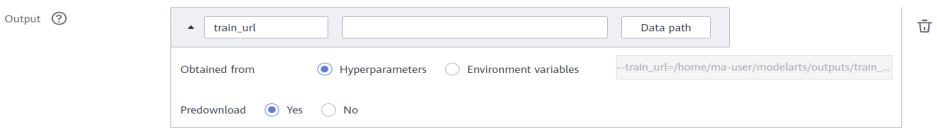
To resume a training job, load a checkpoint and use the checkpoint information to initialize the training status. To do so, add reload ckpt to the code.

Resumable Training and Incremental Training in ModelArts

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

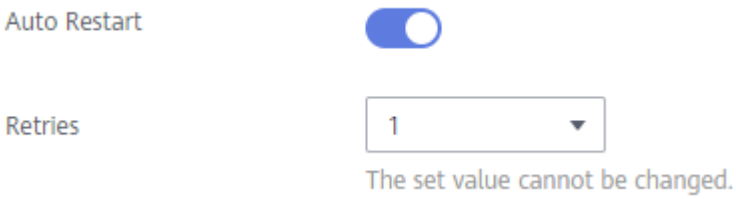
When creating a training job, configure the data path to the training output, save checkpoints in this data path, and set **Predownload** to **Yes**. If you set **Predownload** to **Yes**, the system automatically downloads the **checkpoint** file in the training output data path to a local directory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

Figure 6-9 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.

Figure 6-10 Auto Restart



6.4 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.

Cases

Common cases and solutions to training job suspension are as follows:

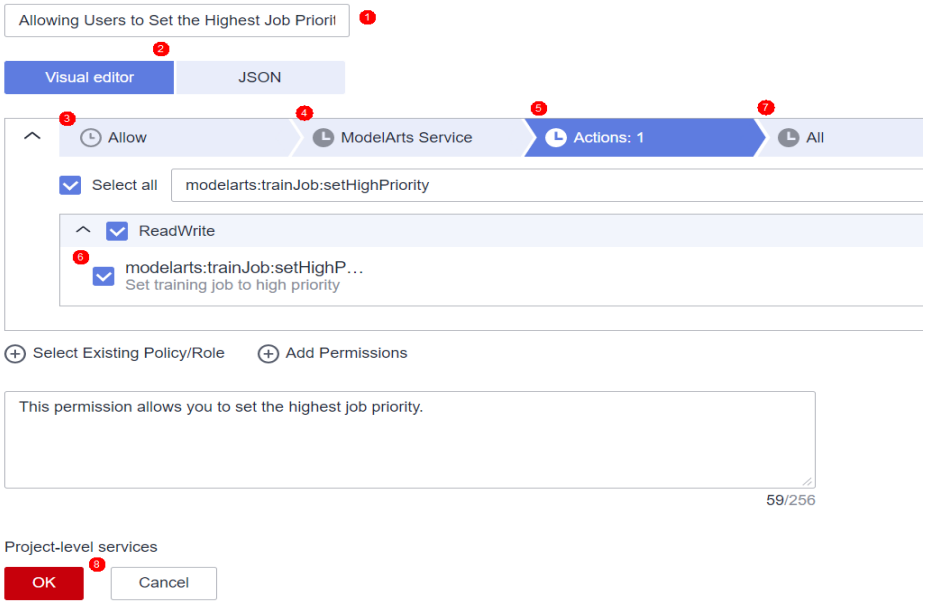
6.5 Permission to Set the Highest Job Priority

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 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 6-11 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.

7 Visualized Model Training

7.1 Introduction to Training Job Visualization

ModelArts notebook of the new version supports TensorBoard and MindInsight for visualizing training jobs. In the development environment, use small datasets to train and debug algorithms, during which you can check algorithm convergence and detect issues to facilitate debugging.

You can create visualization jobs of TensorBoard and MindInsight types on ModelArts.

Both TensorBoard and MindInsight effectively display the change trend of a training job and the data used in the training.

- TensorBoard

  TensorBoard effectively displays the computational graph of TensorFlow in the running process, the trend of all metrics in time, and the data used in the training. For more details about TensorBoard, see **TensorBoard official website**.

  TensorBoard visualization training jobs support only CPU and GPU flavors based on TensorFlow 2.1, and PyTorch 1.4 and 1.8 images. Select images and flavors based on the site requirements.

- MindInsight

  MindInsight visualizes information such as scalars, images, computational graphs, and model hyperparameters during training. It also provides functions such as training dashboard, model lineage, data lineage, and performance debugging, helping you train and debug models efficiently. MindInsight supports MindSpore training jobs. For more information about MindInsight, see **MindSpore official website**.

  The following shows the images and flavors supported by MindInsight visualization training jobs, and select images and flavors based on the site requirements.

  - MindSpore 1.2.0 (CPU or GPU)
  - MindSpore 1.5.x or later (Ascend)

You can use the summary file generated during model training to create a visualization job in Notebook of DevEnviron.

- For details about how to create a MindInsight visualization job in a development environment, see **MindInsight Visualization Jobs**.
- For details about how to create a TensorBoard visualization job in a development environment, see **TensorBoard Visualization Jobs**.

7.2 MindInsight Visualization Jobs

ModelArts notebook of the new version supports MindInsight visualization jobs. In the development environment, use small datasets to train and debug algorithms, during which you can check algorithm convergence and detect issues to facilitate debugging.

MindInsight visualizes information such as scalars, images, computational graphs, and model hyperparameters during training. It also provides functions such as training dashboard, model lineage, data lineage, and performance debugging, helping you train and debug models efficiently. MindInsight supports MindSpore training jobs. For more information about MindInsight, see **MindSpore official website**.

MindSpore allows you to save data into the summary log file and obtain the data on the MindInsight GUI.

Prerequisites

When using MindSpore to compile a training script, add the code for collecting the summary record to the script to ensure that the summary file is generated in the training result.

For details, see **Collecting Summary Record**.

Precautions

- To run a MindInsight training job in a development environment, start MindInsight and then the training process.
- Only one-card single-node training is supported.

Creating a MindInsight Visualization Job in a Development Environment

Step 1 Create a Development Environment and Access It Online

Step 2 Upload the Summary Data

Step 3 Start MindInsight

Step 4 View Visualized Data on the Training Dashboard

Step 1 Create a Development Environment and Access It Online

On the ModelArts management console, choose **DevEnviron > Notebook** to access notebook of the new version and create a MindSpore instance. After the instance is created, click **Open** in the **Operation** column of the instance to access it online.

The following shows the images and flavors supported by MindInsight visualization training jobs, and select images and flavors based on the site requirements.

- MindSpore 1.2.0 (CPU or GPU)

Step 2 Upload the Summary Data

Summary data is required for using MindInsight visualization functions in DevEnviron.

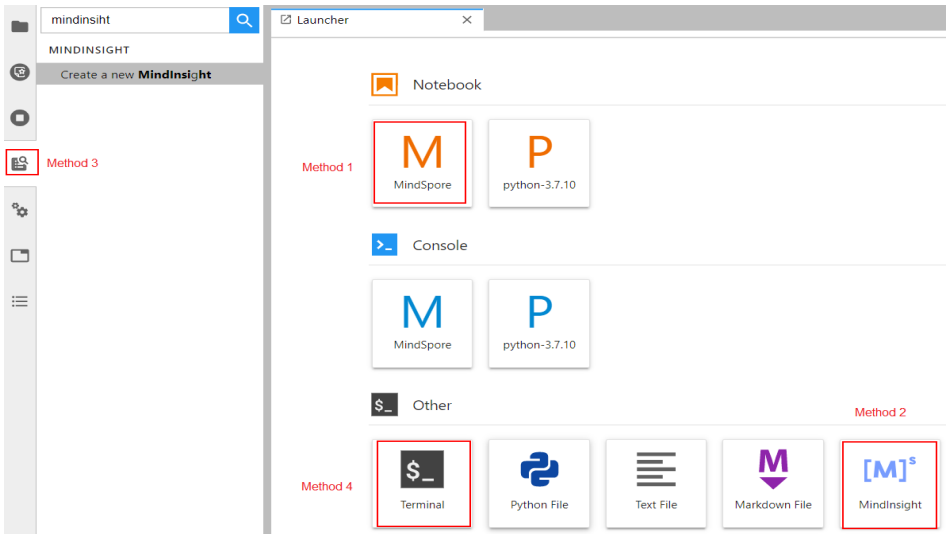
You can upload the summary data to the **/home/ma-user/work/** directory in the development environment or store it in the OBS parallel file system.

- For details about how to upload the summary data to the notebook path **/home/ma-user/work/**, see **Uploading Files to JupyterLab**.
- If you want the notebook development environment to mount the OBS parallel file system directory and read the summary data, upload the summary file generated during model training to the OBS parallel file system When MindInsight is started in a notebook instance, the notebook instance automatically mounts the OBS parallel file system directory and reads the summary data.

Step 3 Start MindInsight

Choose a way you like to start MindInsight in JupyterLab.

Figure 7-1 Starting MindInsight in JupyterLab



Method 1

1. Click MindSpore to go to the JupyterLab development environment. The **.ipynb** file is automatically created.

2. Enter the following command in the dialog box:

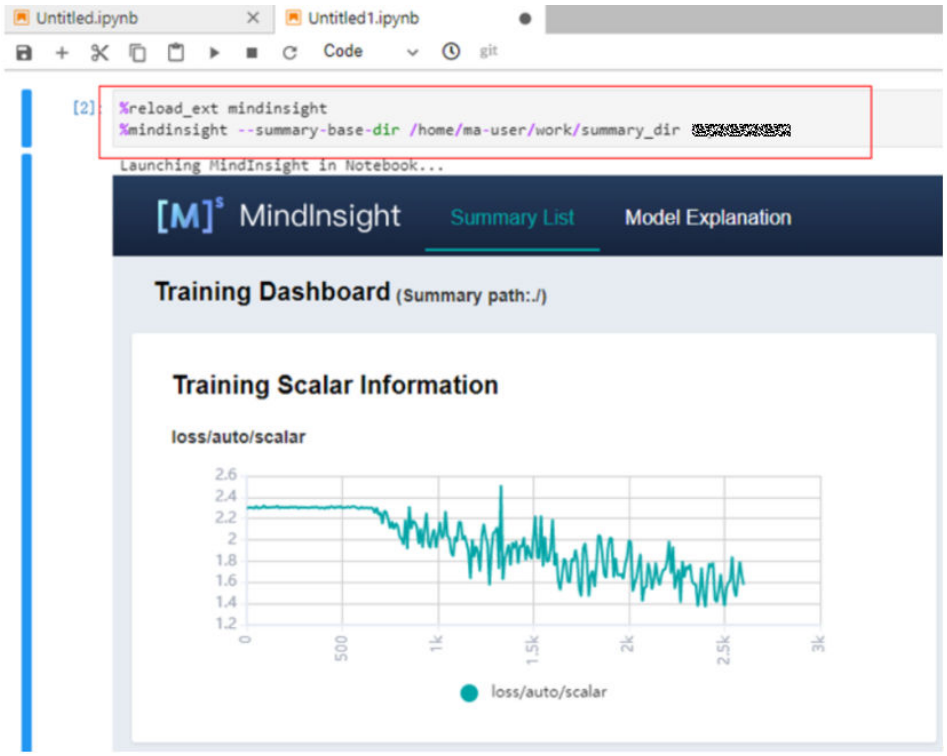
```
%reload_ext mindinsight
%mindinsight --port {PORT} --summary-base-dir {SUMMARY_BASE_DIR}
```

Parameters:

- **port {*PORT*}**: web service port for visualization, which defaults to **8080**. If the default port **8080** is occupied, specify a port ranging from 1 to 65535.
- **summary-base-dir {*SUMMARY_BASE_DIR*}**: data storage path in the development environment
  - Local path of the development environment: **./work/xxx** (relative path) or /**home/ma-user/work/xxx** (absolute path)
  - Path of the OBS parallel file system bucket: **obs://xxx/**

For example:
If the summary data is stored in **/home/ma-user/work/** of the development environment, run the following command:
%mindinsight --summary-base-dir /home/ma-user/work/xxx
or
If the summary data is stored in the OBS parallel file system, run the following command and the development environment automatically mounts the storage path of the OBS parallel file system and reads data.
%mindinsight --summary-base-dir obs://xxx/

Figure 7-2 MindInsight page (1)



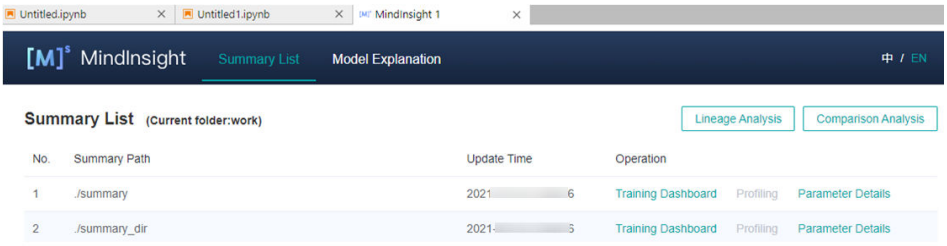
Method 2

Click  to go to the MindInsight page.

The directory **/home/ma-user/work/** is read by default.

All project log names are displayed in the **Runs** area. You can view the logs of the target project in the **Runs** area on the left.

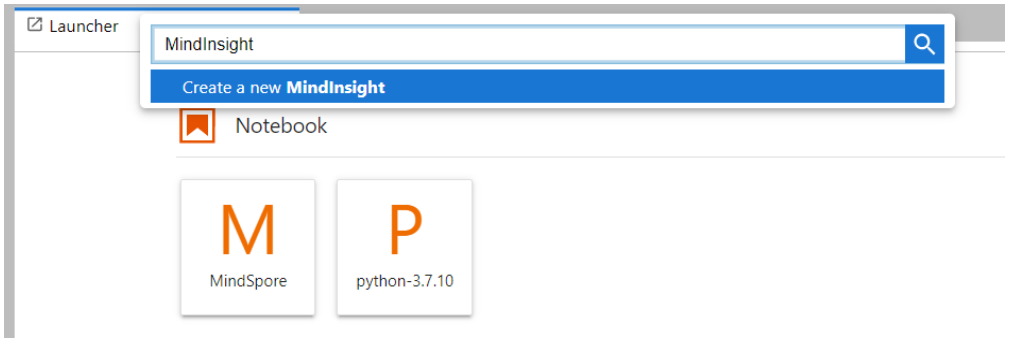
Figure 7-3 MindInsight page (2)



Method 3

1. Choose **View** > **Activate Command Palette**, enter **MindInsight** in the search box, and click **Create a new MindInsight**.

   Figure 7-4 Create a new MindInsight

2. Enter the path of the summary data you want to view or the storage path of the OBS parallel file system, and click **CREATE**.
   - Local path of the development environment: **./summary** (relative path) or **/home/ma-user/work/summary** (absolute path)
   - Path of the OBS parallel file system: **obs://xxx/**

Figure 7-5 Entering the summary data path

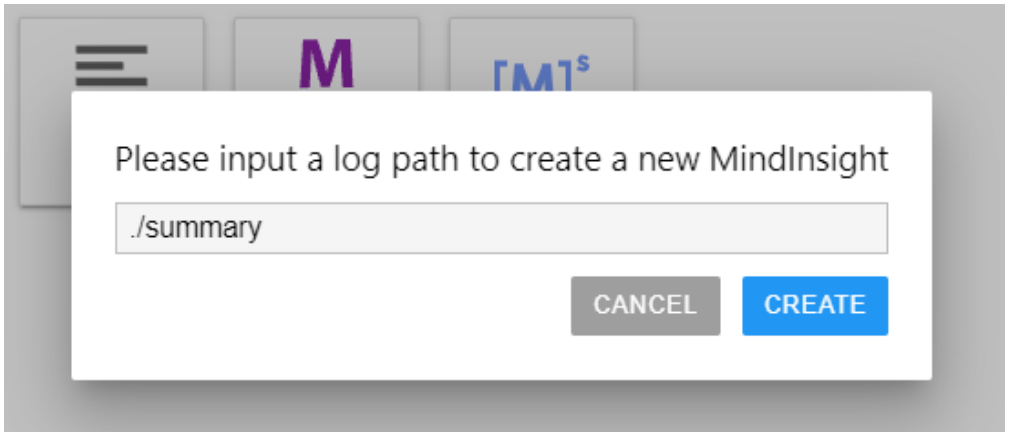
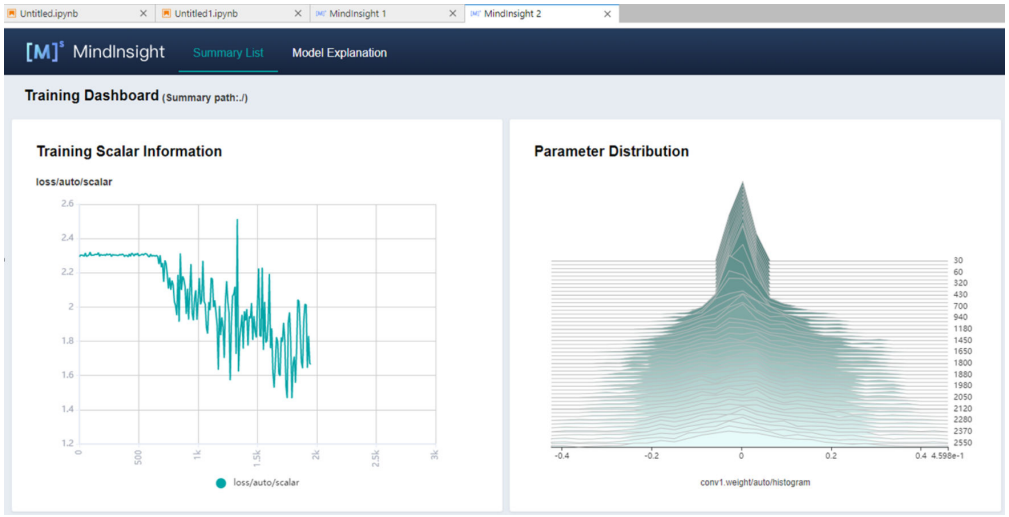


Figure 7-6 MindInsight page (3)



NOTE

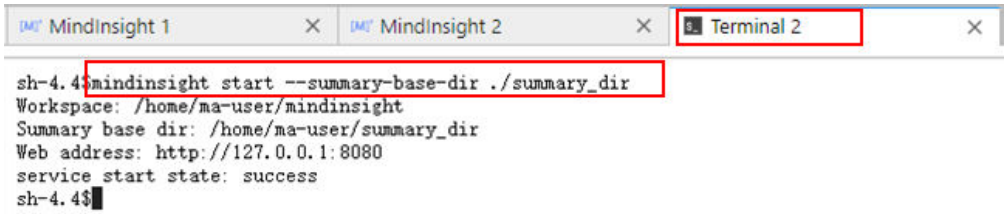
A maximum of 10 MindInsight instances can be started using methods 2 and 3.

Method 4

Click Terminal and run the following command. (In this way, the UI cannot be displayed.)

```
mindinsight start --summary-base-dir ./summary_dir
```


Figure 7-7 Opening MindInsight through Terminal



Step 4 View Visualized Data on the Training Dashboard

The training dashboard is important for MindInsight visualization. The training dashboard allows for scalar visualization, parameter distribution visualization, computational graph visualization, dataset graph visualization, image visualization, and tensor visualization.

For more information, see **Viewing Dashboard** on the MindSpore official website.

Related Operations

To stop a MindInsight instance, perform the following steps:

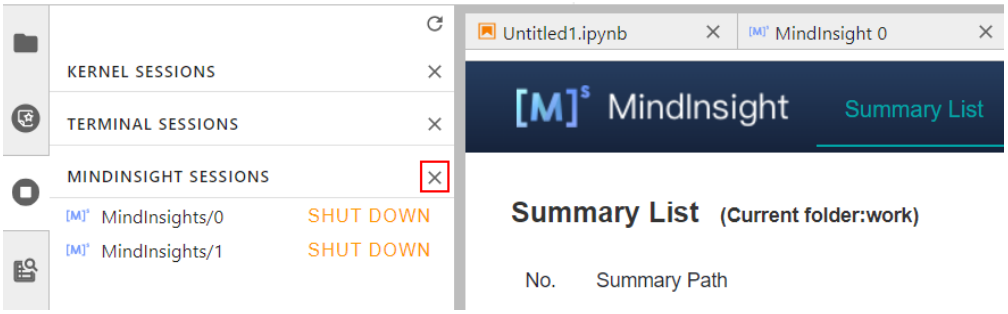
- Method 1: Enter the following command in the **.ipynb** file window of JupyterLab. Replace **8080** with the actual port number for **starting MindInsight**.
  !mindinsight stop --port *8080*
- Method 2: Click  . The MindInsight instance management page is displayed, which shows all started MindInsight instances. Click **SHUT DOWN** next to an instance to stop it.

  Figure 7-8 Clicking SHUT DOWN to stop an instance

- Method 3: Click  in the following figure to close all started MindInsight instances.

Figure 7-9 Stopping all started MindInsight instances



- Method 4 (not recommended): Close the MindInsight window on JupyterLab. In this case, only the visualization window is closed, but the instance is still running on the backend.

7.3 TensorBoard Visualization Jobs

ModelArts supports TensorBoard for visualizing training jobs. TensorBoard is a visualization tool package of TensorFlow. It provides visualization functions and tools required for machine learning experiments.

TensorBoard effectively displays the computational graph of TensorFlow in the running process, the trend of all metrics in time, and the data used in the training.

Prerequisites

When you compile a training script, add the code for collecting the summary record to the script to ensure that the summary file is generated in the training result.

For details about how to add the code for collecting the summary record to a TensorFlow-powered training script, see **TensorFlow official website**.

Precautions

- TensorBoard visualization training jobs support only CPU and GPU flavors based on TensorFlow2.1, and PyTorch1.4, 1.8 or later images. Select images and flavors based on the site requirements.

Process of Creating a TensorBoard Visualization Job in a Development Environment

Step 1 Create a Development Environment and Access It Online

Step 2 Upload the Summary Data

Step 3 Start TensorBoard

Step 4 View Visualized Data on the Training Dashboard

Step 1 Create a Development Environment and Access It Online

On the ModelArts management console, choose **DevEnviron** > **Notebook** to access notebook of the new version and create an instance using a TensorFlow or

PyTorch image. After the instance is created, click **Open** in the **Operation** column of the instance to access it online.

TensorBoard visualization training jobs support only CPU and GPU flavors based on TensorFlow2.1, and PyTorch1.4, 1.8 or later images. Select images and flavors based on the site requirements.

Step 2 Upload the Summary Data

Summary data is required for using TensorBoard visualization functions in DevEnviron.

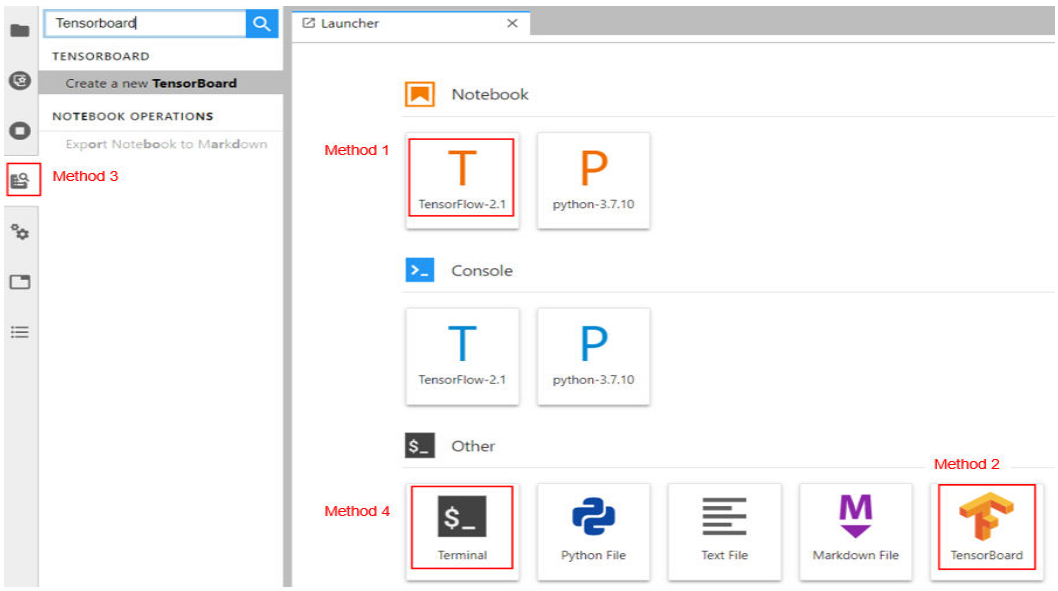
You can upload the summary data to the **/home/ma-user/work/** directory in the development environment or store it in the OBS parallel file system.

- For details about how to upload the summary data to the notebook path **/home/ma-user/work/**, see **Uploading Files to JupyterLab**.
- If you want the notebook development environment to mount the OBS parallel file system directory and read the summary data, upload the summary file generated during model training to the OBS parallel file system When TensorBoard is started in a notebook instance, the notebook instance automatically mounts the OBS parallel file system directory and reads the summary data.

Step 3 Start TensorBoard

Choose a way you like to start TensorBoard in JupyterLab.

Figure 7-10 Starting TensorBoard in JupyterLab



NOTICE

You can upgrade TensorBoard to any version except 2.4.0. After the upgrade, only method 1 starts the new-version TensorBoard. Using other methods will still start TensorBoard 2.1.1.

Method 1

1. Click to go to the JupyterLab development environment. The **.ipynb** file is automatically created.
2. Enter the following command in the dialog box:

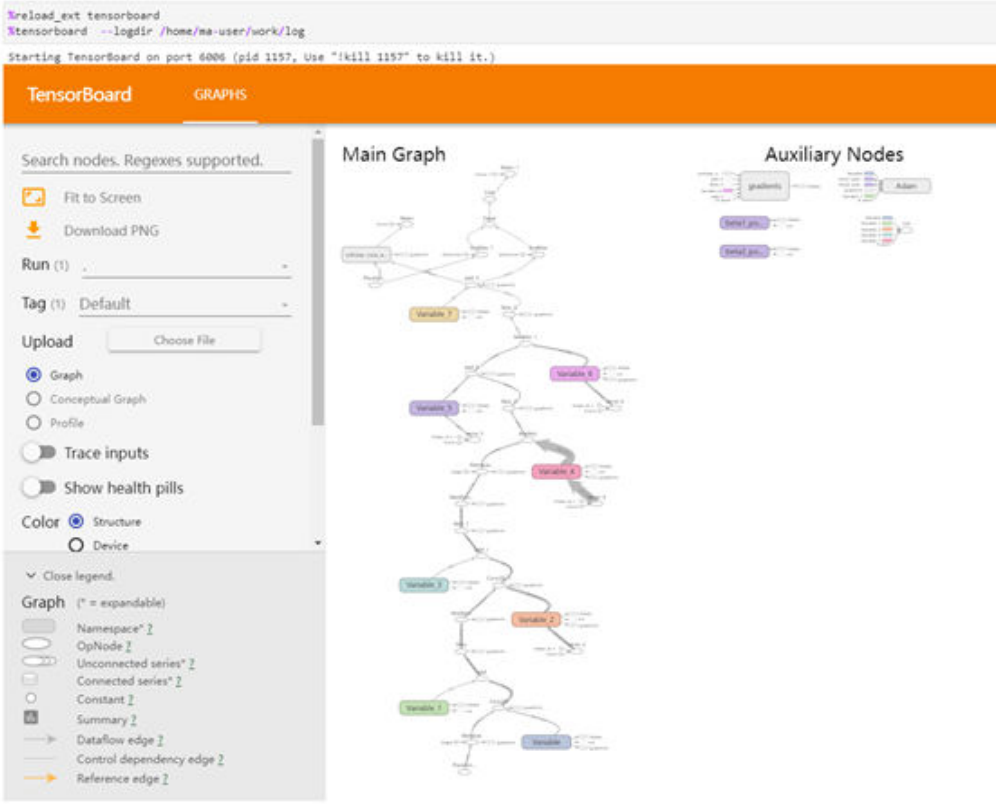
```
%reload_ext ma_tensorboard
%ma_tensorboard  --port {PORT} --logdir {BASE_DIR}
```

   Parameters:

   - **port {***PORT***}**: web service port for visualization, which defaults to **8080**. If the default port **8080** is occupied, specify a port ranging from 1 to 65535.
   - **logdir {***BASE_DIR***}**: data storage path in the development environment
     - Local path of the development environment: **./work/xxx** (relative path) or /**home/ma-user/work/xxx** (absolute path)
     - Path of the OBS parallel file system: **obs://xxx/**

```
For example:
# If the summary data is stored in /home/ma-user/work/ of the development environment, run the following command:
%ma_tensorboard  --port {PORT} --logdir /home/ma-user/work/xxx
or
# If the summary data is stored in the OBS parallel file system, run the following command and the development environment automatically mounts the storage path of the OBS parallel file system and reads data.
%ma_tensorboard  --port {PORT} --logdir obs://xxx/
```


Figure 7-11 TensorBoard page (1)



Method 2

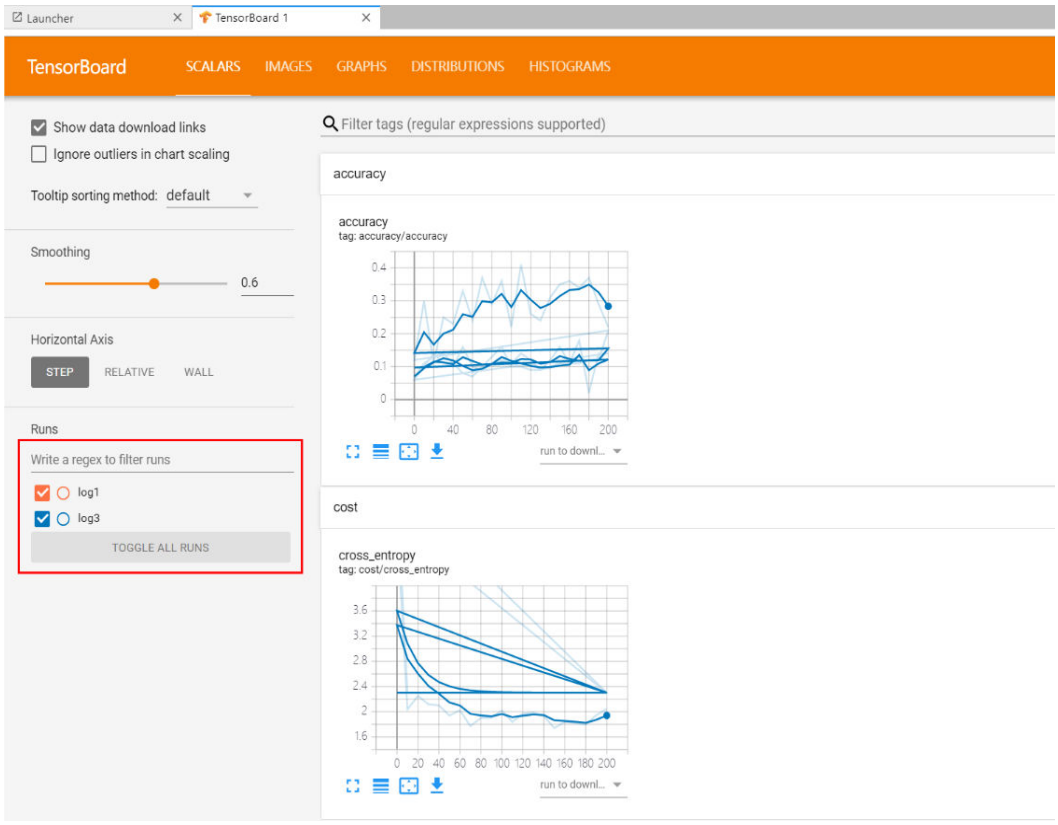
Click 

to go to the TensorBoard page.

The directory **/home/ma-user/work/** is read by default.

All project log names are displayed in the **Runs** area. You can view the logs of the target project in the **Runs** area on the left.

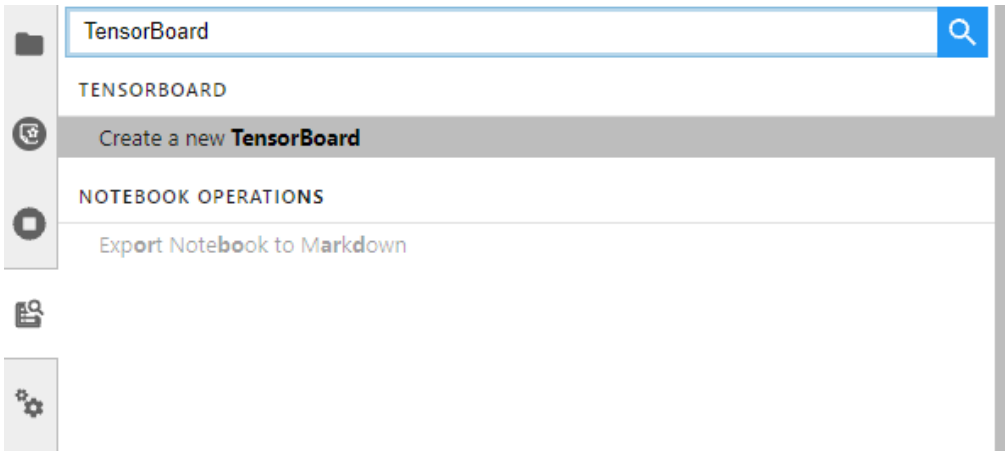
Figure 7-12 TensorBoard page (2)



Method 3

1. Choose **View** > **Activate Command Palette**, enter **TensorBoard** in the search box, and click **Create a new TensorBoard**.

   Figure 7-13 Creating a TensorBoard instance

2. Enter the path of the summary data you want to view or the storage path of the OBS parallel file system.
   - Local path of the development environment: **./summary** (relative path) or **/home/ma-user/work/summary** (absolute path)

– Path of the OBS parallel file system bucket: **obs://xxx/**

Figure 7-14 Entering the summary data path

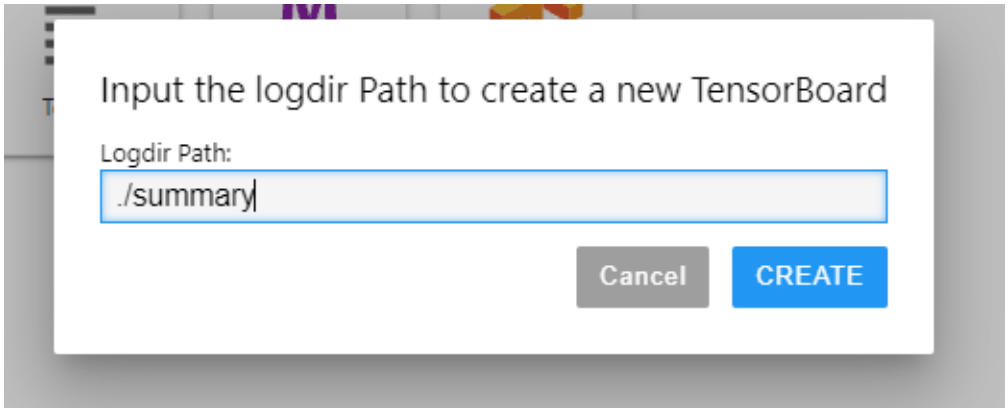
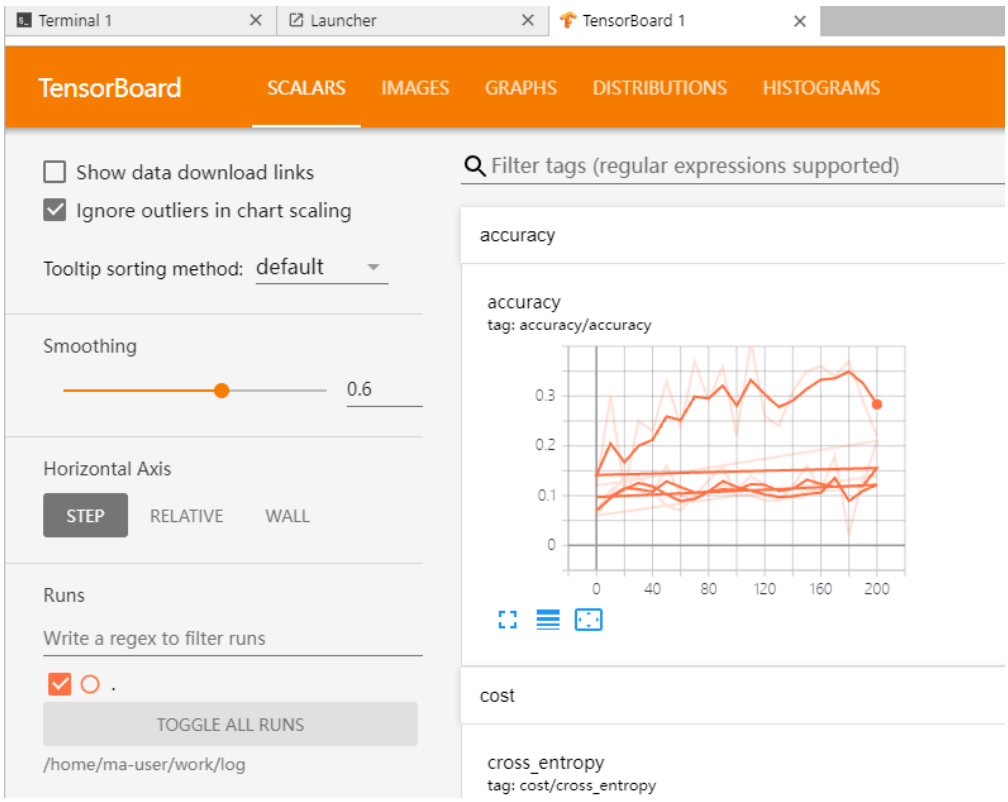


Figure 7-15 TensorBoard page (3)

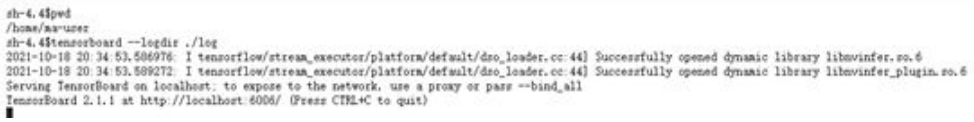


Method 4

Click Terminal and run the following command. (In this way, the UI cannot be displayed.)

```
tensorboard --logdir ./log
```


Figure 7-16 Opening TensorBoard through Terminal



Step 4 View Visualized Data on the Training Dashboard

The training dashboard is important for TensorBoard visualization. The training dashboard allows for scalar visualization, image visualization, and computational graph visualization.

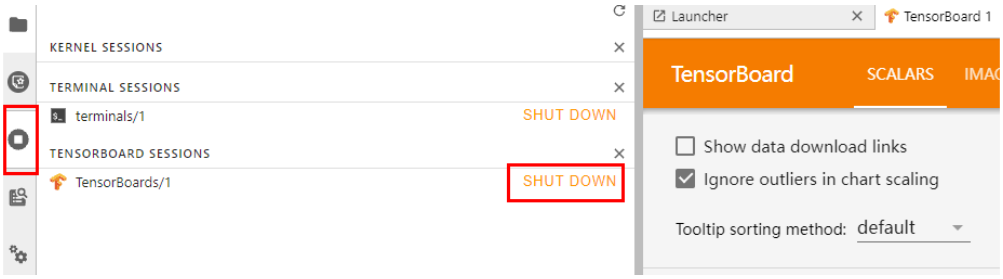
For more functions, see **Get started with TensorBoard**.

Related Operations

To stop a TensorBoard instance, perform the following steps:

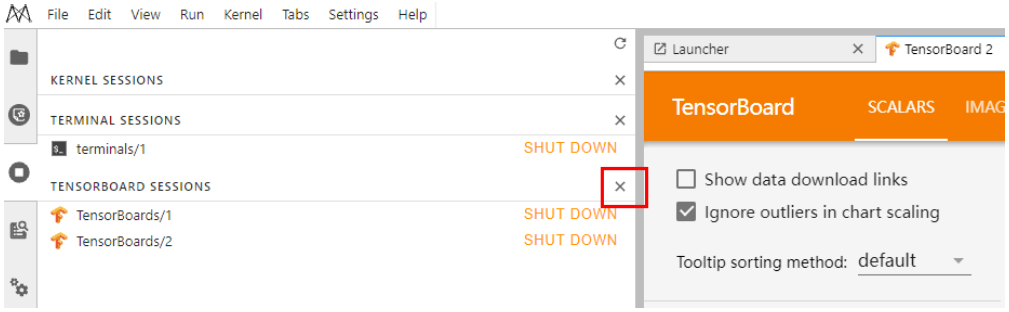
- Method 1: Enter the following command in the **.ipynb** file window of JupyterLab. (Obtain PID on the startup screen or using the command **ps -ef | grep tensorboard**.)
  !kill *PID*
- Method 2: Click  . The TensorBoard instance management page is displayed, which shows all started TensorBoard instances. Click **SHUT DOWN** next to an instance to stop it.

  Figure 7-17 Clicking SHUT DOWN to stop an instance

- Method 3: Click  in the following figure to close all started TensorBoard instances.

  Figure 7-18 Stopping all started TensorBoard instances

- Method 4 (not recommended): Close the TensorBoard window on JupyterLab. In this case, only the visualization window is closed, but the instance is still running on the backend.

8 Distributed Training

8.1 Distributed Training

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

- The development environment refers to the new-version Notebook provided by ModelArts, excluding the old-version Notebook.
- If the notebook 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.

8.2 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**.

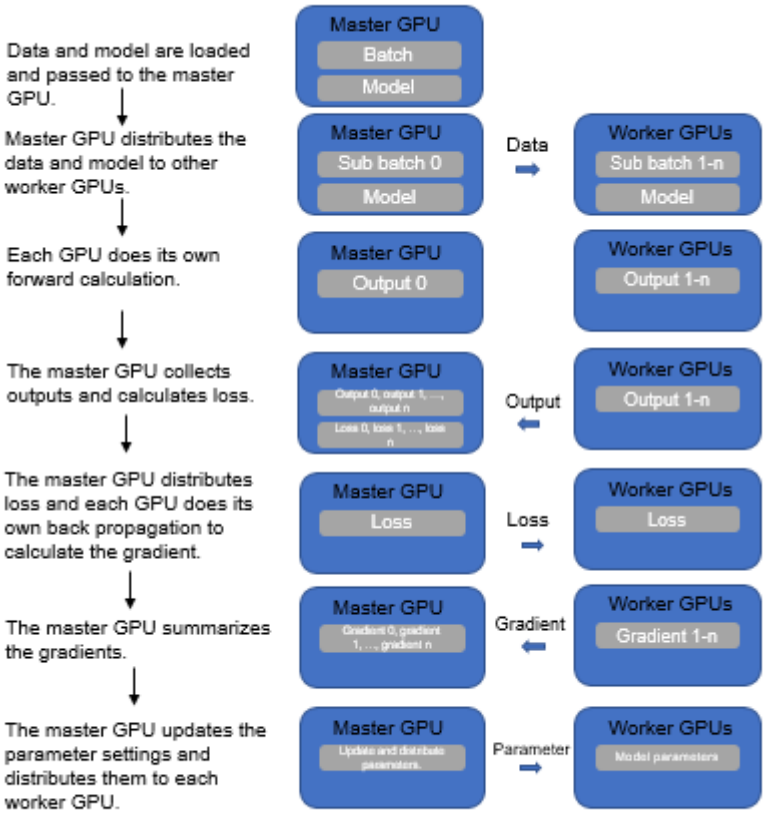
Training Process

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 8-1 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 ###
```


8.3 Multi-Node Multi-Card Training Using DistributedDataParallel

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

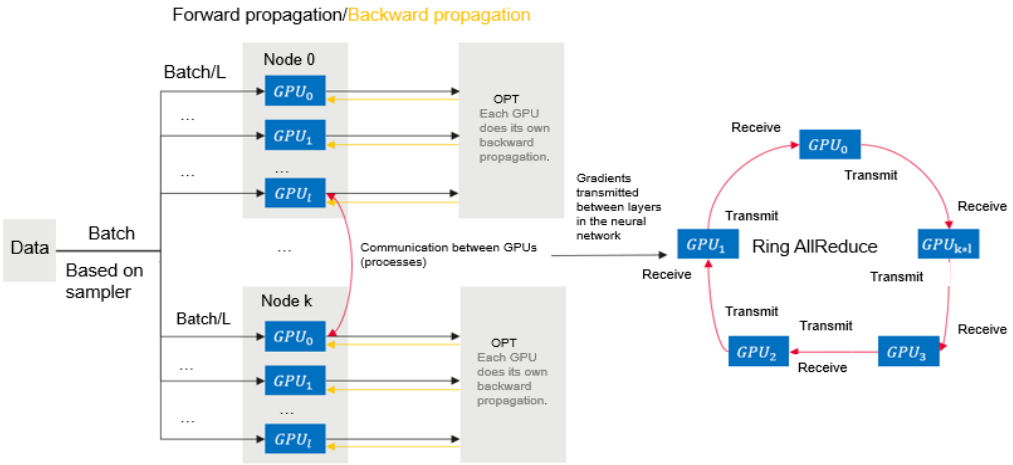
Training Process

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 8-2 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()

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

8.4 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 8-1 Training result comparison

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

8.5 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

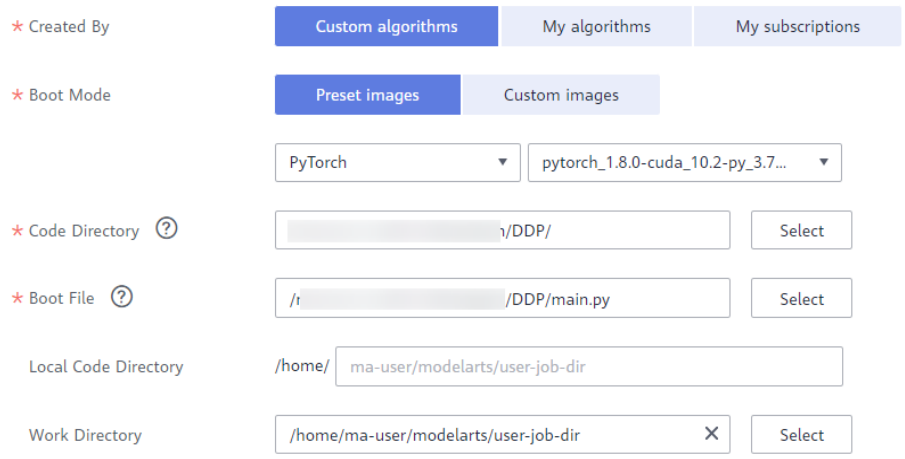
1. How Do I Use Different Datasets in the Sample Code?

- 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 (stored in the **input_dir** folder).

Figure 8-3 Creating a training job



- 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.

